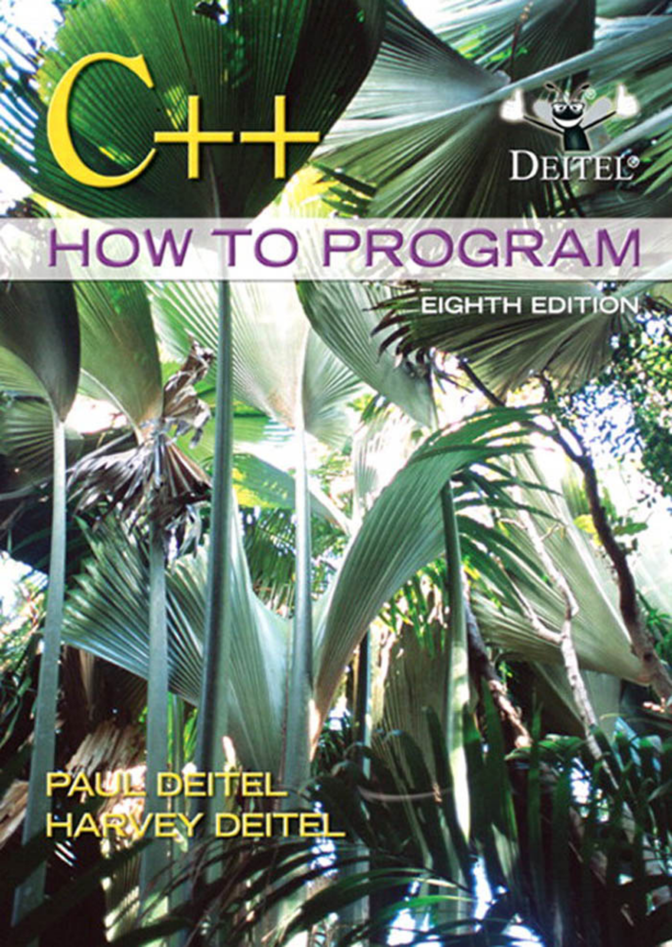

C++
DEITEL®
HOW TO PROGRAM
EIGHTH EDITION
PAUL DEITEL
HARVEY DEITEL

Deitel® How to Program Series Cover Theme

The cover theme for the DEITEL® HOW TO PROGRAM SERIES emphasizes social consciousness issues such as going green, clean energy, recycling, sustainability and more. Within the text, in addition to conventional programming exercises, we've included our Making a Difference exercise set to raise awareness of issues such as global warming, population growth, affordable healthcare, accessibility, privacy of electronic records and more. In this book, you'll use C++ to program applications that relate to these issues. We hope that what you learn in *C++ How to Program,* 8/e will help you to make a difference.

Rainforests

The world's rainforests are often referred to as the "Earth's lungs," the "jewels of the Earth" and the "world's largest pharmacy." Approximately 50% of the world's tropical rainforests are in Central and South America, over 33% are in Asia and Oceania (which consists of Australia, New Zealand and various South Pacific Islands), and 15% are in Africa. Rainforests absorb from the atmosphere vast amounts of carbon dioxide—a gas that many scientists blame for global warming—and they provide approximately 40% of the world's oxygen. They regulate water flow to surrounding areas preventing mudslides and crop loss. Rainforests also support the livelihoods of 1.6 billion people, providing food, fresh water, medicines and more. Approximately 25% of Western medicines used to treat infections, viruses, cancer and more are derived from plants found in rainforests. The U.S. National Cancer Institute has found about 2100 rainforest plant species that are effective against cancer cells. Fewer than one percent of rainforest plant species have been tested for medical use.

Rainforests are being deforested at an alarming rate. According to a March 2010 report by the United Nations Food and Agriculture Organization, deforestation has slowed over the last 10 years, but more than 30 million acres of forests are still lost annually, and they're not easily renewed. The United Nations Environment Programme Plant for the Planet: Billion Tree Campaign is one of many reforestation initiatives. To learn more about how you can make a difference, visit `www.unep.org/billiontreecampaign/index.asp`. For further information visit:

```
www.rain-tree.com/facts.htm
www.savetherainforest.org/savetherainforest_007.htm
en.wikipedia.org/wiki/Rainforest
www.rainforestfoundation.org/
```

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized authoring and corporate training organization. The company offers instructor-led courses delivered at client sites worldwide on programming languages and other software topics such as C++, Visual C++®, C, Java™, C#®, Visual Basic®, Objective-C®, XML®, Python®, JavaScript, object technology, Internet and web programming, and Android and iPhone app development. The company's clients include many of the world's largest companies, as well as government agencies, branches of the military and academic institutions. To learn more about Deitel Pearson Higher Education publications and Dive Into® Series corporate training, e-mail `deitel@deitel.com` or visit `www.deitel.com/training/`. Follow Deitel on Facebook® at `www.deitel.com/deitelfan/` and on Twitter® `@deitel`.

C++

HOW TO PROGRAM

EIGHTH EDITION

Deitel® Series Page

How To Program Series

C++ How to Program, 8/E
C How to Program, 6/E
Java™ How to Program, 9/E
Java™ How to Program, Late Objects Version, 8/E
Internet & World Wide Web How to Program, 4/E
Visual C++® 2008 How to Program, 2/E
Visual Basic® 2010 How to Program
Visual C#® 2010 How to Program, 3/E
Small Java™ How to Program, 6/E
Small C++ How to Program, 5/E

Simply Series

Simply C++: An App-Driven Tutorial Approach
Simply Java™ Programming: An App-Driven Tutorial Approach
Simply C#: An App-Driven Tutorial Approach
Simply Visual Basic® 2008, 3/E: An App-Driven Tutorial Approach

CourseSmart Web Books

`www.deitel.com/books/CourseSmart/`

C++ How to Program, 5/E, 6/E, 7/E & 8/E
Simply C++: An App-Driven Tutorial Approach
Java™ How to Program, 6/E, 7/E, 8/E & 9/E

(continued next column)

(continued)

Simply Visual Basic 2008: An App-Driven Tutorial Approach, 3/E
Visual Basic® 2010 How to Program
Visual Basic® 2008 How to Program
Visual C#® 2010 How to Program, 4/E
Visual C#® 2008 How to Program, 3/E

Deitel® Developer Series

C++ for Programmers
AJAX, Rich Internet Applications and Web Development for Programmers
Android for Programmers: An App-Driven Approach
C# 2010 for Programmers, 3/E
iPhone for Programmers: An App-Driven Approach
Java™ for Programmers
JavaScript for Programmers

LiveLessons Video Learning Products

`www.deitel.com/books/LiveLessons/`

C++ Fundamentals
Java™ Fundamentals
C# Fundamentals
iPhone® App Development Fundamentals
JavaScript Fundamentals
Visual Basic Fundamentals

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please register for the free *Deitel® Buzz Online* e-mail newsletter at:

`www.deitel.com/newsletter/subscribe.html`

follow us on Twitter®

`@deitel`

and become a Deitel & Associates fan on Facebook®

`www.deitel.com/deitelfan/`

To communicate with the authors, send e-mail to:

`deitel@deitel.com`

For information on government and corporate *Dive-Into® Series* on-site seminars offered by Deitel & Associates, Inc. worldwide, visit:

`www.deitel.com/training/`

or write to

`deitel@deitel.com`

For continuing updates on Prentice Hall/Deitel publications visit:

`www.deitel.com`
`www.pearsonhighered.com/deitel/`

Check out our Resource Centers for valuable web resources that will help you master C++, other important programming languages, software, and Internet- and web-related topics:

`www.deitel.com/ResourceCenters.html`

C++ HOW TO PROGRAM

EIGHTH EDITION

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Prentice Hall
Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director: ***Marcia J. Horton***
Editor-in-Chief: ***Michael Hirsch***
Associate Editor: ***Carole Snyder***
Vice President, Marketing: ***Patrice Jones***
Marketing Manager: ***Yezan Alayan***
Senior Marketing Coordinator: ***Kathryn Ferranti***
Vice President, Production: ***Vince O'Brien***
Managing Editor: ***Jeff Holcomb***
Associate Managing Editor: ***Robert Engelhardt***
Operations Specialist: ***Lisa McDowell***
Art Director: ***Linda Knowle***
Cover Design: ***Abbey S. Deitel, Harvey M. Deitel, Marta Samsel***
Cover Photo Credit: ***© James Hardy/PhotoAlto/Getty Images***
Media Editor: ***Daniel Sandin***
Media Project Manager: ***Wanda Rockwell***

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on page vi.

Library of Congress Cataloging-in-Publication Data

Deitel, Paul J.
C++ : how to program / P.J. Deitel, H.M. Deitel. -- 8th ed.
p. cm.
Includes index.
ISBN 978-0-13-266236-9
1. C++ (Computer program language) I. Deitel, Harvey M. II. Title.

QA76.73.C153D45 2012
005.13'3--dc22

2011000245

10 9 8 7 6 5 4 3 2 1

ISBN-10: 0-13-266236-1

ISBN-13: 978-0-13-266236-9

Prentice Hall
is an imprint of

PEARSON

In memory of Ken Olsen,
Founder of Digital Equipment Corporation (DEC):

We are deeply grateful for the opportunities DEC extended to us, enabling us to form and grow Deitel & Associates, Inc.

Paul and Harvey Deitel

Trademarks

Contents

Chapters 25–26 and Appendices F–I are PDF documents posted online at the book's Companion Website, which is accessible from www.pearsonhighered.com/deitel.

Chapters 25–26 and Appendices F–I are PDF documents posted online at the book's Companion Website, which is accessible from www.pearsonhighered.com/deitel.

Preface

"The chief merit of language is clearness …"
—Galen

For the Student

Welcome to the C++ computer programming language and *C++ How to Program, Eighth Edition*! This book presents leading-edge computing technologies, and is particularly appropriate for inroductory course sequences based on the curriculum recommendations of two key professional organizations—the ACM and the IEEE.

The new Chapter 1 presents intriguing facts and figures. Our goal is to get you excited about studying computers and programming. The chapter includes a table of some of the research made possible by computers; current technology trends and hardware discussions; the data hierarchy; social networking; a table of business and technology publications and websites that will help you stay up-to-date with the latest technology news, trends and career opportunities; additional Making a Difference exercises and more.

We focus on software engineering best practices. At the heart of the book is our signature "live-code approach"—programming concepts are presented in the context of complete working programs, rather than in code snippets. Each C++ code example is accompanied by live sample executions, so you can see exactly what each program does when it's run on a computer. All the source code is available at `www.deitel.com/books/cpphtp8/` and `www.pearsonhighered.com/deitel/`.

Much of this Preface is addressed to instructors. Please be sure to read the sections entitled Pedagogic Features; Teaching Approach; Software Used in *C++ How to Program, 8/e*; C++ IDE Resource Kit and CourseSmart Web Books.

We believe that this book and its support materials will give you an informative, interesting, challenging and entertaining C++ educational experience. As you read the book, if you have questions, send an e-mail to `deitel@deitel.com`—we'll respond promptly. For updates on this book, visit `www.deitel.com/books/cpphtp8/`, follow us on Facebook (`www.deitel.com/deitelfan`) and Twitter (`@deitel`), and subscribe to the *Deitel® Buzz Online* newsletter (`www.deitel.com/newsletter/subscribe.html`). Good luck!

New and Updated Features

Here are the updates we've made for *C++ How to Program, 8/e*:

Impending New C++ Standard

- ***Optional sections.*** We cover various features of the new standard (sometimes called C++0x and due late in 2011 or early in 2012) in *optional modular sections* and in Chapter 23. These are *easy to include or omit*. Popular compilers such as Microsoft Visual C++ 2010 and GNU C++ 4.5 already implement many of these features. To enable the new standard features in GNU C++, use the `-std=C++0x` flag when you compile the corresponding programs.

- *Boost C++ Libraries, Technical Report 1 (TR1) and C++0x.* In Chapter 23, we introduce the Boost C++ Libraries, Technical Report 1 (TR1) and C++0x. The free Boost open source libraries are created by members of the C++ community. Technical Report 1 describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. The C++ Standards Committee is revising the C++ Standard—the main goals are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language. The new standard will include many of the libraries in TR1 and changes to the core language. We overview the Boost libraries and provide code examples for the "regular expression" and "smart pointer" libraries. Regular expressions are used to match specific character patterns in text. They can be used, for example, to validate data to ensure that it's in a particular format, to replace parts of one string with another, or to split a string. Many common bugs in C and C++ code are related to pointers, a powerful programming capability you'll study in Chapter 8. Smart pointers help you avoid errors by providing additional functionality to standard pointers.
- ***`unique_ptr`* vs. *`auto_ptr`*.** We replaced our `auto_ptr` example with the impending standard's class `unique_ptr`, which fixes various problems that were associated with class `auto_ptr`. Use of `auto_ptr` is deprecated and `unique_ptr` is already implemented in many popular compilers, including Visual C++ 2010 and GNU C++ 4.5.
- *Initializer lists for user-defined types.* These enable objects of your own types to be initialized using the same syntax as built-in arrays.
- *Range-based **`for`** statement.* A version of the `for` statement that iterates over all the elements of an array or container (such as an object of the `vector` class).
- *Lambda expressions.* These enable you to create anonymous functions that can be passed to other functions as arguments.
- ***`auto`** storage class specifier.* The keyword `auto` can no longer be used as a storage class specifier.
- ***`auto`*.** This keyword now deduces the type of a variable from its initializer.
- ***`nullptr`*.** This keyword is a replacement for assigning zero to a null pointer.
- ***`static_assert`*.** This capability allows you to test certain aspects of the program at compile time.
- *New **`long long`** and **`unsigned long long`** types.* These new types were introduced for use with 64-bit machines.

Pedagogic Features

- *Enhanced Making a Difference exercises set.* We encourage you to use computers and the Internet to research and solve significant social problems. These exercises are meant to increase awareness and discussion of important issues the world is facing. We hope you'll approach them with your own values, politics and beliefs.

Check out our new Making a Difference Resource Center at `www.deitel.com/MakingADifference` for additional ideas you may want to investigate further.

- ***Page numbers for key terms in chapter summaries.*** For key terms that appear in the chapter summaries, we include the page number of each term's defining occurrence in the chapter.
- ***VideoNotes.*** The Companion Website includes 15+ hours of VideoNotes in which co-author Paul Deitel explains in detail most of the programs in the core chapters. Instructors have told us that their students find the VideoNotes valuable for preparing for and reviewing lectures.
- ***Modular presentation.*** We've grouped the chapters into teaching modules. The Chapter Dependency Chart (later in this Preface) reflects the modularization.

Object Technology

- ***Object-oriented programming and design.*** We introduce the basic concepts and terminology of object technology in Chapter 1. Students develop their first customized classes and objects in Chapter 3. Presenting objects and classes early gets students "thinking about objects" immediately and mastering these concepts more thoroughly. [For courses that require a late-objects approach, consider *C++ How to Program, Late Objects Version, Seventh Edition*, which begins with six chapters on programming fundamentals (including two on control statements) and continues with seven chapters that gradually introduce object-oriented programming concepts.]
- ***Integrated case studies.*** We provide several case studies that span multiple sections and chapters. These include development of the `GradeBook` class in Chapters 3–7, the `Time` class in Chapters 9–10, the `Employee` class in Chapters 12–13, and the optional OOD/UML ATM case study in Chapters 25–26.
- ***Integrated `GradeBook` case study.*** The `GradeBook` case study uses classes and objects in Chapters 3–7 to incrementally build a `GradeBook` class that represents an instructor's grade book and performs various calculations based on a set of student grades, such as calculating the average grade, finding the maximum and minimum, and printing a bar chart.
- ***Exception handling.*** We integrate basic exception handling early in the book. Instructors can easily pull more detailed material forward from Chapter 16, Exception Handling: A Deeper Look.
- ***Prefer `vector`s to C arrays.*** C++ offers two types of arrays—`vector` class objects (which we start using in Chapter 7) and C-style, pointer-based arrays. As appropriate, we use class template `vector` instead of C arrays throughout the book. However, we begin by discussing C arrays in Chapter 7 to prepare you for working with legacy code and to use as a basis for building your own customized `Array` class in Chapter 11.
- ***Prefer `string` objects to C strings.*** Similarly, C++ offers two types of strings—`string` class objects (which we use starting in Chapter 3) and C-style, pointer-based strings. We continue to include some early discussions of C strings to give

you practice with pointer manipulations, to illustrate dynamic memory allocation with `new` and `delete` and to prepare you for working with C strings in the legacy code that you'll encounter in industry. In new development, you should favor `string` class objects. We've replaced most occurrences of C strings with instances of C++ class `string` to make programs more robust and eliminate many of the security problems that can be caused by using C strings.

- ***Optional case study: Using the UML to develop an object-oriented design and C++ implementation of an ATM.*** The UML™ (Unified Modeling Language™) is the industry-standard graphical language for modeling object-oriented systems. Chapters 25–26 include an *optional* online case study on object-oriented design using the UML. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical requirements document that specifies the system to be built. We determine the classes needed to implement that system, the attributes the classes need to have, the behaviors the classes need to exhibit and specify how the classes must interact with one another to meet the system requirements. From the design we produce a complete C++ implementation. Students often report having a "light-bulb moment"—the case study helps them "tie it all together" and really understand object orientation.
- ***Standard Template Library (STL).*** This might be one of the most important topics in the book in terms of your appreciation of software reuse. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. Chapter 22 introduces the STL and discusses its three key components—containers, iterators and algorithms. The STL components provide tremendous expressive power, often reducing many lines of code to a single statement.

Other Features

- ***Printed book contains core content; additional chapters are online.*** Several online chapters are included for more advanced courses and for professionals. These are available in searchable PDF format on the book's password-protected Companion Website—see the access card in the front of this book.
- ***Reorganized Chapter 11, Operator Overloading; Class `string`.*** We reorganized this chapter to begin with standard library class `string` so readers can see an elegant use of operator overloading before they implement their own. We also moved the section on proxy classes to the end of Chapter 10, where it's a more natural fit.
- ***Enhanced use of `const`.*** We increased the use of `const` book-wide to encourage better software engineering.
- ***Software engineering concepts.*** Chapter 1 briefly introduces very current software engineering terminology, including agile software development, Web 2.0, Ajax, SaaS (Software as a Service), PaaS (Platform as a Service), cloud computing, web services, open source software, design patterns, refactoring, LAMP and more.
- ***Compilation and linking process for multiple-source-file programs.*** Chapter 3 includes a detailed diagram and discussion of the compilation and linking process that produces an executable program.

- *Function Call Stack Explanation.* In Chapter 6, we provide a detailed discussion with illustrations of the function call stack and activation records to explain how C++ is able to keep track of which function is currently executing, how automatic variables of functions are maintained in memory and how a function knows where to return after it completes execution.
- *Tuned Treatment of Inheritance and Polymorphism.* Chapters 12–13 have been carefully tuned using a concise `Employee` class hierarchy. We use this same treatment in our C++, Java, C# and Visual Basic books—one of our reviewers called it the best he had seen in 25 years as a trainer and consultant.
- *Discussion and illustration of how polymorphism works "under the hood."* Chapter 13 contains a detailed diagram and explanation of how C++ can implement polymorphism, `virtual` functions and dynamic binding internally. This gives students a solid understanding of how these capabilities work.
- *ISO/IEC C++ standard compliance.* We've audited our presentation against the ISO/IEC C++ standard document.
- *Debugger appendices.* We provide two Using the Debugger appendices on the book's Companion Website—Appendix H, Using the Visual Studio Debugger, and Appendix I, Using the GNU C++ Debugger.
- *Code tested on multiple platforms.* We tested the code examples on various popular C++ platforms including GNU C++ on Linux and Microsoft Windows, and Visual C++ on Windows. For the most part, the book's examples port to popular standard-compliant compilers.
- *Game Programming.* Because of limited interest, we've removed from the book Chapter 27, Game Programming with Ogre (which covers only Linux). For instructors who would like to continue using this material with *C++ How to Program, 8/e*, we've included the version from *C++ How to Program, 7/e* on the book's Companion Website.

Our Text + Digital Approach to Content

We surveyed hundreds of instructors teaching C++ courses and learned that most want a book with content focused on their introductory courses. With that in mind, we moved various advanced chapters to the web. Having this content in digital format makes it easily searchable, and gives us the ability to fix errata and add new content as appropriate. The book's Companion Website, which is accessible at

```
www.pearsonhighered.com/deitel/
```

(see the access card at the front of the book) contains the following chapters in *searchable* PDF format:

- Chapter 25, ATM Case Study, Part 1: Object-Oriented Design with the UML
- Chapter 26, ATM Case Study, Part 2: Implementing an Object-Oriented Design
- Game Programming with Ogre (from *C++ How to Program, 7/e*)
- Appendix F, C Legacy Code Topics

- Appendix G, UML 2: Additional Diagram Types
- Appendix H, Using the Visual Studio Debugger
- Appendix I, Using the GNU C++ Debugger

The Companion Website also includes:

- Extensive VideoNotes—watch and listen as co-author Paul Deitel discusses the key features of the code examples in Chapters 2–13 and portions of Chapters 16 and 17.
- Two true/false questions per section with answers for self-review.
- Solutions to approximately half of the solved exercises in the book.

The following materials are posted at the Companion Website and at `www.deitel.com/books/cpphtp8/`:

- An array of function pointers example and additional function pointer exercises (from Chapter 8).
- `String` Class Operator Overloading Case Study (from Chapter 11).
- Building Your Own Compiler exercise descriptions (from Chapter 20).

Dependency Chart

The chart on the next page shows the dependencies among the chapters to help instructors plan their syllabi. *C++ How to Program, 8/e* is appropriate for CS1 and CS2 courses.

Teaching Approach

C++ How to Program, 8/e, contains a rich collection of examples. We stress program clarity and concentrate on building well-engineered software.

Live-code approach. The book is loaded with "live-code" examples—most new concepts are presented in the context of *complete working C++ applications*, followed by one or more executions showing program inputs and outputs. In the few cases where we use a code snippet, we tested it in a complete working program, then copied and pasted it into the book.

Syntax coloring. For readability, we syntax color all the C++ code, similar to the way most C++ integrated-development environments and code editors syntax color code. Our coloring conventions are as follows:

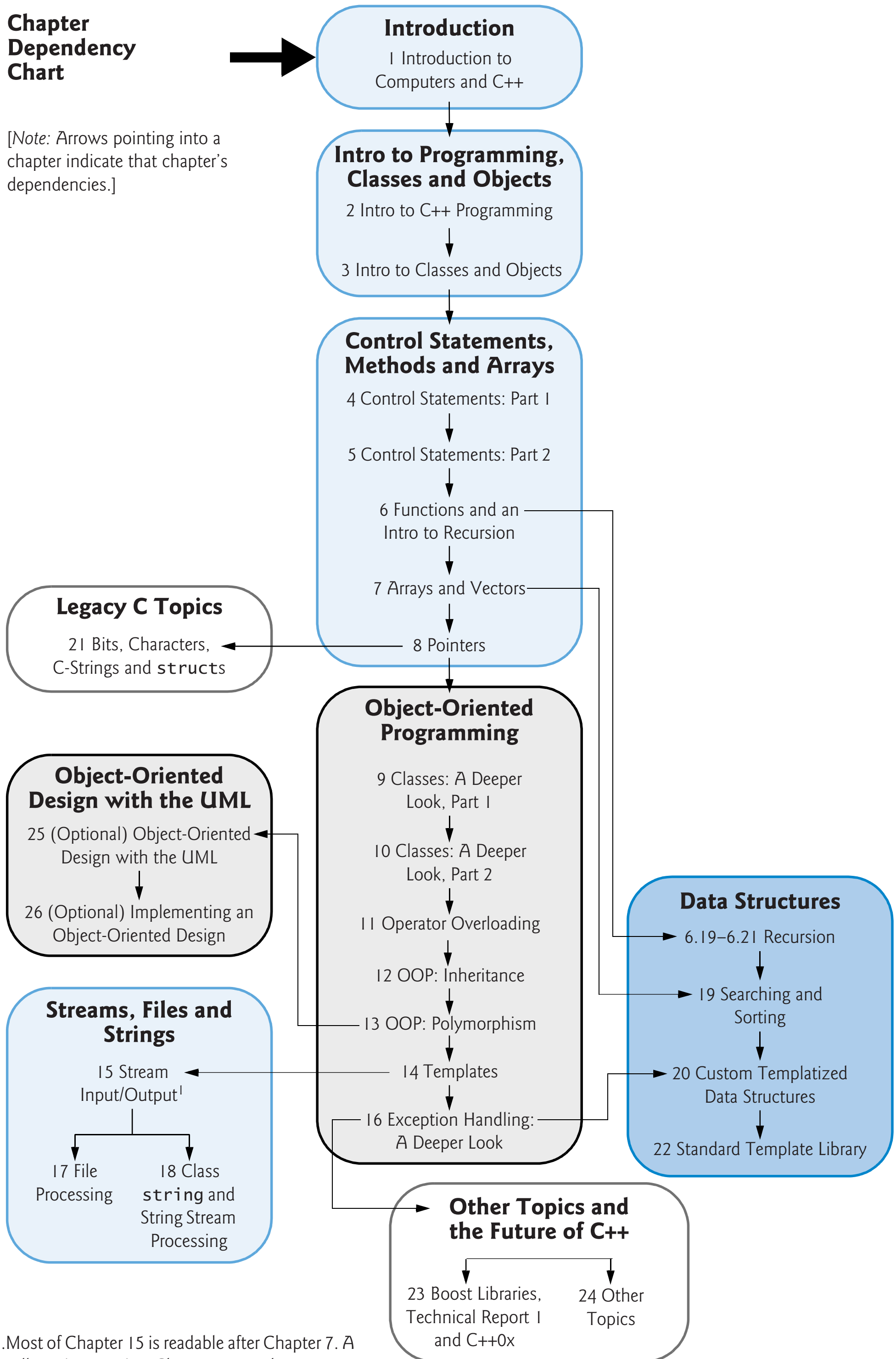
```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in black
```

Code highlighting. We place light blue shaded rectangles around each program's key code segments.

Using fonts for emphasis. We place the key terms and the index's page reference for each defining occurrence in **bold blue** text for easy reference. We emphasize on-screen components in the **bold Helvetica** font (e.g., the **File** menu) and C++ program text in the `Lucida` font (for example, `int x = 5;`).

Chapter Dependency Chart

[*Note:* Arrows pointing into a chapter indicate that chapter's dependencies.]



1. Most of Chapter 15 is readable after Chapter 7. A small portion requires Chapters 12 and 14.

Objectives. The opening quotes are followed by a list of chapter objectives.

Illustrations/ figures. Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

Programming tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined seven decades of programming and teaching experience.

Good Programming Practices

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Errors

Pointing out these Common Programming Errors *reduces the likelihood that you'll make them.*

Error-Prevention Tips

These tips contain suggestions for exposing and removing bugs from your programs; many describe aspects of C++ that prevent bugs from getting into programs in the first place.

Performance Tips

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tips

The Portability Tips *help you write code that will run on a variety of platforms.*

Software Engineering Observations

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Summary bullets. We present a section-by-section bullet-list summary of the chapter with the page references to the defining occurrence for many of the key terms in each section.

Self-review exercises and answers. Extensive self-review exercises *and* answers are included for self study. All of the exercises in the optional ATM case study are fully solved.

Exercises. Each chapter concludes with a substantial set of exercises including:

- simple recall of important terminology and concepts
- What's wrong with this code?
- What does this code do?
- writing individual statements and small portions of functions and classes
- writing complete functions, classes and programs
- major projects.

Please do not write to us requesting access to the Pearson Instructor's Resource Center which contains the book's instructor supplements, including the exercise solutions. Ac-

cess is limited strictly to college instructors teaching from the book. Instructors may obtain access only through their Pearson representatives. Solutions are *not* provided for "project" exercises. Check out our Programming Projects Resource Center for lots of additional exercise and project possibilities (`www.deitel.com/ProgrammingProjects/`).

Index. We've included an extensive index. Defining occurrences of key terms are highlighted with a **bold blue** page number.

Software Used in C++ *How to Program, 8/e*

We wrote *C++ How to Program, 8/e* using Microsoft's free Visual C++ Express Edition (which is available free for download at `www.microsoft.com/express/downloads/`) and the free GNU C++ (`gcc.gnu.org/install/binaries.html`), which is already installed on most Linux systems and can be installed on Mac OS X and Windows systems. Apple includes GNU C++ in their Xcode development tools, which Mac OS X users can download from `developer.apple.com/technologies/tools/xcode.html`.

C++ IDE Resource Kit

Your instructor may have ordered through your college bookstore a Value Pack edition of *C++ How to Program, 8/e* that comes bundled with the C++ IDE Resource Kit. This kit contains CD or DVD versions of:

- Microsoft® Visual Studio 2010 Express Edition (`www.microsoft.com/express/`)
- Dev C++ (`www.bloodshed.net/download.html`)
- NetBeans (`netbeans.org/downloads/index.html`)
- Eclipse (`eclipse.org/downloads/`)
- CodeLite (`codelite.org/LiteEditor/Download`)

You can download these software packages from the websites specified above. The C++ IDE Resource Kit also includes access to a Companion Website containing step-by-step written instructions and VideoNotes to help you get started with each development environment. If your book did not come with the C++ IDE Resource Kit, you can purchase access to the Resource Kit's Companion Website from `www.pearsonhighered.com/cppidekit/`.

CourseSmart Web Books

Today's students and instructors have increasing demands on their time and money. Pearson has responded to that need by offering digital texts and course materials online through CourseSmart. CourseSmart allows faculty to review course materials online, saving time and costs. It offers students a high-quality digital version of the text for less than the cost of a print copy of the text. Students receive the same content offered in the print textbook enhanced by search, note-taking, and printing tools. For more information, visit `www.coursesmart.com`.

Instructor Supplements

The following supplements are available to qualified instructors only through Pearson Education's Instructor Resource Center (`www.pearsonhighered.com/irc`):

- *Solutions Manual* with solutions to the vast majority of the end-of-chapter exercises and Lab Manual exercises. We've added dozens of Making a Difference exercises, most with solutions.
- *Test Item File* of multiple-choice questions (approximately two per book section)
- Customizable PowerPoint® slides containing all the code and figures in the text, plus bulleted items that summarize the key points in the text

If you're not already a registered faculty member, contact your Pearson representative or visit `www.pearsonhighered.com/educator/replocator/`.

Acknowledgments2

We'd like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the guidance, savvy and energy of Michael Hirsch, Editor-in-Chief of Computer Science. Carole Snyder recruited the book's reviewers and managed the review process. Bob Engelhardt managed the book's production.

Reviewers

We wish to acknowledge the efforts of our seventh and eighth edition reviewers. They scrutinized the text and the programs and provided countless suggestions for improving the presentation: Virginia Bailey (Jackson StateUniversity), Thomas J. Borrelli (Rochester Institute of Technology), Chris Cox (Adobe Systems), Gregory Dai (eBay), Peter J. DePasquale (The College of New Jersey), John Dibling (SpryWare), Susan Gauch (University of Arkansas), Doug Gregor (Apple, Inc.), Jack Hagemeister (Washington State University), Williams M. Higdon (University of Indiana), Wing-Ning Li (University of Arkansas), Dean Mathias (Utah State University), Robert A. McLain (Tidewater Community College), April Reagan (Microsoft), José Antonio González Seco (Parliament of Andalusia, Spain), Dave Topham (Ohlone College) and Anthony Williams (author and C++ Standards Committee member).

Well, there you have it! As you read the book, we would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. We hope you enjoy working with *C++ How to Program, Eighth Edition* as much as we enjoyed writing it!

Paul and Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of C++, Java, C#, Visual Basic, C and Internet programming courses to industry clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Stratus, Cambridge Technology Partners, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees from MIT in Electrical Engineering and a Ph.D. in Mathematics from Boston University—at both he studied computing before separate computer science degree programs were created. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., with his son, Paul J. Deitel. He and Paul are the co-authors of dozens of books and LiveLessons multimedia packages. With translations published in Japanese, German, Russian, Chinese, Spanish, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish, the Deitels' texts have earned international recognition. Dr. Deitel has delivered hundreds of professional programming language seminars to major corporations, academic institutions, government organizations and the military.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized corporate training and authoring organization specializing in computer programming languages, Internet and web software technology, object-technology and Android™ and iPhone® education and applications development. The company provides instructor-led courses delivered at client sites worldwide on major programming languages and platforms, such as C++, Visual C++®, C, Java™, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming, Android and iPhone app development, and a growing list of additional programming and software-development courses. The founders of Deitel & Associates, Inc., are Paul J. Deitel and Dr. Harvey M. Deitel. The company's clients include many of the world's largest corporations, government agencies, branches of the military, and academic institutions. Through its 35-year publishing partnership with Prentice Hall/ Pearson Higher Education, Deitel & Associates, Inc., publishes leading-edge programming textbooks, professional books, interactive multimedia *Cyber Classrooms*, and *LiveLessons* DVD-based and web-based video courses. Deitel & Associates, Inc., and the authors can be reached via e-mail at:

```
deitel@deitel.com
```

To learn more about Deitel & Associates, Inc., its publications and its *Dive Into® Series* Corporate Training curriculum delivered at client locations worldwide, visit:

```
www.deitel.com/training/
```

subscribe to the free *Deitel® Buzz Online* e-mail newsletter at:

```
www.deitel.com/newsletter/subscribe.html
```

and follow the authors on Facebook (`www.deitel.com/deitelfan`) and Twitter (`@deitel`).

Individuals wishing to purchase Deitel books, and *LiveLessons* DVD and web-based training courses can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

```
www.pearsonhighered.com
```

Introduction to Computers and C++

1

Man is still the most extraordinary computer of all.
—John F. Kennedy

Good design is good business.
—Thomas J. Watson, Founder of IBM

How wonderful it is that nobody need wait a single moment before starting to improve the world.
—Anne Frank

Objectives

In this chapter you'll learn:

- Exciting recent developments in the computer field.
- Computer hardware, software and networking basics.
- The data hierarchy.
- The different types of programming languages.
- Basic object-technology concepts.
- The importance of the Internet and the web.
- A typical C++ program-development environment.
- To test-drive a C++ application.
- Some key recent software technologies.
- How computers can help you make a difference.

Outline

1.1 Introduction

Welcome to C++—a powerful computer programming language that's appropriate for technically oriented people with little or no programming experience, and for experienced programmers to use in building substantial information systems. You're already familiar with the powerful tasks computers perform. Using this textbook, you'll write instructions commanding computers to perform those kinds of tasks. *Software* (i.e., the instructions you write) controls *hardware* (i.e., computers).

You'll learn *object-oriented programming*—today's key programming methodology. You'll create and work with many *software objects* in this text.

C++ is one of today's most popular software development languages. This text provides an introduction to programming in the version of C++ standardized in the United States through the **American National Standards Institute** (**ANSI**) and worldwide through the efforts of the **International Organization for Standardization** (**ISO**).

In use today are more than a billion general-purpose computers and billions more cell phones, smartphones and handheld devices (such as tablet computers). According to a study by eMarketer, the number of mobile Internet users will reach approximately 134 million by 2013.[1] Other studies have projected smartphone sales to surpass personal computer sales in 2011[2] and tablet sales to account for over 20% of all personal computer sales by 2015.[3] By 2014, the smartphone applications market is expected to exceed $40 billion,[4] which is creating significant opportunities for programming mobile applications.

Computing in Industry and Research

These are exciting times in the computer field. Many of the most influential and successful businesses of the last two decades are technology companies, including Apple, IBM, Hew-

1. `www.circleid.com/posts/mobile_internet_users_to_reach_134_million_by_2013/`.
2. `www.pcworld.com/article/171380/more_smartphones_than_desktop_pcs_by_2011.html`.
3. `www.forrester.com/ER/Press/Release/0,1769,1340,00.html`.
4. *Inc.*, December 2010/January 2011, pages 116–123.

lett Packard, Dell, Intel, Motorola, Cisco, Microsoft, Google, Amazon, Facebook, Twitter, Groupon, Foursquare, Yahoo!, eBay and many more—these are major employers of people who study computer science, information systems or related disciplines. At the time of this writing, Apple was the second most valuable company in the world and *the* most valuable technology company.[5] Computers are also used extensively in academic and industrial research. Figure 1.1 provides just a few examples of exciting ways in which computers are used in research and industry.

Name	Description
Internet	The Internet—a global network of computers—was made possible by the *convergence of computing and communications*. It has its roots in the 1960s, when research funding was supplied by the U.S. Department of Defense. Originally designed to connect the main computer systems of about a dozen universities and research organizations, the Internet today is accessible by billions of computers and computer-controlled devices worldwide. Computers break lengthy transmissions into packets at the sending end, route the packets to their intended receivers and ensure that those packets are received in sequence and without error at the receiving end. According to a study by Forrester Research, the average U.S. online consumer now spends as much time online as watching television (`forrester.com/rb/Research/understanding_changing_needs_of_us_online_consumer,/q/id/57861/t/2`).
Human Genome Project	The Human Genome Project was founded to identify and analyze the 20,000+ genes in human DNA . The project used computer programs to analyze complex genetic data, determine the sequences of the billions of chemical base pairs that make up human DNA and store the information in databases which have been made available to researchers in many fields. This research has led to tremendous innovation and growth in the biotechnology industry.
World Community Grid	World Community Grid (`www.worldcommunitygrid.org`) is a non-profit computing grid. People worldwide donate their unused computer processing power by installing a free secure software program that allows the World Community Grid to harness the excess power when the computers are idle. The computing power is used in place of supercomputers to conduct scientific research projects that are making a difference, including developing affordable solar energy, providing clean water to the developing world, fighting cancer, curing muscular dystrophy, finding influenza antiviral drugs, growing more nutritious rice for regions fighting hunger and more.
Medical imaging	X-ray computed tomography (CT) scans, also called CAT (computerized axial tomography) scans, take X-rays of the body from hundreds of different angles. Computers are used to adjust the intensity of the X-ray, optimizing the scan for each type of tissue, then to combine all of the information to create a 3D image.

Fig. 1.1 | A few uses for computers. (Part 1 of 3.)

5. `www.zdnet.com/blog/apple/apple-becomes-worlds-second-most-valuable-company/9047`.

Name	Description
GPS	Global Positioning System (GPS) devices use a network of satellites to retrieve location-based information. Multiple satellites send time-stamped signals to the device GPS device, which calculates the distance to each satellite based on the time the signal left the satellite and the time the signal was received. The location of each satellite and the distance to each are used to determine the exact location of the device. Based on your location, GPS devices can provide step-by-step directions, help you easily find nearby businesses (restaurants, gas stations, etc.) and points of interest, or help you find your friends.
Microsoft's SYNC®	Many Ford cars now feature Microsoft's SYNC technology, providing speech-synthesis (for reading text messages to you) and speech-recognition capabilities that allow you to use voice commands to browse music, request traffic alerts and more.
AMBER™ Alert	The AMBER (America's Missing: Broadcast Emergency Response) Alert System is used to find abducted children. Law enforcement notifies TV and radio broadcasters and state transportation officials, who then broadcast alerts on TV, radio, computerized highway signs, the Internet and wireless devices. AMBER Alert recently partnered with Facebook. Facebook users can "Like" AMBER Alert pages by location to receive alerts in their news feeds.
Robots	Robots are computerized machines that can perform tasks (including physical tasks), respond to stimuli and more. They can be used for day-to-day tasks (e.g., iRobot's Roomba vacuum), entertainment (such as robotic pets), military combat, space and deep sea exploration, manufacturing and more. In 2004, NASA's remote-controlled Mars rover—which used Java technology—explored the surface to learn about the history of water on the planet.
One Laptop Per Child (OLPC)	One Laptop Per Child (OLPC) is providing low-power, inexpensive, Internet-enabled laptops to poor children worldwide—enabling learning and reducing the digital divide (`one.laptop.org`). By providing these educational resources, OLPC is increasing the opportunities for poor children to learn and make a difference in their communities.
Game programming	The computer game business is larger than the first-run movie business. The most sophisticated video games can cost as much as $100 million to develop. Activision's *Call of Duty 2: Modern Warfare*, released in November 2009, earned $310 million in just one day in North America and the U.K. (`news.cnet.com/8301-13772_3-10396593-52.html?tag=mncol;txt`)! Online *social gaming*, which enables users worldwide to compete with one another, is growing rapidly. Zynga—creator of popular online games such as *Farmville* and *Mafia Wars*—was founded in 2007 and already has over 215 million monthly users. To accommodate the growth in traffic, Zynga is adding nearly 1,000 servers each week (`techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/`)! Video game consoles are also becoming increasingly sophisticated. The Wii Remote uses an *accelerometer* (to detect tilt and acceleration) and a sensor that determines where the device is pointing, allowing the device to respond to motion. By gesturing with the Wii Remote in hand, you can control the video game on the screen.

Fig. 1.1 | A few uses for computers. (Part 2 of 3.)

Name	Description
(cont.)	With Microsoft's Kinect for Xbox 360, you—the player—become the controller. Kinect uses a camera, depth sensor and sophisticated software to follow your body movement, allowing you to control the game (`en.wikipedia.org/wiki/Kinect`). Kinect games include dancing, exercising, playing sports, training virtual animals and more.
Internet TV	Internet TV set-top boxes (such as Apple TV and Google TV) give you access to content—such as games, news, movies, television shows and more—allowing you to access an enormous amount of content on demand; you no longer need to rely on cable or satellite television providers to get content.

Fig. 1.1 | A few uses for computers. (Part 3 of 3.)

1.2 Computers: Hardware and Software

A computer is a device that can perform computations and make logical decisions phenomenally faster than human beings can. Many of today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second! To put that in perspective, a quadrillion-instruction-per-second computer can perform in one second more than 100,000 calculations *for every person on the planet!* And—these "upper limits" are growing quickly!

Computers process data under the control of sets of instructions called **computer programs**. These programs guide the computer through orderly sets of actions specified by people called computer **programmers**. The programs that run on a computer are referred to as **software**. In this book, you'll learn today's key programming methodology that's enhancing programmer productivity, thereby reducing software-development costs—*object-oriented programming*.

A computer consists of various devices referred to as **hardware** (e.g., the keyboard, screen, mouse, hard disks, memory, DVDs and processing units). Computing costs are *dropping dramatically*, owing to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials—it's an ingredient in common sand. Silicon-chip technology has made computing so economical that more than a billion general-purpose computers are in use worldwide, and this is expected to *double* in the next few years.

Computer chips (*microprocessors*) control countless devices. These **embedded systems** include anti-lock brakes in cars, navigation systems, smart home appliances, home security systems, cell phones and smartphones, robots, intelligent traffic intersections, collision avoidance systems, video game controllers and more. The vast majority of the microprocessors produced each year are embedded in devices other than general-purpose computers.[6]

6. `www.eetimes.com/electronics-blogs/industrial-control-designline-blog/4027479/Real-men-program-in-C?pageNumber=1`.

Moore's Law

Every year, you probably expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the costs of hardware supporting these technologies. For many decades, hardware costs have fallen rapidly. Every year or two, the capacities of computers have approximately *doubled* without any increase in price. This remarkable observation often is called **Moore's Law**, named for the person who identified the trend, Gordon Moore, co-founder of Intel—a leading manufacturer of the processors in today's computers and embedded systems. Moore's Law and related observations are especially true in relation to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) they have to hold programs and data over longer periods of time, and their processor speeds—the speeds at which computers execute their programs (i.e., do their work). Similar growth has occurred in the communications field, in which costs have plummeted as enormous demand for communications bandwidth (i.e., information-carrying capacity) has attracted intense competition. We know of no other fields in which technology improves so quickly and costs fall so rapidly. Such phenomenal improvement is truly fostering the *Information Revolution*.

1.3 Data Hierarchy

Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on. Figure 1.2 illustrates a portion of the data hierarchy. Figure 1.3 summarizes the data hierarchy's levels.

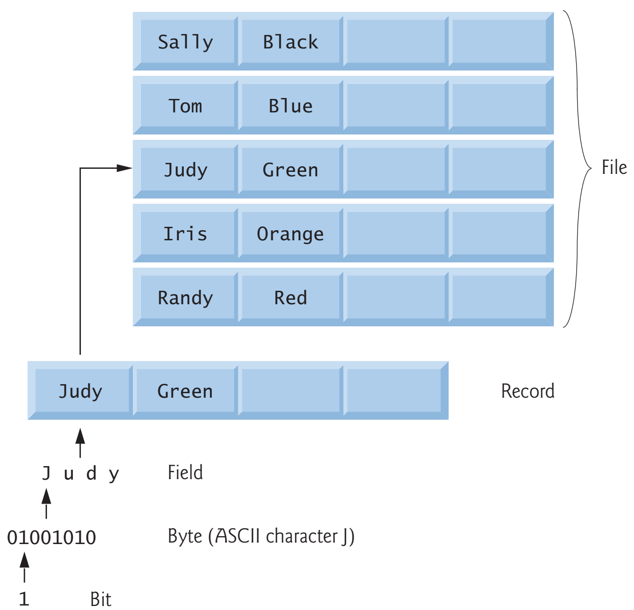


Fig. 1.2 | Data hierarchy.

Level	Description
Bits	The smallest data item in a computer can assume the value `0` or the value `1`. Such a data item is called a **bit** (short for "binary digit"—a digit that can assume one of two values). It's remarkable that the impressive functions performed by computers involve only the simplest manipulations of `0`s and `1`s—*examining a bit's value*, *setting a bit's value* and *reversing a bit's value* (from `1` to `0` or from `0` to `1`).
Characters	It's tedious for people to work with data in the low-level form of bits. Instead, they prefer to work with *decimal digits* (0–9), *letters* (A–Z and a–z), and *special symbols* (e.g., $, @, %, &, *, (,), –, +, ", :, ? and /). Digits, letters and special symbols are known as **characters**. The computer's **character set** is the set of all the characters used to write programs and represent data items. Computers process only `1`s and `0`s, so a computer's character set represents every character as a pattern of `1`s and `0`s. C++ uses the **ASCII (American Standard Code for Information Interchange)** character set (Appendix B).
Fields	Just as characters are composed of bits, **fields** are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters can be used to represent a person's name, and a field consisting of decimal digits could represent a person's age.
Records	Several related fields can be used to compose a **record** (implemented as a `class` in Java). In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses): • Employee identification number (a whole number) • Name (a string of characters) • Address (a string of characters) • Hourly pay rate (a number with a decimal point) • Year-to-date earnings (a number with a decimal point) • Amount of taxes withheld (a number with a decimal point) Thus, a record is a group of related fields. In the preceding example, all the fields belong to the same employee. A company might have many employees and a payroll record for each one.
Files	A **file** is a group of related records. [*Note:* More generally, a file contains arbitrary data in arbitrary formats. In some operating systems, a file is viewed simply as a *sequence of bytes*—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer.] It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information.

Fig. 1.3 | Levels of the data hierarchy.

1.4 Computer Organization

Regardless of differences in physical appearance, computers can be envisioned as divided into various **logical units** or sections (Fig. 1.4).

Logical unit	Description
Input unit	This "receiving" section obtains information (data and computer programs) from **input devices** and places it at the disposal of the other units for processing. Most information is entered into computers through keyboards, touch screens and mouse devices. Other forms of input include speaking to your computer, scanning images and barcodes, reading from secondary storage devices (like hard drives, DVD drives, Blu-ray Disc™ drives and USB flash drives—also called "thumb drives" or "memory sticks"), receiving video from a webcam and having your computer receive information from the Internet (such as when you download videos from YouTube™ or e-books from Amazon). Newer forms of input include reading position data from a GPS device, and motion and orientation information from an accelerometer in a smartphone or game controller.
Output unit	This "shipping" section takes information that the computer has processed and places it on various **output devices** to make it available for use outside the computer. Most information that's output from computers today is displayed on screens, printed on paper, played as audio or video on portable media players (such as Apple's popular iPods) and giant screens in sports stadiums, transmitted over the Internet or used to control other devices, such as robots and "intelligent" appliances.
Memory unit	This rapid-access, relatively low-capacity "warehouse" section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is *volatile*—it's typically lost when the computer's power is turned off. The memory unit is often called either **memory** or **primary memory**. Typical main memories on desktop and notebook computers contain between 1 GB and 8 GB (GB stands for gigabytes; a gigabyte is approximately one billion bytes).
Arithmetic and logic unit (ALU)	This "manufacturing" section performs *calculations*, such as addition, subtraction, multiplication and division. It also contains the *decision* mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they're equal. In today's systems, the ALU is usually implemented as part of the next logical unit, the CPU.
Central processing unit (CPU)	This "administrative" section coordinates and supervises the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices. Many of today's computers have multiple CPUs and, hence, can perform many operations simultaneously. A **multi-core processor** implements multiple processors on a single integrated-circuit chip—a *dual-core processor* has two CPUs and a *quad-core processor* has four CPUs. Today's desktop computers have processors that can execute billions of instructions per second.
Secondary storage unit	This is the long-term, high-capacity "warehousing" section. Programs or data not actively being used by the other units normally are placed on secondary

Fig. 1.4 | Logical units of a computer. (Part 1 of 2.)

Logical unit	Description
Secondary storage unit (cont.)	storage devices (e.g., your *hard drive*) until they're again needed, possibly hours, days, months or even years later. Information on secondary storage devices is *persistent*—it's preserved even when the computer's power is turned off. Secondary storage information takes much longer to access than information in primary memory, but the cost per unit of secondary storage is much less than that of primary memory. Examples of secondary storage devices include CD drives, DVD drives and flash drives, some of which can hold up to 128 GB. Typical hard drives on desktop and notebook computers can hold up to 2 TB (TB stands for terabytes; a terabyte is approximately one trillion bytes).

Fig. 1.4 | Logical units of a computer. (Part 2 of 2.)

1.5 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate *translation* steps. Hundreds of such languages are in use today. These may be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can directly understand only its own **machine language**, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are *machine dependent* (a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans. For example, here's a section of an early machine-language program that adds overtime pay to base pay and stores the result in gross pay:

```
+1300042774
+1400593419
+1200274027
```

Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of **assembly languages**. *Translator programs* called **assemblers** were developed to convert early assembly-language programs to machine language at computer speeds. The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay:

```
load    basepay
add     overpay
store   grosspay
```

Although such code is clearer to humans, it's incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. Translator programs called **compilers** convert high-level language programs into machine language. High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a *single* statement such as

```
grossPay = basePay + overTimePay
```

From the programmer's standpoint, high-level languages are preferable to machine and assembly languages. C++, C, Microsoft's .NET languages (e.g., Visual Basic, Visual C++ and Visual C#) and Java are among the most widely used high-level programming languages.

Compiling a large high-level language program into machine language can take a considerable amount of computer time. *Interpreter* programs were developed to execute high-level language programs directly (without the delay of compilation), although slower than compiled programs run.

1.6 Introduction to Object Technology

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. *Objects*, or more precisely—as we'll see in Chapter 3—the *classes* objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers are discovering that using a modular, object-oriented design and implementation approach can make software-development groups much more productive than was possible with earlier popular techniques like "structured programming"—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

To help you understand objects and their contents, let's begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel "hides" the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Just as you cannot cook meals in the kitchen of a blueprint, you cannot drive a car's engineering drawings. Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

Member Functions and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **member function**, which houses the program statements that actually perform its task. The member function hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In C++, we create a program unit called a **class** to house the set of member functions that perform the class's tasks. For example, a class that represents a bank account might contain one member function to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account's current balance is. A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

Instantiation

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* of a class before a program can perform the tasks that the class's member functions define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

Reuse

Just as a car's engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing*, *debugging* and *performance* tuning. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

Software Engineering Observation 1.1

Use a building-block approach to creating your programs. Avoid reinventing the wheel—use existing pieces wherever possible. This software reuse *is a key benefit of object-oriented programming.*

Messages and Member Function Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **member function call** that tells a member function of the object to perform its task. For example, a program might call a particular bank account object's *deposit* member function to increase the account's balance.

Attributes and Data Members

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a *balance attribute* that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **data members**.

Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and member functions into objects—an object's attributes and member functions are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class "convertible" certainly *is an* object of the more *general* class "automobile," but more *specifically*, the roof can be raised or lowered.

Object-Oriented Analysis and Design (OOAD)

Soon you'll be writing programs in C++. How will you create the **code** (i.e., the program instructions) for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1,000 software developers building the next U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C++ are object oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 3 and 4, then use them in our deeper treatment of object-oriented programming through

Chapter 13. In our *optional* ATM Software Engineering Case Study in Chapters 25–26 we present a simple subset of the UML's features as we guide you through an object-oriented design experience.

1.7 Operating Systems

Operating systems are software systems that make using computers more convenient for users, application developers and system administrators. Operating systems provide services that allow each application to execute safely, efficiently and *concurrently* (i.e., in parallel) with other applications. The software that contains the core components of the operating system is called the **kernel**. Popular desktop operating systems include Linux, Windows 7 and Mac OS X. Popular mobile operating systems used in smartphones and tablets include Google's Android, BlackBerry OS and Apple's iOS (for its iPhone, iPad and iPod Touch devices).

Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS—an enormously popular personal-computer operating system of the time that users interacted with by typing commands. Windows borrowed from many concepts (such as icons, menus and windows) popularized by early Apple Macintosh operating systems and originally developed by Xerox PARC. Windows 7 is Microsoft's latest operating system—its features include enhancements to the user interface, faster startup times, further refinement of security features, touch-screen and multi-touch support, and more. Windows is a *proprietary* operating system—it's controlled by one company exclusively. Windows is by far the world's most widely used operating system.

Linux—An Open-Source Operating System

The Linux operating system is perhaps the greatest success of the *open-source* movement. **Open-source software** is a software development style that departs from the *proprietary* development that dominated software's early years. With open-source development, individuals and companies contribute their efforts in developing, maintaining and evolving software in exchange for the right to use that software for their own purposes, typically at no charge. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster. Open source also encourages more innovation.

Some organizations in the open-source community are the Eclipse Foundation (the Eclipse Integrated Development Environment helps C++ programmers conveniently develop software), the Mozilla Foundation (creators of the Firefox web browser), the Apache Software Foundation (creators of the Apache web server used to develop web-based applications) and SourceForge (which provides the tools for managing open source projects—it has over 260,000 of them under development). Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create a software-based business now than just a few decades ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.[7]

The **Linux** kernel is the core of the most popular open-source, freely distributed, full-featured operating system. It's developed by a loosely organized team of volunteers, and is

7. `developers.facebook.com/opensource/`.

popular in servers, personal computers and embedded systems. Unlike that of proprietary operating systems like Microsoft's Windows and Apple's Mac OS X, Linux source code (the program code) is available to the public for examination and modification and is free to download and install. As a result, users of the operating system benefit from a community of developers actively debugging and improving the kernel, an absence of licensing fees and restrictions, and the ability to completely customize the operating system to meet specific needs.

In 1991, Linus Torvalds, a 21-year-old student at the University of Helsinki, Finland, began developing the Linux kernel as a hobby. (The name Linux is derived from "Linus" and "UNIX"—an operating system developed by Bell Labs in 1969.) Torvalds wished to improve upon the design of Minix, an educational operating system created by Professor Andrew Tanenbaum of the Vrije Universiteit in Amsterdam. The Minix source code was publicly available to allow professors to demonstrate basic operating-system implementation concepts to their students.

Torvalds released the first version of Linux in 1991. The favorable response led to the creation of a community that has continued to develop and support Linux. Developers downloaded, tested, and modified the Linux code, submitting bug fixes and feedback to Torvalds, who reviewed them and applied the improvements to the code.

The 1994 release of Linux included many features commonly found in a mature operating system, making Linux a viable alternative to UNIX. Enterprise systems companies such as IBM and Oracle became increasingly interested in Linux as it continued to stabilize and spread to new platforms.

A variety of issues—such as Microsoft's market power, the small number of user-friendly Linux applications and the diversity of Linux distributions, such as Red Hat Linux, Ubuntu Linux and many others—have prevented widespread Linux use on desktop computers. But Linux has become extremely popular on servers and in embedded systems, such as Google's Android-based smartphones.

Android

Android—the fastest growing mobile and smartphone operating system—is based on the Linux kernel and Java. One benefit of developing Android apps is the openness of the platform. The operating system is open source and free.

The Android operating system was developed by Android, Inc., which was acquired by Google in 2005. In 2007, the Open Handset Alliance™—a consortium of 34 companies initially and 79 by 2010—was formed to continue developing Android. As of December 2010, more than 300,000 Android smartphones were being activated each day![8] Android smartphones are now outselling iPhones.[9] The Android operating system is used in numerous smartphones (such as the Motorola Droid, HTC EVO™ 4G, Samsung Vibrant™ and many more), e-reader devices (such as the Barnes and Noble Nook™), tablet computers (such as the Dell Streak, the Samsung Galaxy Tab and more), in-store touch-screen kiosks, cars, robots and multimedia players.

Android smartphones include the functionality of a mobile phone, Internet client (for web browsing and Internet communication), MP3 player, gaming console, digital camera

8. `www.pcmag.com/article2/0,2817,2374076,00.asp`.
9. `mashable.com/2010/08/02/android-outselling-iphone-2/`.

and more, wrapped into handheld devices with full-color *multitouch screens*—these allow you to control the device with *gestures* involving one touch or multiple simultaneous touches. You can download apps directly onto your Android device through Android Market and other app marketplaces. As of December 2010, there were over 200,000 apps in Google's Android Market.

1.8 Programming Languages

In this section, we provide brief comments on several popular programming languages (Fig. 1.5). In the next section we introduce C++.

Programming language	Description
Fortran	Fortran (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. It's still widely used and its latest versions support object-oriented programming.
COBOL	COBOL (COmmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users based on a language developed by Grace Hopper, a career U.S. Navy officer and computer scientist. COBOL is still widely used for commercial applications that require precise and efficient manipulation of large amounts of data. Its latest version supports object-oriented programming.
Pascal	Research in the 1960s resulted in *structured programming*—a disciplined approach to writing programs that are clearer, easier to test and debug and easier to modify than large programs produced with previous techniques. One of the more tangible results of this research was the development of Pascal by Professor Niklaus Wirth in 1971. It was designed for teaching structured programming and was popular in college courses for several decades.
Ada	Ada, based on Pascal, was developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s. The DOD wanted a single language that would fill most of its needs. The Pascal-based language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. She's credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). Its latest version supports object-oriented programming.
Basic	Basic was developed in the 1960s at Dartmouth College to familiarize novices with programming techniques. Many of its latest versions are object oriented.

Fig. 1.5 | Other programming languages. (Part 1 of 3.)

Programming language	Description
C	C was implemented in 1972 by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system's development language. Today, most of the code for general-purpose operating systems is written in C or C++.
Objective-C	Objective-C is an object-oriented language based on C. It was developed in the early 1980s and later acquired by Next, which in turn was acquired by Apple. It has become the key programming language for the Mac OS X operating system and all iOS-powered devices (such as iPods, iPhones and iPads).
Java	Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in the C++-based object-oriented programming language called Java. A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-control devices. This is sometimes called "write once, run anywhere." Java is used to develop large-scale enterprise applications, to enhance the functionality of web servers (the computers that provide the content we see in our web browsers), to provide applications for consumer devices (e.g., smartphones, television set-top boxes and more) and for many other purposes.
Visual Basic	Microsoft's Visual Basic language was introduced in the early 1990s to simplify the development of Microsoft Windows applications. Its latest versions support object-oriented programming.
Visual C#	Microsoft's three object-oriented primary programming languages are Visual Basic (based on the original Basic), Visual C++ (based on C++) and C# (based on C++ and Java, and developed for integrating the Internet and the web into computer applications).
PHP	PHP is an object-oriented, "open-source" (see Section 1.7) "scripting" language supported by a community of users and developers and is used by numerous websites including Wikipedia and Facebook. PHP is platform independent—implementations exist for all major UNIX, Linux, Mac and Windows operating systems. PHP also supports many databases, including MySQL.
Perl	Perl (Practical Extraction and Report Language), one of the most widely used object-oriented scripting languages for web programming, was developed in 1987 by Larry Wall. It features rich text-processing capabilities and flexibility.
Python	Python, another object-oriented scripting language, was released publicly in 1991. Developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam (CWI), Python draws heavily from Modula-3—a systems programming language. Python is "extensible"—it can be extended through classes and programming interfaces.

Fig. 1.5 | Other programming languages. (Part 2 of 3.)

Programming language	Description
JavaScript	JavaScript is the most widely used scripting language. It's primarily used to add programmability to web pages—for example, animations and interactivity with the user. It's provided with all major web browsers.
Ruby on Rails	Ruby—created in the mid-1990s by Yukihiro Matsumoto—is an open-source, object-oriented programming language with a simple syntax that's similar to Perl and Python. Ruby on Rails combines the scripting language Ruby with the Rails web application framework developed by 37Signals. Their book, *Getting Real* (`getting-real.37signals.com/toc.php`), is a must read for web developers. Many Ruby on Rails developers have reported productivity gains over other languages when developing database-intensive web applications. Ruby on Rails was used to build Twitter's user interface.
Scala	Scala (`www.scala-lang.org/node/273`)—short for "scalable language"—was designed by Martin Odersky, a professor at École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Released in 2003, Scala uses both the object-oriented programming and functional programming paradigms and is designed to integrate with Java. Programming in Scala can reduce the amount of code in your applications significantly. Twitter and Foursquare use Scala.

Fig. 1.5 | Other programming languages. (Part 3 of 3.)

1.9 C++ and a Typical C++ Development Environment

C++ evolved from C, which was developed by Dennis Ritchie at Bell Laboratories. C is available for most computers and is hardware independent. With careful design, it's possible to write C programs that are **portable** to most computers.

The widespread use of C with various kinds of computers (sometimes called **hardware platforms**) unfortunately led to many variations. A standard version of C was needed. The American National Standards Institute (ANSI) cooperated with the International Organization for Standardization (ISO) to standardize C worldwide; the joint standard document was published in 1990 and is referred to as *ANSI/ISO 9899: 1990*.

C99 is the latest ANSI standard for the C programming language. It was developed to evolve the C language to keep pace with increasingly powerful hardware and ever more demanding user requirements. C99 also makes C more consistent with C++. For more information on C and C99, see our book *C How to Program, 6/e* and our C Resource Center (located at `www.deitel.com/C`).

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that "spruce up" the C language, but more importantly, it provides capabilities for object-oriented programming.

You'll begin developing customized, reusable classes and objects in Chapter 3, Introduction to Classes, Objects and Strings. The book is object oriented, where appropriate, from the start and throughout the text.

We also provide an optional automated teller machine (ATM) case study in Chapters 25–26, which contains a complete C++ implementation. The case study presents a carefully paced introduction to object-oriented design using the UML—an industry standard graphical modeling language for developing object-oriented systems. We guide you through a friendly design experience intended for the novice.

C++ Standard Library

C++ programs consist of pieces called **classes** and **functions.** You can program each piece yourself, but most C++ programmers take advantage of the rich collections of classes and functions in the **C++ Standard Library**. Thus, there are really two parts to learning the C++ "world." The first is learning the C++ language itself; the second is learning how to use the classes and functions in the C++ Standard Library. We discuss many of these classes and functions. P. J. Plauger's book, *The Standard C Library* (Upper Saddle River, NJ: Prentice Hall PTR, 1992), is a must read for programmers who need a deep understanding of the ANSI C library functions included in C++. Many special-purpose class libraries are supplied by independent software vendors.

Software Engineering Observation 1.2

*Use a "building-block" approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called **software reuse**, this practice is central to object-oriented programming.*

Software Engineering Observation 1.3

When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

The advantage of creating your own functions and classes is that you'll know exactly how they work. You'll be able to examine the C++ code. The disadvantage is the time-consuming and complex effort that goes into designing, developing and maintaining new functions and classes that are correct and that operate efficiently.

Performance Tip 1.1

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they're written carefully to perform efficiently. This technique also shortens program development time.

Portability Tip 1.1

Using C++ Standard Library functions and classes instead of writing your own improves program portability, because they're included in every C++ implementation.

We now explain the commonly used steps in creating and executing a C++ application using a C++ development environment (illustrated in Figs. 1.6–1.11). C++ systems generally consist of three parts: a program development environment, the language and the C++ Standard Library. C++ programs typically go through six phases: edit, preprocess, compile, link, load and execute. The following discussion explains a typical C++ program development environment.

Phase 1: Creating a Program

Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor* (Fig. 1.6). You type a C++ program (typically referred to as **source code**) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. C++ source code filenames often end with the `.cpp`, `.cxx`, `.cc` or `.C` extensions (note that `C` is in uppercase) which indicate that a file contains C++ source code. See the documentation for your C++ compiler for more information on file-name extensions.

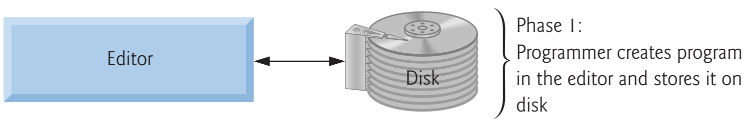


Fig. 1.6 | Typical C++ development environment—editing phase.

Two editors widely used on Linux systems are `vi` and `emacs`. C++ software packages for Microsoft Windows such as Microsoft Visual C++ (`microsoft.com/express`) have editors integrated into the programming environment. You can also use a simple text editor, such as Notepad in Windows, to write your C++ code.

For organizations that develop substantial information systems, **integrated development environments (IDEs)** are available from many major software suppliers. IDEs provide tools that support the software-development process, including editors for writing and editing programs and debuggers for locating **logic errors**—errors that cause programs to execute incorrectly. Popular IDEs include Microsoft® Visual Studio 2010 Express Edition, Dev C++, NetBeans, Eclipse and CodeLite.

Phase 2: Preprocessing a C++ Program

In Phase 2, you give the command to **compile** the program (Fig. 1.7). In a C++ system, a **preprocessor** program executes automatically before the compiler's translation phase begins (so we call preprocessing Phase 2 and compiling Phase 3). The C++ preprocessor obeys commands called **preprocessor directives,** which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually include other text files to be compiled, and perform various text replacements. The most common preprocessor directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in Appendix E, Preprocessor.

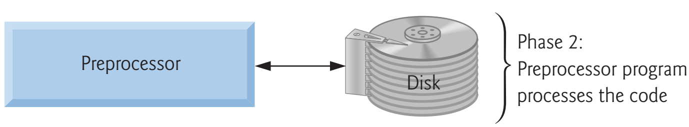


Fig. 1.7 | Typical C++ development environment—preprocessor phase.

Phase 3: Compiling a C++ Program

In Phase 3, the compiler translates the C++ program into machine-language code—also referred to as object code (Fig. 1.8).

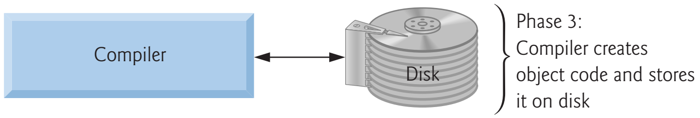


Fig. 1.8 | Typical C++ development environment—compilation phase.

Phase 4: Linking

Phase 4 is called **linking.** C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project (Fig. 1.9). The object code produced by the C++ compiler typically contains "holes" due to these missing parts. A **linker** links the object code with the code for the missing functions to produce an **executable program** (with no missing pieces). If the program compiles and links correctly, an executable image is produced.

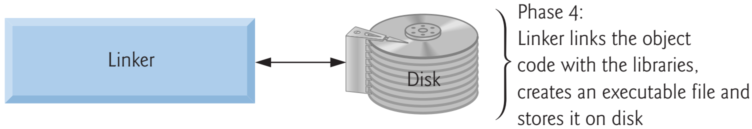


Fig. 1.9 | Typical C++ development environment—linking phase.

Phase 5: Loading

Phase 5 is called **loading.** Before a program can be executed, it must first be placed in memory (Fig. 1.10). This is done by the **loader,** which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.

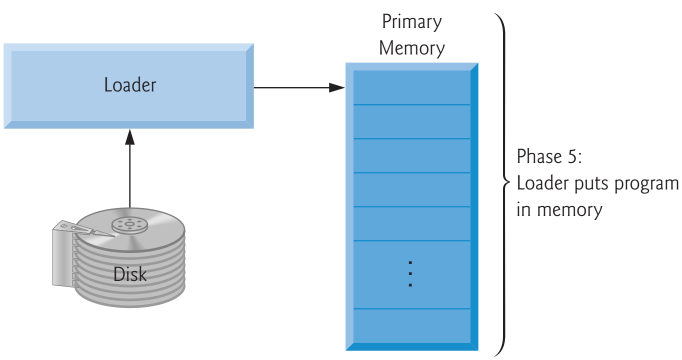


Fig. 1.10 | Typical C++ development environment—loading phase.

Phase 6: Execution

Finally, the computer, under the control of its CPU, **executes** the program one instruction at a time (Fig. 1.11). Some modern computer architectures can execute several instructions in parallel.

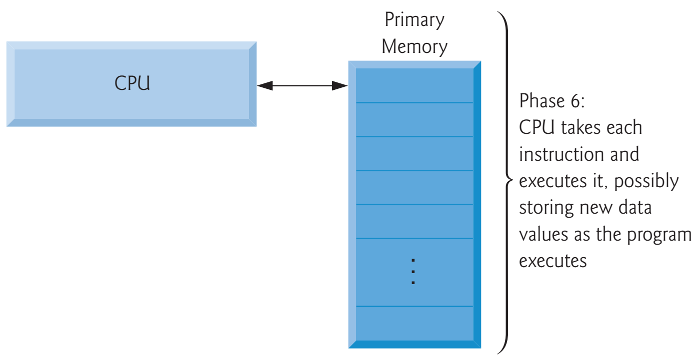


Fig. 1.11 | Typical C++ development environment—execution phase.

Problems That May Occur at Execution Time

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss throughout this book. For example, an executing program might try to divide by zero (an illegal operation for whole-number arithmetic in C++). This would cause the C++ program to display an error message. If this occurred, you'd have to return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections fixed the problem(s). [*Note:* Most programs in C++ input or output data. Certain C++ functions take their input from `cin` (the **standard input stream**; pronounced "see-in"), which is normally the keyboard, but `cin` can be redirected to another device. Data is often output to `cout` (the **standard output stream**; pronounced "see-out"), which is normally the computer screen, but `cout` can be redirected to another device. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to other devices, such as disks and hardcopy printers. There is also a **standard error stream** referred to as **`cerr`**. The `cerr` stream (normally connected to the screen) is used for displaying error messages.

Common Programming Error 1.1

*Errors such as division by zero occur as a program runs, so they're called **runtime errors** or **execution-time errors**. **Fatal runtime errors** cause programs to terminate immediately without having successfully performed their jobs. **Nonfatal runtime errors** allow programs to run to completion, often producing incorrect results.*

1.10 Test-Driving a C++ Application

In this section, you'll run and interact with your first C++ application. You'll begin by running an entertaining guess-the-number game, which picks a number from 1 to 1000 and prompts you to guess it. If your guess is correct, the game ends. If your guess is not correct, the application indicates whether your guess is higher or lower than the correct number. There is no limit on the number of guesses you can make. [*Note:* For this test drive only, we've modified this application from the exercise you'll be asked to create in Chapter 6, Functions and an Introduction to Recursion. Normally this application randomly selects the correct answer as you execute the program. The modified application uses the same correct answer every time the program executes (though this may vary by compiler), so you can use the same guesses we use in this section and see the same results as we walk you through interacting with your first C++ application.]

We'll demonstrate running a C++ application using the Windows **Command Prompt** and a shell on Linux. The application runs similarly on both platforms. Many development environments are available in which you can compile, build and run C++ applications, such as GNU C++, Dev C++, Microsoft Visual C++, CodeLite, NetBeans, Eclipse etc. Consult your instructor for information on your specific development environment.

In the following steps, you'll run the application and enter various numbers to guess the correct number. The elements and functionality that you see in this application are typical of those you'll learn to program in this book. We use fonts to distinguish between features you see on the screen (e.g., the **Command Prompt**) and elements that are not directly related to the screen. We emphasize screen features like titles and menus (e.g., the **File** menu) in a semibold **sans-serif Helvetica** font and to emphasize filenames, text displayed by an application and values you should enter into an application (e.g., `GuessNumber` or `500`) in a `sans-serif Lucida` font. As you've noticed, the **defining occurrence** of each term is set in blue, bold type. For the figures in this section, we point out significant parts of the application. To make these features more visible, we've modified the background color of the **Command Prompt** window (for the Windows test drive only). To modify the **Command Prompt** colors on your system, open a **Command Prompt** by selecting **Start > All Programs > Accessories > Command Prompt**, then right click the title bar and select **Properties**. In the **"Command Prompt" Properties** dialog box that appears, click the **Colors** tab, and select your preferred text and background colors.

Running a C++ Application from the Windows Command Prompt

1. *Checking your setup.* It's important to read the Before You Begin section at `www.deitel.com/books/cpphtp8/` to make sure that you've copied the book's examples to your hard drive correctly.
2. *Locating the completed application.* Open a **Command Prompt** window. To change to the directory for the completed **GuessNumber** application, type **`cd C:\examples\ch01\GuessNumber\Windows`**, then press *Enter* (Fig. 1.12). The command `cd` is used to change directories.

Fig. 1.12 | Opening a **Command Prompt** window and changing the directory.

3. ***Running the* GuessNumber *application.*** Now that you are in the directory that contains the **GuessNumber** application, type the command **`GuessNumber`** (Fig. 1.13) and press *Enter*. [*Note:* `GuessNumber.exe` is the actual name of the application; however, Windows assumes the `.exe` extension by default.]

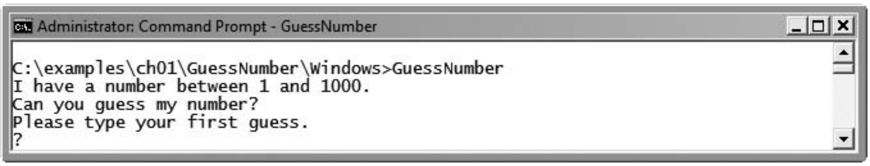


Fig. 1.13 | Running the **GuessNumber** application.

4. ***Entering your first guess.*** The application displays `"Please type your first guess."`, then displays a question mark (`?`) as a prompt on the next line (Fig. 1.13). At the prompt, enter **`500`** (Fig. 1.14).

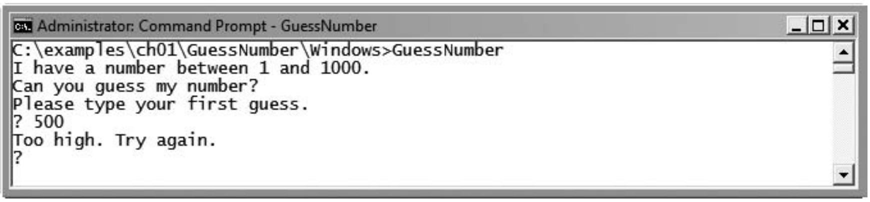


Fig. 1.14 | Entering your first guess.

5. ***Entering another guess.*** The application displays `"Too high. Try again."`, meaning that the value you entered is greater than the number the application chose as the correct guess. So, you should enter a lower number for your next guess. At the prompt, enter **`250`** (Fig. 1.15). The application again displays `"Too high. Try again."`, because the value you entered is still greater than the number that the application chose as the correct guess.

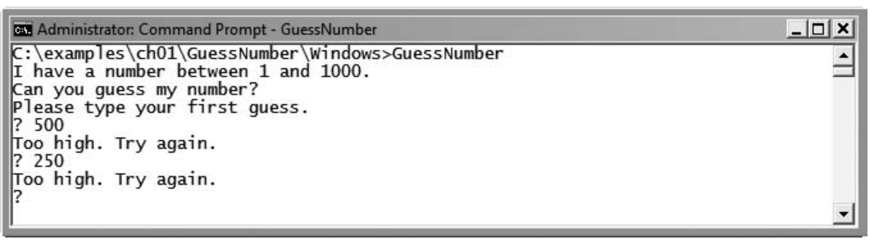


Fig. 1.15 | Entering a second guess and receiving feedback.

6. ***Entering additional guesses.*** Continue to play the game by entering values until you guess the correct number. The application will display `"Excellent! You guessed the number!"` (Fig. 1.16).

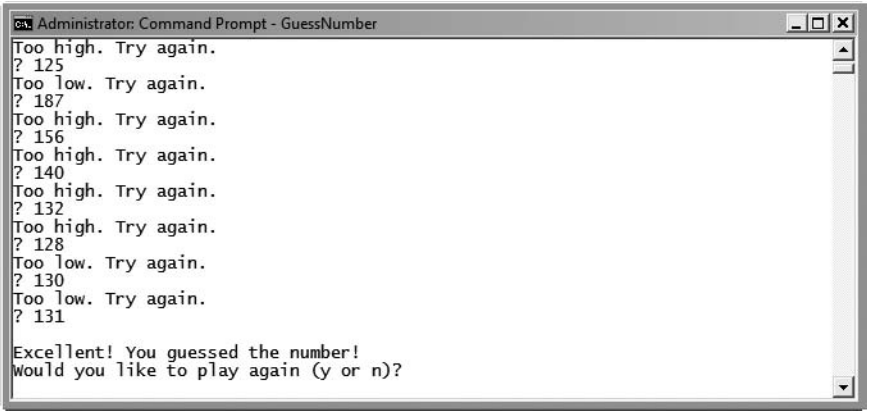


Fig. 1.16 | Entering additional guesses and guessing the correct number.

7. ***Playing the game again or exiting the application.*** After you guess correctly, the application asks if you'd like to play another game (Fig. 1.16). At the `"Would you like to play again (y or n)?"` prompt, entering the one character **`y`** causes the application to choose a new number and displays the message "`Please type your first guess.`" followed by a question mark prompt (Fig. 1.17) so you can make your first guess in the new game. Entering the character **`n`** ends the application and returns you to the application's directory at the **Command Prompt** (Fig. 1.18). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.
8. ***Close the* Command Prompt *window.***

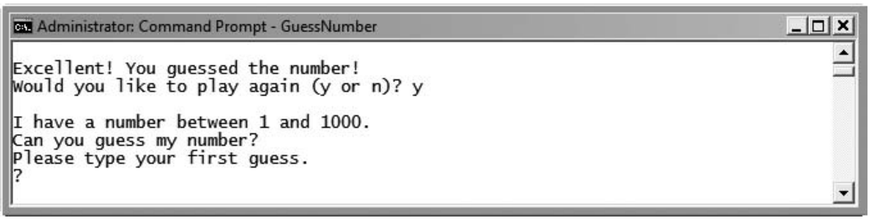


Fig. 1.17 | Playing the game again.

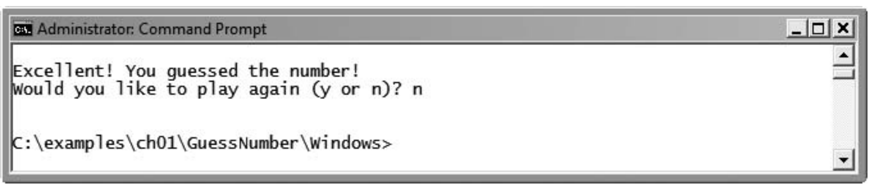


Fig. 1.18 | Exiting the game.

Running a C++ Application Using GNU C++ with Linux

For this test drive, we assume that you know how to copy the examples into your home directory. Please see your instructor if you have any questions regarding copying the files to your Linux system. Also, for the figures in this section, we use a bold highlight to point out the user input required by each step. The prompt in the shell on our system uses the tilde (~) character to represent the home directory, and each prompt ends with the dollar sign ($) character. The prompt will vary among Linux systems.

1. ***Locating the completed application.*** From a Linux shell, change to the completed **GuessNumber** application directory (Fig. 1.19) by typing

```
cd Examples/ch01/GuessNumber/GNU_Linux
```

then pressing *Enter*. The command `cd` is used to change directories.

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.19 | Changing to the **GuessNumber** application's directory.

2. ***Compiling the* GuessNumber *application.*** To run an application on the GNU C++ compiler, you must first compile it by typing

```
g++ GuessNumber.cpp -o GuessNumber
```

as in Fig. 1.20. This command compiles the application and produces an executable file called `GuessNumber`.

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.20 | Compiling the **GuessNumber** application using the `g++` command.

3. ***Running the* GuessNumber *application.*** To run the executable file `GuessNumber`, type `./GuessNumber` at the next prompt, then press *Enter* (Fig. 1.21).

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.21 | Running the **GuessNumber** application.

4. ***Entering your first guess.*** The application displays `"Please type your first guess."`, then displays a question mark (`?`) as a prompt on the next line (Fig. 1.21). At the prompt, enter **500** (Fig. 1.22). [*Note:* This is the same application that we modified and test-drove for Windows, but the outputs could vary based on the compiler being used.]

5. ***Entering another guess.*** The application displays `"Too high. Try again."`, meaning that the value you entered is greater than the number the application chose as the correct guess (Fig. 1.22). At the next prompt, enter **`250`** (Fig. 1.23). This time the application displays `"Too low. Try again."`, because the value you entered is less than the correct guess.
6. ***Entering additional guesses.*** Continue to play the game (Fig. 1.24) by entering values until you guess the correct number. When you guess correctly, the application displays `"Excellent! You guessed the number."`

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Fig. 1.22 | Entering an initial guess.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Fig. 1.23 | Entering a second guess and receiving feedback.

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384
Excellent! You guessed the number.
Would you like to play again (y or n)?
```

Fig. 1.24 | Entering additional guesses and guessing the correct number.

7. ***Playing the game again or exiting the application.*** After you guess the correct number, the application asks if you'd like to play another game. At the `"Would you like to play again (y or n)?"` prompt, entering the one character **`y`** causes the application to choose a new number and displays the message `"Please type your first guess."` followed by a question mark prompt (Fig. 1.25) so you can make your first guess in the new game. Entering the character **`n`** ends the application and returns you to the application's directory in the shell (Fig. 1.26). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

```
Excellent! You guessed the number.
Would you like to play again (y or n)? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.25 | Playing the game again.

```
Excellent! You guessed the number.
Would you like to play again (y or n)? n

~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.26 | Exiting the game.

1.11 Web 2.0: Going Social

The web literally exploded in the mid-to-late 1990s, but the "dot com" economic bust brought hard times in the early 2000s. The resurgence that began in 2004 or so has been named **Web 2.0**. Google is widely regarded as the signature company of Web 2.0. Some other companies with "Web 2.0 characteristics" are YouTube (video sharing), FaceBook (social networking), Twitter (microblogging), Groupon (social commerce), Foursquare (mobile check-in), Salesforce (business software offered as online services), Craigslist (free classified listings), Flickr (photo sharing), Second Life (a virtual world), Skype (Internet telephony) and Wikipedia (a free online encyclopedia).

Google

In 1996, Stanford computer science Ph.D. candidates Larry Page and Sergey Brin began collaborating on a new search engine. In 1997, they changed the name to Google—a play on the mathematical term *googol*, a quantity represented by the number "one" followed by 100 "zeros" (or 10^{100})—a staggeringly large number. Google's ability to return extremely accurate search results quickly helped it become the most widely used search engine and one of the most popular websites in the world.

Google continues to be an innovator in search technologies. For example, Google Goggles is a fascinating mobile app (available on Android and iPhone) that allows you to perform a Google search using a photo rather than entering text. You simply take pictures of a landmarks, books (covers or barcodes), logos, art or wine bottle labels, and Google Goggles scans the photo and returns search results. You can also take a picture of text (for example, a restaurant menu or a sign) and Google Goggles will translate it for you.

Ajax

Ajax is one of the premier Web 2.0 software technologies. Ajax helps Internet-based applications perform like desktop applications—a difficult task, given that such applications suffer transmission delays as data is shuttled back and forth between your computer and server computers on the Internet. Using Ajax, applications like Google Maps have achieved excellent performance and approach the look-and-feel of desktop applications.

Social Applications

Over the last several years, there's been a tremendous increase in the number of social applications on the web. Even though the computer industry is mature, these sites were still able to become phenomenally successful in a relatively short period of time. Figure 1.27 discusses a few of the social applications that are making an impact.

Company	Description
Facebook	Facebook was launched from a Harvard dorm room in 2004 by classmates Mark Zuckerberg, Chris Hughes, Dustin Moskovitz and Eduardo Saverin and is already worth an estimated $70 billion. By January 2011, Facebook was the most active site on the Internet with more than 600 million users—nearly 9% of the Earth's population—who spend 700 billion minutes on Facebook per month (`www.time.com/time/specials/packages/article/0,28804,2036683_2037183,00.html`). At its current rate of growth (about 5% per month), Facebook will reach one billion users in 2012, out of the two billion people on the Internet! The activity on the site makes it extremely attractive for application developers. Each day, over 20 million applications are installed by Facebook users (`www.facebook.com/press/info.php?statistics`).
Twitter	Twitter was founded in 2006 by Jack Dorsey, Evan Williams and Isaac "Biz" Stone—all from the podcast company, Odeo. Twitter has revolutionized *microblogging*. Users post tweets—messages of up to 140 characters in length. Approximately 95 million tweets are posted per day (`twitter.com/about`). You can follow the tweets of friends, celebrities, businesses, government representatives (including the U.S. President, who has 6.3 million followers), etc., or you can follow tweets by subject to track news, trends and more. At the time of this writing, Lady Gaga had the most followers (over 7.7 million). Twitter has become the point of origin for many breaking news stories worldwide.
Groupon	Groupon, a *social commerce* site, was launched by Andrew Mason in 2008. By January 2011, the company was valued around $15 billion, making it the fastest growing company ever! It's now available in hundreds of markets worldwide. Groupon offers one daily deal in each market for restaurants, retailers, services, attractions and more. Deals are activated only after a minimum number of people sign up to buy the product or service. If you sign up for a deal

Fig. 1.27 | Social applications. (Part 1 of 2.)

Company	Description
Groupon (cont.)	and it has yet to meet the minimum, you might be inclined to tell others about the deal by email, Facebook, Twitter, etc. If the deal does not meet the minimum sales, it's cancelled. One of the most successful national Groupon deals to date was a certificate for $50 worth of merchandise from a major apparel company for $25. Over 440,000 vouchers were sold in one day.
Foursquare	Foursquare—launched in 2009 by Dennis Crowley and Naveen Selvadurai—is a mobile *check-in* application that allows you to notify your friends of your whereabouts. You can download the app to your smartphone and link it to your Facebook and Twitter accounts so your friends can follow you from multiple platforms. If you do not have a smartphone, you can check in by text message. Foursquare uses GPS to determine your exact location. Businesses use Foursquare to send offers to users in the area. Launched in March 2009, Foursquare already has over 5 million users worldwide.
Skype	Skype is a software product that allows you to make mostly free voice and video calls over the Internet using a technology called *VoIP (Voice over IP*; IP stands for "Internet Protocol"*)*. Skype was founded in 2003 by Niklas Zennström and Dane Janus Friis. Just two years later, the company was sold to eBay for $2.6 billion.
YouTube	YouTube is a video-sharing site that was founded in 2005. Within one year, the company was purchased by Google for $1.65 billion. YouTube now accounts for 10% of all Internet traffic (`www.webpronews.com/topnews/2010/04/16/facebook-and-youtube-get-the-most-business-internet-traffic`). Within one week of the release of Apple's iPhone 3GS—the first iPhone model to offer video—mobile uploads to YouTube grew 400% (`www.hypebot.com/hypebot/2009/06/youtube-reports-1700-jump-in-mobile-video.html`).

Fig. 1.27 | Social applications. (Part 2 of 2.)

1.12 Software Technologies

Figure 1.28 lists a number of buzzwords that you'll hear in the software development community. We've created Resource Centers on most of these topics, with more on the way.

Technology	Description
Agile software development	**Agile software development** is a set of methodologies that try to get software implemented faster and using fewer resources than previous methodologies. Check out the Agile Alliance (`www.agilealliance.org`) and the Agile Manifesto (`www.agilemanifesto.org`).
Refactoring	**Refactoring** involves reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality. It's widely employed with agile development methodologies. Many IDEs include *refactoring tools* to do major portions of the reworking automatically.

Fig. 1.28 | Software technologies. (Part 1 of 2.)

Technology	Description
Design patterns	**Design patterns** are proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to *reuse* them to develop better-quality software using less time, money and effort.
LAMP	MySQL is an open-source database management system. PHP is the most popular open-source server-side Internet "scripting" language for developing Internet-based applications. **LAMP** is an acronym for the set of open-source technologies that many developers use to build web applications—it stands for Linux, Apache, MySQL and PHP (or Perl or Python—two other languages used for similar purposes).
Software as a Service (SaaS)	Software has generally been viewed as a product; most software still is offered this way. If you want to run an application, you buy a software package from a software vendor—often a CD, DVD or web download. You then install that software on your computer and run it as needed. As new versions of the software appear, you upgrade your software, often requiring significant time and at considerable expense. This process can become cumbersome for organizations with tens of thousands of systems that must be maintained on a diverse array of computer equipment. With **Software as a Service (SaaS)**, the software runs on servers elsewhere on the Internet. When that server is updated, all clients worldwide see the new capabilities—no local installation is needed. You access the service through a browser. Browsers are quite portable, so you can run the same applications on a wide variety of computers from anywhere in the world. Salesforce.com, Google, and Microsoft's Office Live and Windows Live all offer SaaS.
Platform as a Service (PaaS)	**Platform as a Service (PaaS)** provides a computing platform for developing and running applications as a service over the web, rather than installing the tools on your computer. PaaS providers include Google App Engine, Amazon EC2, Bungee Labs and more.
Cloud computing	SaaS and PaaS are examples of **cloud computing** in which software, platforms and infrastructure (e.g., processing power and storage) are hosted on demand over the Internet. This provides users with flexibility, scalability and cost savings. For example, consider a company's data storage needs which can fluctuate significantly over the course of a year. Rather than investing in large-scale storage hardware—which can be costly to purchase, maintain and secure, and would most likely not be used to capacity at all times—the company could purchase cloud-based services (such as Amazon S3, Google Storage, Microsoft Windows Azure™, Nirvanix™ and others) dynamically as needed.
Software Development Kit (SDK)	**Software Development Kits (SDKs)** include the tools and documentation developers use to program applications.

Fig. 1.28 | Software technologies. (Part 2 of 2.)

Figure 1.29 describes software product release categories.

Version	Description
Alpha	An *alpha* version of software is the earliest release of a software product that's still under active development. Alpha versions are often buggy, incomplete and unstable and are released to a relatively small number of developers for testing new features, getting early feedback, etc.
Beta	*Beta* versions are released to a larger number of developers later in the development process after most major bugs have been fixed and new features are nearly complete. Beta software is more stable, but still subject to change.
Release candidates	*Release candidates* are generally *feature complete* and (supposedly) bug free and ready for use by the community, which provides a diverse testing environment—the software is used on different systems, with varying constraints and for a variety of purposes. Any bugs that appear are corrected and eventually the final product is released to the general public. Software companies often distribute incremental updates over the Internet.
Continuous beta	Software that's developed using this approach generally does not have version numbers (for example, Google search or Gmail). The software, which is hosted in the cloud (not installed on your computer), is constantly evolving so that users always have the latest version.

Fig. 1.29 | Software product release terminology.

1.13 Future of C++: TR1, the New C++ Standard and the Open Source Boost Libraries

Bjarne Stroustrup, the creator of C++, has expressed his vision for the future of C++. The main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language.

Throughout the book, we discuss in optional sections various key features of the new C++ standard. In addition, Chapter 23 introduces the Boost C++ Libraries, Technical Report 1 (TR1) and more new C++ features.

Technical Report 1 describes the proposed changes to the C++ Standard Library. These libraries add useful functionality to C++. The C++ Standards Committee is currently finishing the revision of the C++ Standard. The last standard was published in 1998. Work on the new standard began in 2003. At that time, it was referred to as **C++0x** because the standard was scheduled to be released before the end of the decade. The new standard includes most of the libraries in TR1 and changes to the core language.

The **Boost C++ Libraries** are free, open-source libraries created by members of the C++ community. Boost has grown to over 100 libraries, with more being added regularly. Today there are thousands of programmers in the Boost open source community. Boost provides C++ programmers with useful libraries that work well with the existing C++ Standard Library. The Boost libraries can be used by C++ programmers working on a wide variety of platforms with many different compilers. Several of the Boost libraries are included in TR1 and will be part of the new standard. We overview the libraries included in TR1 and provide code examples for the "regular expression" and "smart pointer" libraries.

Regular expressions are used to match specific character patterns in text. They can be used to validate data to ensure that it's in a particular format, to replace parts of one string with another, or to split a string.

Many common bugs in C and C++ code are related to pointers, a powerful programming capability C++ absorbed from C. **Smart pointers** help you avoid errors by providing additional functionality, typically strengthens the process of memory allocation and deallocation.

1.14 Keeping Up-to-Date with Information Technologies

Figure 1.30 lists key technical and business publications that will help you stay up-to-date with the latest news and trends and technology. You can also find a growing list of Internet- and web-related Resource Centers at `www.deitel.com/resourcecenters.html`.

Publication	URL
Bloomberg BusinessWeek	`www.businessweek.com`
CNET	`news.cnet.com`
Computer World	`www.computerworld.com`
Engadget	`www.engadget.com`
eWeek	`www.eweek.com`
Fast Company	`www.fastcompany.com/`
Fortune	`money.cnn.com/magazines/fortune/`
InfoWorld	`www.infoworld.com`
Mashable	`mashable.com`
PCWorld	`www.pcworld.com`
SD Times	`www.sdtimes.com`
Slashdot	`slashdot.org/`
Smarter Technology	`www.smartertechnology.com`
Technology Review	`technologyreview.com`
Techcrunch	`techcrunch.com`
Wired	`www.wired.com`

Fig. 1.30 | Technical and business publications (many are free).

1.15 Wrap-Up

In this chapter we discussed computer hardware, software, programming languages and operating systems. We introduced the basics of object technology. You learned about some of the exciting recent developments in the computer field. We overviewed a typical C++ program development environment and you test-drove a C++ application. We also discussed some key software development terminology.

In Chapter 2, you'll create your first C++ applications. You'll see how programs display messages on the screen and obtain information from the user at the keyboard for processing. You'll see several examples that demonstrate how programs display messages on the screen and obtain information from the user at the keyboard for processing.

Self-Review Exercises

1.1 Fill in the blanks in each of the following statements:

a) The company that popularized personal computing was ________.
b) The computer that made personal computing legitimate in business and industry was the ________.
c) Computers process data under the control of sets of instructions called ________.
d) The key logical units of the computer are the ________, ________, ________, ________, ________ and ________.
e) The three types of languages discussed in the chapter are ________, ________ and ________.
f) The programs that translate high-level language programs into machine language are called ________.
g) ________ is a smartphone operating system based on the Linux kernel and Java.
h) ________ software is generally feature complete and (supposedly) bug free and ready for use by the community.
i) The Wii Remote, as well as many smartphones, uses a(n) ________ which allows the device to respond to motion.

1.2 Fill in the blanks in each of the following sentences about the C++ environment.

a) C++ programs are normally typed into a computer using a(n) ________ program.
b) In a C++ system, a(n) ________ program executes before the compiler's translation phase begins.
c) The ________ program combines the output of the compiler with various library functions to produce an executable program.
d) The ________ program transfers the executable program from disk to memory.

1.3 Fill in the blanks in each of the following statements (based on Section 1.6):

a) Objects have the property of ________—although objects may know how to communicate with one another across well-defined interfaces, they normally are not allowed to know how other objects are implemented.
b) C++ programmers concentrate on creating ________, which contain data members and the member functions that manipulate those data members and provide services to clients.
c) The process of analyzing and designing a system from an object-oriented point of view is called ________.
d) With ________, new classes of objects are derived by absorbing characteristics of existing classes, then adding unique characteristics of their own.
e) ________ is a graphical language that allows people who design software systems to use an industry-standard notation to represent them.
f) The size, shape, color and weight of an object are considered ________ of the object's class.

Answers to Self-Review Exercises

1.1 a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, central processing unit, arithmetic and logic unit, secondary storage unit. e) machine languages, assembly languages, high-level languages. f) compilers. g) Android. h) Release candidate. i) accelerometer.

1.2 a) editor. b) preprocessor. c) linker. d) loader.

1.3 a) information hiding. b) classes. c) object-oriented analysis and design (OOAD). d) inheritance. e) The Unified Modeling Language (UML). f) attributes.

Exercises

1.4 Fill in the blanks in each of the following statements:

a) The logical unit of the computer that receives information from outside the computer for use by the computer is the ________.
b) The process of instructing the computer to solve a problem is called ________.
c) ________ is a type of computer language that uses English-like abbreviations for machine-language instructions.
d) ________ is a logical unit of the computer that sends information which has already been processed by the computer to various devices so that it may be used outside the computer.
e) ________ and ________ are logical units of the computer that retain information.
f) ________ is a logical unit of the computer that performs calculations.
g) ________ is a logical unit of the computer that makes logical decisions.
h) ________ languages are most convenient to the programmer for writing programs quickly and easily.
i) The only language a computer can directly understand is that computer's ________.
j) ________ is a logical unit of the computer that coordinates the activities of all the other logical units.

1.5 Fill in the blanks in each of the following statements:

a) ________ is used to develop large-scale enterprise applications, to enhance the functionality of web servers, to provide applications for consumer devices and for many other purposes.
b) ________ initially became widely known as the development language of the Unix operating system.
c) The Web 2.0 company ________ is the fastest growing company ever.
d) The ________ programming language was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories.

1.6 Fill in the blanks in each of the following statements:

a) C++ programs normally go through six phases—________, ________, ________, ________, ________ and ________.
b) A(n) ________ provides many tools that support the software development process, such as editors for writing and editing programs, debuggers for locating logic errors in programs, and many other features.
c) The command `java` invokes the ________, which executes Java programs.
d) A(n) ________ is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with it.
e) The ________ takes the `.class` files containing the program's bytecodes and transfers them to primary memory.
f) The ________ examines bytecodes to ensure that they're valid.

1.7 You're probably wearing on your wrist one of the world's most common types of objects—a watch. Discuss how each of the following terms and concepts applies to the notion of a watch:

object, attributes, behaviors, class, inheritance (consider, for example, an alarm clock), abstraction, modeling, messages, encapsulation, interface and information hiding.

Making a Difference

Throughout the book we've included Making a Difference exercises in which you'll be asked to work on problems that really matter to individuals, communities, countries and the world. For more information about worldwide organizations working to make a difference, and for related programming project ideas, visit our Making a Difference Resource Center at `www.deitel.com/makingadifference`.

1.8 *(Test Drive: Carbon Footprint Calculator)* Some scientists believe that carbon emissions, especially from the burning of fossil fuels, contribute significantly to global warming and that this can be combatted if individuals take steps to limit their use of carbon-based fuels. Organizations and individuals are increasingly concerned about their "carbon footprints." Websites such as TerraPass

`www.terrapass.com/carbon-footprint-calculator/`

and Carbon Footprint

`www.carbonfootprint.com/calculator.aspx`

provide carbon footprint calculators. Test drive these calculators to determine your carbon footprint. Exercises in later chapters will ask you to program your own carbon footprint calculator. To prepare for this, research the formulas for calculating carbon footprints.

1.9 *(Test Drive: Body Mass Index Calculator)* By recent estimates, two-thirds of the people in the United States are overweight and about half of those are obese. This causes significant increases in illnesses such as diabetes and heart disease. To determine whether a person is overweight or obese, you can use a measure called the body mass index (BMI). The United States Department of Health and Human Services provides a BMI calculator at `www.nhlbisupport.com/bmi/`. Use it to calculate your own BMI. An exercise in Chapter 2 will ask you to program your own BMI calculator. To prepare for this, research the formulas for calculating BMI.

1.10 *(Attributes of Hybrid Vehicles)* In this chapter you learned the basics of classes. Now you'll begin "fleshing out" aspects of a class called "Hybrid Vehicle." Hybrid vehicles are becoming increasingly popular, because they often get much better mileage than purely gasoline-powered vehicles. Browse the web and study the features of four or five of today's popular hybrid cars, then list as many of their hybrid-related attributes as you can. For example, common attributes include city-miles-per-gallon and highway-miles-per-gallon. Also list the attributes of the batteries (type, weight, etc.).

1.11 *(Gender Neutrality)* Many people want to eliminate sexism in all forms of communication. You've been asked to create a program that can process a paragraph of text and replace gender-specific words with gender-neutral ones. Assuming that you've been given a list of gender-specific words and their gender-neutral replacements (e.g., replace "wife" by "spouse," "man" by "person," "daughter" by "child" and so on), explain the procedure you'd use to read through a paragraph of text and manually perform these replacements. How might your procedure generate a strange term like "woperchild," which is actually listed in the Urban Dictionary (`www.urbandictionary.com`)? In Chapter 4, you'll learn that a more formal term for "procedure" is "algorithm," and that an algorithm specifies the steps to be performed and the order in which to perform them.

1.12 *(Privacy)* Some online email services save all email correspondence for some period of time. Suppose a disgruntled employee of one of these online email services were to post all of the email correspondences for millions of people, including yours, on the Internet. Discuss the issues.

1.13 *(Programmer Responsibility and Liability)* As a programmer in industry, you may develop software that could affect people's health or even their lives. Suppose a software bug in one of your programs were to cause a cancer patient to receive an excessive dose during radiation therapy and that the person is either severely injured or dies. Discuss the issues.

1.14 *(2010 "Flash Crash")* An example of the consequences of our excessive dependency on computers was the so-called "flash crash" which occurred on May 6, 2010, when the U.S. stock market fell precipitously in a matter of minutes, wiping out trillions of dollars of investments, and then recovered within minutes. Use the Internet to investigate the causes of this crash and discuss the issues it raises.

Making a Difference Resources

The *Microsoft Image Cup* is a global competition in which students use technology to try to solve some of the world's most difficult problems, such as environmental sustainability, ending hunger, emergency response, literacy, combating HIV/AIDS and more. For more information about the competition and to learn about the projects developed by previous winners, visit `www.imaginecup.com/about`. You can also find several project ideas submitted by worldwide charitable organizations at `www.imaginecup.com/students/imagine-cup-solve-this`. For additional ideas for programming projects that can make a difference, search the web for "making a difference" and visit the following websites:

`www.un.org/millenniumgoals`
The United Nations Millennium Project seeks solutions to major worldwide issues such as environmental sustainability, gender equality, child and maternal health, universal education and more.

`www.ibm.com/smarterplanet/`
The IBM® Smarter Planet website discusses how IBM is using technology to solve issues related to business, cloud computing, education, sustainability and more.

`www.gatesfoundation.org/Pages/home.aspx`
The Bill and Melinda Gates Foundation provides grants to organizations that work to alleviate hunger, poverty and disease in developing countries. In the U.S., the foundation focusses on improving public education, particularly for people with few resources.

`www.nethope.org/`
NetHope is a collaboration of humanitarian organizations worldwide working to solve technology problems such as connectivity, emergency response and more.

`www.rainforestfoundation.org/home`
The Rainforest Foundation works to preserve rainforests and to protect the rights of the indigenous people who call the rainforests home. The site includes a list of things you can do to help.

`www.undp.org/`
The United Nations Development Programme (UNDP) seeks solutions to global challenges such as crisis prevention and recovery, energy and the environment, democratic governance and more.

`www.unido.org`
The United Nations Industrial Development Organization (UNIDO) seeks to reduce poverty, give developing countries the opportunity to participate in global trade, and promote energy efficiency and sustainability.

`www.usaid.gov/`
USAID promotes global democracy, health, economic growth, conflict prevention, humanitarian aid and more.

`www.toyota.com/ideas-for-good/`
Toyota's Ideas for Good website describes several Toyota technologies that are making a difference—including their Advanced Parking Guidance System, Hybrid Synergy Drive®, Solar Powered Ventilation System, T.H.U.M.S. (Total Human Model for Safety) and Touch Tracer Display. You can participate in the Ideas for Good challenge by submitting a short essay or video describing how these technologies can be used for other good purposes.

Introduction to C++ Programming

2

What's in a name? that which we call a rose
By any other name would smell as sweet.
—William Shakespeare

When faced with a decision, I always ask, "What would be the most fun?"
—Peggy Walker

High thoughts must have high language.
—Aristophanes

One person can make a difference and every person should try.
—John F. Kennedy

Objectives

In this chapter you'll learn:

- To write simple computer programs in C++.
- To write simple input and output statements.
- To use fundamental types.
- Basic computer memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write simple decision-making statements.

Outline

2.1 Introduction

We now introduce C++ programming, which facilitates a disciplined approach to program design. Most of the C++ programs you'll study in this book process information and display results. In this chapter, we present five examples that demonstrate how your programs can display messages and obtain information from the user for processing. The first three examples simply display messages on the screen. The next obtains two numbers from a user, calculates their sum and displays the result. The accompanying discussion shows you how to perform arithmetic calculations and save their results for later use. The fifth example demonstrates decision-making by showing you how to compare two numbers, then display messages based on the comparison results. We analyze each program one line at a time to help you ease your way into C++ programming.

2.2 First Program in C++: Printing a Line of Text

C++ uses notations that may appear strange to nonprogrammers. We now consider a simple program that prints a line of text (Fig. 2.1). This program illustrates several important features of the C++ language.

```
1  // Fig. 2.1: fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11 } // end function main
```

```
Welcome to C++!
```

Fig. 2.1 | Text-printing program.

Comments

Lines 1 and 2

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
```

each begin with **`//`**, indicating that the remainder of each line is a **comment**. You insert comments to document your programs and to help other people read and understand them. Comments do not cause the computer to perform any action when the program is run—they're ignored by the C++ compiler and do not cause any machine-language object code to be generated. The comment `Text-printing program` describes the purpose of the program. A comment beginning with `//` is called a **single-line comment** because it terminates at the end of the current line. [*Note:* You also may use C's style in which a comment—possibly containing many lines—begins with `/*` and ends with `*/`.]

Good Programming Practice 2.1

Every program should begin with a comment that describes the purpose of the program.

`#include` Preprocessor Directive

Line 3

```
#include <iostream> // allows program to output data to the screen
```

is a **preprocessor directive**, which is a message to the C++ preprocessor (introduced in Section 1.9). Lines that begin with **`#`** are processed by the preprocessor *before* the program is compiled. This line notifies the preprocessor to include in the program the contents of the **input/output stream header `<iostream>`**. This header must be included for any program that outputs data to the screen or inputs data from the keyboard using C++'s stream input/output. The program in Fig. 2.1 outputs data to the screen, as we'll soon see. We discuss headers in more detail in Chapter 6 and explain the contents of `<iostream>` in Chapter 15.

Common Programming Error 2.1

Forgetting to include the `<iostream>` header in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message.

Blank Lines and White Space

Line 4 is simply a blank line. You use blank lines, space characters and tab characters (i.e., "tabs") to make programs easier to read. Together, these characters are known as **white space**. White-space characters are normally ignored by the compiler.

The `main` Function

Line 5

```
// function main begins program execution
```

is another single-line comment indicating that program execution begins at the next line. Line 6

```
int main()
```

is a part of every C++ program. The parentheses after `main` indicate that **`main`** is a program building block called a **function**. C++ programs typically consist of one or more functions and classes (as you'll learn in Chapter 3). Exactly one function in every program *must* be named `main`. Figure 2.1 contains only one function. C++ programs begin executing at function `main`, even if `main` is not the first function in the program. The keyword `int` to the left of `main` indicates that `main` "returns" an integer (whole number) value. A **keyword** is a word in code that is reserved by C++ for a specific use. The complete list of C++ keywords can be found in Fig. 4.3. We'll explain what it means for a function to "return a value" when we demonstrate how to create your own functions in Section 3.3. For now, simply include the keyword `int` to the left of `main` in each of your programs.

The **left brace**, **`{`**, (line 7) must *begin* the **body** of every function. A corresponding **right brace**, **`}`**, (line 11) must *end* each function's body.

An Output Statement

Line 8

```
std::cout << "Welcome to C++!\n"; // display message
```

instructs the computer to **perform an action**—namely, to print the **string** of characters contained between the double quotation marks. A string is sometimes called a **character string** or a **string literal**. We refer to characters between double quotation marks simply as strings. White-space characters in strings are not ignored by the compiler.

The entire line 8, including `std::cout`, the **`<<` operator**, the string `"Welcome to C++!\n"` and the **semicolon** (**`;`**), is called a **statement**. Every C++ statement must end with a semicolon (also known as the **statement terminator**). Preprocessor directives (like `#include`) do not end with a semicolon. Output and input in C++ are accomplished with **streams** of characters. Thus, when the preceding statement is executed, it sends the stream of characters `Welcome to C++!\n` to the **standard output stream object**—**`std::cout`**—which is normally "connected" to the screen.

Common Programming Error 2.2

*Omitting the semicolon at the end of a C++ statement is a syntax error. The **syntax** of a programming language specifies the rules for creating proper programs in that language. A **syntax error** occurs when the compiler encounters code that violates C++'s language rules (i.e., its syntax). The compiler normally issues an error message to help you locate and fix the incorrect code. Syntax errors are also called **compiler errors**, **compile-time errors** or **compilation errors**, because the compiler detects them during the compilation phase. You cannot execute your program until you correct all the syntax errors in it. As you'll see, some compilation errors are not syntax errors.*

Good Programming Practice 2.2

Indent the body of each function one level within the braces that delimit the function's body. This makes a program's functional structure stand out and makes the program easier to read.

Good Programming Practice 2.3

Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We prefer three spaces per level of indent.

The `std` Namespace

The `std::` before `cout` is required when we use names that we've brought into the program by the preprocessor directive `#include <iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to "namespace" `std`. The names `cin` (the standard input stream) and `cerr` (the standard error stream)—introduced in Chapter 1—also belong to namespace `std`. Namespaces are an advanced C++ feature that we discuss in depth in Chapter 24, Other Topics. For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and `cerr` in a program. This can be cumbersome—in Fig. 2.13, we introduce the `using` directive, which will enable you to omit `std::` before each use of a name in the `std` namespace.

The Stream Insertion Operator and Es'cape Sequences

The `<<` operator is referred to as the **stream insertion operator**. When this program executes, the value to the operator's right, the right **operand**, is inserted in the output stream. Notice that the operator points in the direction of where the data goes. The right operand's characters normally print exactly as they appear between the double quotes. However, the characters `\n` are not printed on the screen (Fig. 2.1). The backslash (`\`) is called an **escape character**. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. It causes the **cursor** (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

Escape sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\'`	Single quote. Use to print a single quote character.
`\"`	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

The `return` Statement

Line 10

```
return 0; // indicate that program ended successfully
```

is one of several means we'll use to **exit a function**. When the **`return` statement** is used at the end of `main`, as shown here, the value `0` indicates that the program has *terminated successfully*. The right brace, `}`, (line 11) indicates the end of function `main`. According to the C++ standard, if program execution reaches the end of `main` without encountering a `return` statement, it's assumed that the program terminated successfully—exactly as when the last statement in main is a `return` statement with the value `0`. For that reason, we *omit* the `return` statement at the end of `main` in subsequent programs.

2.3 Modifying Our First C++ Program

We now present two examples that modify the program of Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

Printing a Single Line of Text with Multiple Statements

`Welcome to C++!` can be printed several ways. For example, Fig. 2.3 performs stream insertion in multiple statements (lines 8–9), yet produces the same output as the program of Fig. 2.1. [*Note:* From this point forward, we use a yellow background to highlight the key features each program introduces.] Each stream insertion resumes printing where the previous one stopped. The first stream insertion (line 8) prints `Welcome` followed by a space, and because this string did not end with `\n`, the second stream insertion (line 9) begins printing on the *same* line immediately following the space.

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10 } // end function main
```

```
Welcome to C++!
```

Fig. 2.3 | Printing a line of text with multiple statements.

Printing Multiple Lines of Text with a Single Statement

A single statement can print multiple lines by using newline characters, as in line 8 of Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, place two newline characters back to back, as in line 8.

```
1  // Fig. 2.4: fig02_04.cpp
2  // Printing multiple lines of text with a single statement.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome\nto\n\nC++!\n";
9  } // end function main
```

```
Welcome
to

C++!
```

Fig. 2.4 | Printing multiple lines of text with a single statement.

2.4 Another C++ Program: Adding Integers

Our next program uses the input stream object `std::cin` and the stream extraction operator, >>, to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result using `std::cout`. Figure 2.5 shows the program and sample inputs and outputs. In the sample execution, we highlight the user's input in bold.

```
1  // Fig. 2.5: fig02_05.cpp
2  // Addition program that displays the sum of two integers.
3  #include <iostream> // allows program to perform input and output
4
5  // function main begins program execution
6  int main()
7  {
8     // variable declarations
9     int number1; // first integer to add
10    int number2; // second integer to add
11    int sum; // sum of number1 and number2
12
13    std::cout << "Enter first integer: "; // prompt user for data
14    std::cin >> number1; // read first integer from user into number1
15
16    std::cout << "Enter second integer: "; // prompt user for data
17    std::cin >> number2; // read second integer from user into number2
18
19    sum = number1 + number2; // add the numbers; store result in sum
20
21    std::cout << "Sum is " << sum << std::endl; // display sum; end line
22 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5 | Addition program that displays the sum of two integers entered at the keyboard.

The comments in lines 1 and 2 state the name of the file and the purpose of the program. The C++ preprocessor directive in line 3 includes the contents of the `<iostream>` header. The program begins execution with function `main` (line 6). The left brace (line 7) begins `main`'s body and the corresponding right brace (line 22) ends it.

Variable Declarations

Lines 9–11

```
int number1; // first integer to add
int number2; // second integer to add
int sum; // sum of number1 and number2
```

are **declarations**. The identifiers `number1`, `number2` and `sum` are the names of **variables**. A variable is a location in the computer's memory where a value can be stored for use by a program. These declarations specify that the variables `number1`, `number2` and `sum` are data of type **`int`**, meaning that these variables will hold **integer** values, i.e., whole numbers such as 7, –11, 0 and 31914. All variables *must* be declared with a *name* and a *data type before* they can be used in a program. Several variables of the same type may be declared in one declaration or in multiple declarations. We could have declared all three variables in one declaration by using a **comma-separated list** as follows:

```
int number1, number2, sum;
```

This makes the program less readable and prevents us from providing comments that describe each variable's purpose.

Good Programming Practice 2.4

Place a space after each comma (`,`) to make programs more readable.

Fundamental Types

We'll soon discuss the type `double` for specifying real numbers, and the type `char` for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and –11.19. A `char` variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (e.g., `$` or `*`). Types such as `int`, `double` and `char` are called **fundamental types**. Fundamental-type names are *keywords* and therefore *must* appear in all lowercase letters. Appendix C contains the complete list of fundamental types.

Identifiers

A variable name (such as `number1`) is any valid **identifier** that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (`_`) that does not begin with a digit. C++ is **case sensitive**—uppercase and lowercase letters are different, so `a1` and `A1` are different identifiers.

Portability Tip 2.1

C++ allows identifiers of any length, but your C++ implementation may restrict identifier lengths. Use identifiers of 31 characters or fewer to ensure portability.

Good Programming Practice 2.5

*Choosing meaningful identifiers makes a program **self-documenting**—a person can understand the program simply by reading it rather than having to refer to manuals or comments.*

Good Programming Practice 2.6

Avoid using abbreviations in identifiers. This improves program readability.

Good Programming Practice 2.7

Do not use identifiers that begin with underscores and double underscores, because C++ compilers may use names like that for their own purposes internally. This will prevent the names you choose from being confused with names the compilers choose.

Placement of Variable Declarations

Declarations of variables can be placed almost anywhere in a program, but they must appear *before* their corresponding variables are used in the program. For example, in the program of Fig. 2.5, the declaration in line 9

```
    int number1; // first integer to add
```

could have been placed immediately before line 14

```
    std::cin >> number1; // read first integer from user into number1
```

the declaration in line 10

```
    int number2; // second integer to add
```

could have been placed immediately before line 17

```
    std::cin >> number2; // read second integer from user into number2
```

and the declaration in line 11

```
    int sum; // sum of number1 and number2
```

could have been placed immediately before line 19

```
    sum = number1 + number2; // add the numbers; store result in sum
```

Good Programming Practice 2.8

Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out and contributes to program clarity.

Obtaining the First Value from the User

Line 13

```
    std::cout << "Enter first integer: "; // prompt user for data
```

displays `Enter first integer:` followed by a space. This message is called a **prompt** because it directs the user to take a specific action. We like to pronounce the preceding statement as "`std::cout` *gets* the character string `"Enter first integer: "`." Line 14

```
    std::cin >> number1; // read first integer from user into number1
```

uses the **standard input stream object `cin`** (of namespace `std`) and the **stream extraction operator, `>>`**, to obtain a value from the keyboard. Using the stream extraction operator with `std::cin` takes character input from the standard input stream, which is usually the keyboard. We like to pronounce the preceding statement as, "`std::cin` *gives* a value to `number1`" or simply "`std::cin` *gives* `number1`."

When the computer executes the preceding statement, it waits for the user to enter a value for variable `number1`. The user responds by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the computer. The computer converts the character representation of the number to an integer and assigns (i.e., copies) this number (or **value**) to the variable `number1`. Any subsequent references to `number1` in this program will use this same value.

The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialog, it's often called **interactive computing.**

Obtaining the Second Value from the User

Line 16

```
    std::cout << "Enter second integer: "; // prompt user for data
```

prints `Enter second integer:` on the screen, prompting the user to take action. Line 17

```
std::cin >> number2; // read second integer from user into number2
```

obtains a value for variable `number2` from the user.

Calculating the Sum of the Values Input by the User

The assignment statement in line 19

```
sum = number1 + number2; // add the numbers; store result in sum
```

adds the values of variables `number1` and `number2` and assigns the result to variable `sum` using the **assignment operator =**. The statement is read as, "`sum` *gets* the value of `number1 + number2`." Most calculations are performed in assignment statements. The = operator and the + operator are called **binary operators** because each has two operands. In the case of the + operator, the two operands are `number1` and `number2`. In the case of the preceding = operator, the two operands are `sum` and the value of the expression `number1 + number2`.

> **Good Programming Practice 2.9**
> *Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.*

Displaying the Result

Line 21

```
std::cout << "Sum is " << sum << std::endl; // display sum; end line
```

displays the character string `Sum is` followed by the numerical value of variable `sum` followed by `std::endl`—a so-called **stream manipulator**. The name `endl` is an abbreviation for "end line" and belongs to namespace `std`. The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer." This simply means that, on some systems where outputs accumulate in the machine until there are enough to "make it worthwhile" to display them on the screen, `std::endl` forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

The preceding statement outputs multiple values of different types. The stream insertion operator "knows" how to output each type of data. Using multiple stream insertion operators (<<) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**. It's unnecessary to have multiple statements to output multiple pieces of data.

Calculations can also be performed in output statements. We could have combined the statements in lines 19 and 21 into the statement

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

thus eliminating the need for the variable `sum`.

A powerful feature of C++ is that you can create your own data types called classes (we introduce this capability in Chapter 3 and explore it in depth in Chapters 9 and 10). You can then "teach" C++ how to input and output values of these new data types using the >> and << operators (this is called **operator overloading**—a topic we explore in Chapter 11).

2.5 Memory Concepts

Variable names such as `number1`, `number2` and `sum` actually correspond to **locations** in the computer's memory. Every variable has a *name*, a *type*, a *size* and a *value*.

In the addition program of Fig. 2.5, when the statement in line 14

```
std::cin >> number1; // read first integer from user into number1
```

is executed, the integer typed by the user is placed into a memory location to which the name `number1` has been assigned by the compiler. Suppose the user enters `45` as the value for `number1`. The computer will place `45` into the location `number1`, as shown in Fig. 2.6. When a value is placed in a memory location, the value *overwrites* the previous value in that location; thus, placing a new value into a memory location is said to be **destructive**.

Returning to our addition program, suppose the user enters `72` when the statement

```
std::cin >> number2; // read second integer from user into number2
```

is executed. This value is placed into the location `number2`, and memory appears as in Fig. 2.7. The variables' locations are not necessarily adjacent in memory.

Once the program has obtained values for `number1` and `number2`, it adds these values and places the total into the variable `sum`. The statement

```
sum = number1 + number2; // add the numbers; store result in sum
```

replaces whatever value was stored in `sum`. The calculated sum of `number1` and `number2` is placed into variable `sum` without regard to what value may already be in `sum`—that value is *lost*). After `sum` is calculated, memory appears as in Fig. 2.8. The values of `number1` and `number2` appear exactly as they did before the calculation. These values were used, but *not* destroyed, as the computer performed the calculation. Thus, when a value is read out of a memory location, the process is **nondestructive**.

Fig. 2.6 | Memory location showing the name and value of variable `number1`.

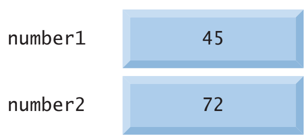


Fig. 2.7 | Memory locations after storing values the variables for `number1` and `number2`.

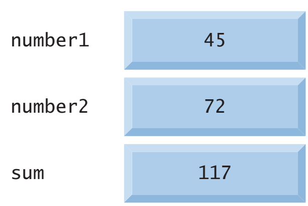


Fig. 2.8 | Memory locations after calculating and storing the `sum` of `number1` and `number2`.

2.6 Arithmetic

Most programs perform arithmetic calculations. Figure 2.9 summarizes the C++ **arithmetic operators**. Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication and the **percent sign** (`%`) is the modulus operator that will be discussed shortly. The arithmetic operators in Fig. 2.9 are all *binary* operators, i.e., operators that take two operands. For example, the expression `number1 + number2` contains the binary operator `+` and the two operands `number1` and `number2`.

Integer division (i.e., where both the numerator and the denominator are integers) yields an integer quotient; for example, the expression `7 / 4` evaluates to `1` and the expression `17 / 5` evaluates to `3`. Any fractional part in integer division is *discarded* (i.e., **truncated**)—no rounding occurs.

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm or $b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 2.9 | Arithmetic operators.

C++ provides the **modulus operator**, `%`, that yields the remainder after integer division. The modulus operator can be used *only* with integer operands. The expression `x % y` yields the *remainder* after `x` is divided by `y`. Thus, `7 % 4` yields `3` and `17 % 5` yields `2`. In later chapters, we discuss many interesting applications of the modulus operator, such as determining whether one number is a multiple of another (a special case of this is determining whether a number is odd or even).

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C++ must be entered into the computer in **straight-line form**. Thus, expressions such as "`a` divided by `b`" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally *not* acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C++ expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c` we write `a * ( b + c )`.

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by these **rules of operator precedence**, which are generally the same as those in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the "highest level of precedence." In cases of **nested**, or **embedded**, **parentheses**, such as

```
( a * ( b + c ) )
```

the operators in the *innermost* pair of parentheses are applied first.

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from *left to right*. Multiplication, division and modulus are said to be on the *same* level of precedence.

3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from *left to right*. Addition and subtraction also have the *same* level of precedence.

The set of rules of operator precedence defines the order in which C++ applies operators. When we say that certain operators are applied from left to right, we are referring to the **associativity** of the operators. For example, the addition operators (+) in the expression

```
a + b + c
```

associate from left to right, so `a + b` is calculated first, then `c` is added to that sum to determine the whole expression's value. We'll see that some operators associate from *right to left*. Figure 2.10 summarizes these rules of operator precedence. We expand this table as we introduce additional C++ operators. A complete precedence chart is included in Appendix A.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. [*Caution:* If you have an expression such as `(a + b) * (c - d)` in which two sets of parentheses are not nested, but appear "on the same level," the C++ Standard does *not* specify the order in which these parenthesized subexpressions will be evaluated.]
`*`, `/`, `%`	Multiplication, Division, Modulus	Evaluated second. If there are several, they're evaluated left to right.
`+` `-`	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Fig. 2.10 | Precedence of arithmetic operators.

Sample Algebraic and C++ Expressions

Now consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent. The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

C++: `m = ( a + b + c + d + e ) / 5;`

The parentheses are required because division has *higher* precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

C++: `y = m * x + b;`

No parentheses are required. The multiplication is applied first because multiplication has a higher precedence than addition.

The following example contains modulus (%), multiplication, division, addition, subtraction and assignment operations:

Algebra: $z = pr\%q + w/x - y$

C++:

```
z  =  p  *  r  %  q  +  w  /  x  -  y;
   6     1     2     4     3     5
```

The circled numbers under the statement indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated *first* in left-to-right order (i.e., they associate from left to right) because they have *higher precedence* than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. The assignment operator is applied *last* because its precedence is *lower* than that of any of the arithmetic operators.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial $y = ax^2 + bx + c$:

```
y  =  a  *  x  *  x  +  b  *  x  +  c;
   6     1     2     4     3     5
```

The circled numbers under the statement indicate the order in which C++ applies the operators. *There is no arithmetic operator for exponentiation in C++*, so we've represented x^2 as `x * x`. We'll soon discuss the standard library function `pow` ("power") that performs exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until Chapter 6.

Suppose variables `a`, `b`, `c` and `x` in the preceding second-degree polynomial are initialized as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.11 illustrates the order in which the operators are applied and the final value of the expression.

As in algebra, it's acceptable to place *unnecessary* parentheses in an expression to make the expression clearer. These are called **redundant parentheses**. For example, the preceding assignment statement could be parenthesized as follows:

```
y = ( a * x * x ) + ( b * x ) + c;
```

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;` `2 * 5 is 10`	*(Leftmost multiplication)*
Step 2.	`y = 10 * 5 + 3 * 5 + 7;` `10 * 5 is 50`	*(Leftmost multiplication)*
Step 3.	`y = 50 + 3 * 5 + 7;` `3 * 5 is 15`	*(Multiplication before addition)*
Step 4.	`y = 50 + 15 + 7;` `50 + 15 is 65`	*(Leftmost addition)*
Step 5.	`y = 65 + 7;` `65 + 7 is 72`	*(Last addition)*
Step 6.	`y = 72`	*(Last operation—place* `72` *in* `y`*)*

Fig. 2.11 | Order in which a second-degree polynomial is evaluated.

2.7 Decision Making: Equality and Relational Operators

We now introduce a simple version of C++'s **`if` statement** that allows a program to take alternative action based on whether a **condition** is true or false. If the condition is true, the statement in the body of the `if` statement is executed. If the condition is false, the body statement is not executed. We'll see an example shortly.

Conditions in `if` statements can be formed by using the **equality operators** and **relational operators** summarized in Fig. 2.12. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is lower than that of the relational operators, and associate left to right.

Common Programming Error 2.3

*Reversing the order of the pair of symbols in the operators `!=`, `>=` and `<=` (by writing them as `=!`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `!=` as `=!` will not be a syntax error, but almost certainly will be a **logic error** that has an effect at execution time. You'll understand why when you learn about logical operators in Chapter 5. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing, but usually produces incorrect results.*

Common Programming Error 2.4

Confusing the equality operator `==` with the assignment operator `=` results in logic errors. Read the equality operator should be read "is equal to" or "double equals," and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." As we discuss in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.

Standard algebraic equality or relational operator	C++ equality or relational operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`
Equality operators			
=	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`

Fig. 2.12 | Equality and relational operators.

Using the `if` Statement

The following example (Fig. 2.13) uses six `if` statements to compare two numbers input by the user. If the condition in any of these `if` statements is satisfied, the output statement associated with that `if` statement is executed.

```cpp
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
   int number1; // first integer to compare
   int number2; // second integer to compare

   cout << "Enter two integers to compare: "; // prompt user for data
   cin >> number1 >> number2; // read two integers from user

   if ( number1 == number2 )
      cout << number1 << " == " << number2 << endl;

   if ( number1 != number2 )
      cout << number1 << " != " << number2 << endl;

   if ( number1 < number2 )
      cout << number1 << " < " << number2 << endl;
```

Fig. 2.13 | Comparing integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

```
27
28        if ( number1 > number2 )
29           cout << number1 << " > " << number2 << endl;
30
31        if ( number1 <= number2 )
32           cout << number1 << " <= " << number2 << endl;
33
34        if ( number1 >= number2 )
35           cout << number1 << " >= " << number2 << endl;
36     } // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Fig. 2.13 | Comparing integers using `if` statements, relational operators and equality operators. (Part 2 of 2.)

`using` Directives

Lines 6–8

```
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are **`using` directives** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. We can now write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, respectively, in the remainder of the program.

In place of lines 6–8, many programmers prefer to use the directive

```
using namespace std;
```

which enables a program to use *all* the names in any standard C++ header (such as `<iostream>`) that a program might include. From this point forward in the book, we'll use the preceding directive in our programs.

Variable Declarations and Reading the Inputs from the User

Lines 13–14

```
int number1; // first integer to compare
int number2; // second integer to compare
```

declare the variables used in the program.

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we're allowed to write `cin` (instead of `std::cin`) because of line 7. First a value is read into variable `number1`, then a value is read into variable `number2`.

Comparing Numbers

The `if` statement in lines 19–20

```
if ( number1 == number2 )
   cout << number1 << " == " << number2 << endl;
```

compares the values of variables `number1` and `number2` to test for equality. If the values are equal, the statement in line 20 displays a line of text indicating that the numbers are equal. If the conditions are `true` in one or more of the `if` statements starting in lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Each `if` statement in Fig. 2.13 has a single statement in its body and each body statement is indented. In Chapter 4 we show how to specify `if` statements with multiple-statement bodies (by enclosing the body statements in a pair of braces, `{ }`, creating what's called a **compound statement** or a **block**).

Good Programming Practice 2.10

Indent the statement(s) in the body of an `if` statement to enhance readability.

Common Programming Error 2.5

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence *with the `if` statement and* always *executes, often causing the program to produce incorrect results.*

White Space

Note the use of white space in Fig. 2.13. Recall that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It's a syntax error to split identifiers, strings (such as `"hello"`) and constants (such as the number `1000`) over several lines.

Good Programming Practice 2.11

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

Operator Precedence

Figure 2.14 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. All these operators, with the exception of the assignment operator `=`, associate from left to right. Addition is left-associative, so an expression like `x + y + z` is evaluated as if it had been written `(x + y) + z`. The assignment operator `=` associates from *right to left*, so an expression such as `x = y = 0` is evaluated as if it had been written `x = (y = 0)`, which, as we'll soon see, first assigns `0` to `y`, then assigns the *result* of that assignment—`0`—to `x`.

Operators	Associativity	Type
`()`	*[See caution in Fig. 2.10]*	grouping parentheses
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	stream insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`=`	right to left	assignment

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Good Programming Practice 2.12

Refer to the operator precedence and associativity chart (Appendix A) when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.

2.8 Wrap-Up

You learned many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. In particular, you learned to use the output stream object `cout` and the input stream object `cin` to build simple interactive programs. We explained how variables are stored in and retrieved from memory. You also learned how to use arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the associativity of the operators. You also learned how C++'s `if` statement allows a program to make decisions. Finally, we introduced the equality and relational operators, which you use to form conditions in `if` statements.

The non-object-oriented applications presented here introduced you to basic programming concepts. As you'll see in Chapter 3, C++ applications typically contain just a few lines of code in function `main`—these statements normally create the objects that perform the work of the application, then the objects "take over from there." In Chapter 3, you'll learn how to implement your own classes and use objects of those classes in applications.

Summary

Section 2.2 First Program in C++: Printing a Line of Text

- Single-line comments (p. 39) begin with `//`. You insert comments to document your programs and improve their readability.

- Comments do not cause the computer to perform any action (p. 40) when the program is run—they're ignored by the compiler and do not cause any machine-language object code to be generated.
- A preprocessor directive (p. 39) begins with `#` and is a message to the C++ preprocessor. Preprocessor directives are processed before the program is compiled and don't end with a semicolon.
- The line `#include <iostream>` (p. 39) tells the C++ preprocessor to include the contents of the input/output stream header, which contains information necessary to compile programs that use `std::cin` (p. 43) and `std::cout` (p. 40) and the stream insertion (<<, p. 40) and stream extraction (>>, p. 43) operators.
- White space (i.e., blank lines, space characters and tab characters, p. 39) makes programs easier to read. White-space characters outside of literals are ignored by the compiler.
- C++ programs begin executing at `main` (p. 39), even if `main` does not appear first in the program.
- The keyword `int` to the left of `main` indicates that `main` "returns" an integer value.
- The body (p. 40) of every function must be contained in braces (`{` and `}`).
- A string (p. 40) in double quotes is sometimes referred to as a character string, message or string literal. White-space characters in strings are *not* ignored by the compiler.
- Every statement (p. 40) must end with a semicolon (also known as the statement terminator).
- Output and input in C++ are accomplished with streams (p. 40) of characters.
- The output stream object `std::cout`—normally connected to the screen—is used to output data. Multiple data items can be output by concatenating stream insertion (<<) operators.
- The input stream object `std::cin`—normally connected to the keyboard—is used to input data. Multiple data items can be input by concatenating stream extraction (>>) operators.
- The notation `std::cout` specifies that we are using `cout` from "namespace" `std`.
- When a backslash (i.e., an escape character) is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence (p. 41).
- The newline escape sequence `\n` (p. 41) moves the cursor to the beginning of the next line on the screen.
- A message that directs the user to take a specific action is known as a prompt (p. 45).
- C++ keyword `return` (p. 41) is one of several means to exit a function.

Section 2.4 Another C++ Program: Adding Integers

- All variables (p. 43) in a C++ program must be declared before they can be used.
- A variable name is any valid identifier (p. 44) that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (_). Identifiers cannot start with a digit. Identifiers can be any length, but some systems or C++ implementations may impose length restrictions.
- C++ is case sensitive (p. 44).
- Most calculations are performed in assignment statements (p. 46).
- A variable is a location in memory (p. 47) where a value can be stored for use by a program.
- Variables of type `int` (p. 44) hold integer values, i.e., whole numbers such as 7, –11, 0, 31914.

Section 2.5 Memory Concepts

- Every variable stored in the computer's memory has a name, a value, a type and a size.
- Whenever a new value is placed in a memory location, the process is destructive (p. 47); i.e., the new value replaces the previous value in that location. The previous value is lost.

- When a value is read from memory, the process is nondestructive (p. 47); i.e., a copy of the value is read, leaving the original value undisturbed in the memory location.
- The `std::endl` stream manipulator (p. 46) outputs a newline, then "flushes the output buffer."

Section 2.6 Arithmetic

- C++ evaluates arithmetic expressions (p. 48) in a precise sequence determined by the rules of operator precedence (p. 48) and associativity (p. 49).
- Parentheses may be used to group expressions.
- Integer division (p. 48) yields an integer quotient. Any fractional part in integer division is truncated.
- The modulus operator, `%` (p. 48), yields the remainder after integer division.

Section 2.7 Decision Making: Equality and Relational Operators

- The `if` statement (p. 51) allows a program to take alternative action based on whether a condition is met. The format for an `if` statement is

```
if ( condition )
   statement;
```

  If the condition is true, the statement in the body of the `if` is executed. If the condition is not met, i.e., the condition is false, the body statement is skipped.
- Conditions in `if` statements are commonly formed by using equality and relational operators (p. 51). The result of using these operators is always the value true or false.
- The `using` directive (p. 53)

```
using std::cout;
```

  informs the compiler where to find `cout` (namespace `std`) and eliminates the need to repeat the `std::` prefix. The directive

```
using namespace std;
```

  enables the program to use all the names in any included C++ standard library header.

Self-Review Exercises

2.1 Fill in the blanks in each of the following.

a) Every C++ program begins execution at the function ________.
b) A(n) ________ begins the body of every function and a(n) ________ ends the body.
c) Every C++ statement ends with a(n) ________.
d) The escape sequence `\n` represents the ________ character, which causes the cursor to position to the beginning of the next line on the screen.
e) The ________ statement is used to make decisions.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why. Assume the statement `using std::cout;` is used.

a) Comments cause the computer to print the text after the `//` on the screen when the program is executed.
b) The escape sequence `\n`, when output with `cout` and the stream insertion operator, causes the cursor to position to the beginning of the next line on the screen.
c) All variables must be declared before they're used.
d) All variables must be given a type when they're declared.
e) C++ considers the variables `number` and `NuMbEr` to be identical.
f) Declarations can appear almost anywhere in the body of a C++ function.
g) The modulus operator (`%`) can be used only with integer operands.

h) The arithmetic operators `*`, `/`, `%`, `+` and – all have the same level of precedence.
i) A C++ program that prints three lines of output must contain three statements using `cout` and the stream insertion operator.

2.3 Write a single C++ statement to accomplish each of the following (assume that `using` directives have not been used):
a) Declare the variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int`.
b) Prompt the user to enter an integer. End your prompting message with a colon (`:`) followed by a space and leave the cursor positioned after the space.
c) Read an integer from the user at the keyboard and store it in integer variable `age`.
d) If the variable `number` is not equal to `7`, print `"The variable number is not equal to 7"`.
e) Print the message `"This is a C++ program"` on one line.
f) Print the message `"This is a C++ program"` on two lines. End the first line with `C++`.
g) Print the message `"This is a C++ program"` with each word on a separate line.
h) Print the message `"This is a C++ program"`. Separate each word from the next by a tab.

2.4 Write a statement (or comment) to accomplish each of the following (assume that `using` directives have been used for `cin`, `cout` and `endl`):
a) State that a program calculates the product of three integers.
b) Declare the variables `x`, `y`, `z` and `result` to be of type `int` (in separate statements).
c) Prompt the user to enter three integers.
d) Read three integers from the keyboard and store them in the variables `x`, `y` and `z`.
e) Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
f) Print `"The product is "` followed by the value of the variable `result`.
g) Return a value from `main` indicating that the program terminated successfully.

2.5 Using the statements you wrote in Exercise 2.4, write a complete program that calculates and displays the product of three integers. Add comments to the code where appropriate. [*Note:* You'll need to write the necessary `using` directives.]

2.6 Identify and correct the errors in each of the following statements (assume that the statement `using std::cout;` is used):
a)
```
if ( c < 7 );
   cout << "c is less than 7\n";
```
b)
```
if ( c => 7 )
   cout << "c is equal to or greater than 7\n";
```

Answers to Self-Review Exercises

2.1 a) `main`. b) left brace (`{`), right brace (`}`). c) semicolon. d) newline. e) `if`.

2.2 a) False. Comments do not cause any action to be performed when the program is executed. They're used to document programs and improve their readability.
b) True.
c) True.
d) True.
e) False. C++ is case sensitive, so these variables are unique.
f) True.
g) True.
h) False. The operators `*`, `/` and `%` have the same precedence, and the operators `+` and `-` have a lower precedence.
i) False. One statement with `cout` and multiple `\n` escape sequences can print several lines.

2.3 a)
```
int c, thisIsAVariable, q76354, number;
```

b) `std::cout << "Enter an integer: ";`

c) `std::cin >> age;`

d)
```
if ( number != 7 )
   std::cout << "The variable number is not equal to 7\n";
```

e) `std::cout << "This is a C++ program\n";`

f) `std::cout << "This is a C++\nprogram\n";`

g) `std::cout << "This\nis\na\nC++\nprogram\n";`

h) `std::cout << "This\tis\ta\tC++\tprogram\n";`

2.4 a) `// Calculate the product of three integers`

b)
```
int x;
int y;
int z;
int result;
```

c) `cout << "Enter three integers: ";`

d) `cin >> x >> y >> z;`

e) `result = x * y * z;`

f) `cout << "The product is " << result << endl;`

g) `return 0;`

2.5 (See program below.)

```
// Calculate the product of three integers
#include <iostream> // allows program to perform input and output
using namespace std; // program uses names from the std namespace

// function main begins program execution
int main()
{
   int x; // first integer to multiply
   int y; // second integer to multiply
   int z; // third integer to multiply
   int result; // the product of the three integers

   cout << "Enter three integers: "; // prompt user for data
   cin >> x >> y >> z; // read three integers from user
   result = x * y * z; // multiply the three integers; store result
   cout << "The product is " << result << endl; // print result; end line
} // end function main
```

2.6 a) *Error:* Semicolon after the right parenthesis of the condition in the `if` statement.
Correction: Remove the semicolon after the right parenthesis. [*Note:* The result of this error is that the output statement executes whether or not the condition in the `if` statement is true.] The semicolon after the right parenthesis is a null (or empty) statement that does nothing. We'll learn more about the null statement in Chapter 4.

b) *Error:* The relational operator `=>`.
Correction: Change `=>` to `>=`, and you may want to change "equal to or greater than" to "greater than or equal to" as well.

Exercises

2.7 Discuss the meaning of each of the following objects:

a) `std::cin`

b) `std::cout`

2.8 Fill in the blanks in each of the following:

a) ________ are used to document a program and improve its readability.
b) The object used to print information on the screen is ________.
c) A C++ statement that makes a decision is ________.
d) Most calculations are normally performed by ________ statements.
e) The ________ object inputs values from the keyboard.

2.9 Write a single C++ statement or line that accomplishes each of the following:

a) Print the message `"Enter two numbers"`.
b) Assign the product of variables `b` and `c` to variable `a`.
c) State that a program performs a payroll calculation (i.e., use text that helps to document a program).
d) Input three integer values from the keyboard into integer variables `a`, `b` and `c`.

2.10 State which of the following are *true* and which are *false*. If *false*, explain your answers.

a) C++ operators are evaluated from left to right.
b) The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
c) The statement `cout << "a = 5;";` is a typical example of an assignment statement.
d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.
e) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Fill in the blanks in each of the following:

a) What arithmetic operations are on the same level of precedence as multiplication? ________.
b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? ________.
c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a(n) ________.

2.12 What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer "nothing." Assume `x = 2` and `y = 3`.

a) `cout << x;`
b) `cout << x + x;`
c) `cout << "x=";`
d) `cout << "x = " << x;`
e) `cout << x + y << " = " << y + x;`
f) `z = x + y;`
g) `cin >> x >> y;`
h) `// cout << "x + y = " << x + y;`
i) `cout << "\n";`

2.13 Which of the following C++ statements contain variables whose values are replaced?

a) `cin >> b >> c >> d >> e >> f;`
b) `p = i + j + k + 7;`
c) `cout << "variables whose values are replaced";`
d) `cout << "a = 5";`

2.14 Given the algebraic equation $y = ax^3 + 7$, which of the following, if any, are correct C++ statements for this equation?

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * ( x + 7 );`
c) `y = ( a * x ) * x * ( x + 7 );`
d) `y = (a * x) * x * x + 7;`

e) `y = a * ( x * x * x ) + 7;`
f) `y = a * x * ( x * x + 7 );`

2.15 *(Order of Evalution)* State the order of evaluation of the operators in each of the following C++ statements and show the value of `x` after each statement is performed.

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

2.16 *(Arithmetic)* Write a program that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference, and quotient of the two numbers.

2.17 *(Printing)* Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Do this several ways:

a) Using one statement with one stream insertion operator.
b) Using one statement with four stream insertion operators.
c) Using four statements.

2.18 *(Comparing Integers)* Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words `"is larger."` If the numbers are equal, print the message `"These numbers are equal."`

2.19 *(Arithmetic, Smallest and Largest)* Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialog should appear as follows:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20 *(Diameter, Circumference and Area of a Circle)* Write a program that reads in the radius of a circle as an integer and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Do all calculations in output statements. [*Note:* In this chapter, we've discussed only integer constants and variables. In Chapter 4 we discuss floating-point numbers, i.e., values that can have decimal points.]

2.21 *(Displaying Shapes with Asterisks)* Write a program that prints a box, an oval, an arrow and a diamond as follows:

```
*********         ***           *             *
*       *       *     *        ***           * *
*       *      *       *      *****         *   *
*       *      *       *        *          *     *
*       *      *       *        *         *       *
*       *      *       *        *          *     *
*       *      *       *        *           *   *
*       *       *     *         *            * *
*********         ***           *             *
```

2.22 What does the following code print?

```
cout << "*\n**\n***\n****\n*****" << endl;
```

2.23 *(Largest and Smallest Integers)* Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.24 *(Odd or Even)* Write a program that reads an integer and determines and prints whether it's odd or even. [*Hint:* Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.]

2.25 *(Multiples)* Write a program that reads in two integers and determines and prints if the first is a multiple of the second. [*Hint:* Use the modulus operator.]

2.26 *(Checkerboard Pattern)* Display the following checkerboard pattern with eight output statements, then display the same pattern using as few statements as possible.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.27 *(Integer Equivalent of a Character)* Here is a peek ahead. In this chapter you learned about integers and the type `int`. C++ can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C++ uses small integers internally to represent each different character. The set of characters a computer uses and the corresponding integer representations for those characters are called that computer's **character set**. You can print a character by enclosing that character in single quotes, as with

```
cout << 'A'; // print an uppercase A
```

You can print the integer equivalent of a character using `static_cast` as follows:

```
cout << static_cast< int >( 'A' ); // print 'A' as an integer
```

This is called a **cast** operation (we formally introduce casts in Chapter 4). When the preceding statement executes, it prints the value 65 (on systems that use the **ASCII character set**). Write a program that prints the integer equivalent of a character typed at the keyboard. Store the input in a variable of type `char`. Test your program several times using uppercase letters, lowercase letters, digits and special characters (like `$`).

2.28 *(Digits of an Integer)* Write a program that inputs a five-digit integer, separates the integer into its digits and prints them separated by three spaces each. [*Hint:* Use the integer division and modulus operators.] For example, if the user types in 42339, the program should print:

```
4   2   3   3   9
```

2.29 *(Table)* Using the techniques of this chapter, write a program that calculates the squares and cubes of the integers from 0 to 10. Use tabs to print the following neatly formatted table of values:

```
integer square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

Making a Difference

2.30 *(Body Mass Index Calculator)* We introduced the body mass index (BMI) calculator in Exercise 1.9. The formulas for calculating BMI are

$$BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}$$

or

$$BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}$$

Create a BMI calculator application that reads the user's weight in pounds and height in inches (or, if you prefer, the user's weight in kilograms and height in meters), then calculates and displays the user's body mass index. Also, the application should display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

```
BMI VALUES
Underweight: less than 18.5
Normal:      between 18.5 and 24.9
Overweight:  between 25 and 29.9
Obese:       30 or greater
```

[*Note:* In this chapter, you learned to use the `int` type to represent whole numbers. The BMI calculations when done with `int` values will both produce whole-number results. In Chapter 4 you'll learn to use the `double` type to represent numbers with decimal points. When the BMI calculations are performed with `double`s, they'll both produce numbers with decimal points—these are called "floating-point" numbers.]

2.31 *(Car-Pool Savings Calculator)* Research several car-pooling websites. Create an application that calculates your daily driving cost, so that you can estimate how much money could be saved by car pooling, which also has other advantages such as reducing carbon emissions and reducing traffic congestion. The application should input the following information and display the user's cost per day of driving to work:

a) Total miles driven per day.
b) Cost per gallon of gasoline.
c) Average miles per gallon.
d) Parking fees per day.
e) Tolls per day.

3 Introduction to Classes, Objects and Strings

Nothing can have value without being an object of utility.
—Karl Marx

Your public servants serve you right.
—Adlai E. Stevenson

Knowing how to answer one who speaks,
To reply to one who sends a message.
—Amenemopel

Objectives

In this chapter you'll learn:

- How to define a class and use it to create an object.
- How to implement a class's behaviors as member functions.
- How to implement a class's attributes as data members.
- How to call a member function of an object to perform a task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to initialize an object's data when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.
- How to use objects of class `string`.

Outline

3.1 Introduction

In Chapter 2, you created simple programs that displayed messages to the user, obtained information from the user, performed calculations and made decisions. In this chapter, you'll begin writing programs that employ the basic concepts of *object-oriented programming* that we introduced in Section 1.6. One common feature of every program in Chapter 2 was that all the statements that performed tasks were located in function `main`. Typically, the programs you develop in this book will consist of function `main` and one or more *classes*, each containing *data members* and *member functions*. If you become part of a development team in industry, you might work on software systems that contain hundreds, or even thousands, of classes. In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs in C++.

We present a carefully paced sequence of complete working programs to demonstrate creating and using your own classes. These examples begin our integrated case study on developing a grade-book class that instructors can use to maintain student test scores. We also introduce the C++ standard library class `string`.

3.2 Defining a Class with a Member Function

We begin with an example (Fig. 3.1) that consists of class `GradeBook` (lines 8–16)—which, when it's fully developed in Chapter 7, will represent a grade book that an instructor can use to maintain student test scores—and a `main` function (lines 19–23) that creates a `GradeBook` object. Function `main` uses this object and its member function to display a message on the screen welcoming the instructor to the grade-book program.

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define class GradeBook with a member function displayMessage,
3  // create a GradeBook object, and call its displayMessage function.
4  #include <iostream>
5  using namespace std;
```

Fig. 3.1 | Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object and call its `displayMessage` function. (Part 1 of 2.)

```
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    // function that displays a welcome message to the GradeBook user
12    void displayMessage()
13    {
14       cout << "Welcome to the Grade Book!" << endl;
15    } // end function displayMessage
16 }; // end class GradeBook
17
18 // function main begins program execution
19 int main()
20 {
21    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
22    myGradeBook.displayMessage(); // call object's displayMessage function
23 } // end main
```

```
Welcome to the Grade Book!
```

Fig. 3.1 | Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object and call its `displayMessage` function. (Part 2 of 2.)

Class `GradeBook`

Before function `main` (lines 19–23) can create a `GradeBook` object, we must tell the compiler what member functions and data members belong to the class. The `GradeBook` **class definition** (lines 8–16) contains a member function called `displayMessage` (lines 12–15) that displays a message on the screen (line 14). We need to make an object of class `GradeBook` (line 21) and call its `displayMessage` member function (line 22) to get line 14 to execute and display the welcome message. We'll soon explain lines 21–22 in detail.

The class definition begins in line 8 with the keyword `class` followed by the class name `GradeBook`. By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **Pascal case**, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

Every class's **body** is enclosed in a pair of left and right braces (`{` and `}`), as in lines 9 and 16. The class definition terminates with a semicolon (line 16).

Common Programming Error 3.1

Forgetting the semicolon at the end of a class definition is a syntax error.

Recall that the function `main` is always called automatically when you execute a program. Most functions do *not* get called automatically. As you'll soon see, you must call member function `displayMessage` *explicitly* to tell it to perform its task.

Line 10 contains the keyword **`public`**, which is an **access specifier**. Lines 12–15 define member function `displayMessage`. This member function appears after access specifier `public:` to indicate that the function is "available to the public"—that is, it can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any). Access specifiers are always followed by a colon (`:`). For the remainder of the text, when we refer to the access specifier `public`, we'll omit the colon as we did in this sentence. Section 3.4 introduces the access specifier, `private`. Later in the book we'll study the access specifier `protected`.

Each function in a program performs a task and may *return a value* when it completes its task—for example, a function might perform a calculation, then return the result of that calculation. When you define a function, you must specify a **return type** to indicate the type of the value returned by the function when it completes its task. In line 12, keyword **`void`** to the left of the function name `displayMessage` is the function's return type. Return type `void` indicates that `displayMessage` will *not* return any data to its **calling function** (in this example, line 22 of `main`, as we'll see in a moment) when it completes its task. In Fig. 3.5, you'll see an example of a function that *does* return a value.

The name of the member function, `displayMessage`, follows the return type (line 12). By convention, function names begin with a *lowercase* first letter and all subsequent words in the name begin with a capital letter. This capitalization style is often refered to as **camel case** and is also used for variable names. The parentheses after the member function name indicate that this is a function. An empty set of parentheses, as shown in line 12, indicates that this member function does *not* require additional data to perform its task. You'll see an example of a member function that *does* require additional data in Section 3.3. Line 12 is commonly referred to as a **function header**. Every function's *body* is delimited by left and right braces (`{` and `}`), as in lines 13 and 15.

The *body of a function* contains statements that perform the function's task. In this case, member function `displayMessage` contains one statement (line 14) that displays the message `"Welcome to the Grade Book!"`. After this statement executes, the function has completed its task.

Testing Class **`GradeBook`**

Next, we'd like to use class `GradeBook` in a program. As you saw in Chapter 2, the function `main` (lines 19–23) begins the execution of every program.

In this program, we'd like to call class `GradeBook`'s `displayMessage` member function to display the welcome message. Typically, you cannot call a member function of a class until you create an object of that class. (As you'll learn in Section 10.6, `static` member functions are an exception.) Line 21 creates an object of class `GradeBook` called `myGradeBook`. The variable's type is `GradeBook`—the class we defined in lines 8–16. When we declare variables of type `int`, as we did in Chapter 2, the compiler knows what `int` is—it's a *fundamental type* that's "built into" C++. In line 21, however, the compiler does *not* automatically know what type `GradeBook` is—it's a **user-defined type**. We tell the compiler what `GradeBook` is by including the *class definition* (lines 8–16). If we omitted these lines, the compiler would issue an error message. Each class you create becomes a new *type* that can be used to create objects. You can define new class types as needed; this is one reason why C++ is known as an **extensible language**.

Line 22 *calls* the member function `displayMessage` using variable `myGradeBook` followed by the **dot operator** (`.`), the function name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` function to perform its task. At the beginning of line 22, "`myGradeBook.`" indicates that `main` should use the `GradeBook` object that was created in line 21. The empty parentheses in line 12 indicate that member function `displayMessage` does *not* require additional data to perform its task, which is why we called this function with empty parentheses in line 22. (In Section 3.3, you'll see how to pass data to a function.) When `displayMessage` completes its task, the program reaches the end of `main` (line 23) and terminates.

UML Class Diagram for Class GradeBook

Recall from Section 1.6 that the UML is a standardized graphical language used by software developers to represent their object-oriented systems. In the UML, each class is modeled in a **UML class diagram** as a rectangle with three compartments. Figure 3.2 presents a class diagram for class `GradeBook` (Fig. 3.1). The *top compartment* contains the class's name centered horizontally and in boldface type. The *middle compartment* contains the class's attributes, which correspond to data members in C++. This compartment is currently empty, because class `GradeBook` does not have any attributes. (Section 3.4 presents a version of class `GradeBook` with an attribute.) The *bottom compartment* contains the class's operations, which correspond to member functions in C++. The UML models operations by listing the operation name followed by a set of parentheses. Class `GradeBook` has only one member function, `displayMessage`, so the bottom compartment of Fig. 3.2 lists one operation with this name. Member function `displayMessage` does *not* require additional information to perform its tasks, so the parentheses following `displayMessage` in the class diagram are *empty*, just as they are in the member function's header in line 12 of Fig. 3.1. The *plus sign (+)* in front of the operation name indicates that `displayMessage` is a *public* operation in the UML (i.e., a `public` member function in C++).

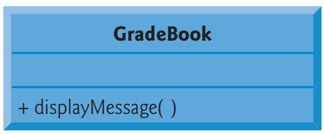


Fig. 3.2 | UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation.

3.3 Defining a Member Function with a Parameter

In our car analogy from Section 1.6, we mentioned that pressing a car's gas pedal sends a message to the car to perform a task—make the car go faster. But *how fast* should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car includes *both* the *task to perform* and *additional information that helps the car perform the task*. This additional information is known as a **parameter**—the *value* of the parameter helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies values—called **arguments**—for each of the function's parameters. For example, to make a deposit into a bank account, suppose a `deposit` member function of an `Account` class specifies a parameter that represents the *deposit amount*. When the `deposit` member function is called, an argument value representing the deposit amount is copied to the member function's parameter. The member function then adds that amount to the account balance.

Defining and Testing Class **GradeBook**

Our next example (Fig. 3.3) redefines class `GradeBook` (lines 9–18) with a `displayMessage` member function (lines 13–17) that displays the course name as part of the welcome message. The new version of `displayMessage` requires a parameter (`courseName` in line 13) that represents the course name to output.

```
1  // Fig. 3.3: fig03_03.cpp
2  // Define class GradeBook with a member function that takes a parameter,
3  // create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  #include <string> // program uses C++ standard string class
6  using namespace std;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // function that displays a welcome message to the GradeBook user
13    void displayMessage( string courseName )
14    {
15       cout << "Welcome to the grade book for\n" << courseName << "!"
16          << endl;
17    } // end function displayMessage
18 }; // end class GradeBook
19
20 // function main begins program execution
21 int main()
22 {
23    string nameOfCourse; // string of characters to store the course name
24    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
25
26    // prompt for and input course name
27    cout << "Please enter the course name:" << endl;
28    getline( cin, nameOfCourse ); // read a course name with blanks
29    cout << endl; // output a blank line
30
31    // call myGradeBook's displayMessage function
32    // and pass nameOfCourse as an argument
33    myGradeBook.displayMessage( nameOfCourse );
34 } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Fig. 3.3 | Define class `GradeBook` with a member function that takes a parameter, create a `GradeBook` object and call its `displayMessage` function.

Before discussing the new features of class `GradeBook`, let's see how the new class is used in `main` (lines 21–34). Line 23 creates a variable of type **`string`** called `nameOfCourse` that will be used to store the course name entered by the user. A variable of type `string` represents a string of characters such as "`CS101 Introduction to C++ Programming`". A string is actually an *object* of the C++ Standard Library class `string`. This class is defined in **header `<string>`**, and the name `string`, like `cout`, belongs to namespace `std`. To enable lines 13 and 23 to compile, line 5 includes the `<string>` header. The `using` directive in line 6 allows us to simply write `string` in line 23 rather than `std::string`. For now, you can think of `string` variables like variables of other types such as `int`. You'll learn additional `string` capabilities in Section 3.8 and in Chapter 18.

Line 24 creates an object of class `GradeBook` named `myGradeBook`. Line 27 prompts the user to enter a course name. Line 28 reads the name from the user and assigns it to the `nameOfCourse` variable, using the library function **`getline`** to perform the input. Before we explain this line of code, let's explain why we cannot simply write

```
cin >> nameOfCourse;
```

to obtain the course name. In our sample program execution, we use the course name "`CS101 Introduction to C++ Programming`," which contains multiple words separated by blanks. (Recall that we highlight user-supplied input in bold.) When `cin` is used with the stream extraction operator, it reads characters *until the first white-space character is reached*. Thus, only "`CS101`" would be read by the preceding statement. The rest of the course name would have to be read by subsequent input operations.

In this example, we'd like the user to type the complete course name and press *Enter* to submit it to the program, and we'd like to store the entire course name in the `string` variable `nameOfCourse`. The function call `getline( cin, nameOfCourse )` in line 28 reads characters (*including* the space characters that separate the words in the input) from the standard input stream object `cin` (i.e., the keyboard) until the newline character is encountered, places the characters in the `string` variable `nameOfCourse` and *discards* the newline character. When you press *Enter* while typing program input, a newline is inserted in the input stream. The `<string>` header must be included in the program to use function `getline`, which belongs to namespace `std`.

Line 33 calls `myGradeBook`'s `displayMessage` member function. The `nameOfCourse` variable in parentheses is the *argument* that's passed to member function `displayMessage` so that it can perform its task. The value of variable `nameOfCourse` in `main` is copied to member function `displayMessage`'s parameter `courseName` in line 13. When you execute this program, member function `displayMessage` outputs as part of the welcome message the course name you type (in our sample execution, `CS101 Introduction to C++ Programming`).

More on Arguments and Parameters

To specify in a function definition that the function requires data to perform its task, you place additional information in the function's **parameter list**, which is located in the parentheses following the function name. The parameter list may contain any number of parameters, including none at all (represented by empty parentheses as in Fig. 3.1, line 12) to indicate that a function does *not* require any parameters. Member function `displayMessage`'s parameter list (Fig. 3.3, line 13) declares that the function requires one parameter. Each parameter specifies a *type* and an *identifier*. The type `string` and the identifier `courseName` indicate that member function `displayMessage` requires a `string` to perform its task. The member function body uses the parameter `courseName` to access the value that's passed to the function in the function call (line 33 in `main`). Lines 15–16 display parameter `courseName`'s value as part of the welcome message. The parameter variable's name (`courseName` in line 13) can be the *same* as or *different* from the argument variable's name (`nameOfCourse` in line 33)—you'll learn why in Chapter 6.

A function can specify multiple parameters by separating each from the next with a comma. The number and order of arguments in a function call *must match* the number and order of parameters in the parameter list of the called member function's header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header. (As you'll learn in subsequent chapters, an argument's type and its corresponding parameter's type need not always be *identical*, but they must be "consistent.") In our example, the one `string` argument in the function call (i.e., `nameOfCourse`) *exactly matches* the one `string` parameter in the member-function definition (i.e., `courseName`).

Updated UML Class Diagram for Class **GradeBook**

The UML class diagram of Fig. 3.4 models class `GradeBook` of Fig. 3.3. Like the class `GradeBook` defined in Fig. 3.1, this `GradeBook` class contains `public` member function `displayMessage`. However, this version of `displayMessage` has a parameter. The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its own data types similar to those of C++. The UML is language independent—it's used with many different programming languages—so its terminology does not exactly match that of C++. For example, the UML type `String` corresponds to the C++ type `string`. Member function `displayMessage` of class `GradeBook` (Fig. 3.3, lines 13–17) has a `string` parameter named `courseName`, so Fig. 3.4 lists `courseName : String` between the parentheses following the operation name `displayMessage`. This version of the `GradeBook` class still does *not* have any data members.

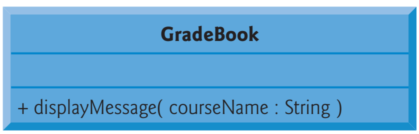


Fig. 3.4 | UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation with a `courseName` parameter of UML type `String`.

3.4 Data Members, *set* Functions and *get* Functions

In Chapter 2, we declared all of a program's variables in its `main` function. Variables declared in a function definition's body are known as **local variables** and can be used *only* from the line of their declaration in the function to closing right brace (`}`) of the block in which they're declared. A local variable must be declared *before* it can be used in a function. A local variable cannot be accessed outside the function in which it's declared. *When a function terminates, the values of its local variables are lost.* (You'll see an exception to this in Chapter 6 when we discuss `static` local variables.)

A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called **data members** and are declared *inside* a class definition but *outside* the bodies of the class's member-function definitions. Each object of a class maintains its own copy of its attributes in memory. These attributes exist throughout the life of the object. The example in this section demonstrates a `GradeBook` class that contains a `courseName` data member to represent a particular `GradeBook` object's course name.

`GradeBook` Class with a Data Member, a set *Function and a* get *Function*

In our next example, class `GradeBook` (Fig. 3.5) maintains the course name as a *data member* so that it can be used or modified at any time during a program's execution. The class contains member functions `setCourseName`, `getCourseName` and `displayMessage`. Member function `setCourseName` stores a course name in a `GradeBook` data member. Member function `getCourseName` obtains the course name from that data member. Member function `displayMessage`—which now specifies no parameters—still displays a welcome message that includes the course name. However, as you'll see, the function now obtains the course name by calling another function in the same class—`getCourseName`.

```cpp
// Fig. 3.5: fig03_05.cpp
// Define class GradeBook that contains a courseName data member
// and member functions to set and get its value;
// Create and manipulate a GradeBook object with these functions.
#include <iostream>
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   // function that sets the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function that gets the course name
   string getCourseName()
   {
      return courseName; // return the object's courseName
   } // end function getCourseName

   // function that displays a welcome message
   void displayMessage()
   {
      // this statement calls getCourseName to get the
      // name of the course this GradeBook represents
      cout << "Welcome to the grade book for\n" << getCourseName() << "!"
         << endl;
   } // end function displayMessage
```

Fig. 3.5 | Defining and testing class `GradeBook` with a data member and *set* and *get* functions. (Part 1 of 2.)

```
33 private:
34    string courseName; // course name for this GradeBook
35 }; // end class GradeBook
36
37 // function main begins program execution
38 int main()
39 {
40    string nameOfCourse; // string of characters to store the course name
41    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
42
43    // display initial value of courseName
44    cout << "Initial course name is: " << myGradeBook.getCourseName()
45       << endl;
46
47    // prompt for, input and set course name
48    cout << "\nPlease enter the course name:" << endl;
49    getline( cin, nameOfCourse ); // read a course name with blanks
50    myGradeBook.setCourseName( nameOfCourse ); // set the course name
51
52    cout << endl; // outputs a blank line
53    myGradeBook.displayMessage(); // display message with new course name
54 } // end main
```

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Fig. 3.5 | Defining and testing class `GradeBook` with a data member and *set* and *get* functions. (Part 2 of 2.)

A typical instructor teaches multiple courses, each with its own course name. Line 34 declares that `courseName` is a variable of type `string`. Because the variable is declared in the class definition (lines 10–35) but outside the bodies of the class's member-function definitions (lines 14–17, 20–23 and 26–32), the variable is a *data member*. Every instance (i.e., object) of class `GradeBook` contains one copy of each of the class's data members—if there are two `GradeBook` objects, each has its *own* copy of `courseName` (one per object), as you'll see in the example of Fig. 3.7. A benefit of making `courseName` a data member is that all the member functions of the class can manipulate any data members that appear in the class definition (in this case, `courseName`).

Access Specifiers `public` *and* `private`

Most data-member declarations appear after the **`private`** access specifier. Variables or functions declared after access specifier `private` (and before the next access specifier if there is one) are accessible only to member functions of the class for which they're declared (or to "friends" of the class, as you'll see in Chapter 10, Classes: A Deeper Look, Part 2). Thus, data member `courseName` can be used only in member functions `setCourseName`, `getCourseName` and `displayMessage` of class `GradeBook` (or to "friends" of the class, if there were any).

Software Engineering Observation 3.1

Generally, data members should be declared `private` *and member functions should be declared* `public`*.*

Error-Prevention Tip 3.1

Make the data members of a class `private` *and the member functions of the class* `public`*. This facilitates debugging because problems with data manipulations are localized to either the class's member functions or the friends of the class.*

Common Programming Error 3.2

An attempt by a function, which is not a member of a particular class (or a friend of that class) to access a `private` *member of that class is a compilation error.*

The *default access* for class members is `private` so all members after the class header and before the first access specifier (if there are any) are `private`. The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing.

Declaring data members with access specifier `private` is known as **data hiding**. When a program creates a `GradeBook` object, data member `courseName` is *encapsulated* (hidden) in the object and can be accessed only by member functions of the object's class. In class `GradeBook`, member functions `setCourseName` and `getCourseName` manipulate the data member `courseName` directly.

Member Functions `setCourseName` *and* `getCourseName`

Member function `setCourseName` (lines 14–17) does not return any data when it completes its task, so its return type is `void`. The member function receives one parameter—`name`—which represents the course name that will be passed to it as an argument (as we'll see in line 50 of `main`). Line 16 assigns `name` to data member `courseName`. In this example, `setCourseName` does not *validate* the course name—i.e., the function does *not* check that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. Suppose, for instance, that a university can print student transcripts containing course names of only 25 characters or fewer. In this case, we might want class `GradeBook` to ensure that its data member `courseName` never contains more than 25 characters. We discuss validation in Section 3.8.

Member function `getCourseName` (lines 20–23) returns a particular `GradeBook` object's `courseName`. The member function has an empty parameter list, so it does *not* require additional data to perform its task. The function specifies that it returns a `string`. When a function that specifies a return type other than `void` is called and completes its task, the function uses a **`return` statement** (as in line 22) to *return a result* to its calling function. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls member function `getCourseName` on a `GradeBook` object, the statement expects to receive the `GradeBook`'s course name (in this case, a `string`, as specified by the function's return type).

If you have a function `square` that returns the square of its argument, the statement

```
result = square( 2 );
```

returns `4` from function `square` and assigns to variable `result` the value `4`. If you have a function `maximum` that returns the largest of three integer arguments, the statement

```
biggest = maximum( 27, 114, 51 );
```

returns `114` from function `maximum` and assigns to variable `biggest` the value `114`.

The statements in lines 16 and 22 each use variable `courseName` (line 34) even though it was *not* declared in any of the member functions. We can do this because `courseName` is a *data member* of the class.

Member Function `displayMessage`

Member function `displayMessage` (lines 26–32) does *not* return any data when it completes its task, so its return type is `void`. The function does *not* receive parameters, so its parameter list is empty. Lines 30–31 output a welcome message that includes the value of data member `courseName`. Line 30 calls member function `getCourseName` to obtain the value of `courseName`. Member function `displayMessage` could also access data member `courseName` directly, just as member functions `setCourseName` and `getCourseName` do. We explain shortly why it's preferable to call member function `getCourseName` to obtain the value of `courseName`.

Testing Class `GradeBook`

The `main` function (lines 38–54) creates one object of class `GradeBook` and uses each of its member functions. Line 41 creates a `GradeBook` object named `myGradeBook`. Lines 44–45 display the initial course name by calling the object's `getCourseName` member function. The first line of the output does not show a course name, because the object's `courseName` data member (i.e., a `string`) is initially empty—by default, the initial value of a `string` is the so-called **empty string**, i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

Line 48 prompts the user to enter a course name. Local `string` variable `nameOfCourse` (declared in line 40) is set to the course name entered by the user, which is obtained by the call to the `getline` function (line 49). Line 50 calls object `myGradeBook`'s `setCourseName` member function and supplies `nameOfCourse` as the function's argument. When the function is called, the argument's value is copied to parameter `name` (line 14) of member function `setCourseName`. Then the parameter's value is assigned to data member `courseName` (line 16). Line 52 skips a line; then line 53 calls object `myGradeBook`'s `displayMessage` member function to display the welcome message containing the course name.

Software Engineering with Set *and* Get *Functions*

A class's `private` data members can be manipulated only by member functions of that class (and by "friends" of the class). So a **client of an object**—that is, any statement that calls the object's member functions from *outside* the object—calls the class's `public` member functions to request the class's services for particular objects of the class. This is why the statements in function `main` call member functions `setCourseName`, `getCourseName` and `displayMessage` on a `GradeBook` object. Classes often provide `public` member functions to allow clients of the class to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` data members. These member function names need not begin with `set` or `get`, but this naming convention is common. In this example, the member function that *sets* the `courseName` data member is called `setCourseName`, and the member function that *gets* the value of the `courseName` data member is called `getCourseName`. *Set* functions are sometimes called **mutators** (because they mutate, or change, values), and *get* functions are also called **accessors** (because they access values).

Recall that declaring data members with access specifier `private` enforces data hiding. Providing `public` *set* and *get* functions allows clients of a class to access the hidden data, but only *indirectly*. The client knows that it's attempting to modify or obtain an object's data, but the client does *not* know *how* the object performs these operations. In some cases, a class may internally represent a piece of data one way, but expose that data to clients in a different way. For example, suppose a `Clock` class represents the time of day as a `private int` data member `time` that stores the number of seconds since midnight. However, when a client calls a `Clock` object's `getTime` member function, the object could return the time with hours, minutes and seconds in a `string` in the format `"HH:MM:SS"`. Similarly, suppose the `Clock` class provides a *set* function named `setTime` that takes a `string` parameter in the `"HH:MM:SS"` format. Using `string` capabilities presented in Chapter 18, the `setTime` function could convert this `string` to a number of seconds, which the function stores in its `private` data member. The *set* function could also check that the value it receives represents a valid time (e.g., `"12:30:45"` is valid but `"42:85:70"` is not). The *set* and *get* functions allow a client to interact with an object, but the object's `private` data remains safely encapsulated (i.e., hidden) in the object itself.

The *set* and *get* functions of a class also should be used by other member functions *within* the class to manipulate the class's `private` data, although these member functions *can* access the `private` data directly. In Fig. 3.5, member functions `setCourseName` and `getCourseName` are `public` member functions, so they're accessible to clients of the class, as well as to the class itself. Member function `displayMessage` calls member function `getCourseName` to obtain the value of data member `courseName` for display purposes, even though `displayMessage` can access `courseName` directly—accessing a data member via its *get* function creates a better, more robust class (i.e., a class that's easier to maintain and less likely to malfunction). If we decide to change the data member `courseName` in some way, the `displayMessage` definition will *not* require modification—only the bodies of the *get* and *set* functions that directly manipulate the data member will need to change. For example, suppose we want to represent the course name as two separate data members—`courseNumber` (e.g., `"CS101"`) and `courseTitle` (e.g., `"Introduction to C++ Programming"`). Member function `displayMessage` can still issue a single call to member function `getCourseName` to obtain the full course name to display as part of the welcome message. In this case, `getCourseName` would need to build and return a `string` containing the `courseNumber` followed by the `courseTitle`. Member function `displayMessage` could continue to display the complete course title "`CS101 Introduction to C++ Programming`." The benefits of calling a *set* function from another member function of the same class will become clear when we discuss validation in Section 3.8.

Good Programming Practice 3.1

Always try to localize *the effects of changes to a class's data members by accessing and manipulating the data members through their* get *and* set *functions.*

Software Engineering Observation 3.2

Write programs that are clear and easy to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified.

GradeBook's UML Class Diagram with a Data Member and* set *and* get *Functions

Figure 3.6 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.5. This diagram models `GradeBook`'s data member `courseName` as an attribute in the middle compartment. The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute `courseName` is `String`, which corresponds to `string` in C++. Data member `courseName` is `private` in C++, so the class diagram lists a minus sign (-) in front of the corresponding attribute's name. The minus sign in the UML is equivalent to the `private` access specifier in C++. Class `GradeBook` contains three `public` member functions, so the class diagram lists three operations in the third compartment. Operation `setCourseName` has a `String` parameter called `name`. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Member function `getCourseName` of class `GradeBook` has a `string` return type in C++, so the class diagram shows a `String` return type in the UML. Operations `setCourseName` and `displayMessage` do not return values (i.e., they return `void` in C++), so the UML class diagram does not specify a return type after the parentheses of these operations.

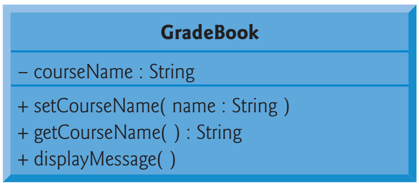


Fig. 3.6 | UML class diagram for class `GradeBook` with a private `courseName` attribute and public operations `setCourseName`, `getCourseName` and `displayMessage`.

3.5 Initializing Objects with Constructors

As mentioned in Section 3.4, when an object of class `GradeBook` (Fig. 3.5) is created, its data member `courseName` is initialized to the empty string by default. What if you want to provide a course name when you *create* a `GradeBook` object? Each class you declare can provide a **constructor** that can be used to initialize an object of the class when the object is created. A constructor is a special member function that must be defined with the same name as the class, so that the compiler can distinguish it from the class's other member functions. An important difference between constructors and other functions is that *constructors cannot return values*, so they *cannot* specify a return type (not even `void`). Normally, constructors are declared `public`.

C++ requires a constructor call for each object that's created, which helps ensure that each object is initialized properly before it's used in a program. The constructor call occurs *implicitly* when the object is created. If a class does not *explicitly* include a constructor, the compiler provides a **default constructor**—that is, a constructor with *no* parameters. For example, when line 41 of Fig. 3.5 creates a `GradeBook` object, the default constructor is called. The default constructor provided by the compiler creates a `GradeBook` object without giving any initial values to the object's fundamental type data members. [*Note:* For data members that are objects of other classes, the default constructor implicitly calls each data member's default constructor to ensure that the data member is initialized properly. This is why the `string` data member `courseName` (in Fig. 3.5) was initialized to the empty string—the default constructor for class `string` sets the `string`'s value to the empty string. You'll learn more about initializing data members that are objects of other classes in Section 10.3.]

In the example of Fig. 3.7, we specify a course name for a `GradeBook` object when the object is created (e.g., line 46). In this case, the argument `"CS101 Introduction to C++ Programming"` is passed to the `GradeBook` object's constructor (lines 14–17) and used to initialize the `courseName`. Figure 3.7 defines a modified `GradeBook` class containing a constructor with a `string` parameter that receives the initial course name.

```
1  // Fig. 3.7: fig03_07.cpp
2  // Instantiating multiple objects of the GradeBook class and using
3  // the GradeBook constructor to specify the course name
4  // when each GradeBook object is created.
5  #include <iostream>
6  #include <string> // program uses C++ standard string class
7  using namespace std;
8
9  // GradeBook class definition
10 class GradeBook
11 {
12 public:
13    // constructor initializes courseName with string supplied as argument
14    GradeBook( string name )
15    {
16       setCourseName( name ); // call set function to initialize courseName
17    } // end GradeBook constructor
18
19    // function to set the course name
20    void setCourseName( string name )
21    {
22       courseName = name; // store the course name in the object
23    } // end function setCourseName
24
25    // function to get the course name
26    string getCourseName()
27    {
28       return courseName; // return object's courseName
29    } // end function getCourseName
30
31    // display a welcome message to the GradeBook user
32    void displayMessage()
33    {
34       // call getCourseName to get the courseName
35       cout << "Welcome to the grade book for\n" << getCourseName()
36          << "!" << endl;
37    } // end function displayMessage
38 private:
39    string courseName; // course name for this GradeBook
40 }; // end class GradeBook
```

Fig. 3.7 | Instantiating multiple objects of the `GradeBook` class and using the `GradeBook` constructor to specify the course name when each `GradeBook` object is created. (Part 1 of 2.)

```
41
42 // function main begins program execution
43 int main()
44 {
45    // create two GradeBook objects
46    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
47    GradeBook gradeBook2( "CS102 Data Structures in C++" );
48
49    // display initial value of courseName for each GradeBook
50    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
51       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
52       << endl;
53 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Fig. 3.7 | Instantiating multiple objects of the `GradeBook` class and using the `GradeBook` constructor to specify the course name when each `GradeBook` object is created. (Part 2 of 2.)

Defining a Constructor

Lines 14–17 of Fig. 3.7 define a constructor for class `GradeBook`. Notice that the constructor has the *same* name as its class, `GradeBook`. A constructor specifies in its parameter list the data it requires to perform its task. When you create a new object, you place this data in the parentheses that follow the object name (as we did in lines 46–47). Line 14 indicates that class `GradeBook`'s constructor has a `string` parameter called `name`. Line 14 does not specify a return type, because constructors *cannot* return values (or even `void`).

Line 16 in the constructor's body passes the constructor's parameter `name` to member function `setCourseName` (lines 20–23), which simply assigns the value of its parameter to data member `courseName`. You might be wondering why we make the call to `setCourseName` in line 16—the constructor certainly could perform the assignment `courseName = name`. In Section 3.8, we modify `setCourseName` to perform validation (ensuring that, in this case, the `courseName` is 25 or fewer characters in length). At that point the benefits of calling `setCourseName` from the constructor will become clear. Both the constructor (line 14) and the `setCourseName` function (line 20) use a parameter called `name`. You can use the same parameter names in different functions because the parameters are *local* to each function; they do not interfere with one another.

Testing Class **`GradeBook`**

Lines 43–53 of Fig. 3.7 define the `main` function that tests class `GradeBook` and demonstrates initializing `GradeBook` objects using a constructor. Line 46 creates and initializes `GradeBook` object `gradeBook1`. When this line executes, the `GradeBook` constructor (lines 14–17) is called (implicitly by C++) with the argument `"CS101 Introduction to C++ Programming"` to initialize `gradeBook1`'s course name. Line 47 repeats this process for `GradeBook` object `gradeBook2`, this time passing the argument `"CS102 Data Structures in C++"` to initialize `gradeBook2`'s course name. Lines 50–51 use each object's `getCourseName` member function to obtain the course names and show that they were indeed initialized when the objects were created. The output confirms that each `GradeBook` object maintains its *own* copy of data member `courseName`.

Two Ways to Provide a Default Constructor for a Class

Any constructor that takes no arguments is called a default constructor. A class can get a default constructor in one of two ways:

1. The compiler implicitly creates a default constructor in a class that does not define a constructor. Such a constructor does *not* initialize the class's data members, but *does* call the default constructor for each data member that's an object of another class. An uninitialized variable typically contains a "garbage" value.
2. You explicitly define a constructor that takes no arguments. Such a default constructor will call the default constructor for each data member that's an object of another class and will perform additional initialization specified by you.

If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class. For each version of class `GradeBook` in Fig. 3.1, Fig. 3.3 and Fig. 3.5 the compiler *implicitly* defined a default constructor.

Error-Prevention Tip 3.2

Unless no initialization of your class's data members is necessary (almost never), provide a constructor to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.

Software Engineering Observation 3.3

Data members can be initialized in a constructor, or their values may be set later after the object is created. However, it's a good software engineering practice to ensure that an object is fully initialized before *the client code invokes the object's member functions. You should not rely on the client code to ensure that an object gets initialized properly.*

Adding the Constructor to Class `GradeBook`'s UML Class Diagram

The UML class diagram of Fig. 3.8 models the `GradeBook` class of Fig. 3.7, which has a constructor with a `name` parameter of type `string` (represented by type `String` in the UML). Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name. By convention, you list the class's constructor before other operations in the third compartment.

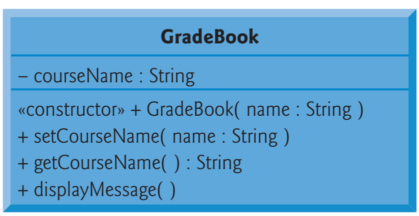


Fig. 3.8 | UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of UML type `String`.

3.6 Placing a Class in a Separate File for Reusability

One of the benefits of creating class definitions is that, when packaged properly, your classes can be *reused* by other programmers. For example, you can reuse C++ Standard Library type `string` in any C++ program by including the header `<string>` (and, as you'll see, by being able to link to the library's object code).

Programmers who wish to use our `GradeBook` class cannot simply include the file from Fig. 3.7 in another program. As you learned in Chapter 2, function `main` begins the execution of every program, and every program must have *exactly one* `main` function. If other programmers include the code from Fig. 3.7, they get extra "baggage"—our `main` function—and their programs will then have two `main` functions. Attempting to compile a program with two `main` functions produces an error when the compiler tries to compile the second `main` function it encounters. So, placing `main` in the same file with a class definition *prevents that class from being reused* by other programs. In this section, we demonstrate how to make class `GradeBook` reusable by *separating it into another file* from the `main` function.

Headers

Each of the previous examples in the chapter consists of a single `.cpp` file, also known as a **source-code file**, that contains a `GradeBook` class definition and a `main` function. When building an object-oriented C++ program, it's customary to define *reusable* source code (such as a class) in a file that by convention has a `.h` filename extension—known as a **header**. Programs use `#include` preprocessor directives to include headers and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `GradeBook`.

Our next example separates the code from Fig. 3.7 into two files—`GradeBook.h` (Fig. 3.9) and `fig03_10.cpp` (Fig. 3.10). As you look at the header in Fig. 3.9, notice that it contains only the `GradeBook` class definition (lines 8–38), the appropriate headers and a `using` directive. The `main` function that *uses* class `GradeBook` is defined in the source-code file `fig03_10.cpp` (Fig. 3.10) in lines 8–18. To help you prepare for the larger programs you'll encounter later in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a **driver program**). You'll soon learn how a source-code file with `main` can use the class definition found in a header to create objects of a class.

```
1  // Fig. 3.9: GradeBook.h
2  // GradeBook class definition in a separate file from main.
3  #include <iostream>
4  #include <string> // class GradeBook uses C++ standard string class
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    // constructor initializes courseName with string supplied as argument
12    GradeBook( string name )
13    {
```

Fig. 3.9 | `GradeBook` class definition in a separate file from `main`. (Part 1 of 2.)

```
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 3.9 | GradeBook class definition in a separate file from `main`. (Part 2 of 2.)

```
// Fig. 3.10: fig03_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
} // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Fig. 3.10 | Including class `GradeBook` from file `GradeBook.h` for use in `main`.

Including a Header That Contains a User-Defined Class

A header such as `GradeBook.h` (Fig. 3.9) cannot be used as a complete program, because it does not contain a `main` function. If you try to compile and link `GradeBook.h` by itself to create an executable application, Microsoft Visual C++ 2010 produces the linker error message:

```
error LNK2001: unresolved external symbol _mainCRTStartup
```

To compile and link with GNU C++ on Linux, you must first include the header in a `.cpp` source-code file, then GNU C++ produces a linker error message containing:

```
undefined reference to 'main'
```

This error indicates that the linker could not locate the program's `main` function. To test class `GradeBook` (defined in Fig. 3.9), you must write a separate source-code file containing a `main` function (such as Fig. 3.10) that instantiates and uses objects of the class.

The compiler doesn't know what a `GradeBook` is because it's a user-defined type. In fact, the compiler doesn't even know the classes in the C++ Standard Library. To help it understand how to use a class, we must explicitly provide the compiler with the class's definition—that's why, for example, to use type `string`, a program must include the `<string>` header. This enables the compiler to determine the amount of memory that it must reserve for each `string` object and ensure that a program calls a `string`'s member functions correctly.

To create `GradeBook` objects `gradeBook1` and `gradeBook2` in lines 11–12 of Fig. 3.10, the compiler must know the *size* of a `GradeBook` object. While objects conceptually contain data members and member functions, C++ objects actually contain *only* data. The compiler creates only *one* copy of the class's member functions and *shares* that copy among all the class's objects. Each object, of course, needs its own copy of the class's data members, because their contents can vary among objects (such as two different `BankAccount` objects having two different balances). The member-function code, however, is *not modifiable*, so it can be shared among all objects of the class. Therefore, the size of an object depends on the amount of memory required to store the class's data members. By including `GradeBook.h` in line 4, we give the compiler access to the information it needs (Fig. 3.9, line 37) to determine the size of a `GradeBook` object and to determine whether objects of the class are used correctly (in lines 11–12 and 15–16 of Fig. 3.10).

Line 4 instructs the C++ preprocessor to replace the directive with a copy of the contents of `GradeBook.h` (i.e., the `GradeBook` class definition) *before* the program is compiled. When the source-code file `fig03_10.cpp` is compiled, it now contains the `GradeBook` class definition (because of the `#include`), and the compiler is able to determine how to create `GradeBook` objects and see that their member functions are called correctly. Now that the class definition is in a header (without a `main` function), we can include that header in *any* program that needs to reuse our `GradeBook` class.

How Headers Are Located

Notice that the name of the `GradeBook.h` header in line 4 of Fig. 3.10 is enclosed in quotes (`" "`) rather than angle brackets (`< >`). Normally, a program's source-code files and user-defined headers are placed in the *same* directory. When the preprocessor encounters a header name in quotes, it attempts to locate the header in the same directory as the file in which the `#include` directive appears. If the preprocessor cannot find the header in that directory, it searches for it in the same location(s) as the C++ Standard Library headers. When the preprocessor encounters a header name in angle brackets (e.g., <iostream>), it assumes that the header is part of the C++ Standard Library and does *not* look in the directory of the program that's being preprocessed.

Error-Prevention Tip 3.3

To ensure that the preprocessor can locate headers correctly, #include preprocessor directives should place user-defined headers names in quotes (e.g., "GradeBook.h") and place C++ Standard Library headers names in angle brackets (e.g., <iostream>).

Additional Software Engineering Issues

Now that class GradeBook is defined in a header, the class is *reusable*. Unfortunately, placing a class definition in a header as in Fig. 3.9 still *reveals the entire implementation of the class to the class's clients*—GradeBook.h is simply a text file that anyone can open and read. Conventional software engineering wisdom says that to use an object of a class, the client code needs to know only what member functions to call, what arguments to provide to each member function and what return type to expect from each member function. *The client code does not need to know how those functions are implemented.*

If client code *does* know how a class is implemented, the programmer might write client code based on the class's implementation details. Ideally, if that implementation changes, the class's clients should not have to change. *Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.*

In Section 3.7, we show how to break up the GradeBook class into two files so that

1. the class is *reusable*,
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect, and
3. the clients do *not* know how the class's member functions are implemented.

3.7 Separating Interface from Implementation

In the preceding section, we showed how to promote software reusability by separating a class definition from the client code (e.g., function main) that uses the class. We now introduce another fundamental principle of good software engineering—**separating interface from implementation**.

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands. The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.

Similarly, the **interface of a class** describes *what* services a class's clients can use and how to *request* those services, but not *how* the class carries out the services. A class's `public` interface consists of the class's `public` member functions (also known as the class's **`public` services**). For example, class `GradeBook`'s interface (Fig. 3.9) contains a constructor and member functions `setCourseName`, `getCourseName` and `displayMessage`. `GradeBook`'s clients (e.g., `main` in Fig. 3.10) *use* these functions to request the class's services. As you'll soon see, you can specify a class's interface by writing a class definition that lists *only* the member-function names, return types and parameter types.

Separating the Interface from the Implementation

In our prior examples, each class definition contained the complete definitions of the class's `public` member functions and the declarations of its `private` data members. However, it's better software engineering to define member functions *outside* the class definition, so that their implementation details can be *hidden* from the client code. This practice *ensures* that you do not write client code that depends on the class's implementation details. If you were to do so, the client code would be more likely to "break" if the class's implementation changed. Given that one class could have many clients, such a change could cause wide-ranging problems in a software system.

The program of Figs. 3.11–3.13 separates class `GradeBook`'s interface from its implementation by splitting the class definition of Fig. 3.9 into two files—the header `GradeBook.h` (Fig. 3.11) in which class `GradeBook` is defined, and the source-code file `GradeBook.cpp` (Fig. 3.12) in which `GradeBook`'s member functions are defined. By convention, member-function definitions are placed in a source-code file of the same base name (e.g., `GradeBook`) as the class's header but with a `.cpp` filename extension. The source-code file `fig03_13.cpp` (Fig. 3.13) defines function `main` (the client code). The code and output of Fig. 3.13 are identical to that of Fig. 3.10. Figure 3.14 shows how this three-file program is compiled from the perspectives of the `GradeBook` class programmer and the client-code programmer—we'll explain this figure in detail.

`GradeBook.h`*: Defining a Class's Interface with Function Prototypes*

Header `GradeBook.h` (Fig. 3.11) contains another version of `GradeBook`'s class definition (lines 9–18). This version is similar to the one in Fig. 3.9, but the function definitions in Fig. 3.9 are replaced here with **function prototypes** (lines 12–15) that *describe the class's* `public` *interface without revealing the class's member-function implementations.* A function prototype is a *declaration* of a function that tells the compiler the function's name, its return type and the types of its parameters. Also, the header still specifies the class's `private` data member (line 17) as well. Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class. Including the header `GradeBook.h` in the client code (line 5 of Fig. 3.13) provides the compiler with the information it needs to ensure that the client code calls the member functions of class `GradeBook` correctly.

The function prototype in line 12 (Fig. 3.11) indicates that the constructor requires one `string` parameter. Recall that constructors don't have return types, so no return type appears in the function prototype. Member function `setCourseName`'s function prototype indicates that `setCourseName` requires a `string` parameter and does not return a value (i.e., its return type is `void`). Member function `getCourseName`'s function prototype indicates that the function does not require parameters and returns a `string`. Finally, member

```
1 // Fig. 3.11: GradeBook.h
2 // GradeBook class definition. This file presents GradeBook's public
3 // interface without revealing the implementations of GradeBook's member
4 // functions, which are defined in GradeBook.cpp.
5 #include <string> // class GradeBook uses C++ standard string class
6 using namespace std;
7
8 // GradeBook class definition
9 class GradeBook
10 {
11 public:
12    GradeBook( string ); // constructor that initializes courseName
13    void setCourseName( string ); // function that sets the course name
14    string getCourseName(); // function that gets the course name
15    void displayMessage(); // function that displays a welcome message
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

Fig. 3.11 | GradeBook class definition containing function prototypes that specify the interface of the class.

function `displayMessage`'s function prototype (line 15) specifies that `displayMessage` does not require parameters and does not return a value. These function prototypes are the same as the corresponding function headers in Fig. 3.9, except that the parameter names (which are *optional* in prototypes) are not included and each function prototype *must* end with a semicolon.

Good Programming Practice 3.2

Although parameter names in function prototypes are optional (they're ignored by the compiler), many programmers use these names for documentation purposes.

Error-Prevention Tip 3.4

Parameter names in a function prototype (which, again, are ignored by the compiler) can be misleading if the names used do not match those used in the function definition. For this reason, many programmers create function prototypes by copying the first line of the corresponding function definitions (when the source code for the functions is available), then appending a semicolon to the end of each prototype.

`GradeBook.cpp`: Defining Member Functions in a Separate Source-Code File

Source-code file `GradeBook.cpp` (Fig. 3.12) *defines* class `GradeBook`'s member functions, which were *declared* in lines 12–15 of Fig. 3.11. The definitions appear in lines 9–32 and are nearly identical to the member-function definitions in lines 12–35 of Fig. 3.9.

Each member-function name in the function headers (lines 9, 15, 21 and 27) is preceded by the class name and `::`, which is known as the **binary scope resolution operator**. This "ties" each member function to the (now separate) `GradeBook` class definition (Fig. 3.11), which declares the class's member functions and data members. Without "`GradeBook::`" preceding each function name, these functions would *not* be recognized by the compiler as member functions of class `GradeBook`—the compiler would consider them

```
1  // Fig. 3.12: GradeBook.cpp
2  // GradeBook member-function definitions. This file contains
3  // implementations of the member functions prototyped in GradeBook.h.
4  #include <iostream>
5  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // constructor initializes courseName with string supplied as argument
9  GradeBook::GradeBook( string name )
10 {
11    setCourseName( name ); // call set function to initialize courseName
12 } // end GradeBook constructor
13
14 // function to set the course name
15 void GradeBook::setCourseName( string name )
16 {
17    courseName = name; // store the course name in the object
18 } // end function setCourseName
19
20 // function to get the course name
21 string GradeBook::getCourseName()
22 {
23    return courseName; // return object's courseName
24 } // end function getCourseName
25
26 // display a welcome message to the GradeBook user
27 void GradeBook::displayMessage()
28 {
29    // call getCourseName to get the courseName
30    cout << "Welcome to the grade book for\n" << getCourseName()
31       << "!" << endl;
32 } // end function displayMessage
```

Fig. 3.12 | GradeBook member-function definitions represent the implementation of class GradeBook.

"free" or "loose" functions, like `main`. These are also called *global functions*. Such functions cannot access `GradeBook`'s `private` data or call the class's member functions, without specifying an object. So, the compiler would *not* be able to compile these functions. For example, lines 17 and 23 that access variable `courseName` would cause compilation errors because `courseName` is not declared as a local variable in each function—the compiler would not know that `courseName` is already declared as a data member of class `GradeBook`.

Common Programming Error 3.3

When defining a class's member functions outside *that class, omitting the class name and binary scope resolution operator (`::`) preceding the function names causes errors.*

To indicate that the member functions in `GradeBook.cpp` are part of class `GradeBook`, we must first include the `GradeBook.h` header (line 5 of Fig. 3.12). This allows us to access the class name `GradeBook` in the `GradeBook.cpp` file. When compiling `GradeBook.cpp`, the compiler uses the information in `GradeBook.h` to ensure that

1. the first line of each member function (lines 9, 15, 21 and 27) matches its prototype in the `GradeBook.h` file—for example, the compiler ensures that `getCourseName` accepts no parameters and returns a `string`, and that
2. each member function knows about the class's data members and other member functions—for example, lines 17 and 23 can access variable `courseName` because it's declared in `GradeBook.h` as a data member of class `GradeBook`, and lines 11 and 30 can call functions `setCourseName` and `getCourseName`, respectively, because each is declared as a member function of the class in `GradeBook.h` (and because these calls conform with the corresponding prototypes).

Testing Class **GradeBook**

Figure 3.13 performs the same `GradeBook` object manipulations as Fig. 3.10. Separating `GradeBook`'s interface from the implementation of its member functions does *not* affect the way that this client code uses the class. It affects only how the program is compiled and linked, which we discuss in detail shortly.

```
1  // Fig. 3.13: fig03_13.cpp
2  // GradeBook class demonstration after separating
3  // its interface from its implementation.
4  #include <iostream>
5  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // function main begins program execution
9  int main()
10 {
11    // create two GradeBook objects
12    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
13    GradeBook gradeBook2( "CS102 Data Structures in C++" );
14
15    // display initial value of courseName for each GradeBook
16    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
17       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
18       << endl;
19 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Fig. 3.13 | `GradeBook` class demonstration after separating its interface from its implementation.

As in Fig. 3.10, line 5 of Fig. 3.13 includes the `GradeBook.h` header so that the compiler can ensure that `GradeBook` objects are created and manipulated correctly in the client code. Before executing this program, the source-code files in Fig. 3.12 and Fig. 3.13 must both be compiled, then linked together—that is, the member-function calls in the client code need to be tied to the implementations of the class's member functions—a job performed by the linker.

The Compilation and Linking Process

The diagram in Fig. 3.14 shows the compilation and linking process that results in an executable `GradeBook` application that can be used by instructors. Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class. So, the diagram shows what's required by both the class-implementation programmer and the client-code programmer. The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the `GradeBook` application user, respectively. [*Note:* Figure 3.14 is *not* a UML diagram.]

A class-implementation programmer responsible for creating a reusable `GradeBook` class creates the header `GradeBook.h` and the source-code file `GradeBook.cpp` that

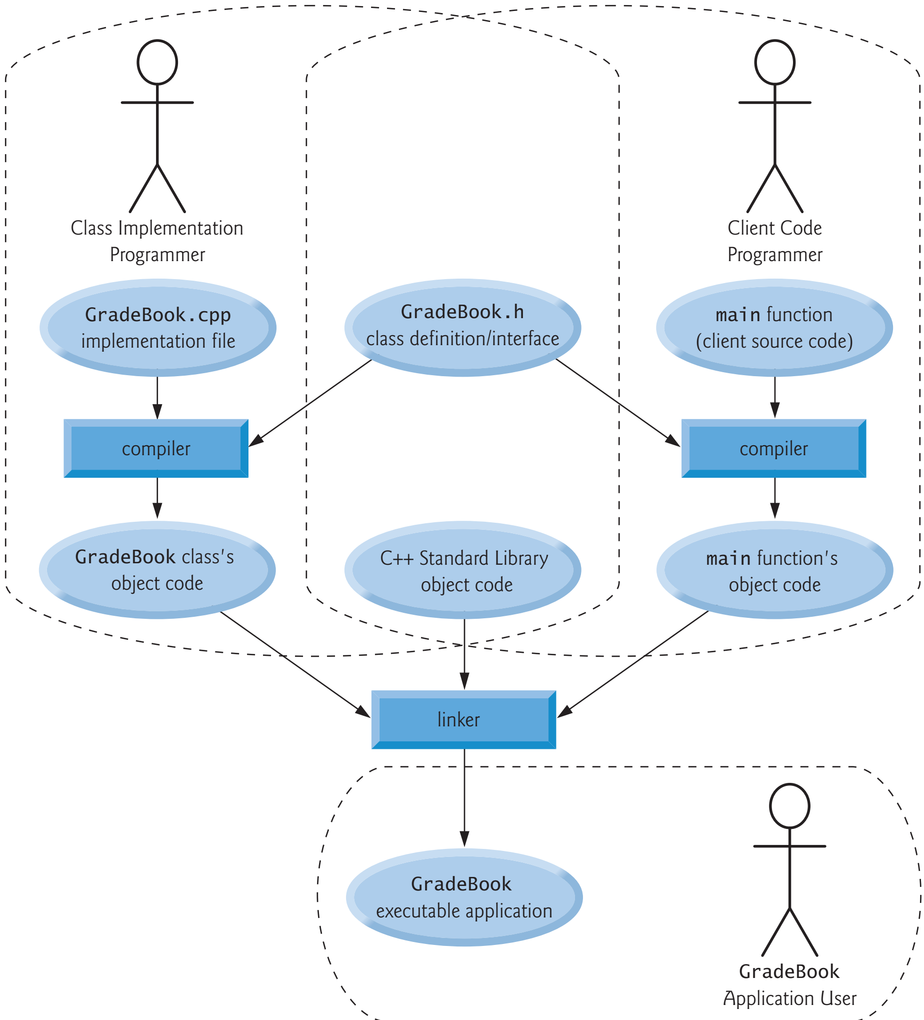


Fig. 3.14 | Compilation and linking process that produces an executable application.

#includes the header, then compiles the source-code file to create GradeBook's object code. To hide the class's member-function implementation details, the class-implementation programmer would provide the client-code programmer with the header GradeBook.h (which specifies the class's interface and data members) and the GradeBook object code (i.e., the machine-language instructions that represent GradeBook's member functions). The client-code programmer is *not* given GradeBook.cpp, so the client remains unaware of how GradeBook's member functions are implemented.

The client code needs to know only GradeBook's interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the GradeBook.h header, the client-code programmer must have access to this file and must #include it in the client's source-code file. When the client code is compiled, the compiler uses the class definition in GradeBook.h to ensure that the main function creates and manipulates objects of class GradeBook correctly.

To create the executable GradeBook application, the last step is to link

1. the object code for the main function (i.e., the client code),
2. the object code for class GradeBook's member-function implementations and
3. the C++ Standard Library object code for the C++ classes (e.g., string) used by the class-implementation programmer and the client-code programmer.

The linker's output is the *executable* GradeBook application that instructors can use to manage their students' grades. Compilers and IDEs typically invoke the linker for you after compiling your code.

For further information on compiling multiple-source-file programs, see your compiler's documentation. We provide links to various C++ compilers in our C++ Resource Center at www.deitel.com/cplusplus/.

3.8 Validating Data with *set* Functions

In Section 3.4, we introduced *set* functions for allowing clients of a class to modify the value of a private data member. In Fig. 3.5, class GradeBook defines member function setCourseName to simply assign a value received in its parameter name to data member courseName. This member function does not ensure that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. As we stated earlier, suppose that a university can print student transcripts containing course names of only 25 characters or less. If the university uses a system containing GradeBook objects to generate the transcripts, we might want class GradeBook to ensure that its data member courseName never contains more than 25 characters. The program of Figs. 3.15–3.17 enhances class GradeBook's member function setCourseName to perform this **validation** (also known as **validity checking**).

GradeBook Class Definition

Notice that GradeBook's class definition (Fig. 3.15)—and hence, its interface—is identical to that of Fig. 3.11. Since the interface remains unchanged, clients of this class need not be changed when the definition of member function setCourseName is modified. This enables clients to take advantage of the improved GradeBook class simply by linking the client code to the updated GradeBook's object code.

```cpp
// Fig. 3.15: GradeBook.h
// GradeBook class definition presents the public interface of
// the class. Member-function definitions appear in GradeBook.cpp.
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor that initializes a GradeBook object
   void setCourseName( string ); // function that sets the course name
   string getCourseName(); // function that gets the course name
   void displayMessage(); // function that displays a welcome message
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 3.15 | GradeBook class definition.

Validating the Course Name with **`GradeBook`** *Member Function* **`setCourseName`**

The enhancement to class `GradeBook` is in the definition of `setCourseName` (Fig. 3.16, lines 16–29). The `if` statement in lines 18–19 determines whether parameter `name` contains a valid course name (i.e., a `string` of 25 or fewer characters). If the course name is valid, line 19 stores it in data member `courseName`. Note the expression `name.length()` in line 18. This is a member-function call just like `myGradeBook.displayMessage()`. The C++ Standard Library's `string` class defines a member function **`length`** that returns the number of characters in a `string` object. Parameter `name` is a `string` object, so the call `name.length()` returns the number of characters in `name`. If this value is less than or equal to `25`, `name` is valid and line 19 executes.

```cpp
1  // Fig. 3.16: GradeBook.cpp
2  // Implementations of the GradeBook member-function definitions.
3  // The setCourseName function performs validation.
4  #include <iostream>
5  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // constructor initializes courseName with string supplied as argument
9  GradeBook::GradeBook( string name )
10 {
11    setCourseName( name ); // validate and store courseName
12 } // end GradeBook constructor
13
14 // function that sets the course name;
15 // ensures that the course name has at most 25 characters
16 void GradeBook::setCourseName( string name )
17 {
```

Fig. 3.16 | Member-function definitions for class `GradeBook` with a *set* function that validates the length of data member `courseName`. (Part 1 of 2.)

```
18    if ( name.length() <= 25 ) // if name has 25 or fewer characters
19       courseName = name; // store the course name in the object
20
21    if ( name.length() > 25 ) // if name has more than 25 characters
22    {
23       // set courseName to first 25 characters of parameter name
24       courseName = name.substr( 0, 25 ); // start at 0, length of 25
25
26       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
27          << "Limiting courseName to first 25 characters.\n" << endl;
28    } // end if
29 } // end function setCourseName
30
31 // function to get the course name
32 string GradeBook::getCourseName()
33 {
34    return courseName; // return object's courseName
35 } // end function getCourseName
36
37 // display a welcome message to the GradeBook user
38 void GradeBook::displayMessage()
39 {
40    // call getCourseName to get the courseName
41    cout << "Welcome to the grade book for\n" << getCourseName()
42       << "!" << endl;
43 } // end function displayMessage
```

Fig. 3.16 | Member-function definitions for class `GradeBook` with a *set* function that validates the length of data member `courseName`. (Part 2 of 2.)

The `if` statement in lines 21–28 handles the case in which `setCourseName` receives an invalid course name (i.e., a name that is more than 25 characters long). Even if parameter `name` is too long, we still want to leave the `GradeBook` object in a **consistent state**—that is, a state in which the object's data member `courseName` contains a valid value (i.e., a `string` of 25 characters or less). Thus, we truncate the specified course name and assign the first 25 characters of `name` to the `courseName` data member (unfortunately, this could truncate the course name awkwardly). Standard class `string` provides member function **`substr`** (short for "substring") that returns a new `string` object created by copying part of an existing `string` object. The call in line 24 (i.e., `name.substr( 0, 25 )`) passes two integers (`0` and `25`) to `name`'s member function `substr`. These arguments indicate the portion of the string `name` that `substr` should return. The first argument specifies the starting position in the original `string` from which characters are copied—the first character in every string is considered to be at position 0. The second argument specifies the number of characters to copy. Therefore, the call in line 24 returns a 25-character substring of `name` starting at position 0 (i.e., the first 25 characters in `name`). For example, if `name` holds the value `"CS101 Introduction to Programming in C++"`, `substr` returns `"CS101 Introduction to Pro"`. After the call to `substr`, line 24 assigns the substring returned by `substr` to data member `courseName`. In this way, `setCourseName` ensures that `courseName` is always assigned a string containing 25 or fewer characters. If the member function has to truncate the course name to make it valid, lines 26–27 display a warning message.

The `if` statement in lines 21–28 contains two body statements—one to set the `courseName` to the first 25 characters of parameter `name` and one to print an accompanying message to the user. Both statements should execute when `name` is too long, so we place them in a pair of braces, `{ }`. Recall from Chapter 2 that this creates a block. You'll learn more about placing multiple statements in a control statement's body in Chapter 4.

The statement in lines 26–27 could also appear without a stream insertion operator at the start of the second line of the statement, as in:

```
cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
   "Limiting courseName to first 25 characters.\n" << endl;
```

The C++ compiler combines adjacent string literals, even if they appear on separate lines of a program. Thus, in the statement above, the C++ compiler would combine the string literals `"\" exceeds maximum length (25).\n"` and `"Limiting courseName to first 25 characters.\n"` into a single string literal that produces output identical to that of lines 26–27 in Fig. 3.16. This behavior allows you to print lengthy strings by breaking them across lines in your program without including additional stream insertion operations.

Testing Class GradeBook

Figure 3.17 demonstrates the modified version of class `GradeBook` (Figs. 3.15–3.16) featuring validation. Line 12 creates a `GradeBook` object named `gradeBook1`. Recall that the `GradeBook` constructor calls `setCourseName` to initialize data member `courseName`. In previous versions of the class, the benefit of calling `setCourseName` in the constructor was not evident. Now, however, *the constructor takes advantage of the validation* provided by `setCourseName`. The constructor simply calls `setCourseName`, *rather than duplicating* its validation code. When line 12 of Fig. 3.17 passes an initial course name of `"CS101 Introduction to Programming in C++"` to the `GradeBook` constructor, the constructor passes this value to `setCourseName`, where the actual initialization occurs. Because this course name contains more than 25 characters, the body of the second `if` statement executes, causing `courseName` to be initialized to the truncated 25-character course name `"CS101 Introduction to Pro"` (the truncated part is highlighted in red in line 12). The output in Fig. 3.17 contains the warning message output by lines 26–27 of Fig. 3.16 in member function `setCourseName`. Line 13 creates another `GradeBook` object called `gradeBook2`—the valid course name passed to the constructor is exactly 25 characters.

Lines 16–19 of Fig. 3.17 display the truncated course name for `gradeBook1` (we highlight this in blue in the program output) and the course name for `gradeBook2`. Line 22 calls `gradeBook1`'s `setCourseName` member function directly, to change the course name in the `GradeBook` object to a shorter name that does not need to be truncated. Then, lines 25–28 output the course names for the `GradeBook` objects again.

```
1  // Fig. 3.17: fig03_17.cpp
2  // Create and manipulate a GradeBook object; illustrate validation.
3  #include <iostream>
4  #include "GradeBook.h" // include definition of class GradeBook
5  using namespace std;
```

Fig. 3.17 | Creating and manipulating a `GradeBook` object in which the course name is limited to 25 characters in length. (Part 1 of 2.)

```

// function main begins program execution
int main()
{
   // create two GradeBook objects;
   // initial course name of gradeBook1 is too long
   GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
   GradeBook gradeBook2( "CS102 C++ Data Structures" );

   // display each GradeBook's courseName
   cout << "gradeBook1's initial course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's initial course name is: "
      << gradeBook2.getCourseName() << endl;

   // modify myGradeBook's courseName (with a valid-length string)
   gradeBook1.setCourseName( "CS101 C++ Programming" );

   // display each GradeBook's courseName
   cout << "\ngradeBook1's course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's course name is: "
      << gradeBook2.getCourseName() << endl;
} // end main
```

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures
```

Fig. 3.17 | Creating and manipulating a `GradeBook` object in which the course name is limited to 25 characters in length. (Part 2 of 2.)

Additional Notes on Set *Functions*

A `public` *set* function such as `setCourseName` should carefully scrutinize any attempt to modify the value of a data member (e.g., `courseName`) to ensure that the new value is appropriate for that data item. For example, an attempt to *set* the day of the month to 37 should be rejected, an attempt to *set* a person's weight to zero or a negative value should be rejected, an attempt to *set* a grade on an exam to 185 (when the proper range is zero to 100) should be rejected, and so on

Software Engineering Observation 3.4

Making data members `private` and controlling access, especially write access, to those data members through `public` member functions helps ensure data integrity.

Error-Prevention Tip 3.5

The benefits of data integrity are not automatic simply because data members are made `private`—you must provide appropriate validity checking and report the errors.

A class's *set* functions can return values to the class's clients indicating that attempts were made to assign invalid data to objects of the class. A client can test the return value of a *set* function to determine whether the attempt to modify the object was successful and to take appropriate action. In C++, clients of objects can be notified of invalid values via the *exception-handling mechanism*, which we begin discussing in Chapter 7 and present in-depth in Chapter 16. To keep the program of Figs. 3.15–3.17 simple at this early point in the book, `setCourseName` in Fig. 3.16 just prints an appropriate message.

3.9 Wrap-Up

In this chapter, you created user-defined classes, and created and used objects of those classes. We declared data members of a class to maintain data for each object of the class. We also defined member functions that operate on that data. You learned how to call an object's member functions to request the services the object provides and how to pass data to those member functions as arguments. We discussed the difference between a local variable of a member function and a data member of a class. We also showed how to use a constructor to specify initial values for an object's data members. You learned how to separate the interface of a class from its implementation to promote good software engineering. We presented a diagram that shows the files that class-implementation programmers and client-code programmers need to compile the code they write. We demonstrated how *set* functions can be used to validate an object's data and ensure that objects are maintained in a consistent state. UML class diagrams were used to model classes and their constructors, member functions and data members. In the next chapter, we begin our introduction to control statements, which specify the order in which a function's actions are performed.

Summary

Section 3.2 Defining a Class with a Member Function

- A class definition (p. 66) contains the data members and member functions that define the class's attributes and behaviors, respectively.
- A class definition begins with the keyword `class` followed immediately by the class name.
- By convention, the name of a user-defined class (p. 67) begins with a capital letter and, for readability, each subsequent word in the class name begins with a capital letter.
- Every class's body (p. 66) is enclosed in a pair of braces (`{` and `}`) and ends with a semicolon.
- Member functions that appear after access specifier `public` (p. 66) can be called by other functions in a program and by member functions of other classes.
- Access specifiers are always followed by a colon (`:`).
- Keyword `void` (p. 67) is a special return type which indicates that a function will perform a task but will not return any data to its calling function when it completes its task.
- By convention, function names (p. 67) begin with a lowercase first letter and all subsequent words in the name begin with a capital letter.
- An empty set of parentheses after a function name indicates that the function does not require additional data to perform its task.
- Every function's body is delimited by left and right braces (`{` and `}`).
- Typically, you cannot call a member function until you create an object of its class.
- Each new class you create becomes a new type in C++.

- In the UML, each class is modeled in a class diagram (p. 68) as a rectangle with three compartments, which (top to bottom) contain the class's name, attributes and operations, respectively.
- The UML models operations as the operation name followed by parentheses. A plus sign (+) preceding the name indicates a `public` operation (i.e., a `public` member function in C++).

Section 3.3 Defining a Member Function with a Parameter

- A member function can require one or more parameters (p. 68) that represent additional data it needs to perform its task. A function call supplies an argument (p. 68) for each function parameter.
- A member function is called by following the object name with a dot (`.`) operator (p. 67), the function name and a set of parentheses containing the function's arguments.
- A variable of C++ Standard Library class `string` (p. 69) represents a string of characters. This class is defined in header `<string>`, and the name `string` belongs to namespace `std`.
- Function `getline` (from header `<string>`, p. 70) reads characters from its first argument until a newline character is encountered, then places the characters (not including the newline) in the `string` variable specified as its second argument. The newline character is discarded.
- A parameter list (p. 70) may contain any number of parameters, including none at all (represented by empty parentheses) to indicate that a function does not require any parameters.
- The number of arguments in a function call must match the number of parameters in the parameter list of the called member function's header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header.
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
- The UML has its own data types. Not all the UML data types have the same names as the corresponding C++ types. The UML type `String` corresponds to the C++ type `string`.

Section 3.4 Data Members, set *Functions and* get *Functions*

- Variables declared in a function's body are local variables (p. 71) and can be used only from the point of their declaration in the function to the immediately following closing right brace (`}`).
- A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it's declared.
- Data members (p. 71) normally are `private` (p. 73). Variables or functions declared `private` are accessible only to member functions of the class in which they're declared, or to friends of the class.
- When a program creates (instantiates) an object, its `private` data members are encapsulated (hidden, p. 74) in the object and can be accessed only by member functions of the object's class.
- When a function that specifies a return type other than `void` is called and completes its task, the function returns a result to its calling function.
- By default, the initial value of a `string` is the empty string (p. 75)—i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.
- A class often provides `public` member functions to allow the class's clients to *set* or *get* (p. 75) `private` data members. The names of these member functions normally begin with *set* or *get*.
- *Set* and *get* functions allow clients of a class to indirectly access the hidden data. The client does not know how the object performs these operations.
- A class's *set* and *get* functions should be used by other member functions of the class to manipulate the class's `private` data. If the class's data representation is changed, member functions that access the data only via the *set* and *get* functions will not require modification.
- A `public` *set* function should carefully scrutinize any attempt to modify the value of a data member to ensure that the new value is appropriate for that data item.

- The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. Private attributes are preceded by a minus sign (–) in the UML.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams do not specify return types for operations that do not return values.

Section 3.5 Initializing Objects with Constructors

- Each class should provide a constructor (p. 77) to initialize an object of the class when the object is created. A constructor must be defined with the same name as the class.
- A difference between constructors and functions is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`.
- C++ requires a constructor call at the time each object is created, which helps ensure that every object is initialized before it's used in a program.
- A constructor with no parameters is a default constructor (p. 77). If you do not provide a constructor, the compiler provides a default constructor. You can also define a default constructor explicitly. If you define a constructor for a class, C++ will not create a default constructor.
- The UML models constructors as operations in a class diagram's third compartment with the word "constructor" between guillemets (« and ») before the constructor's name.

Section 3.6 Placing a Class in a Separate File for Reusability

- Class definitions, when packaged properly, can be reused by programmers worldwide.
- It's customary to define a class in a header (p. 81) that has a `.h` filename extension.

Section 3.7 Separating Interface from Implementation

- If the class's implementation changes, the class's clients should not be required to change.
- Interfaces define and standardize the ways in which things such as people and systems interact.
- A class's `public` interface (p. 85) describes the `public` member functions that are made available to the class's clients. The interface describes *what* services (p. 85) clients can use and how to *request* those services, but does not specify *how* the class carries out the services.
- Separating interface from implementation (p. 84) makes programs easier to modify. Changes in the class's implementation do not affect the client as long as the class's interface remains unchanged.
- A function prototype (p. 85) contains a function's name, its return type and the number, types and order of the parameters the function expects to receive.
- Once a class is defined and its member functions are declared (via function prototypes), the member functions should be defined in a separate source-code file.
- For each member function defined outside of its corresponding class definition, the function name must be preceded by the class name and the binary scope resolution operator (`::`, p. 86).

Section 3.8 Validating Data with set *Functions*

- Class `string`'s `length` member function (p. 91) returns the number of characters in a `string`.
- Class `string`'s member function `substr` (p. 92) returns a new `string` containing a copy of part of an existing `string`. The first argument specifies the starting position in the original `string`. The second specifies the number of characters to copy.

Self-Review Exercises

3.1 Fill in the blanks in each of the following:

a) Every class definition contains the keyword public followed immediately by the class's name.

b) A class definition is typically stored in a file with the .h filename extension.

c) Each parameter in a function header specifies both a(n) type and a(n) identifier.
d) When each object of a class maintains its own copy of an attribute, the variable that represents the attribute is also known as a(n) ________.
e) Keyword `public` is a(n) ________.
f) Return type ________ indicates that a function will perform a task but will not return any information when it completes its task.
g) Function ________ from the `<string>` library reads characters until a newline character is encountered, then copies those characters into the specified `string`.
h) When a member function is defined outside the class definition, the function header must include the class name and the ________, followed by the function name to "tie" the member function to the class definition.
i) The source-code file and any other files that use a class can include the class's header via a(n) ________ preprocessor directive.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) By convention, function names begin with a capital letter and all subsequent words in the name begin with a capital letter.
b) Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.
c) Data members or member functions declared with access specifier `private` are accessible to member functions of the class in which they're declared.
d) Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.
e) Every function's body is delimited by left and right braces (`{` and `}`).
f) Any source-code file that contains `int main()` can be used to execute a program.
g) The types of arguments in a function call must be consistent with the types of the corresponding parameters in the function prototype's parameter list.

3.3 What is the difference between a local variable and a data member?

3.4 Explain the purpose of a function parameter. What's the difference between a parameter and an argument?

Answers to Self-Review Exercises

3.1 a) `class`. b) `.h`. c) type, name. d) data member. e) access specifier. f) `void`. g) `getline`. h) binary scope resolution operator (`::`). i) `#include`.

3.2 a) False. Function names begin with a lowercase letter and all subsequent words in the name begin with a capital letter. b) True. c) True. d) False. Such variables are local variables and can be used only in the member function in which they're declared. e) True. f) True. g) True.

3.3 A local variable is declared in the body of a function and can be used only from its declaration to the closing brace of the block in which it's declared. A data member is declared in a class, but not in the body of any of the class's member functions. Every object of a class has a separate copy of the class's data members. Data members are accessible to all member functions of the class.

3.4 A parameter represents additional information that a function requires to perform its task. Each parameter required by a function is specified in the function header. An argument is the value supplied in the function call. When the function is called, the argument value is passed into the function parameter so that the function can perform its task.

Exercises

3.5 ***(Function Prototypes and Definitions)*** Explain the difference between a function prototype and a function definition.

3.6 *(Default Constructor)* What's a default constructor? How are an object's data members initialized if a class has only an implicitly defined default constructor?

3.7 *(Data Members)* Explain the purpose of a data member.

3.8 *(Header and Source-Code Files)* What's a header? What's a source-code file? Discuss the purpose of each.

3.9 *(Using a Class Without a `using` Directive)* Explain how a program could use class `string` without inserting a `using` directive.

3.10 *(Set and Get Functions)* Explain why a class might provide a *set* function and a *get* function for a data member.

3.11 *(Modifying Class `GradeBook`)* Modify class `GradeBook` (Figs. 3.11–3.12) as follows:

a) Include a second `string` data member that represents the course instructor's name.
b) Provide a *set* function to change the instructor's name and a *get* function to retrieve it.
c) Modify the constructor to specify course name and instructor name parameters.
d) Modify function `displayMessage` to output the welcome message and course name, then the `string` `"This course is presented by: "` followed by the instructor's name.

Use your modified class in a test program that demonstrates the class's new capabilities.

3.12 *(`Account` Class)* Create an `Account` class that a bank might use to represent customers' bank accounts. Include a data member of type `int` to represent the account balance. [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it's greater than or equal to 0. If not, set the balance to 0 and display an error message indicating that the initial balance was invalid. Provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print a message indicating `"Debit amount exceeded account balance."` Member function `getBalance` should return the current balance. Create a program that creates two `Account` objects and tests the member functions of class `Account`.

3.13 *(`Invoice` Class)* Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four data members—a part number (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`) and a price per item (type `int`). [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Your class should have a constructor that initializes the four data members. Provide a *set* and a *get* function for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an `int` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0`. Write a test program that demonstrates class `Invoice`'s capabilities.

3.14 *(`Employee` Class)* Create a class called `Employee` that includes three pieces of information as data members—a first name (type `string`), a last name (type `string`) and a monthly salary (type `int`). [*Note:* In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)—called floating-point values—to represent dollar amounts.] Your class should have a constructor that initializes the three data members. Provide a *set* and a *get* function for each data member. If the monthly salary is not positive, set it to `0`. Write a test program that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10 percent raise and display each `Employee`'s yearly salary again.

3.15 *(`Date` Class)* Create a class called `Date` that includes three pieces of information as data members—a month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1–12; if it isn't, set the month to 1. Provide a *set* and a *get* function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test program that demonstrates class `Date`'s capabilities.

Making a Difference

3.16 *(Target-Heart-Rate Calculator)* While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that is 50–85% of your maximum heart rate. [Note: *These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.*] Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* functions. The class also should include a function `getAge` that calculates and returns the person's age (in years), a function `getMaxiumumHeartRate` that calculates and returns the person's maximum heart rate and a function `getTargetHeartRate` that calculates and returns the person's target heart rate. Since you do not yet know how to obtain the current date from the computer, function `getAge` should prompt the user to enter the current month, day and year before calculating the person's age. Write an application that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

3.17 *(Computerization of Health Records)* A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and in emergencies, could save lives. In this exercise, you'll design a "starter" `HealthProfile` class for a person. The class attributes should include the person's first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide *set* and *get* functions. The class also should include functions that calculate and return the user's age in years, maximum heart rate and target-heart-rate range (see Exercise 3.16), and body mass index (BMI; see Exercise 2.30). Write an application that prompts for the person's information, instantiates an object of class `HealthProfile` for that person and prints the information from that object—including the person's first name, last name, gender, date of birth, height and weight—then calculates and prints the person's age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the "BMI values" chart from Exercise 2.30. Use the same technique as Exercise 3.16 to calculate the person's age.

4

Control Statements: Part 1

Let's all move one place on.
—Lewis Carroll

The wheel is come full circle.
—William Shakespeare

How many apples fell on Newton's head before he took the hint!
—Robert Frost

All the evolution we know of proceeds from the vague to the definite.
—Charles Sanders Peirce

Objectives

In this chapter you'll learn:

- Basic problem-solving techniques.
- To develop algorithms through the process of top-down, stepwise refinement.
- To use the `if` and `if...else` selection statements to choose among alternative actions.
- To use the `while` repetition statement to execute statements in a program repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Outline

4.1 Introduction

Before writing a program to solve a problem, we must have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a program, we must also understand the available building blocks and employ proven program construction techniques. In this chapter and in Chapter 5, Control Statements: Part 2, we discuss these issues as we present the theory and principles of structured programming. The concepts presented here are crucial to building effective classes and manipulating objects.

In this chapter, we introduce C++'s `if`, `if...else` and `while` statements, three of the building blocks that allow you to specify the logic required for member functions to perform their tasks. We devote a portion of this chapter (and Chapters 5 and 7) to further developing the `GradeBook` class. In particular, we add a member function to the `GradeBook` class that uses control statements to calculate the average of a set of student grades. Another example demonstrates additional ways to combine control statements. We introduce C++'s assignment, increment and decrement operators. These additional operators abbreviate and simplify many program statements.

4.2 Algorithms

Any solvable computing problem can be solved by executing of a series of actions in a specific order. A **procedure** for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which the actions execute

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work prepared to make critical decisions. Suppose the same steps are performed in a different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements (actions) execute is called **program control**. This chapter investigates program control using C++'s **control statements**.

4.3 Pseudocode

Pseudocode (or "fake" code) is an artificial and informal language that helps you develop algorithms without having to worry about the details of C++ language syntax. The pseudocode we present is helpful for developing algorithms that will be converted to structured C++ programs. Pseudocode is similar to everyday English; it's convenient and user friendly, although it isn't an actual computer programming language.

Pseudocode does *not* execute on computers. Rather, it helps you "think out" a program before attempting to write it in a programming language, such as C++.

The style of pseudocode we present consists purely of characters, so you can type pseudocode conveniently, using any editor program. A carefully prepared pseudocode program can easily be converted to a corresponding C++ program. In many cases, this simply requires replacing pseudocode statements with C++ equivalents.

Pseudocode normally describes only **executable statements**, which cause specific actions to occur after you convert a program from pseudocode to C++ and the program is compiled and run on a computer. Declarations (that do not have initializers or do not involve constructor calls) are not executable statements. For example, the declaration

```
int counter;
```

tells the compiler the type of variable `counter` and instructs the compiler to reserve space in memory for the variable. This declaration does not cause any action—such as input, output or a calculation—to occur when the program executes. We typically do not include variable declarations in our pseudocode. Some programmers choose to list variables and mention their purposes at the beginning of pseudocode programs.

Let's look at an example of pseudocode that may be written to help a programmer create the addition program of Fig. 2.5. This pseudocode (Fig. 4.1) corresponds to the algorithm that inputs two integers from the user, adds these integers and displays their sum. We show the complete pseudocode listing here—we'll show how to *create* pseudocode from a problem statement later in the chapter.

Lines 1–2 correspond to the statements in lines 13–14 of Fig. 2.5. Notice that the pseudocode statements are simply English statements that convey what task is to be performed in C++. Likewise, lines 4–5 correspond to the statements in lines 16–17 of Fig. 2.5 and lines 7–8 correspond to the statements in lines 19 and 21 of Fig. 2.5.

```
1  Prompt the user to enter the first integer
2  Input the first integer
3
4  Prompt the user to enter the second integer
5  Input the second integer
6
7  Add first integer and second integer, store result
8  Display result
```

Fig. 4.1 | Pseudocode for the addition program of Fig. 2.5.

4.4 Control Structures

Normally, statements in a program execute one after the other in the order in which they're written. This is called **sequential execution**. Various C++ statements we'll soon discuss enable you to specify that *the next statement to execute may be other than the next one in sequence*. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The finger of blame was pointed at the **goto statement**, which allows you to specify a transfer of control to one of a wide range of possible destinations in a program (creating what's often called "spaghetti code"). The notion of so-called **structured programming** became almost synonymous with "**goto elimination**."

The research of Böhm and Jacopini[1] demonstrated that programs could be written without any goto statements. It became the challenge of the era for programmers to shift their styles to "goto-less programming." It was not until the 1970s that programmers started taking structured programming seriously. The results have been impressive, as software development groups have reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes is that structured programs are clearer, are easier to debug, test and modify and are more likely to be bug-free in the first place.

Böhm and Jacopini's work demonstrated that all programs could be written in terms of only three **control structures**, namely, the **sequence structure**, the **selection structure** and the **repetition structure**. The term "control structures" comes from the field of computer science. When we introduce C++'s implementations of control structures, we'll refer to them in the terminology of the C++ standard document as "control statements."

Sequence Structure in C++

The *sequence structure* is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they're written—that is, in sequence. The UML **activity diagram** of Fig. 4.2 illustrates a typical sequence structure in which two calculations are performed in order. C++ allows you to have as many actions as

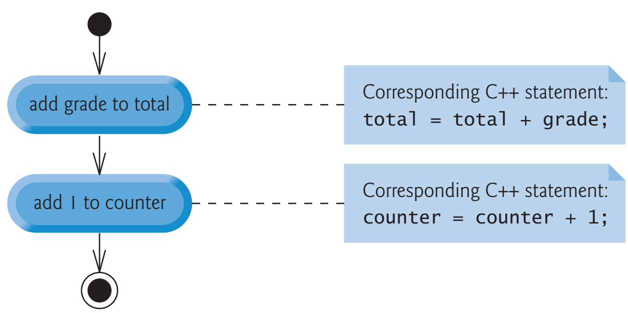


Fig. 4.2 | Sequence-structure activity diagram.

1. Böhm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366–371.

you want in a sequence structure. As you'll soon see, anywhere a single action may be placed, you may place several actions in sequence.

In this figure, the two statements add a grade to a `total` variable and add the value 1 to a `counter` variable. Such statements might appear in a program that averages several student grades. To calculate an average, the total of the grades being averaged is divided by the number of grades. A counter variable would be used to keep track of the number of values being averaged. You'll see similar statements in the program of Section 4.8.

An activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, such as the sequence structure in Fig. 4.2. Activity diagrams are composed of special-purpose symbols, such as **action state symbols** (a rectangle with its left and right sides replaced with arcs curving outward), **diamonds** and **small circles**; these symbols are connected by **transition arrows**, which represent the flow of the activity.

Activity diagrams clearly show how control structures operate. Consider the sequence-structure activity diagram of Fig. 4.2. It contains two **action states** that represent actions to perform. Each action state contains an **action expression**—e.g., "add grade to total" or "add 1 to counter"—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram are called transition arrows. These arrows represent **transitions**, which indicate the *order* in which the actions represented by the action states occur—the program that implements the activities illustrated by the activity diagram in Fig. 4.2 first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the diagram represents the activity's **initial state**—the *beginning of the workflow* before the program performs the modeled activities. The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the **final state**—the *end of the workflow* after the program performs its activities.

Figure 4.2 also includes rectangles with the upper-right corners folded over. These are called **notes** in the UML—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.2 uses UML notes to show the C++ code associated with each action state in the activity diagram. A **dotted line** connects each note with the element that the note describes. Activity diagrams normally do not show the C++ code that implements the activity. We use notes for this purpose here to illustrate how the diagram relates to C++ code. For more information on the UML, see our optional case study, which appears in Chapters 25–26, and visit our UML Resource Center at `www.deitel.com/UML/`.

Selection Statements in C++

C++ provides three types of selection statements (discussed in this chapter and Chapter 5). The `if` selection statement either performs (selects) an action if a condition is true or skips the action if the condition is false. The `if...else` selection statement performs an action if a condition is true or performs a different action if the condition is false. The `switch` selection statement (Chapter 5) performs one of *many* different actions, depending on the value of an integer expression.

The `if` selection statement is a **single-selection statement** because it selects or ignores a single action (or, as you'll soon see, a single group of actions). The `if...else` statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The `switch` selection statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

Repetition Statements in C++

C++ provides three types of repetition statements (also called **looping statements** or **loops**) for performing statements repeatedly while a condition (called the **loop-continuation condition**) remains true. These are the **`while`**, **`do…while`** and **`for`** statements. (Chapter 5 presents the `do…while` and `for` statements.) The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will *not* execute. The `do…while` statement performs the action (or group of actions) in its body at least once.

Each of the words `if`, `else`, `switch`, `while`, `do` and `for` is a C++ keyword. Keywords must *not* be used as identifiers, such as variable names, and must be spelled with only lowercase letters. Figure 4.3 provides a complete list of C++ keywords.

Common Programming Error 4.1

Using a keyword as an identifier is a syntax error.

C++ Keywords				
Keywords common to the C and C++ programming languages				
`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			
C++-only keywords				
`and`	`and_eq`	`asm`	`bitand`	`bitor`
`bool`	`catch`	`class`	`compl`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`export`	`false`
`friend`	`inline`	`mutable`	`namespace`	`new`
`not`	`not_eq`	`operator`	`or`	`or_eq`
`private`	`protected`	`public`	`reinterpret_cast`	`static_cast`
`template`	`this`	`throw`	`true`	`try`
`typeid`	`typename`	`using`	`virtual`	`wchar_t`
`xor`	`xor_eq`			
C++0x keywords				
`alignof`	`axiom`	`char16_t`	`char32_t`	`concept`
`concept_map`	`constexpr`	`decltype`	`late_check`	`nullptr`
`requires`	`static_assert`	`thread_local`		

Fig. 4.3 | C++ keywords.

Summary of Control Statements in C++

C++ has only three kinds of control structures, which from this point forward we refer to as control statements: the sequence statement, selection statements (three types—`if`, `if…else` and `switch`) and repetition statements (three types—`while`, `for` and `do…while`). Each program combines these control statements as appropriate for the algorithm the program implements. We can model each control statement as an activity diagram with initial and final states that represent a control statement's entry point and exit point, respectively. These **single-entry/single-exit control statements** make it easy to build programs—control statements are attached to one another by connecting the exit point of one to the entry point of the next. This is similar to the way a child stacks building blocks, so we call this **control-statement stacking**. You'll see that there's only one other way to connect control statements—called **control-statement nesting**, in which one control statement is contained *inside* another.

Software Engineering Observation 4.1

Any C++ program can be constructed from only seven different types of control statements (sequence, `if`, `if…else`, `switch`, `while`, `do…while` and `for`) combined in only two ways (control-statement stacking and control-statement nesting).

4.5 `if` Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The pseudocode statement

If student's grade is greater than or equal to 60
    Print "Passed"

determines whether the condition "student's grade is greater than or equal to 60" is `true` or `false`. If the condition is `true`, "Passed" is printed and the next pseudocode statement in order is "performed" (remember that pseudocode is not a real programming language). If the condition is `false`, the print statement is ignored and the next pseudocode statement in order is performed. The indentation of the second line is optional, but it's recommended because it emphasizes the inherent structure of structured programs.

The preceding pseudocode *If* statement can be written in C++ as

```
if ( grade >= 60 )
   cout << "Passed";
```

The C++ code corresponds closely to the pseudocode. This is one of the properties of pseudocode that make it such a useful program development tool.

Figure 4.4 illustrates the single-selection `if` statement. It contains what is perhaps the most important symbol in an activity diagram—the diamond or **decision symbol**, which indicates that a decision is to be made. A decision symbol indicates that the workflow will continue along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets above or next to the transition arrow). If a particular guard condition is true, the workflow enters the action state to which that transition arrow points. In Fig. 4.4, if the grade is greater than or equal to 60, the program prints "Passed" to the screen, then transitions to the final state of this activity. If the grade is less than 60, the program immediately transitions to the final state without displaying a message.

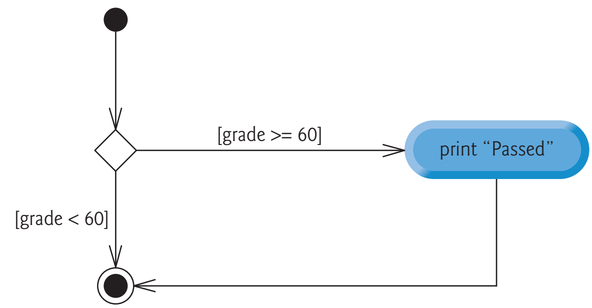


Fig. 4.4 | `if` single-selection statement activity diagram.

You saw in Chapter 2 that decisions can be based on conditions containing relational or equality operators. Actually, in C++, a decision can be based on any expression—if the expression evaluates to zero, it's treated as false; if the expression evaluates to nonzero, it's treated as true. C++ provides the data type **`bool`** for variables that can hold only the values **`true`** and **`false`**—each of these is a C++ keyword.

Portability Tip 4.1

For compatibility with earlier versions of C, which used integers for Boolean values, the `bool` *value* `true` *also can be represented by any nonzero value (compilers typically use 1) and the* `bool` *value* `false` *also can be represented as the value zero.*

The `if` statement is a single-entry/single-exit statement. We'll see that the activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made and final states.

Envision seven bins, each containing only empty UML activity diagrams of one of the seven types of control statements. Your task, then, is assembling a program from the activity diagrams of as many of each type of control statement as the algorithm demands, combining the activity diagrams in only two possible ways (stacking or nesting), then filling in the action states and decisions with action expressions and guard conditions in a manner appropriate to form a structured implementation for the algorithm. We'll continue discussing the variety of ways in which actions and decisions may be written.

4.6 `if…else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`; otherwise the action is skipped. The `if…else` double-selection statement allows you to specify an action to perform when the condition is true and a different action to perform when the condition is `false`. For example, the pseudocode statement

If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"

prints “Passed” if the student’s grade is greater than or equal to 60, but prints “Failed” if the student’s grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is “performed.”

The preceding pseudocode *If…Else* statement can be written in C++ as

```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```

The body of the `else` is also indented.

Good Programming Practice 4.1

Whatever indentation convention you choose should be applied consistently. It’s difficult to read programs that do not obey uniform spacing conventions.

Good Programming Practice 4.2

If there are several levels of indentation, each level should be indented the same additional amount of space to promote readability and maintainability.

Figure 4.5 illustrates the the `if...else` statement’s flow of control.

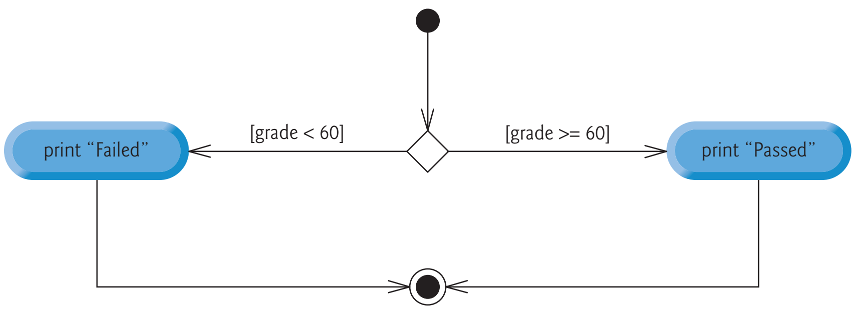


Fig. 4.5 | `if...else` double-selection statement activity diagram.

Conditional Operator (?:)

C++ provides the **conditional operator** (`?:`), which is closely related to the `if...else` statement. The conditional operator is C++’s only **ternary operator**—it takes three operands. The operands, together with the conditional operator, form a **conditional expression**. The first operand is a condition, the second operand is the value for the entire conditional expression if the condition is `true` and the third operand is the value for the entire conditional expression if the condition is `false`. For example, the output statement

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

contains a conditional expression, `grade >= 60 ? "Passed" : "Failed"`, that evaluates to the string `"Passed"` if the condition `grade >= 60` is `true`, but evaluates to `"Failed"` if the condition is `false`. Thus, the statement with the conditional operator performs essentially the same as the preceding `if...else` statement. As we’ll see, the precedence of the conditional operator is low, so the parentheses in the preceding expression are required.

Error-Prevention Tip 4.1

To avoid precedence problems (and for clarity), place conditional expressions (that appear in larger expressions) in parentheses.

The values in a conditional expression also can be actions to execute. For example, the following conditional expression also prints "Passed" or "Failed":

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

The preceding conditional expression is read, "If `grade` is greater than or equal to `60`, then `cout << "Passed"`; otherwise, `cout << "Failed"`." This, too, is comparable to the preceding `if...else` statement. Conditional expressions can appear in some program locations where `if...else` statements cannot.

Nested `if...else` *Statements*

Nested `if...else` statements test for multiple cases by placing `if...else` selection statements inside other `if...else` selection statements. For example, the following pseudocode `if...else` statement prints `A` for exam grades greater than or equal to 90, `B` for grades in the range 80 to 89, `C` for grades in the range 70 to 79, `D` for grades in the range 60 to 69 and `F` for all other grades:

```
If student's grade is greater than or equal to 90
    Print "A"
Else
    If student's grade is greater than or equal to 80
        Print "B"
    Else
        If student's grade is greater than or equal to 70
            Print "C"
        Else
            If student's grade is greater than or equal to 60
                Print "D"
            Else
                Print "F"
```

This pseudocode can be written in C++ as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else
   if ( studentGrade >= 80 ) // 80-89 gets "B"
      cout << "B";
   else
      if ( studentGrade >= 70 ) // 70-79 gets "C"
         cout << "C";
      else
         if ( studentGrade >= 60 ) // 60-69 gets "D"
            cout << "D";
         else // less than 60 gets "F"
            cout << "F";
```

If `studentGrade` is greater than or equal to 90, the first four conditions are `true`, but only the output statement after the first test executes. Then, the program skips the `else`-part of the "outermost" `if…else` statement. Most write the preceding `if…else` statement as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
   cout << "A";
else if ( studentGrade >= 80 ) // 80-89 gets "B"
   cout << "B";
else if ( studentGrade >= 70 ) // 70-79 gets "C"
   cout << "C";
else if ( studentGrade >= 60 ) // 60-69 gets "D"
   cout << "D";
else // less than 60 gets "F"
   cout << "F";
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right, which can force lines to wrap.

Performance Tip 4.1

A nested `if…else` statement can perform much faster than a series of single-selection `if` statements because of the possibility of early exit after one of the conditions is satisfied.

Performance Tip 4.2

In a nested `if…else` statement, test the conditions that are more likely to be `true` at the beginning of the nested statement. This will enable the nested `if…else` statement to run faster by exiting earlier than if infrequently occurring cases were tested first.

Dangling-`else` Problem

The C++ compiler always associates an `else` with the immediately preceding `if` unless told to do otherwise by the placement of braces (`{` and `}`). This behavior can lead to what's referred to as the **dangling-`else` problem**. For example,

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
else
   cout << "x is <= 5";
```

appears to indicate that if `x` is greater than `5`, the nested `if` statement determines whether `y` is also greater than `5`. If so, `"x and y are > 5"` is output. Otherwise, it appears that if `x` is not greater than `5`, the `else` part of the `if…else` outputs `"x is <= 5"`.

Beware! This nested `if…else` statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
   else
      cout << "x is <= 5";
```

in which the body of the first `if` is a nested `if…else`. The outer `if` statement tests whether `x` is greater than `5`. If so, execution continues by testing whether `y` is also greater than `5`.

If the second condition is true, the proper string—`"x and y are > 5"`—is displayed.[7] However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know that `x` is greater than `5`.

To force the nested `if...else` statement to execute as originally intended, we can write it as follows:

```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x and y are > 5";
}
else
   cout << "x is <= 5";
```

The braces (`{}`) indicate to the compiler that the second `if` statement is in the body of the first `if` and that the `else` is associated with the first `if`. Exercises 4.23–4.24 further investigate the dangling-`else` problem.

Blocks

The `if` selection statement expects only one statement in its body. Similarly, the `if` and `else` parts of an `if...else` statement each expect only one body statement. To include *several* statements in the body of an `if` or in either part of an `if...else`, enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a **compound statement** or a **block**. We use the term "block" from this point forward.

Software Engineering Observation 4.2

A block can be placed anywhere in a program that a single statement can be placed.

The following example includes a block in the `else` part of an `if...else` statement.

```
if ( studentGrade >= 60 )
   cout << "Passed.\n";
else
{
   cout << "Failed.\n";
   cout << "You must take this course again.\n";
}
```

In this case, if `studentGrade` is less than 60, the program executes *both* statements in the body of the `else` and prints

```
Failed.
You must take this course again.
```

Notice the braces surrounding the two statements in the `else` clause. These braces are important. Without the braces, the statement

```
cout << "You must take this course again.\n";
```

would be outside the body of the `else` part of the `if` and would execute regardless of whether the grade was less than 60. This is a logic error.

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have no statement at all, which is called a **null statement** or an **empty statement**. The null statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 4.2

Placing a semicolon after the condition in an `if` *statement leads to a logic error in single-selection* `if` *statements and a syntax error in double-selection* `if...else` *statements (when the* `if` *part contains an actual body statement).*

4.7 while Repetition Statement

A **repetition statement** specifies that a program should repeat an action while some condition remains true. The pseudocode statement

While there are more items on my shopping list
Purchase next item and cross it off my list

describes the repetition that occurs during a shopping trip. The condition, "there are more items on my shopping list" is either true or false. If it's true, then the action, "Purchase next item and cross it off my list" is performed. This action will be performed repeatedly while the condition remains true. The statement contained in the *While* repetition statement constitutes the body of the *While*, which can be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition statement executes.

As an example of C++'s `while` repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable `product` has been initialized to `3`. When the following `while` repetition statement finishes executing, `product` contains the result:

```
int product = 3;

while ( product <= 100 )
   product = 3 * product;
```

When the `while` statement begins execution, `product`'s value is 3. Each repetition multiplies `product` by `3`, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, the `while` statement condition—`product <= 100`—becomes `false`. This terminates the repetition, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

Common Programming Error 4.3

Not providing, in the body of a `while` *statement, an action that eventually causes the condition in the* `while` *to become false normally results in a logic error called an* infinite loop, *in which the repetition statement never terminates. This can make a program appear to "hang" or "freeze" if the loop body does not contain statements that interact with the user.*

The UML activity diagram of Fig. 4.6 illustrates the flow of control that corresponds to the preceding `while` statement. Once again, the symbols in the diagram (besides the initial state, transition arrows, a final state and three notes) represent an action state and a decision. This diagram also introduces the UML's **merge symbol**, which joins two flows of activity into one flow of activity. The UML represents *both* the merge symbol and the decision symbol as diamonds. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they *both* flow into the decision that determines whether the loop should begin (or continue) executing. The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows. A decision symbol has one transition arrow pointing *to* the diamond and two or more transition arrows pointing out *from* the diamond to indicate possible transitions from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing *to* the diamond and only one transition arrow pointing *from* the diamond, to indicate multiple activity flows merging to continue the activity. Unlike the decision symbol, the merge symbol does *not* have a counterpart in C++ code.

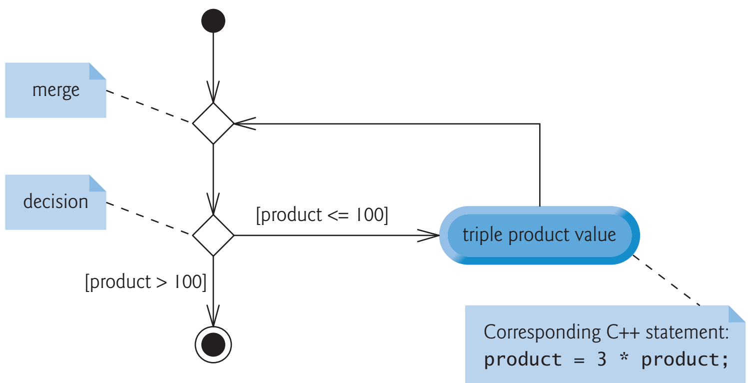


Fig. 4.6 | `while` repetition statement UML activity diagram.

The diagram of Fig. 4.6 clearly shows the repetition of the `while` statement discussed earlier in this section. The transition arrow emerging from the action state points to the merge, which transitions back to the decision that's tested each time through the loop until the guard condition `product > 100` becomes true. Then the `while` statement exits (reaches its final state) and control passes to the next statement in sequence in the program.

Performance Tip 4.3

A small performance improvement for code that executes many times in a loop can result in substantial overall performance improvement.

4.8 Formulating Algorithms: Counter-Controlled Repetition

To illustrate how programmers develop algorithms, this section and Section 4.9 solve two variations of a class average problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (0 to 100) for this quiz are available to you. Calculate and display the total of the grades and the class average.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each of the grades, calculate the average and print the result.

Pseudocode Algorithm with Counter-Controlled Repetition

Let's use pseudocode to list the *actions* to execute and specify the *order* in which these actions should occur. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** to control the number of times a group of statements will execute (also known as the number of **iterations** of the loop).

Counter-controlled repetition is often called **definite repetition** because the number of repetitions is known *before* the loop begins executing. In this example, repetition terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm (Fig. 4.7) and a version of class `GradeBook` (Fig. 4.8–Fig. 4.9) that implements the algorithm in a C++ member function. The section then presents an application (Fig. 4.10) that demonstrates the algorithm in action. In Section 4.9 we demonstrate how to use pseudocode to develop such an algorithm from scratch.

> **Software Engineering Observation 4.3**
> *Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. The process of producing a working C++ program from the algorithm is typically straightforward.*

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Prompt the user to enter the next grade
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the total of the grades for all students in the class
Print the class average
```

Fig. 4.7 | Pseudocode for solving the class average problem with counter-controlled repetition.

Note the references in the pseudocode algorithm of Fig. 4.7 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A **counter** is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

Enhancing **GradeBook** *Validation*

Let's consider an enhancement we made to class `GradeBook`. In Fig. 3.16, our `setCourseName` member function validated the course name by testing whether the course name's length was less than or equal to 25 characters, using an `if` statement. If this was true, the course name would be set. This code was followed by an `if` statement that tested whether the course name's length was larger than 25 characters (in which case the course name would be shortened). The second `if` statement's condition is the exact opposite of the first `if` statement's condition. If one condition evaluates to `true`, the other must evaluate to `false`. Such a situation is ideal for an `if...else` statement, so we've modified our code, replacing the two `if` statements with one `if...else` statement (lines 18–25 of Fig. 4.9).

Implementing Counter-Controlled Repetition in Class **GradeBook**

Class `GradeBook` (Fig. 4.8–Fig. 4.9) contains a constructor (declared in line 11 of Fig. 4.8 and defined in lines 9–12 of Fig. 4.9) that assigns a value to the class's data member `courseName` (declared in line 17 of Fig. 4.8). Lines 16–26, 29–32 and 35–29 of Fig. 4.9 define member functions `setCourseName`, `getCourseName` and `displayMessage`, respectively. Lines 42–68 define member function `determineClassAverage`, which implements the class average algorithm described by the pseudocode in Fig. 4.7.

```
1  // Fig. 4.8: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

Fig. 4.8 | Class average problem using counter-controlled repetition: `GradeBook` header.

```
1  // Fig. 4.9: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with counter-controlled repetition.
4  #include <iostream>
5  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // constructor initializes courseName with string supplied as argument
9  GradeBook::GradeBook( string name )
10 {
```

Fig. 4.9 | Class average problem using counter-controlled repetition: `GradeBook` source code file. (Part 1 of 3.)

```
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function to set the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage()
{
   int total; // sum of grades entered by user
   int gradeCounter; // number of the grade to be entered next
   int grade; // grade value entered by user
   int average; // average of grades

   // initialization phase
   total = 0; // initialize total
   gradeCounter = 1; // initialize loop counter

   // processing phase
   while ( gradeCounter <= 10 ) // loop 10 times
   {
      cout << "Enter grade: "; // prompt for input
      cin >> grade; // input next grade
      total = total + grade; // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter by 1
   } // end while

```

Fig. 4.9 | Class average problem using counter-controlled repetition: `GradeBook` source code file. (Part 2 of 3.)

```
62     // termination phase
63     average = total / 10; // integer division yields integer result
64
65     // display total and average of grades
66     cout << "\nTotal of all 10 grades is " << total << endl;
67     cout << "Class average is " << average << endl;
68  } // end function determineClassAverage
```

Fig. 4.9 | Class average problem using counter-controlled repetition: `GradeBook` source code file. (Part 3 of 3.)

Lines 44–47 (Fig. 4.9) declare local variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`. Variable `grade` stores the user input. Notice that the preceding declarations appear in the body of member function `determineClassAverage`.

In this chapter's versions of class `GradeBook`, we simply read and process a set of grades. The averaging calculation is performed in member function `determineClassAverage` using local variables—we do not preserve any information about student grades in the class's data members. In Chapter 7, Arrays and Vectors, we modify class `GradeBook` to maintain the grades in memory using a data member that refers to a data structure known as an array. This allows a `GradeBook` object to perform various calculations on a set of grades without requiring the user to enter the grades multiple times.

Lines 50–51 initialize `total` to 0 and `gradeCounter` to 1 before they're used in calculations. Counter variables are normally initialized to zero or one, depending on their use. An uninitialized variable contains a **"garbage" value** (also called an **undefined value**)—the value last stored in the memory location reserved for that variable. The variables `grade` and `average` (for the user input and calculated average, respectively) need not be initialized—their values will be assigned as they're input or calculated later in the function.

Common Programming Error 4.4

Not initializing counters and totals can lead to logic errors.

Error-Prevention Tip 4.2

Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they're used.

Good Programming Practice 4.3

Declare each variable on a separate line with its own comment for readability.

Line 54 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to `10`. While this condition remains true, the `while` statement repeatedly executes the statements between the braces that delimit its body (lines 55–60).

Line 56 displays the prompt `"Enter grade: "`. This line corresponds to the pseudocode statement *"Prompt the user to enter the next grade."* Line 57 reads the grade entered by the user and assigns it to variable `grade`. This line corresponds to the pseudocode statement *"Input the next grade."* Recall that variable `grade` was not initialized earlier in the program, because the program obtains the value for `grade` from the user during each iteration of the loop. Line 58 adds the new grade entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 59 adds `1` to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes `gradeCounter` to exceed `10`. At that point the `while` loop terminates because its condition (line 54) becomes false.

When the loop terminates, line 63 performs the averaging calculation and assigns its result to the variable `average`. Line 66 displays the text `"Total of all 10 grades is "` followed by variable `total`'s value. Line 67 then displays the text `"Class average is "` followed by variable `average`'s value. Member function `determineClassAverage` then returns control to the calling function (i.e., `main` in Fig. 4.10).

Demonstrating Class **GradeBook**

Figure 4.10 contains this application's `main` function, which creates an object of class `GradeBook` and demonstrates its capabilities. Line 9 of Fig. 4.10 creates a new `GradeBook` object called `myGradeBook`. The string in line 9 is passed to the `GradeBook` constructor (lines 9–12 of Fig. 4.9). Line 11 of Fig. 4.10 calls `myGradeBook`'s `displayMessage` member function to display a welcome message to the user. Line 12 then calls `myGradeBook`'s `determineClassAverage` member function to allow the user to enter 10 grades, for which the member function then calculates and prints the average—the member function performs the algorithm shown in the pseudocode of Fig. 4.7.

```
1  // Fig. 4.10: fig04_10.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object myGradeBook and
8     // pass course name to constructor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.determineClassAverage(); // find average of 10 grades
13 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
```

Fig. 4.10 | Class average problem using counter-controlled repetition: Creating an object of class `GradeBook` (Fig. 4.8–Fig. 4.9) and invoking its `determineClassAverage` member function. (Part 1 of 2.)

```
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 4.10 | Class average problem using counter-controlled repetition: Creating an object of class `GradeBook` (Fig. 4.8–Fig. 4.9) and invoking its `determineClassAverage` member function. (Part 2 of 2.)

Notes on Integer Division and Truncation

The averaging calculation performed in response to the function call in line 12 of Fig. 4.10 produces an integer result. The sample execution indicates that the sum of the grade values is 846, which, when divided by 10, should yield 84.6—a number with a decimal point. However, the result of the calculation `total / 10` (line 63 of Fig. 4.9) is the integer 84, because `total` and 10 are both integers. Dividing two integers results in integer division—any fractional part of the calculation is *lost* (i.e., **truncated**). We'll see how to obtain a result that includes a decimal point from the averaging calculation in the next section.

Common Programming Error 4.5

Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

In Fig. 4.9, if line 63 used `gradeCounter` rather than 10, the output for this program would display an incorrect value, 76. This would occur because in the final iteration of the `while` statement, `gradeCounter` was incremented to the value 11 in line 59.

Common Programming Error 4.6

*Using a loop's counter-control variable in a calculation after the loop often causes a common logic error called an **off-by-one error**. In a counter-controlled loop that counts up by one each time through the loop, the loop terminates when the counter's value is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10).*

4.9 Formulating Algorithms: Sentinel-Controlled Repetition

Let's generalize the class average problem. Consider the following problem:

> *Develop a class average program that processes grades for an arbitrary number of students each time it's run.*

In the previous example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. The program must process an *arbitrary* number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

To solve this problem, we can use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." After typing the legitimate grades, the user types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called **indefinite repetition** because the number of repetitions is *not* known before the loop begins executing.

The sentinel value must be chosen so that it's not confused with an acceptable input value. Grades are normally nonnegative integers, so –1 is an acceptable sentinel value. Thus, a run of the program might process inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89. Since –1 is the sentinel value, it should not enter into the averaging calculation.

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

We approach the class average program with a technique called **top-down, stepwise refinement**, a technique that's essential to the development of well-structured programs. We begin with a pseudocode representation of the **top**—a single statement that conveys the overall function of the program:

Determine the class average for the quiz for an arbitrary number of students

The top is, in effect, a *complete* representation of a program. Unfortunately, the top (as in this case) rarely conveys sufficient detail from which to write a program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following **first refinement**.

Initialize variables
Input, sum and count the quiz grades
Calculate and print the total of all student grades and the class average

This refinement uses only the sequence structure—these steps execute in order.

Software Engineering Observation 4.4

Each refinement, as well as the top itself, is a complete *specification of the algorithm; only the level of detail varies.*

Software Engineering Observation 4.5

Many programs can be divided logically into three phases: an initialization phase *that initializes the program variables; a* processing phase *that inputs data values and adjusts program variables (such as counters and totals) accordingly; and a* termination phase *that calculates and outputs the final results.*

Proceeding to the Second Refinement

The preceding *Software Engineering Observation* is often all you need for the first refinement in the top-down process. In the **second refinement**, we commit to specific variables. In this example, we need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it's input by the user and a variable to hold the calculated average. The pseudocode statement

Initialize variables

can be refined as follows:

Initialize total to zero
Initialize counter to zero

Only the variables *total* and *counter* need to be initialized before they're used. The variables *average* and *grade* (for the calculated average and the user input, respectively) need not be initialized, because their values will be replaced as they're calculated or input.

The pseudocode statement

Input, sum and count the quiz grades

requires a repetition statement (i.e., a loop) that successively inputs each grade. We don't know in advance how many grades are to be processed, so we'll use **sentinel-controlled repetition**. The user enters legitimate grades one at a time. After entering the last legitimate grade, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is then

```
Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)
```

In pseudocode, we do *not* use braces around the statements that form the body of the *While* structure. We simply *indent* the statements under the *While* to show that they belong to the *While*. Again, pseudocode is only an informal program development aid.

The pseudocode statement

Calculate and print the total of all student grades and the class average

can be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of the grades for all students in the class
    Print the class average
else
    Print "No grades were entered"
```

We test for the possibility of *division by zero*—normally a **fatal logic error** that, if undetected, would cause the program to fail (often called **"crashing"**). The complete second refinement of the pseudocode for the class average problem is shown in Fig. 4.11.

Common Programming Error 4.7

An attempt to divide by zero normally causes a fatal runtime error.

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of the grades for all students in the class
    Print the class average
else
    Print "No grades were entered"
```

Fig. 4.11 | Class average problem pseudocode algorithm with sentinel-controlled repetition.

Error-Prevention Tip 4.3

When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur. We'll say more about dealing with these kinds of errors when we discuss exception handling.

The pseudocode in Fig. 4.11 solves the more general class average problem. This algorithm required only two levels of refinement. Sometimes more levels are necessary.

Software Engineering Observation 4.6

Terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C++. Typically, implementing the C++ program is then straightforward.

Software Engineering Observation 4.7

Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this method might work for simple and familiar problems, it can lead to serious difficulties in large, complex projects.

Implementing Sentinel-Controlled Repetition in Class `GradeBook`

Figures 4.12–4.13 show class `GradeBook` containing member function `determineClassAverage` that implements the pseudocode algorithm of Fig. 4.11 (this class is demonstrated in Fig. 4.14). Although each grade entered is an integer, the averaging calculation is likely to produce a number with a decimal point—in other words, a real number or **float-**

ing-point number (e.g., 7.33, 0.0975 or 1000.12345). The type `int` cannot represent such a number, so this class must use another type to do so. C++ provides several data types for storing floating-point numbers in memory, including **`float`** and **`double`**. The primary difference between these types is that, compared to `float` variables, `double` variables can typically store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's **precision**). This program introduces a special operator called a **cast operator** to force the averaging calculation to produce a floating-point numeric result.

```cpp
// Fig. 4.12: GradeBook.h
// Definition of class GradeBook that determines a class average.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor initializes course name
   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void determineClassAverage(); // averages grades entered by the user
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 4.12 | Class average problem using sentinel-controlled repetition: `GradeBook` header.

```cpp
// Fig. 4.13: GradeBook.cpp
// Member-function definitions for class GradeBook that solves the
// class average program with sentinel-controlled repetition.
#include <iostream>
#include <iomanip> // parameterized stream manipulators
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function to set the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 1 of 3.)

```
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage()
{
   int total; // sum of grades entered by user
   int gradeCounter; // number of grades entered
   int grade; // grade value
   double average; // number with decimal point for average

   // initialization phase
   total = 0; // initialize total
   gradeCounter = 0; // initialize loop counter

   // processing phase
   // prompt for input and read grade from user
   cout << "Enter grade or -1 to quit: ";
   cin >> grade; // input grade or sentinel value

   // loop until sentinel value read from user
   while ( grade != -1 ) // while grade is not -1
   {
      total = total + grade; // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter

      // prompt for input and read next grade from user
      cout << "Enter grade or -1 to quit: ";
      cin >> grade; // input grade or sentinel value
   } // end while

```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: GradeBook source code file. (Part 2 of 3.)

```
70       // termination phase
71       if ( gradeCounter != 0 ) // if user entered at least one grade...
72       {
73          // calculate average of all grades entered
74          average = static_cast< double >( total ) / gradeCounter;
75
76          // display total and average (with two digits of precision)
77          cout << "\nTotal of all " << gradeCounter << " grades entered is "
78             << total << endl;
79          cout << "Class average is " << setprecision( 2 ) << fixed << average
80             << endl;
81       } // end if
82       else // no grades were entered, so output appropriate message
83          cout << "No grades were entered" << endl;
84    } // end function determineClassAverage
```

Fig. 4.13 | Class average problem using sentinel-controlled repetition: `GradeBook` source code file. (Part 3 of 3.)

```
// Fig. 4.14: fig04_14.cpp
// Create GradeBook object and invoke its determineClassAverage function.
#include "GradeBook.h" // include definition of class GradeBook

int main()
{
   // create GradeBook object myGradeBook and
   // pass course name to constructor
   GradeBook myGradeBook( "CS101 C++ Programming" );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.determineClassAverage(); // find average of 10 grades
} // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

Fig. 4.14 | Class average problem using sentinel-controlled repetition: Creating a `GradeBook` object and invoking its `determineClassAverage` member function.

This example stacks control statements on top of one another—the `while` statement (lines 60–68 of Fig. 4.13) is immediately followed by an `if...else` statement (lines 71–83) in sequence. Much of the code in this program is identical to the code in Fig. 4.9, so we concentrate on the new features and issues.

Line 48 declares the `double` variable `average`. Recall that we used an `int` variable in the preceding example to store the class average. Using type `double` in the current example allows us to store the class average calculation's result as a floating-point number. Line 52 initializes the variable `gradeCounter` to `0`, because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, the program increments variable `gradeCounter` only when the user enters a valid grade value and the program completes the processing of the grade. Finally, notice that both input statements (lines 57 and 67) are preceded by an output statement that prompts the user for input.

Good Programming Practice 4.4

Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values. In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

Program Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. 4.9. In counter-controlled repetition, each iteration of the `while` statement (lines 54–60 of Fig. 4.9) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 56–57 of Fig. 4.13) before reaching the `while`. This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` (line 62) and increments `gradeCounter` (line 63). Then lines 66–67 in the loop's body prompt for and input the next value from the user. Next, program control reaches the closing right brace (`}`) of the body in line 68, so execution continues with the test of the `while`'s condition (line 60). The condition uses the most recent `grade` input by the user to determine whether the loop's body should execute again. The value of variable `grade` is always input from the user immediately before the program tests the `while` condition. This allows the program to determine whether the value just input is the sentinel value *before* the program processes that value (i.e., adds it to the `total` and increments `gradeCounter`). If the sentinel value is input, the loop terminates, and the program does not add the value –1 to the `total`.

After the loop terminates, the `if...else` statement in lines 71–83 executes. The condition in line 71 determines whether any grades were entered. If none were, the `else` part (lines 82–83) of the `if...else` statement executes and displays the message `"No grades were entered"` and the member function returns control to the calling function.

Notice the block in the `while` loop in Fig. 4.13. Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly, as follows:

```
// loop until sentinel value read from user
while ( grade != -1 )
   total = total + grade; // add grade to total
gradeCounter = gradeCounter + 1; // increment counter

// prompt for input and read next grade from user
cout << "Enter grade or -1 to quit: ";
cin >> grade;
```

This would cause an infinite loop in the program if the user did not input `-1` for the first grade (in line 57).

Common Programming Error 4.8

Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces, even if the body contains only a single statement.

Floating-Point Number Precision and Memory Requirements

Variables of type `float` represent **single-precision floating-point numbers** and have approximately seven significant digits on most 32-bit systems. Variables of type `double` represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide approximately 15 significant digits on most 32-bit systems—approximately double the precision of `float` variables. Most programmers represent floating-point numbers with type `double`. In fact, C++ treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default. Such values in the source code are known as **floating-point constants**. See Appendix C, Fundamental Types, for the ranges of values for `float`s and `double`s.

In conventional arithmetic, floating-point numbers often arise as a result of division—when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a *fixed* amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

Common Programming Error 4.9

Using floating-point numbers in a manner that assumes they're represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results. Floating-point numbers are represented only approximately.

Although floating-point numbers are not always 100 percent precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Due to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we use type `double` throughout the book.

Converting Between Fundamental Types Explicitly and Implicitly

The variable `average` is declared to be of type `double` (line 48 of Fig. 4.13) to capture the fractional result of our calculation. However, `total` and `gradeCounter` are both integer variables. Recall that dividing two integers results in integer division, in which any fractional part of the calculation is lost truncated). In the following statement:

```
average = total / gradeCounter;
```

the division occurs *first*—the result's fractional part is lost before it's assigned to `average`. To perform a floating-point calculation with integers, we must create temporary floating-point values. C++ provides the **unary cast operator** to accomplish this task. Line 74 uses the cast operator **`static_cast`**`<double>(total)` to create a *temporary* floating-point copy of its operand in parentheses—total. Using a cast operator in this manner is called **explicit conversion**. The value stored in total is still an integer.

The calculation now consists of a floating-point value (the temporary double version of total) divided by the integer gradeCounter. The compiler knows how to evaluate *only* expressions in which the operand types are *identical*. To ensure that the operands are of the same type, the compiler performs an operation called **promotion** (also called **implicit conversion**) on selected operands. For example, in an expression containing values of data types int and double, C++ **promotes** int operands to double values. In our example, we are treating total as a double (by using the unary cast operator), so the compiler promotes gradeCounter to double, allowing the calculation to be performed—the result of the floating-point division is assigned to average. In Chapter 6, Functions and an Introduction to Recursion, we discuss all the fundamental data types and their order of promotion.

Cast operators are available for use with every data type and with class types as well. The static_cast operator is formed by following keyword static_cast with angle brackets (< and >) around a data-type name. The cast operator is a **unary operator**—an operator that takes only one operand. In Chapter 2, we studied the binary arithmetic operators. C++ also supports unary versions of the plus (+) and minus (-) operators, so that you can write such expressions as -7 or +5. Cast operators have higher precedence than other unary operators, such as unary + and unary -. This precedence is higher than that of the **multiplicative operators** *, / and %, and lower than that of parentheses. We indicate the cast operator with the notation static_cast<*type*>() in our precedence charts.

Formatting for Floating-Point Numbers

The formatting capabilities in Fig. 4.13 are discussed here briefly and explained in depth in Chapter 15, Stream Input/Output. The call to **setprecision** in line 79 (with an argument of 2) indicates that double variable average should be printed with two digits of **precision** to the right of the decimal point (e.g., 92.37). This call is referred to as a **parameterized stream manipulator** (because of the 2 in parentheses). Programs that use these calls must contain the preprocessor directive (line 5)

```
#include <iomanip>
```

The manipulator endl is a **nonparameterized stream manipulator** (because it isn't followed by a value or expression in parentheses) and does not require the <iomanip> header. If the precision is not specified, floating-point values are normally output with *six* digits of precision (i.e., the **default precision** on most 32-bit systems today), although we'll see an exception to this in a moment.

The stream manipulator **fixed** (line 79) indicates that floating-point values should be output in so-called **fixed-point format**, as opposed to **scientific notation**. Scientific notation is a way of displaying a number as a floating-point number between the values of 1.0 and 10.0, multiplied by a power of 10. For instance, the value 3,100.0 would be displayed in scientific notation as 3.1×10^3. Scientific notation is useful when displaying values that are very large or very small. Formatting using scientific notation is discussed further in Chapter 15. Fixed-point formatting, on the other hand, is used to force a floating-point number to display a specific number of digits. Specifying fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole number amount, such as 88.00. Without the fixed-point formatting option, such a value prints in C++ as 88 without the trailing zeros and without the decimal point. When the stream manipulators `fixed` and `setprecision` are used in a program, the *printed* value is **rounded** to the number of decimal positions indicated by the value passed to `setprecision` (e.g., the value `2` in line 79), although the value in memory remains unaltered. For example, the values 87.946 and 67.543 are output as 87.95 and 67.54, respectively. It's also possible to *force* a decimal point to appear by using stream manipulator **`showpoint`**. If `showpoint` is specified without `fixed`, then trailing zeros will not print. Like `endl`, stream manipulators `fixed` and `showpoint` do not use parameters, nor do they require the `<iomanip>` header. Both can be found in header `<iostream>`.

Lines 79 and 80 of Fig. 4.13 output the class average *rounded* to the nearest hundredth and with exactly two digits to the right of the decimal point. The parameterized stream manipulator (line 79) indicates that variable `average`'s value should be displayed with two digits of precision to the right of the decimal point—indicated by `setprecision( 2 )`. The three grades entered during the sample execution of the program in Fig. 4.14 total 257, which yields the average 85.666666….

4.10 Formulating Algorithms: Nested Control Statements

For the next example, we once again formulate an algorithm by using pseudocode and top-down, stepwise refinement, and write a corresponding C++ program. We've seen that control statements can be stacked on top of one another (in sequence). Here, we examine the only other structured way control statements can be connected, namely, by **nesting** one control statement within another. Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results indicating the number of students who passed and the number who failed.*
> 4. *If more than eight students passed the exam, print the message "Bonus to instructor!"*

After reading the problem statement carefully, we make the following observations:

1. The program must process test results for 10 students. A *counter-controlled loop* can be used because the number of test results is known in advance.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine whether the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2. (Exercise 4.20 considers the consequences of this assumption.)
3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.

4. After the program has processed all the results, it must decide whether more than eight students passed the exam.

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

Analyze exam results and decide whether a bonus should be paid

Once again, it's important to emphasize that the top is a *complete* representation of the program, but several refinements are likely to be needed before the pseudocode evolves naturally into a C++ program.

Our first refinement is

Initialize variables
Input the 10 exam results, and count passes and failures
Display a summary of the exam results and decide whether a bonus should be paid

Here, too, even though we have a *complete* representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input. The last variable is *not* initialized, because its value is read from the user during each iteration of the loop.

The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

Notice that only the counters are initialized at the start of the algorithm.

The pseudocode statement

Input the 10 exam results, and count passes and failures

requires a loop that successively inputs the result of each exam. Here it's known in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., **nested** within the loop), an `if...else` statement will determine whether each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter
```

We use blank lines to isolate the *If...Else* control structure, which improves readability.

The pseudocode statement

Display a summary of the exam results and decide whether a bonus should be paid

can be refined as follows:

Display the number of passes
Display the number of failures

If more than eight students passed
Display "Bonus to instructor!"

The complete second refinement appears in Fig. 4.15. Blank lines set off the *While* structure for readability. This pseudocode is now sufficiently refined for conversion to C++.

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Display the number of passes
Display the number of failures

If more than eight students passed
    Display "Bonus to instructor!"
```

Fig. 4.15 | Pseudocode for examination-results problem.

Conversion to Class Analysis

The program that implements the pseudocode algorithm is shown in Fig. 4.16. This example does not contain a class—it contains just a source code file with function `main` performing all the application's work. In this chapter and in Chapter 3, you've seen examples consisting of one class (including the header and source code files for this class), as well as another source code file testing the class. This source code file contained function `main`, which created an object of the class and called its member functions. Occasionally, when it does not make sense to try to create a *reusable* class to demonstrate a concept, we'll use an example contained entirely within the `main` function of a *single* source code file.

Lines 9–12 declare the variables used to process the examination results. We've taken advantage of a feature of C++ that allows variable initialization to be incorporated into

```cpp
// Fig. 4.16: fig04_16.cpp
// Examination-results problem: Nested control statements.
#include <iostream>
using namespace std;

int main()
{
   // initializing variables in declarations
   int passes = 0; // number of passes
   int failures = 0; // number of failures
   int studentCounter = 1; // student counter
   int result; // one exam result (1 = pass, 2 = fail)

   // process 10 students using counter-controlled loop
   while ( studentCounter <= 10 )
   {
      // prompt user for input and obtain value from user
      cout << "Enter result (1 = pass, 2 = fail): ";
      cin >> result; // input result

      // if...else nested in while
      if ( result == 1 )          // if result is 1,
         passes = passes + 1;     // increment passes;
      else                        // else result is not 1, so
         failures = failures + 1; // increment failures

      // increment studentCounter so loop eventually terminates
      studentCounter = studentCounter + 1;
   } // end while

   // termination phase; display number of passes and failures
   cout << "Passed " << passes << "\nFailed " << failures << endl;

   // determine whether more than eight students passed
   if ( passes > 8 )
      cout << "Bonus to instructor!" << endl;
} // end main
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4
```

Fig. 4.16 | Examination-results problem: Nested control statements. (Part 1 of 2.)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

Fig. 4.16 | Examination-results problem: Nested control statements. (Part 2 of 2.)

declarations (`passes` is initialized to `0`, `failures` is initialized to `0` and `studentCounter` is initialized to `1`). Looping programs may require initialization at the beginning of *each* repetition; such reinitialization normally would be performed by assignment statements rather than in declarations or by moving the declarations inside the loop bodies.

The `while` statement (lines 15–29) loops 10 times. Each iteration inputs and processes one exam result. The `if...else` statement (lines 22–25) for processing each result is nested in the `while` statement. If the `result` is `1`, the `if...else` statement increments `passes`; otherwise, it assumes the `result` is `2` and increments `failures`. Line 28 increments `studentCounter` before the loop condition is tested again in line 15. After 10 values have been input, the loop terminates and line 32 displays the number of `passes` and the number of `failures`. The `if` statement in lines 35–36 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

Figure 4.16 shows the input and output from two sample executions of the program. At the end of the second sample execution, the condition in line 35 is true—more than eight students passed the exam, so the program outputs a message indicating that the instructor should receive a bonus.

4.11 Assignment Operators

C++ provides several **assignment operators** for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the **addition assignment operator `+=`** as

```
c += 3;
```

which adds the value of the expression on the operator's right to the value of the variable on the operator's left and stores the result in the left-side variable. Any statement of the form

variable = *variable operator expression*;

in which the same *variable* appears on both sides of the assignment operator and *operator* is one of the binary operators `+`, `-`, `*`, `/`, or `%` (or a few others we'll discuss later in the text), can be written in the form

variable operator= *expression*;

Thus the assignment `c += 3` adds `3` to `c`. Figure 4.17 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 4.17 | Arithmetic assignment operators.

4.12 Increment and Decrement Operators

In addition to the arithmetic assignment operators, C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary **increment operator**, `++`, and the unary **decrement operator**, `--`, which are summarized in Fig. 4.18. A program can increment by 1 the value of a variable called `c` using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. An increment or decrement operator that's prefixed to (placed *before*) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed *after*) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postincrement	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	predecrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postdecrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

Fig. 4.18 | Increment and decrement operators.

Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **preincrementing** (or **predecrementing**) the variable. Preincrementing (or predecrementing) causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **postincrementing** (or **postdecrementing**) the variable. Postincrementing (or postdecrementing) causes the *current* value of the variable to be used in the expression in which it appears, then the variable's value is incremented (decremented) by 1.

Good Programming Practice 4.5

Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

Figure 4.19 demonstrates the difference between the prefix increment and postfix increment versions of the ++ increment operator. The decrement operator (--) works similarly.

```
1  // Fig. 4.19: fig04_19.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int c;
9
10    // demonstrate postincrement
11    c = 5; // assign 5 to c
12    cout << c << endl; // print 5
13    cout << c++ << endl; // print 5 then postincrement
14    cout << c << endl; // print 6
15
16    cout << endl; // skip a line
17
18    // demonstrate preincrement
19    c = 5; // assign 5 to c
20    cout << c << endl; // print 5
21    cout << ++c << endl; // preincrement then print 6
22    cout << c << endl; // print 6
23 } // end main
```

```
5
5
6

5
6
6
```

Fig. 4.19 | Preincrementing and postincrementing.

Line 11 initializes c to 5, and line 12 outputs c's initial value. Line 13 outputs the value of the expression c++. This postincrements the variable c, so c's original value (5) is output, then c's value is incremented. Thus, line 13 outputs c's initial value (5) again. Line 14 outputs c's new value (6) to prove that the variable's value was incremented in line 13.

Line 19 resets c's value to 5, and line 20 outputs that value. Line 21 outputs the value of the expression ++c. This expression preincrements c, so its value is incremented, then the new value (6) is output. Line 22 outputs c's value again to show that the value of c is still 6 after line 21 executes.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. The three assignment statements in Fig. 4.16:

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

When you increment (++) or decrement (--) an integer variable in a statement by itself, the preincrement and postincrement forms have the same logical effect, and the predecrement and postdecrement forms have the same logical effect. It's only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

Common Programming Error 4.10

Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference, e.g., writing `++(x + 1)`*, is a syntax error.*

Figure 4.20 shows the precedence and associativity of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column indicates the associativity of the operators at each level of precedence. Notice that the conditional operator (?:), the unary operators preincrement (++), predecrement (--), plus (+) and minus (-), and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. All other operators in Fig. 4.20 associate from left to right. The third column names the various groups of operators.

Operators	Associativity	Type
`::`	left to right	scope resolution
`()`	*[See caution in Fig. 2.10]*	grouping parentheses

Fig. 4.20 | Operator precedence for the operators encountered so far in the text. (Part 1 of 2.)

Operators	Associativity	Type
`++ -- static_cast<type>()`	left to right	unary (postfix)
`++ -- + -`	right to left	unary (prefix)
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`<< >>`	left to right	insertion/extraction
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 4.20 | Operator precedence for the operators encountered so far in the text. (Part 2 of 2.)

4.13 Wrap-Up

This chapter presented basic problem-solving techniques that you use in building classes and developing member functions for these classes. We demonstrated how to construct an algorithm (i.e., an approach to solving a problem) in pseudocode, then how to refine the algorithm through pseudocode development, resulting in C++ code that can be executed as part of a function. You learned how to use top-down, stepwise refinement to plan out the actions that a function must perform and the order in which it must perform them.

You learned that only three types of control structures—sequence, selection and repetition—are needed to develop any algorithm. We demonstrated two of C++'s selection statements—the `if` single-selection statement and the `if...else` double-selection statement. The `if` statement is used to execute a set of statements based on a condition—if the condition is true, the statements execute; if it isn't, the statements are skipped. The `if...else` double-selection statement is used to execute one set of statements if a condition is true, and another set of statements if the condition is false. We then discussed the `while` repetition statement, where a set of statements are executed repeatedly as long as a condition is true. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced assignment operators, which can be used for abbreviating statements. We presented the increment and decrement operators, which can be used to add or subtract the value 1 from a variable. In the next chapter, we continue our discussion of control statements, introducing the `for`, `do...while` and `switch` statements.

Summary

Section 4.2 Algorithms

- An algorithm (p. 102) is a procedure for solving a problem in terms of the actions to execute and the order in which to execute them.
- Specifying the order in which statements execute in a program is called program control (p. 103).

Section 4.3 Pseudocode

- Pseudocode (p. 103) helps you think out a program before writing it in a programming language.

Section 4.4 Control Structures

- An activity diagram models the workflow (also called the activity, p. 105) of a software system.
- Activity diagrams (p. 105) are composed of symbols, such as action state symbols, diamonds and small circles, that are connected by transition arrows representing the flow of the activity.
- Like pseudocode, activity diagrams help you develop and represent algorithms.
- An action state is represented as a rectangle with its left and right sides replaced with arcs curving outward. The action expression (p. 105) appears inside the action state.
- The arrows in an activity diagram represent transitions (p. 105), which indicate the order in which the actions represented by action states occur.
- The solid circle in an activity diagram represents the initial state (p. 105)—the beginning of the workflow before the program performs the modeled actions.
- The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the final state (p. 105)—the end of the workflow after the program performs its actions.
- Rectangles with the upper-right corners folded over are called notes (p. 105) in the UML. A dotted line (p. 105) connects each note with the element that the note describes.
- A decision symbol (p. 107) in an activity diagram indicates that a decision is to be made. The workflow follows a path determined by the associated guard conditions. Each transition arrow emerging from a decision symbol has a guard condition. If a guard condition is true, the workflow enters the action state to which the transition arrow points.
- There are three types of control structures (p. 104)—sequence, selection and repetition.
- The sequence structure is built in—by default, statements execute in the order they appear.
- A selection structure chooses among alternative courses of action.

Section 4.5 `if` Selection Statement

- The `if` single-selection statement (p. 107) either performs (selects) an action if a condition is true, or skips the action if the condition is false.

Section 4.6 `if...else` Double-Selection Statement

- The `if...else` double-selection statement (p. 108) performs (selects) an action if a condition is true and performs a different action if the condition is false.
- To include several statements in an `if`'s body (or the body of an `else` for an `if...else` statement), enclose the statements in braces (`{` and `}`). A set of statements contained in braces is called a block (p. 112). A block can be placed anywhere in a program that a single statement can be placed.
- A null statement (p. 113), indicating that no action is to be taken, is indicated by a semicolon (`;`).

Section 4.7 `while` Repetition Statement

- A repetition statement (p. 113) repeats an action while some condition remains true.
- A UML merge symbol (p. 114) has two or more transition arrows pointing to the diamond and only one pointing from it, to indicate multiple activity flows merging to continue the activity.

Section 4.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition (p. 115) is used when the number of repetitions is known before a loop begins executing, i.e., when there is definite repetition.

- The stream manipulator `fixed` (p. 129) indicates that floating-point values should be output in so-called fixed-point format, as opposed to scientific notation.

Section 4.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Top-down, stepwise refinement (p. 121) is a process for refining pseudocode by maintaining a complete representation of the program during each refinement.
- Sentinel-controlled repetition (p. 122) is used when the number of repetitions is not known before a loop begins executing, i.e., when there is indefinite repetition.
- A value that contains a fractional part is referred to as a floating-point number and is represented approximately by data types such as `float` and `double` (p. 124).
- The unary cast operator `static_cast<double>` (p. 124) can be used to create a temporary floating-point copy of its operand.
- Unary operators (p. 129) take only one operand; binary operators take two.
- The parameterized stream manipulator `setprecision` (p. 129) indicates the number of digits of precision that should be displayed to the right of the decimal point.

Section 4.10 Formulating Algorithms: Nested Control Statements

- A nested control statement (p. 130) appears in the body of another control statement.

Section 4.11 Assignment Operators

- The arithmetic operators `+=`, `-=`, `*=`, `/=` and `%=` abbreviate assignment expressions (p. 134).

Section 4.12 Increment and Decrement Operators

- The increment (`++`) and decrement (`--`) operators (p. 135) increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, then its new value is used in the expression in which it appears. If the operator is postfixed to the variable, the variable is first used in the expression in which it appears, then the variable's value is incremented or decremented by 1.

Self-Review Exercises

4.1 Answer each of the following questions.

a) All programs can be written in terms of three types of control structures: ________, ________ and ________.

b) The ________ selection statement is used to execute one action when a condition is `true` or a different action when that condition is `false`.

c) Repeating a set of instructions a specific number of times is called __________ repetition.

d) When it isn't known in advance how many times a set of statements will be repeated, a(n) ________ value can be used to terminate the repetition.

4.2 Write four different C++ statements that each add 1 to integer variable `x`.

4.3 Write C++ statements to accomplish each of the following:

a) In one statement, assign the sum of the current value of `x` and `y` to `z` and postincrement the value of `x`.

b) Determine whether the value of the variable `count` is greater than 10. If it is, print `"Count is greater than 10."`

c) Predecrement the variable `x` by 1, then subtract it from the variable `total`.

d) Calculate the remainder after `q` is divided by `divisor` and assign the result to `q`. Write this statement two different ways.

4.4 Write C++ statements to accomplish each of the following tasks.

a) Declare variables `sum` and `x` to be of type `int`.
b) Set variable `x` to `1`.
c) Set variable `sum` to `0`.
d) Add variable `x` to variable `sum` and assign the result to variable `sum`.
e) Print `"The sum is: "` followed by the value of variable `sum`.

4.5 Combine the statements that you wrote in Exercise 4.4 into a program that calculates and prints the sum of the integers from 1 to 10. Use the `while` statement to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

4.6 State the values of *each* of these `int` variables after the calculation is performed. Assume that, when each statement begins executing, all variables have the integer value 5.

a) `product *= x++;`
b) `quotient /= ++x;`

4.7 Write single C++ statements or portions of statements that do the following:

a) Input integer variable `x` with `cin` and `>>`.
b) Input integer variable `y` with `cin` and `>>`.
c) Set integer variable `i` to `1`.
d) Set integer variable `power` to `1`.
e) Multiply variable `power` by `x` and assign the result to `power`.
f) Preincrement variable `i` by `1`.
g) Determine whether `i` is less than or equal to `y`.
h) Output integer variable `power` with `cout` and `<<`.

4.8 Write a C++ program that uses the statements in Exercise 4.7 to calculate `x` raised to the `y` power. The program should have a `while` repetition statement.

4.9 Identify and correct the errors in each of the following:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```
b)
```
cin << value;
```
c)
```
if ( gender == 1 )
   cout << "Woman" << endl;
else;
   cout << "Man" << endl;
```

4.10 What's wrong with the following `while` repetition statement?

```
while ( z >= 0 )
   sum += z;
```

Answers to Self-Review Exercises

4.1 a) Sequence, selection and repetition. b) `if...else`. c) Counter-controlled or definite. d) Sentinel, signal, flag or dummy.

4.2
```
x = x + 1;
x += 1;
++x;
x++;
```

4.3 a) `z = x++ + y;`

b) `if ( count > 10 )`
```
   cout << "Count is greater than 10" << endl;
```
c) `total -= --x;`

d) `q %= divisor;`
`q = q % divisor;`

4.4 a) `int sum;`
`int x;`

b) `x = 1;`

c) `sum = 0;`

d) `sum += x;`
or
`sum = sum + x;`

e) `cout << "The sum is: " << sum << endl;`

4.5 See the following code:

```
// Exercise 4.5 Solution: ex04_05.cpp
// Calculate the sum of the integers from 1 to 10.
#include <iostream>
using namespace std;

int main()
{
   int sum; // stores sum of integers 1 to 10
   int x; // counter

   x = 1; // count from 1
   sum = 0; // initialize sum

   while ( x <= 10 ) // loop 10 times
   {
      sum += x; // add x to sum
      ++x; // increment x
   } // end while

   cout << "The sum is: " << sum << endl;
} // end main
```

```
The sum is: 55
```

4.6 a) `product = 25, x = 6;`

b) `quotient = 0, x = 6;`

```
// Exercise 4.6 Solution: ex04_06.cpp
// Calculate the value of product and quotient.
#include <iostream>
using namespace std;

int main()
{
   int x = 5;
   int product = 5;
   int quotient = 5;

   // part a
   product *= x++; // part a statement
```

```
    cout << "Value of product after calculation: " << product << endl;
    cout << "Value of x after calculation: " << x << endl << endl;

    // part b
    x = 5; // reset value of x
    quotient /= ++x; // part b statement
    cout << "Value of quotient after calculation: " << quotient << endl;
    cout << "Value of x after calculation: " << x << endl << endl;
} // end main
```

```
Value of product after calculation: 25
Value of x after calculation: 6

Value of quotient after calculation: 0
Value of x after calculation: 6
```

4.7 a) `cin >> x;`
b) `cin >> y;`
c) `i = 1;`
d) `power = 1;`
e) `power *= x;`
or
`power = power * x;`
f) `++i;`
g) `if ( i <= y )`
h) `cout << power << endl;`

4.8 See the following code:

```
// Exercise 4.8 Solution: ex04_08.cpp
// Raise x to the y power.
#include <iostream>
using namespace std;

int main()
{
   int x; // base
   int y; // exponent
   int i; // counts from 1 to y
   int power; // used to calculate x raised to power y

   i = 1; // initialize i to begin counting from 1
   power = 1; // initialize power

   cout << "Enter base as an integer: ";  // prompt for base
   cin >> x; // input base

   cout << "Enter exponent as an integer: "; // prompt for exponent
   cin >> y; // input exponent

   // count from 1 to y and multiply power by x each time
   while ( i <= y )
   {
      power *= x;
      ++i;
   } // end while

   cout << power << endl; // display result
} // end main
```

```
Enter base as an integer: 2
Enter exponent as an integer: 3
8
```

4.9 a) *Error:* Missing the closing right brace of the `while` body.
Correction: Add closing right brace after the statement `c++;`.
b) *Error:* Used stream insertion instead of stream extraction.
Correction: Change << to >>.
c) *Error:* Semicolon after `else` results in a logic error. The second output statement will always be executed.
Correction: Remove the semicolon after `else`.

4.10 The value of the variable `z` is never changed in the `while` statement. Therefore, if the loop-continuation condition `(z >= 0)` is initially `true`, an infinite loop is created. To prevent the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

Exercises

4.11 *(Correct the Code Errors)* Identify and correct the error(s) in each of the following:

a)
```
if ( age >= 65 );
   cout << "Age is greater than or equal to 65" << endl;
else
   cout << "Age is less than 65 << endl";
```
b)
```
if ( age >= 65 )
   cout << "Age is greater than or equal to 65" << endl;
else;
   cout << "Age is less than 65 << endl";
```
c)
```
int x = 1, total;

while ( x <= 10 )
{
   total += x;
   ++x;
}
```
d)
```
While ( x <= 100 )
   total += x;
   ++x;
```
e)
```
while ( y > 0 )
{
   cout << y << endl;
   ++y;
}
```

4.12 *(What Does this Program Do?)* What does the following program print?

```
// Exercise 4.12: ex04_12.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
   int y; // declare y
```

```
   int x = 1; // initialize x
   int total = 0; // initialize total

   while ( x <= 10 ) // loop 10 times
   {
      y = x * x; // perform calculation
      cout << y << endl; // output result
      total += y; // add y to total
      x++; // increment counter x
   } // end while

   cout << "Total is " << total << endl; // display result
} // end main
```

For Exercises 4.13–4.16, perform each of these steps:

a) Read the problem statement.
b) Formulate the algorithm using pseudocode and top-down, stepwise refinement.
c) Write a C++ program.
d) Test, debug and execute the C++ program.

4.13 *(Gas Mileage)* Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several trips by recording miles driven and gallons used for each trip. Develop a C++ program that uses a `while` statement to input the miles driven and gallons used for each trip. The program should calculate and display the miles per gallon obtained for each trip and print the combined miles per gallon obtained for all tankfuls up to this point.

```
Enter miles driven (-1 to quit): 287
Enter gallons used: 13
MPG this trip: 22.076923
Total MPG: 22.076923

Enter miles driven (-1 to quit): 200
Enter gallons used: 10
MPG this trip: 20.000000
Total MPG: 21.173913

Enter the miles driven (-1 to quit): 120
Enter gallons used: 5
MPG this trip: 24.000000
Total MPG: 21.678571

Enter the miles used (-1 to quit): -1
```

4.14 *(Credit Limits)* Develop a C++ program that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) Account number (an integer)
b) Balance at the beginning of the month
c) Total of all items charged by this customer this month
d) Total of all credits applied to this customer's account this month
e) Allowed credit limit

The program should use a `while` statement to input each of these facts, calculate the new balance (= beginning balance + charges – credits) and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit Limit Exceeded."

```
Enter account number (or -1 to quit): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
New balance is 5894.78
Account:      100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter Account Number (or -1 to quit): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
New balance is 802.45

Enter Account Number (or -1 to quit): -1
```

4.15 *(Sales Commission Calculator)* A large company pays its salespeople on a commission basis. The salespeople each receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of chemicals in a week receives $200 plus 9% of $5000, or a total of $650. Develop a C++ program that uses a `while` statement to input each salesperson's gross sales for last week and calculates and displays that salesperson's earnings. Process one salesperson's figures at a time.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

4.16 *(Salary Calculator)* Develop a C++ program that uses a `while` statement to determine the gross pay for each of several employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time-and-a-half" for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee's gross pay.

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the employee ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the employee ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the employee ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```

4.17 *(Find the Largest)* The process of finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest inputs the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a C++ program that uses a `while` statement to determine and print the largest number of 10 numbers input by the user. Your program should use three variables, as follows:

`counter:`	A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
`number:`	The current number input to the program.
`largest:`	The largest number found so far.

4.18 *(Tabular Output)* Write a C++ program that uses a `while` statement and the tab escape sequence `\t` to print the following table of values:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.19 *(Find the Two Largest Numbers)* Using an approach similar to that in Exercise 4.17, find the *two* largest values among the 10 numbers. [*Note:* You must input each number only once.]

4.20 *(Validating User Input)* The examination-results program of Fig. 4.16 assumes that any value input by the user that's not a 1 must be a 2. Modify the application to validate its inputs. On any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

4.21 *(What Does this Program Do?)* What does the following program print?

```cpp
// Exercise 4.21: ex04_21.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
   int count = 1; // initialize count

   while ( count <= 10 ) // loop 10 times
   {
      // output line of text
      cout << ( count % 2 ? "****" : "++++++++" ) << endl;
      ++count; // increment count
   } // end while
} // end main
```

4.22 *(What Does this Program Do?)* What does the following program print?

```cpp
// Exercise 4.22: ex04_22.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
```

```
   int row = 10; // initialize row
   int column; // declare column

   while ( row >= 1 ) // loop until row < 1
   {
      column = 1; // set column to 1 as iteration begins

      while ( column <= 10 ) // loop 10 times
      {
         cout << ( row % 2 ? "<" : ">" ); // output
         ++column; // increment column
      } // end inner while

      --row; // decrement row
      cout << endl; // begin new output line
   } // end outer while
} // end main
```

4.23 *(Dangling-`else` Problem)* State the output for each of the following when `x` is `9` and `y` is `11` and when `x` is `11` and `y` is `9`. The compiler ignores the indentation in a C++ program. The C++ compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{}`. On first glance, you may not be sure which `if` and `else` match, so this is referred to as the "dangling-`else`" problem. We eliminated the indentation from the following code to make the problem more challenging. [*Hint:* Apply indentation conventions you've learned.]

a)
```
if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
cout << "*****" << endl;
}
else
{
cout << "#####" << endl;
cout << "$$$$$" << endl;
}
```

4.24 *(Another Dangling-else Problem)* Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. The compiler ignores indentation in a C++ program. We eliminated the indentation from the following code to make the problem more challenging. [*Note:* It's possible that no modification is necessary.]

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

a) Assuming `x = 5` and `y = 8`, the following output is produced.

```
@@@@@
$$$$$
&&&&&
```

b) Assuming `x = 5` and `y = 8`, the following output is produced.

```
@@@@@
```

c) Assuming `x = 5` and `y = 8`, the following output is produced.

```
@@@@@
&&&&&
```

d) Assuming `x = 5` and `y = 7`, the following output is produced. [*Note:* The last three output statements after the `else` are all part of a block.]

```
#####
$$$$$
&&&&&
```

4.25 *(Square of Asterisks)* Write a program that reads in the size of the side of a square then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

```
*****
*   *
*   *
*   *
*****
```

4.26 *(Palindromes)* A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it's a palindrome. [*Hint:* Use the division and modulus operators to separate the number into its individual digits.]

4.27 *(Printing the Decimal Equivalent of a Binary Number)* Input an integer containing only 0s and 1s (i.e., a "binary" integer) and print its decimal equivalent. Use the modulus and division operators to pick off the "binary" number's digits one at a time from right to left. Much as in the decimal number system, where the rightmost digit has a positional value of 1, the next digit left has a positional value of 10, then 100, then 1000, and so on, in the binary number system the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as 2 * 100 + 3 * 10 + 4 * 1. The decimal equivalent of binary 1101 is 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 or 1 + 0 + 4 + 8, or 13. [*Note:* To learn more about binary numbers, refer to Appendix D.]

4.28 *(Checkerboard Pattern of Asterisks)* Write a program that displays the following checkerboard pattern. Your program must use only three output statements, one of each of the following forms:

```
cout << "* ";
cout << ' ';
cout << endl;
```

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

4.29 ***(Multiples of 2 with an Infinite Loop)*** Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your `while` loop should not terminate (i.e., you should create an infinite loop). To do this, simply use the keyword `true` as the expression for the `while` statement. What happens when you run this program?

4.30 ***(Calculating a Circle's Diameter, Circumference and Area)*** Write a program that reads the radius of a circle (as a `double` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π.

4.31 What's wrong with the following statement? Provide the correct statement to accomplish what the programmer was probably trying to do.

```
cout << ++( x + y );
```

4.32 ***(Sides of a Triangle)*** Write a program that reads three nonzero `double` values and determines and prints whether they could represent the sides of a triangle.

4.33 ***(Sides of a Right Triangle)*** Write a program that reads three nonzero integers and determines and prints whether they're the sides of a right triangle.

4.34 ***(Factorial)*** The factorial of a nonnegative integer n is written $n!$ (pronounced "n factorial") and is defined as follows:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (for values of n greater than 1)

and

$n! = 1$ (for $n = 0$ or $n = 1$).

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120. Use `while` statements in each of the following:

a) Write a program that reads a nonnegative integer and computes and prints its factorial.
b) Write a program that estimates the value of the mathematical constant e by using the formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

c) Write a program that computes the value of e^x by using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

Making a Difference

4.35 ***(Enforcing Privacy with Cryptography)*** The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise you'll investigate a simple scheme for encrypting and decrypting data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and encrypt it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and decrypts it (by reversing the encryption scheme) to form the original number. [*Optional reading project:* Research "public key cryptography" in general and the PGP (Pretty Good Privacy) specific public key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.]

4.36 ***(World Population Growth)*** World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable cropland and other limited resources. There is evidence that growth has been slowing in recent years and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues online. *Be sure to investigate various viewpoints.* Get estimates for the current world population and its growth rate (the percentage by which it is likely to increase this year). Write a program that calculates world population growth each year for the next 75 years, *using the simplifying assumption that the current growth rate will stay constant*. Print the results in a table. The first column should display the year from year 1 to year 75. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the year in which the population would be double what it is today, if this year's growth rate were to persist.

5 Control Statements: Part 2

Not everything that can be counted counts, and not every thing that counts can be counted.
—Albert Einstein

Who can control his fate?
—William Shakespeare

The used key is always bright.
—Benjamin Franklin

Intelligence … is the faculty of making artificial objects, especially tools to make tools.
—Henri Bergson

Objectives

In this chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use `for` and `do`…`while` to execute statements in a program repeatedly.
- To implement multiple selection using the `switch` selection statement.
- How `break` and `continue` alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.

Outline

5.1 Introduction

In this chapter, we continue our presentation of structured programming by introducing C++'s remaining control statements. The control statements we study here and those you learned in Chapter 4 will help you build and manipulate objects. We continue our early emphasis on object-oriented programming that began with a discussion of basic concepts in Chapter 1 and extensive object-oriented code examples and exercises in Chapters 3–4.

In this chapter, we demonstrate the `for`, `do...while` and `switch` statements. Through short examples using `while` and `for`, we explore counter-controlled repetition. We expand the `GradeBook` class to use a `switch` statement to count the number of A, B, C, D and F grades in a set of letter grades entered by the user. We introduce the `break` and `continue` program control statements. We discuss the logical operators, which enable you to use more powerful conditional expressions. We also examine the common error of confusing the equality (`==`) and assignment (`=`) operators, and how to avoid it.

5.2 Essentials of Counter-Controlled Repetition

This section uses the `while` repetition statement to formalize the elements required to perform counter-controlled repetition. Counter-controlled repetition requires

1. the **name of a control variable** (or loop counter)
2. the **initial value** of the control variable
3. the **loop-continuation condition** that tests for the **final value** of the control variable (i.e., whether looping should continue)
4. the **increment** (or **decrement**) by which the control variable is modified each time through the loop.

The program in Fig. 5.1 prints the numbers from 1 to 10. The declaration in line 8 *names* the control variable (`counter`), declares it to be an integer, reserves space for it in memory and sets it to an *initial value* of `1`. Declarations that require initialization are *executable* statements. In C++, it's more precise to call a declaration that also reserves memory a **definition**. Because definitions are declarations, too, we'll use the term "declaration" except when the distinction is important.

Variable `counter` (line 8) also could have been declared and initializeed with

```
int counter; // declare control variable
counter = 1; // initialize control variable to 1
```

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int counter = 1; // declare and initialize control variable
9
10    while ( counter <= 10 ) // loop-continuation condition
11    {
12       cout << counter << " ";
13       ++counter; // increment control variable by 1
14    } // end while
15
16    cout << endl; // output a newline
17 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.1 | Counter-controlled repetition.

Line 13 *increments* the loop counter by 1 each time the loop's body is performed. The loop-continuation condition (line 10) in the `while` statement determines whether the value of the control variable is less than or equal to `10` (the final value for which the condition is `true`). The body of this `while` executes even when the control variable is `10`. The loop terminates when the control variable is greater than `10` (i.e., when `counter` is `11`).

Figure 5.1 can be made more concise by initializing `counter` to `0` and by replacing the `while` statement with

```
   while ( ++counter <= 10 ) // loop-continuation condition
      cout << counter << " ";
```

This code saves a statement, because the incrementing is done in the `while` condition before the condition is tested. Also, the code eliminates the braces around the body of the `while`, because the `while` now contains only *one* statement. Coding in such a condensed fashion can lead to programs that are more difficult to read, debug, modify and maintain.

Common Programming Error 5.1

Floating-point values are approximate, so controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.

Error-Prevention Tip 5.1

Control counting loops with integer values.

Good Programming Practice 5.1

Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of indentation.

5.3 for Repetition Statement

In addition to `while`, C++ provides the **for repetition statement**, which specifies the counter-controlled repetition details in a single line of code. To illustrate the power of `for`, let's rewrite the program of Fig. 5.1. The result is shown in Fig. 5.2.

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // for statement header includes initialization,
9     // loop-continuation condition and increment.
10    for ( int counter = 1; counter <= 10; ++counter )
11       cout << counter << " ";
12
13    cout << endl; // output a newline
14 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.2 | Counter-controlled repetition with the `for` statement.

When the `for` statement (lines 10–11) begins executing, the control variable `counter` is declared and initialized to 1. Then, the loop-continuation condition (line 10 between the semicolons) `counter <= 10` is checked. The initial value of `counter` is 1, so the condition is satisfied and the body statement (line 11) prints the value of `counter`, namely 1. Then, the expression `++counter` increments control variable `counter` and the loop begins again with the loop-continuation test. The control variable is now equal to 2, so the final value is not exceeded and the program performs the body statement again. This process continues until the loop body has executed 10 times and the control variable `counter` is incremented to 11—this causes the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the `for` statement (in this case, the output statement in line 13).

for Statement Header Components

Figure 5.3 takes a closer look at the `for` statement header (line 10) of Fig. 5.2. Notice that the `for` statement header "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable. If there's more than one statement in the body of the `for`, braces are required to enclose the body of the loop.

If you incorrectly wrote `counter < 10` as the loop-continuation condition in Fig. 5.2, then the loop would execute only 9 times. This is a common **off-by-one error**.

Common Programming Error 5.2

Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a `while` or `for` statement can cause off-by-one errors.

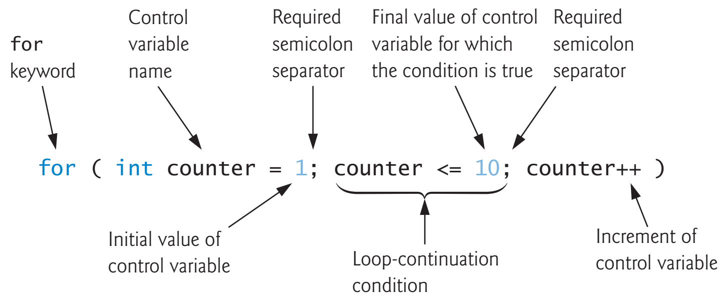


Fig. 5.3 | `for` statement header components.

Good Programming Practice 5.2

Using the final value in the condition of a `while` *or* `for` *statement and using the* `<=` *relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 10` *(which is an off-by-one error) or* `counter < 11` *(which is nevertheless correct). Many programmers prefer so-called* ***zero-based counting****, in which, to count 10 times through the loop,* `counter` *would be initialized to zero and the loop-continuation test would be* `counter < 10`*.*

The general form of the `for` statement is

```
for ( initialization; loopContinuationCondition; increment )
   statement
```

where the *initialization* expression initializes the loop's control variable, *loopContinuationCondition* determines whether the loop should continue executing and *increment* increments the control variable. In most cases, the `for` statement can be represented by an equivalent `while` statement, as follows:

```
initialization;

while ( loopContinuationCondition )
{
   statement
   increment;
}
```

There's an exception to this rule, which we'll discuss in Section 5.7.

If the *initialization* expression declares the control variable (i.e., its type is specified before its name), the control variable can be used *only* in the body of the `for` statement—the control variable will be *unknown* outside the `for` statement. This restricted use of the control variable name is known as the variable's **scope**. The scope of a variable specifies where it can be used in a program. Scope is discussed in detail in Chapter 6.

As we'll see, the *initialization* and *increment* expressions can be comma-separated lists of expressions. The commas, as used in these expressions, are **comma operators**, which guarantee that lists of expressions evaluate from left to right. The comma operator has the lowest precedence of all C++ operators. *The value and type of a comma-separated list of expressions is the value and type of the rightmost expression.* The comma operator is often used in `for` statements. Its primary application is to enable you to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single `for` statement that must be initialized and incremented.

Good Programming Practice 5.3

Place only expressions involving the control variables in the initialization and increment sections of a `for` statement.

The three expressions in the `for` statement header are optional (but the two semicolon separators are required). If the *loopContinuationCondition* is omitted, C++ assumes that the condition is true, thus creating an *infinite loop*. One might omit the *initialization* expression if the control variable is initialized earlier in the program. One might omit the *increment* expression if the increment is calculated by statements in the body of the `for` or if no increment is needed.

The increment expression in the `for` statement acts as a stand-alone statement at the end of the body of the `for`. Therefore, for integer counters, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are all equivalent in the *increment* expression (when no other code appears there). The integer variable being incremented here does not appear in a larger expression, so both preincrementing and postincrementing actually have the *same* effect.

Common Programming Error 5.3

Placing a semicolon immediately to the right of the right parenthesis of a `for` header makes the body of that `for` statement an empty statement. This is usually a logic error.

The initialization, loop-continuation condition and increment expressions of a `for` statement can contain arithmetic expressions. For example, if `x = 2` and `y = 10`, and `x` and `y` are not modified in the loop body, the `for` header

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to

```
for ( int j = 2; j <= 80; j += 5 )
```

The "increment" of a `for` statement can be negative, in which case it's really a *decrement* and the loop actually counts *downward* (as shown in Section 5.4).

If the loop-continuation condition is initially false, the body of the `for` statement is not performed. Instead, execution proceeds with the statement following the `for`.

Frequently, the control variable is printed or used in calculations in the body of a `for` statement, but this is not required. It's common to use the control variable for controlling repetition while never mentioning it in the body of the `for` statement.

Error-Prevention Tip 5.2

Although the value of the control variable can be changed in the body of a `for` statement, avoid doing so, because this practice can lead to subtle logic errors.

for Statement UML Activity Diagram

The `for` repetition statement's UML activity diagram is similar to that of the `while` statement (Fig. 4.6). Figure 5.4 shows the activity diagram of the `for` statement in Fig. 5.2. The diagram makes it clear that initialization occurs once *before* the loop-continuation test is evaluated the first time, and that incrementing occurs each time through the loop *after* the body statement executes. Note that (besides an initial state, transition arrows, a merge, a final state and several notes) the diagram contains only *action states* and a *decision*.

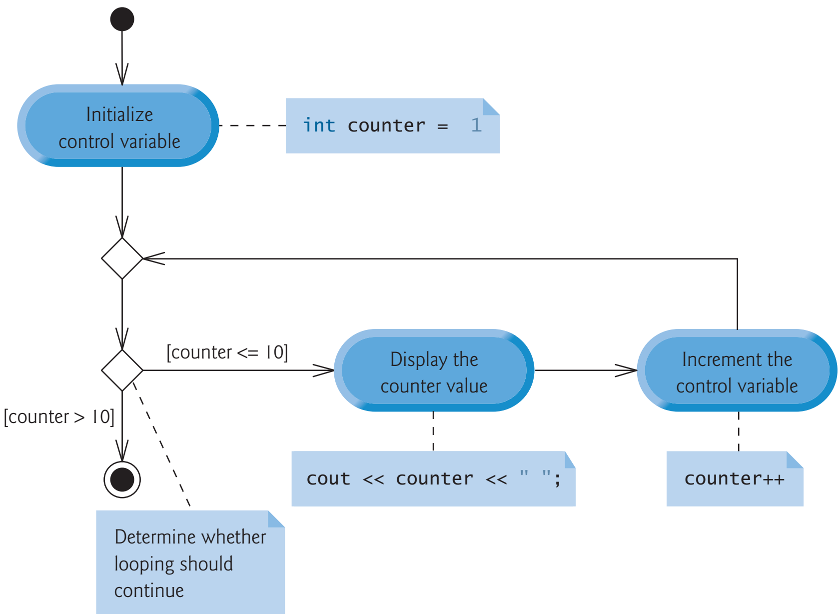


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

5.4 Examples Using the for Statement

The following examples show methods of varying the control variable in a `for` statement. In each case, we write the appropriate `for` statement header. Note the change in the relational operator for loops that *decrement* the control variable.

a) Vary the control variable from `1` to `100` in increments of `1`.

```
for ( int i = 1; i <= 100; ++i )
```

b) Vary the control variable from `100` down to `1` in decrements of `1`.

```
for ( int i = 100; i >= 1; --i )
```

c) Vary the control variable from `7` to `77` in steps of `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from `20` down to `2` in steps of `-2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.

```
for ( int i = 2; i <= 17; i += 3 )
```

f) Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55.

```
for ( int i = 99; i >= 55; i -= 11 )
```

Common Programming Error 5.4

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (such as incorrectly using `i <= 1` *instead of* `i >= 1` *in a loop counting down to 1) is a logic error that yields incorrect results when the program runs.*

Application: Summing the Even Integers from 2 to 20

The program of Fig. 5.5 uses a `for` statement to sum the even integers from 2 to 20. Each iteration of the loop (lines 11–12) adds control variable `number`'s value to variable `total`.

```
1  // Fig. 5.5: fig05_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int total = 0; // initialize total
9
10    // total even integers from 2 through 20
11    for ( int number = 2; number <= 20; number += 2 )
12       total += number;
13
14    cout << "Sum is " << total << endl; // display results
15 } // end main
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

The body of the `for` statement in Fig. 5.5 actually could be merged into the increment portion of the `for` header by using the *comma operator* as follows:

```
for ( int number = 2; // initialization
      number <= 20; // loop continuation condition
      total += number, number += 2 ) // total and increment
   ; // empty body
```

Good Programming Practice 5.4

Although statements preceding a `for` *and statements in the body of a* `for` *often can be merged into the* `for` *header, doing so can make the program more difficult to read, maintain, modify and debug.*

Application: Compound Interest Calculations

Consider the following problem statement:

> *A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:*
>
> $a = p\ (\ 1 + r\ )^n$
>
> *where*
>
> p *is the original amount invested (i.e., the principal),*
> r *is the annual interest rate,*
> n *is the number of years and*
> a *is the amount on deposit at the end of the* n*th year.*

The `for` statement (Fig. 5.6, lines 21–28) performs the indicated calculation for each of the 10 years the money remains on deposit, varying a control variable from 1 to 10 in increments of 1. C++ does *not* include an exponentiation operator, so we use the **standard library function `pow`** (line 24). The function `pow(x, y)` calculates the value of `x` raised to the `y`th power. In this example, the algebraic expression $(1 + r)^n$ is written as `pow(1.0 + rate, year)`, where variable `rate` represents r and variable `year` represents n. Function `pow` takes two arguments of type `double` and returns a `double` value.

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // standard C++ math library
6  using namespace std;
7  
8  int main()
9  {
10    double amount; // amount on deposit at end of each year
11    double principal = 1000.0; // initial amount before interest
12    double rate = .05; // interest rate
13 
14    // display headers
15    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
16 
17    // set floating-point number format
18    cout << fixed << setprecision( 2 );
19 
20    // calculate amount on deposit for each of ten years
21    for ( int year = 1; year <= 10; ++year )
22    {
23       // calculate new amount for specified year
24       amount = principal * pow( 1.0 + rate, year );
25 
26       // display the year and the amount
27       cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
28    } // end for
29 } // end main
```

Fig. 5.6 | Compound interest calculations with `for`. (Part 1 of 2.)

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 5.6 | Compound interest calculations with for. (Part 2 of 2.)

This program will not compile without including header <cmath> (line 5). Function pow requires two double arguments. Variable year is an integer. Header <cmath> includes information that tells the compiler to convert the value of year to a temporary double representation before calling the function. This information is contained in pow's function prototype. Chapter 6 summarizes other math library functions.

Common Programming Error 5.5

Forgetting to include the appropriate header when using standard library functions (e.g., <cmath> in a program that uses math library functions) is a compilation error.

A Caution about Using Type float or double for Monetary Amounts

Lines 10–12 declare the double variables amount, principal and rate. We did this for simplicity because we're dealing with fractional parts of dollars, and we need a type that allows decimal points in its values. Unfortunately, this can cause trouble. Here's a simple explanation of what can go wrong when using float or double to represent dollar amounts (assuming setprecision(2) is used to specify two digits of precision when printing): Two dollar amounts stored in the machine could be 14.234 (which prints as 14.23) and 18.673 (which prints as 18.67). When these amounts are added, they produce the internal sum 32.907, which prints as 32.91. Thus your printout could appear as

```
  14.23
+ 18.67
-------
  32.91
```

but a person adding the individual numbers as printed would expect the sum 32.90! You've been warned! In the exercises, we explore the use of integers to perform monetary calculations. [*Note:* Some third-party vendors sell C++ class libraries that perform precise monetary calculations.]

Good Programming Practice 5.5

Do not use variables of type float or double to perform monetary calculations. The imprecision of floating-point numbers can cause incorrect monetary values.

Using Stream Manipulators to Format Numeric Output

The output statement in line 18 before the for loop and the output statement in line 27 in the for loop combine to print the values of the variables year and amount with the formatting specified by the parameterized stream manipulators `setprecision` and **`setw`** and the nonparameterized stream manipulator `fixed`. The stream manipulator `setw(4)` specifies that the next value output should appear in a **field width** of 4—i.e., `cout` prints the value with *at least* 4 character positions. If the value to be output is *less* than 4 character positions wide, the value is **right justified** in the field by default. If the value to be output is *more* than 4 character positions wide, the field width is extended to accommodate the entire value. To indicate that values should be output **left justified**, simply output nonparameterized stream manipulator **`left`** (found in header `<iostream>`). Right justification can be restored by outputting nonparameterized stream manipulator **`right`**.

The other formatting in the output statements indicates that variable `amount` is printed as a fixed-point value with a decimal point (specified in line 18 with the stream manipulator `fixed`) right justified in a field of 21 character positions (specified in line 27 with `setw(21)`) and two digits of precision to the right of the decimal point (specified in line 18 with manipulator `setprecision(2)`). We applied the stream manipulators `fixed` and `setprecision` to the output stream (i.e., `cout`) before the `for` loop because these format settings remain in effect until they're changed—such settings are called **sticky settings** and they do not need to be applied during each iteration of the loop. However, the field width specified with `setw` applies only to the next value output. We discuss C++'s powerful input/output formatting capabilities in Chapter 15, Stream Input/Output.

The calculation `1.0 + rate`, which appears as an argument to the `pow` function, is contained in the body of the `for` statement. In fact, this calculation produces the *same* result during each iteration of the loop, so repeating it is wasteful—it should be performed once before the loop.

Performance Tip 5.1

Avoid placing expressions whose values do not change inside loops—but, even if you do, many of today's sophisticated optimizing compilers will automatically place such expressions outside *the loops in the generated machine-language code.*

Performance Tip 5.2

Many compilers contain optimization features that improve the performance of the code you write, but it's still better to write good code from the start.

Be sure to try our Peter Minuit problem in Exercise 5.29. This problem demonstrates the wonders of compound interest.

5.5 do…while Repetition Statement

The `do…while` repetition statement is similar to the `while` statement. In the `while` statement, the loop-continuation condition test occurs at the beginning of the loop *before* the body of the loop executes. The `do…while` statement tests the loop-continuation condition *after* the loop body executes; therefore, *the loop body always executes at least once*. When a `do…while` terminates, execution continues with the statement after the `while` clause. It's not necessary to use braces in the `do…while` statement if there's only one statement in the body; however, most programmers include the braces to avoid confusion between the `while` and `do…while` statements. For example,

```
while ( condition )
```

normally is regarded as the header of a `while` statement. A `do...while` with no braces around the single statement body appears as

```
do
   statement
while ( condition );
```

which can be confusing. You might misinterpret the last line—`while(` *condition* `);`—as a `while` statement containing as its body an empty statement. Thus, the `do...while` with one statement often is written as follows to avoid confusion:

```
do
{
   statement
} while ( condition );
```

Figure 5.7 uses a `do...while` statement to print the numbers 1–10. Upon entering the `do...while` statement, line 12 outputs `counter`'s value and line 13 increments `counter`. Then the program evaluates the loop-continuation test at the bottom of the loop (line 14). If the condition is true, the loop continues from the first body statement in the `do...while` (line 12). If the condition is false, the loop terminates and the program continues with the next statement after the loop (line 16).

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int counter = 1; // initialize counter
9
10    do
11    {
12       cout << counter << " "; // display counter
13       ++counter; // increment counter
14    } while ( counter <= 10 ); // end do...while
15
16    cout << endl; // output a newline
17 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.7 | `do...while` repetition statement.

`do...while` Statement UML Activity Diagram

Figure 5.8 contains the `do...while` statement's UML activity diagram, which makes it clear that the loop-continuation condition is not evaluated until *after* the loop performs its body at least once. Compare this activity diagram with that of the `while` statement (Fig. 4.6).

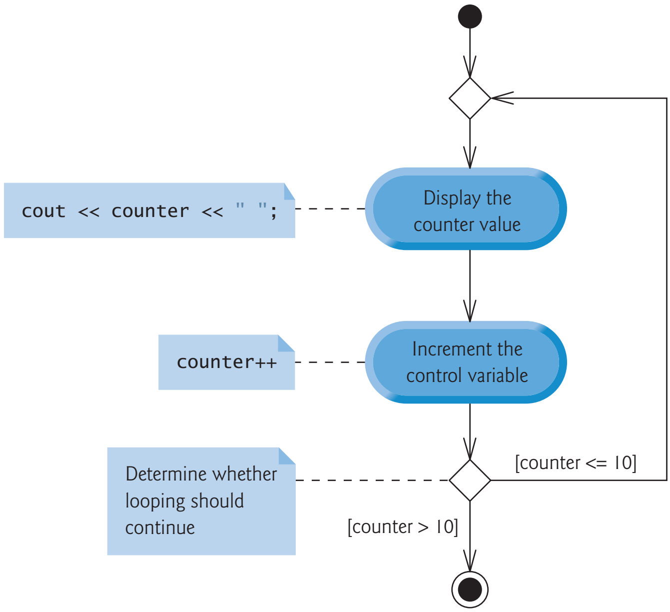


Fig. 5.8 | UML activity diagram for the `do...while` repetition statement of Fig. 5.7.

5.6 `switch` Multiple-Selection Statement

C++ provides the **`switch` multiple-selection** statement to perform many different actions based on the possible values of a variable or expression. Each action is associated with the value of a **constant integral expression** (i.e., any combination of character and integer constants that evaluates to a constant integer value).

`GradeBook` Class with `switch` Statement to Count A, B, C, D and F Grades

This next version of the `GradeBook` class asks the user to enter a set of letter grades, then displays a summary of the number of students who received each grade. The class uses a `switch` to determine whether each grade entered is an A, B, C, D or F and to increment the appropriate grade counter. Class `GradeBook` is defined in Fig. 5.9, and its member-function definitions appear in Fig. 5.10. Figure 5.11 shows sample inputs and outputs of the `main` program that uses class `GradeBook` to process a set of grades.

Like earlier versions of the class definition, the `GradeBook` class definition (Fig. 5.9) contains function prototypes for member functions `setCourseName` (line 12), `getCourseName` (line 13) and `displayMessage` (line 14), as well as the class's constructor (line 11). The class definition also declares `private` data member `courseName` (line 18).

Class `GradeBook` (Fig. 5.9) now contains five additional `private` data members (lines 19–23)—counter variables for each grade category (i.e., A, B, C, D and F). The class also contains two additional `public` member functions—`inputGrades` and `displayGradeReport`. Member function `inputGrades` (declared in line 15) reads an arbitrary number of letter grades from the user using sentinel-controlled repetition and updates the appropriate grade counter for each grade entered. Member function `displayGradeReport` (declared in line 16) outputs a report containing the number of students who received each letter grade.

```
1  // Fig. 5.9: GradeBook.h
2  // Definition of class GradeBook that counts A, B, C, D and F grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void inputGrades(); // input arbitrary number of grades from user
16    void displayGradeReport(); // display a report based on the grades
17 private:
18    string courseName; // course name for this GradeBook
19    int aCount; // count of A grades
20    int bCount; // count of B grades
21    int cCount; // count of C grades
22    int dCount; // count of D grades
23    int fCount; // count of F grades
24 }; // end class GradeBook
```

Fig. 5.9 | GradeBook class definition.

Source-code file `GradeBook.cpp` (Fig. 5.10) contains the member-function definitions for class `GradeBook`. Notice that lines 13–17 in the constructor initialize the five grade counters to `0`—when a `GradeBook` object is first created, no grades have been entered yet. As you'll soon see, these counters are incremented in member function `inputGrades` as the user enters grades. The definitions of member functions `setCourseName`, `getCourseName` and `displayMessage` are identical to those in the earlier versions of class `GradeBook`.

```
1  // Fig. 5.10: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a switch statement to count A, B, C, D and F grades.
4  #include <iostream>
5  #include "GradeBook.h" // include definition of class GradeBook
6  using namespace std;
7
8  // constructor initializes courseName with string supplied as argument;
9  // initializes counter data members to 0
10 GradeBook::GradeBook( string name )
11 {
12    setCourseName( name ); // validate and store courseName
```

Fig. 5.10 | GradeBook class uses `switch` statement to count letter grades. (Part 1 of 3.)

```
   aCount = 0; // initialize count of A grades to 0
   bCount = 0; // initialize count of B grades to 0
   cCount = 0; // initialize count of C grades to 0
   dCount = 0; // initialize count of D grades to 0
   fCount = 0; // initialize count of F grades to 0
} // end GradeBook constructor

// function to set the course name; limits name to 25 or fewer characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// input arbitrary number of grades from user; update grade counter
void GradeBook::inputGrades()
{
   int grade; // grade entered by user

   cout << "Enter the letter grades." << endl
      << "Enter the EOF character to end input." << endl;

   // loop until user types end-of-file key sequence
   while ( ( grade = cin.get() ) != EOF )
   {
      // determine which grade was entered
      switch ( grade ) // switch statement nested in while
      {
         case 'A': // grade was uppercase A
         case 'a': // or lowercase a
            ++aCount; // increment aCount
            break; // necessary to exit switch
```

Fig. 5.10 | GradeBook class uses `switch` statement to count letter grades. (Part 2 of 3.)

```
66
67          case 'B': // grade was uppercase B
68          case 'b': // or lowercase b
69             ++bCount; // increment bCount
70             break; // exit switch
71
72          case 'C': // grade was uppercase C
73          case 'c': // or lowercase c
74             ++cCount; // increment cCount
75             break; // exit switch
76
77          case 'D': // grade was uppercase D
78          case 'd': // or lowercase d
79             ++dCount; // increment dCount
80             break; // exit switch
81
82          case 'F': // grade was uppercase F
83          case 'f': // or lowercase f
84             ++fCount; // increment fCount
85             break; // exit switch
86
87          case '\n': // ignore newlines,
88          case '\t': // tabs,
89          case ' ': // and spaces in input
90             break; // exit switch
91
92          default: // catch all other characters
93             cout << "Incorrect letter grade entered."
94                << " Enter a new grade." << endl;
95             break; // optional; will exit switch anyway
96       } // end switch
97    } // end while
98 } // end function inputGrades
99
100 // display a report based on the grades entered by user
101 void GradeBook::displayGradeReport()
102 {
103    // output summary of results
104    cout << "\n\nNumber of students who received each letter grade:"
105       << "\nA: " << aCount // display number of A grades
106       << "\nB: " << bCount // display number of B grades
107       << "\nC: " << cCount // display number of C grades
108       << "\nD: " << dCount // display number of D grades
109       << "\nF: " << fCount // display number of F grades
110       << endl;
111 } // end function displayGradeReport
```

Fig. 5.10 | GradeBook class uses switch statement to count letter grades. (Part 3 of 3.)

Reading Character Input

The user enters letter grades for a course in member function `inputGrades` (lines 49–98). In the `while` header, in line 57, the parenthesized assignment `(grade = cin.get())` executes first. The `cin.get()` function reads one character from the keyboard and stores that character in integer variable `grade` (declared in line 51). Normally, characters are stored in variables of type **char**; however, characters can be stored in any integer data type, because types short, int and long are guaranteed to be at least as big as type char. Thus, we can treat a character either as an integer or as a character, depending on its use. For example, the statement

```
cout << "The character (" << 'a' << ") has the value "
   << static_cast< int > ( 'a' ) << endl;
```

prints the character a and its integer value as follows:

```
The character (a) has the value 97
```

The integer 97 is the character's numerical representation in the computer. Most computers today use the Unicode character set in which 97 represents the lowercase letter 'a'. Appendix B shows the characters and decimal equivalents from the **ASCII** (**American Standard Code for Information Interchange**) **character set**, which is a subset of Unicode.

Generally, assignment statements have the value that's assigned to the variable on the left side of the =. Thus, the value of the assignment expression grade = cin.get() is the same as the value returned by cin.get() and assigned to the variable grade.

The fact that assignment expressions have values can be useful for assigning the same value to *several* variables. For example,

```
a = b = c = 0;
```

first evaluates c = 0 (because the = operator associates from right to left). The variable b is then assigned the value of c = 0 (which is 0). Then, a is assigned the value of b = (c = 0) (which is also 0). In the program, the value of grade = cin.get() is compared with the value of EOF (a symbol whose acronym stands for "end-of-file"). We use EOF (which normally has the value –1) as the sentinel value. *However, you do not type the value –1, nor do you type the letters EOF as the sentinel value.* Rather, you type a system-dependent keystroke combination that means "end-of-file" to indicate that you have no more data to enter. EOF is a symbolic integer constant that is included into the program via the <iostream> header[1]. If the value assigned to grade is equal to EOF, the while loop (lines 57–97) terminates. We've chosen to represent the characters entered into this program as ints, because EOF has type int.

On UNIX/Linux systems and many others, end-of-file is entered by typing

<Ctrl> d

on a line by itself. This notation means to press and hold down the *Ctrl* key, then press the *d* key. On other systems such as Microsoft Windows, end-of-file can be entered by typing

<Ctrl> z

[*Note:* In some cases, you must press *Enter* after the preceding key sequence. Also, the characters ^Z sometimes appear on the screen to represent end-of-file, as shown in Fig. 5.11.]

Portability Tip 5.1

The keystroke combinations for entering end-of-file are system dependent.

---

1. To compile this program on Linux, you'll also need to include the header <cstdio> which defines the EOF constant.

Portability Tip 5.2

Testing for the symbolic constant `EOF` rather than `-1` makes programs more portable. The ANSI/ISO C standard, from which C++ adopts the definition of `EOF`, states that `EOF` is a negative integral value, so `EOF` could have different values on different systems.

In this program, the user enters grades at the keyboard. When the user presses the *Enter* (or the *Return*) key, the characters are read by the `cin.get()` function, one character at a time. If the character entered is not end-of-file, the flow of control enters the `switch` statement (lines 60–96), which increments the appropriate letter-grade counter.

`switch` Statement Details

The `switch` statement consists of a series of **`case` labels** and an optional **`default` case**. These are used in this example to determine which counter to increment, based on a grade. When the flow of control reaches the `switch`, the program evaluates the expression in the parentheses (i.e., `grade`) following keyword `switch` (line 60). This is called the **controlling expression**. The `switch` statement compares the value of the controlling expression with each `case` label. Assume the user enters the letter `C` as a grade. The program compares `C` to each `case` in the `switch`. If a match occurs (`case 'C':` in line 72), the program executes the statements for that `case`. For the letter `C`, line 74 increments `cCount` by `1`. The `break` statement (line 75) causes program control to proceed with the first statement after the `switch`—in this program, control transfers to line 97. This line marks the end of the body of the `while` loop that inputs grades (lines 57–97), so control flows to the `while`'s condition (line 57) to determine whether the loop should continue executing.

The `case`s in our `switch` explicitly test for the lowercase and uppercase versions of the letters A, B, C, D and F. Note the `case`s in lines 62–63 that test for the values `'A'` and `'a'` (both of which represent the grade A). Listing `case`s consecutively with no statements between them enables the `case`s to perform the same set of statements—when the controlling expression evaluates to either `'A'` or `'a'`, the statements in lines 64–65 will execute. Each `case` can have multiple statements. The `switch` selection statement does not require braces around multiple statements in each `case`.

Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` *and* subsequent `case`s execute until a `break` statement or the end of the `switch` is encountered. This feature is perfect for writing a concise program that displays the iterative song "The Twelve Days of Christmas" in Exercise 5.28.

Common Programming Error 5.6

Forgetting a `break` statement when one is needed in a `switch` statement is a logic error.

Common Programming Error 5.7

Omitting the space between the word `case` and the integral value tested in a `switch` statement—e.g., writing `case3:` instead of `case 3:`—is a logic error. The `switch` statement will not perform the appropriate actions when the controlling expression has a value of 3.

Providing a `default` Case

If no match occurs between the controlling expression's value and a `case` label, the `default` case (lines 92–95) executes. We use the `default` case in this example to process all controlling-expression values that are neither valid grades nor newline, tab or space characters. If no match occurs, the `default` case executes, and lines 93–94 print an error message indicating that an incorrect letter grade was entered. If no match occurs in a `switch` statement that does not contain a `default` case, program control continues with the first statement after the `switch`.

Error-Prevention Tip 5.3

Provide a `default` case in `switch` statements. Cases not explicitly tested in a `switch` statement without a `default` case are ignored. Including a `default` case focuses you on the need to process exceptional conditions. There are situations in which no `default` processing is needed. Although the `case` clauses and the `default` case clause in a `switch` statement can occur in any order, it's common practice to place the `default` clause last.

Good Programming Practice 5.6

The last `case` in a `switch` statement does not require a `break` statement. Some programmers include this `break` for clarity and for symmetry with other cases.

Ignoring Newline, Tab and Blank Characters in Input

Lines 87–90 in the `switch` statement of Fig. 5.10 cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause problems. To have the program read the characters, we must send them to the computer by pressing the *Enter* key. This places a newline character in the input after the character we wish to process. Often, this newline character must be specially processed. By including these `case`s in our `switch` statement, we prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input.

Testing Class **GradeBook**

Figure 5.11 creates a `GradeBook` object (line 8). Line 10 invokes the its `displayMessage` member function to output a welcome message to the user. Line 11 invokes member function object's `inputGrades` to read a set of grades from the user and keep track of how many students received each grade. The output window in Fig. 5.11 shows an error message displayed in response to entering an invalid grade (i.e., `E`). Line 12 invokes `GradeBook` member function `displayGradeReport` (defined in lines 101–111 of Fig. 5.10), which outputs a report based on the grades entered (as in the output in Fig. 5.11).

```
1  // Fig. 5.11: fig05_11.cpp
2  // Create GradeBook object, input grades and display grade report.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13 } // end main
```

Fig. 5.11 | Creating a `GradeBook` object and calling its member functions. (Part 1 of 2.)

```
Welcome to the grade book for
CS101 C++ Programming!

Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z


Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 5.11 | Creating a `GradeBook` object and calling its member functions. (Part 2 of 2.)

`switch` Statement UML Activity Diagram

Figure 5.12 shows the UML activity diagram for the general `switch` multiple-selection statement. Most `switch` statements use a `break` in each `case` to terminate the `switch` statement after processing the `case`. Figure 5.12 emphasizes this by including `break` statements in the activity diagram. Without the `break` statement, control would not transfer to the first statement after the `switch` statement after a `case` is processed. Instead, control would transfer to the next `case`'s actions.

The diagram makes it clear that the `break` statement at the end of a `case` causes control to exit the `switch` statement immediately. Again, note that (besides an initial state, transition arrows, a final state and several notes) the diagram contains action states and decisions. Also, the diagram uses merge symbols to merge the transitions from the `break` statements to the final state.

When using the `switch` statement, remember that each `case` can be used to test only a *constant* integral expression—any combination of character constants and integer constants that evaluates to a constant integer value. A character constant is represented as the specific character in single quotes, such as `'A'`. An integer constant is simply an integer value. Also, each `case` label can specify only one constant integral expression.

Common Programming Error 5.8

Specifying a nonconstant integral expression in a `switch`'s `case` label is a syntax error.

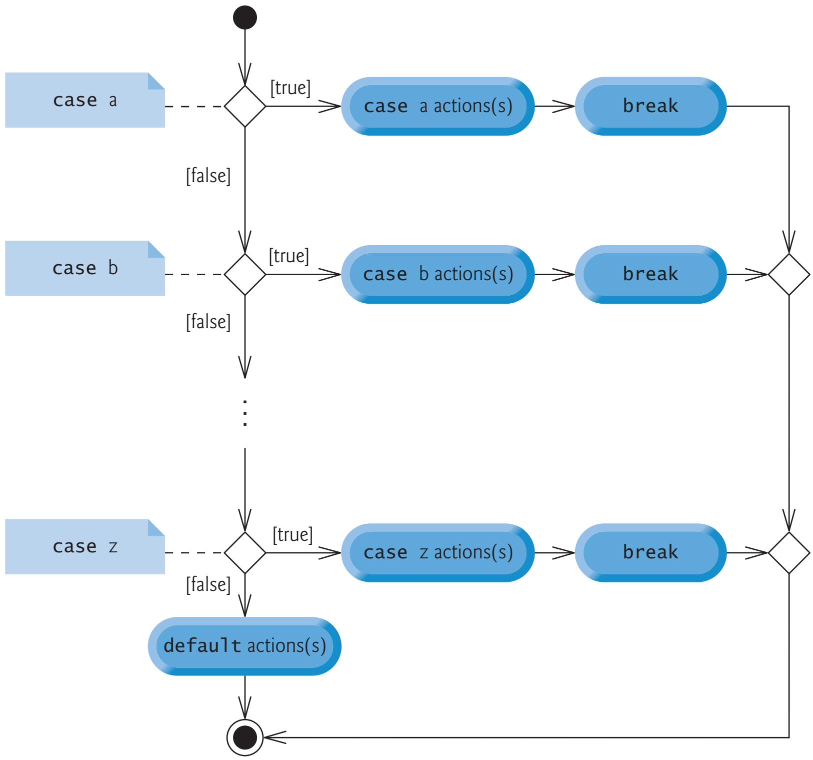


Fig. 5.12 | `switch` multiple-selection statement UML activity diagram with `break` statements.

Common Programming Error 5.9

Providing `case` labels with identical values in a `switch` statement is a compilation error.

In Chapter 13, we present a more elegant way to implement `switch` logic. We'll use a technique called polymorphism to create programs that are often clearer, more concise, easier to maintain and easier to extend than programs that use `switch` logic.

Notes on Data Types

C++ has *flexible data type sizes* (see Appendix C, Fundamental Types). Different applications, for example, might need integers of different sizes. C++ provides several integer types. The range of integer values for each type depends on the particular computer's hardware. In addition to the types `int` and `char`, C++ provides the types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`). The minimum range of values for `short` integers is –32,768 to 32,767. For the vast majority of integer calculations, `long` integers are sufficient. The minimum range of values for `long` integers is –2,147,483,648 to 2,147,483,647. On most computers, `int`s are equivalent either to `short` or to `long`. The range of values for an `int` is at least the same as that for `short` integers and no larger than that for `long` integers. The data type `char` can be used to represent any of the characters in the computer's character set. It also can be used to represent small integers.

5.7 break and continue Statements

C++ also provides statements `break` and `continue` to alter the flow of control. The preceding section showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in a repetition statement.

break Statement

The **`break` statement**, when executed in a `while`, `for`, `do…while` or `switch` statement, causes immediate exit from that statement. Program execution continues with the next statement. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement. Figure 5.13 demonstrates the `break` statement (line 13) exiting a `for` repetition statement.

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int count; // control variable also used after loop terminates
9
10    for ( count = 1; count <= 10; ++count ) // loop 10 times
11    {
12       if ( count == 5 )
13          break; // break loop only if count is 5
14
15       cout << count << " ";
16    } // end for
17
18    cout << "\nBroke out of loop at count = " << count << endl;
19 } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.13 | `break` statement exiting a `for` statement.

When the `if` statement detects that `count` is `5`, the `break` statement executes. This terminates the `for` statement, and the program proceeds to line 18 (immediately after the `for` statement), which displays a message indicating the control variable value that terminated the loop. The `for` statement fully executes its body only four times instead of 10. The control variable `count` is defined outside the `for` statement header, so that we can use the control variable both in the loop's body and after the loop completes its execution.

continue Statement

The **`continue` statement**, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of that statement and proceeds with the next iteration of the loop. In `while` and `do…while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes. In the `for` statement, the increment expression executes, then the loop-continuation test evaluates.

Figure 5.14 uses the `continue` statement (line 11) in a `for` statement to skip the output statement (line 13) when the nested `if` (lines 10–11) determines that the value of `count` is `5`. When the `continue` statement executes, program control continues with the increment of the control variable in the `for` header (line 8) and loops five more times.

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     for ( int count = 1; count <= 10; ++count ) // loop 10 times
9     {
10       if ( count == 5 ) // if count is 5,
11          continue;      // skip remaining code in loop
12
13       cout << count << " ";
14    } // end for
15
16    cout << "\nUsed continue to skip printing 5" << endl;
17 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.14 | `continue` statement terminating an iteration of a `for` statement.

In Section 5.3, we stated that the `while` statement could be used in most cases to represent the `for` statement. The one exception occurs when the increment expression in the `while` statement follows the `continue` statement. In this case, the increment does not execute before the program tests the loop-continuation condition, and the `while` does not execute in the same manner as the `for`.

Good Programming Practice 5.7

Some programmers feel that `break` and `continue` violate structured programming. The effects of these statements can be achieved by structured programming techniques we soon will learn, so these programmers do not use `break` and `continue`. Most programmers consider the use of `break` in `switch` statements acceptable.

Software Engineering Observation 5.1

There's a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary.

5.8 Logical Operators

So far we've studied only **simple conditions**, such as `counter <= 10`, `total > 1000` and `number != sentinelValue`. We expressed these conditions in terms of the relational operators

>, <, >= and <=, and the equality operators == and `!=`. Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested `if` or `if...else` statements.

C++ provides **logical operators** that are used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), `||` (logical OR) and `!` (logical NOT, also called logical negation).

Logical AND (&&) Operator

Suppose that we wish to ensure that two conditions are *both* `true` before we choose a certain path of execution. In this case, we can use the **&&** (**logical AND**) operator, as follows:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

This `if` statement contains two simple conditions. The condition `gender == 1` is used here to determine whether a person is a female. The condition `age >= 65` determines whether a person is a senior citizen. The simple condition to the left of the && operator evaluates first. If necessary, the simple condition to the right of the && operator evaluates next. As we'll discuss shortly, the right side of a logical AND expression is evaluated *only* if the left side is `true`. The `if` statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is `true` if and only if *both* of the simple conditions are `true`. Finally, if this combined condition is indeed `true`, the statement in the `if` statement's body increments the count of `seniorFemales`. If either (or both) of the simple conditions are `false`, then the program skips the incrementing and proceeds to the statement following the `if`. The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == 1 ) && ( age >= 65 )
```

Common Programming Error 5.10

Although 3 < x < 7 is a mathematically correct condition, it does not evaluate as you might expect in C++. Use (3 < x && x < 7) to get the proper evaluation in C++.

Figure 5.15 summarizes the && operator. The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2*. Such tables are often called **truth tables**. C++ evaluates to `false` or `true` all expressions that include relational operators, equality operators and/or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.15 | && (logical AND) operator truth table.

Logical OR (`||`) Operator

Now let's consider the `||` (**logical OR**) operator. Suppose we wish to ensure that either *or* both of two conditions are `true` before we choose a certain path of execution. In this case, we use the `||` operator, as in the following program segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   cout << "Student grade is A" << endl;
```

This preceding condition contains two simple conditions. The simple condition `semesterAverage >= 90` evaluates to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The simple condition `finalExam >= 90` evaluates to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

and awards the student an "A" if *either or both* of the simple conditions are `true`. The message "`Student grade is A`" prints unless both of the simple conditions are `false`. Figure 5.16 is a truth table for the logical OR operator (`||`).

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 5.16 | `||` (logical OR) operator truth table.

The `&&` operator has a higher precedence than the `||` operator. Both operators associate from left to right. An expression containing `&&` or `||` operators evaluates only until the truth or falsehood of the expression is known. Thus, evaluation of the expression

```
( gender == 1 ) && ( age >= 65 )
```

stops immediately if `gender` is not equal to `1` (i.e., the entire expression is `false`) and continues if `gender` is equal to `1` (i.e., the entire expression could still be `true` if the condition `age >= 65` is `true`). This performance feature for the evaluation of logical AND and logical OR expressions is called **short-circuit evaluation**.

Performance Tip 5.3

In expressions using operator `&&`, if the separate conditions are independent of one another, make the condition most likely to be `false` the leftmost condition. In expressions using operator `||`, make the condition most likely to be `true` the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.

Logical Negation (`!`) Operator

C++ provides the `!` (**logical NOT**, also called **logical negation**) operator to "reverse" a condition's meaning. The unary logical negation operator has only a single condition as an operand. The unary logical negation operator is placed *before* a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is `false`, such as in the following program segment:

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

You can often avoid the `!` operator by using an appropriate relational or equality operator. For example, the preceding `if` statement also can be written as follows:

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```

This flexibility often can help you express a condition in a more "natural" or convenient manner. Figure 5.17 is a truth table for the logical negation operator (`!`).

expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.17 | ! (logical negation) operator truth table.

Logical Operators Example

Figure 5.18 demonstrates the logical operators by producing their truth tables. The output shows each expression that's evaluated and its `bool` result. By default, `bool` values `true` and `false` are displayed by `cout` and the stream insertion operator as `1` and `0`, respectively. We use **stream manipulator `boolalpha`** (a sticky manipulator) in line 9 to specify that the value of each `bool` expression should be displayed as either the word "true" or the word "false." For example, the result of the expression `false && false` in line 10 is `false`, so the second line of output includes the word "false." Lines 9–13 produce the truth table for `&&`. Lines 16–20 produce the truth table for `||`. Lines 23–25 produce the truth table for `!`.

```
1  // Fig. 5.18: fig05_18.cpp
2  // Logical operators.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // create truth table for && (logical AND) operator
9     cout << boolalpha << "Logical AND (&&)"
10       << "\nfalse && false: " << ( false && false )
11       << "\nfalse && true: " << ( false && true )
12       << "\ntrue && false: " << ( true && false )
13       << "\ntrue && true: " << ( true && true ) << "\n\n";
```

Fig. 5.18 | Logical operators. (Part 1 of 2.)

```

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)"
      << "\nfalse || false: " << ( false || false )
      << "\nfalse || true: " << ( false || true )
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

   // create truth table for ! (logical negation) operator
   cout << "Logical NOT (!)"
      << "\n!false: " << ( !false )
      << "\n!true: " << ( !true ) << endl;
} // end main
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

Fig. 5.18 | Logical operators. (Part 2 of 2.)

Summary of Operator Precedence and Associativity

Figure 5.19 adds the logical and comma operators to the operator precedence and associativity chart. The operators are shown from top to bottom, in decreasing order of precedence.

Operators	Associativity	Type
`::`	left to right	scope resolution
`()`	*[See caution in Fig. 2.10]*	grouping parentheses
`++` `--` `static_cast<` *type* `>()`	left to right	unary (postfix)
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality

Fig. 5.19 | Operator precedence and associativity. (Part 1 of 2.)

Operators	Associativity	Type
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=  +=  -=  *=  /=  %=`	right to left	assignment
`,`	left to right	comma

Fig. 5.19 | Operator precedence and associativity. (Part 2 of 2.)

5.9 Confusing the Equality (==) and Assignment (=) Operators

There's one error that C++ programmers, no matter how experienced, tend to make so frequently that we feel it requires a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes this so damaging is that it ordinarily does not cause syntax errors—statements with these errors tend to compile correctly and the programs run to completion, often generating incorrect results through runtime logic errors. Some compilers issue a warning when = is used in a context where == is expected.

Two aspects of C++ contribute to these problems. One is that *any expression that produces a value can be used in the decision portion of any control statement*. If the value of the expression is zero, it's treated as `false`, and if the value is nonzero, it's treated as `true`. The second is that assignments produce a value—namely, the value assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

but we accidentally write

```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

The first `if` statement properly awards a bonus to the person whose `payCode` is equal to 4. The second one—with the error—evaluates the assignment expression in the `if` condition to the constant 4. *Any nonzero value is interpreted as* `true`, so this condition is always `true` and the person *always* receives a bonus regardless of what the actual paycode is! Even worse, the paycode has been modified when it was only supposed to be examined!

Common Programming Error 5.11

Using operator == for assignment and using operator = for equality are logic errors.

Error-Prevention Tip 5.4

Programmers normally write conditions such as `x == 7` *with the variable name on the left and the constant on the right. By placing the constant on the left, as in* `7 == x`*, you'll be protected by the compiler if you accidentally replace the == operator with = . The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.*

Variable names are said to be ***lvalues*** (for "left values") because they can be used on the *left* side of an assignment operator. Constants are said to be ***rvalues*** (for "right values") because they can be used on only the *right* side of an assignment operator. *Lvalues* can also be used as *rvalues*, but not vice versa.

There's another equally unpleasant situation. Suppose you want to assign a value to a variable with a simple statement like

```
x = 1;
```

but instead write

```
x == 1;
```

Here, too, this is not a syntax error. Rather, the compiler simply evaluates the conditional expression. If `x` is equal to `1`, the condition is `true` and the expression evaluates to the value `true`. If `x` is not equal to `1`, the condition is `false` and the expression evaluates to the value `false`. Regardless of the expression's value, there's no assignment operator, so the value simply is lost. The value of `x` remains unaltered, probably causing an execution-time logic error. Unfortunately, we do not have a handy trick available to help you with this problem!

Error-Prevention Tip 5.5

Use your text editor to search for all occurrences of = *in your program and check that you have the correct assignment operator or logical operator in each place.*

5.10 Structured Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, so should programmers design programs. Our field is younger than architecture is, and our collective wisdom is sparser. We've learned that structured programming produces programs that are easier than unstructured programs to understand, test, debug, modify, and even prove correct in a mathematical sense.

Figure 5.20 uses activity diagrams to summarize C++'s control statements. The initial and final states indicate the *single entry point* and the *single exit point* of each control statement. Arbitrarily connecting individual symbols in an activity diagram can lead to unstructured programs. Therefore, the programming profession uses only a limited set of control statements that can be combined in only two simple ways to build structured programs.

For simplicity, only *single-entry/single-exit control statements* are used—there's only one way to enter and only one way to exit each control statement. Connecting control statements in sequence to form structured programs is simple—the final state of one control statement is connected to the initial state of the next—that is, they're placed one after another in a program. We've called this control-statement *stacking*. The rules for forming structured programs also allow for control statements to be *nested*.

Figure 5.21 shows the rules for forming structured programs. The rules assume that action states may be used to indicate any action. The rules also assume that we begin with the so-called *simplest activity diagram* (Fig. 5.22), consisting of only an initial state, an action state, a final state and transition arrows.

Applying the rules of Fig. 5.21 always results in an activity diagram with a neat, building-block appearance. For example, repeatedly applying Rule 2 to the simplest

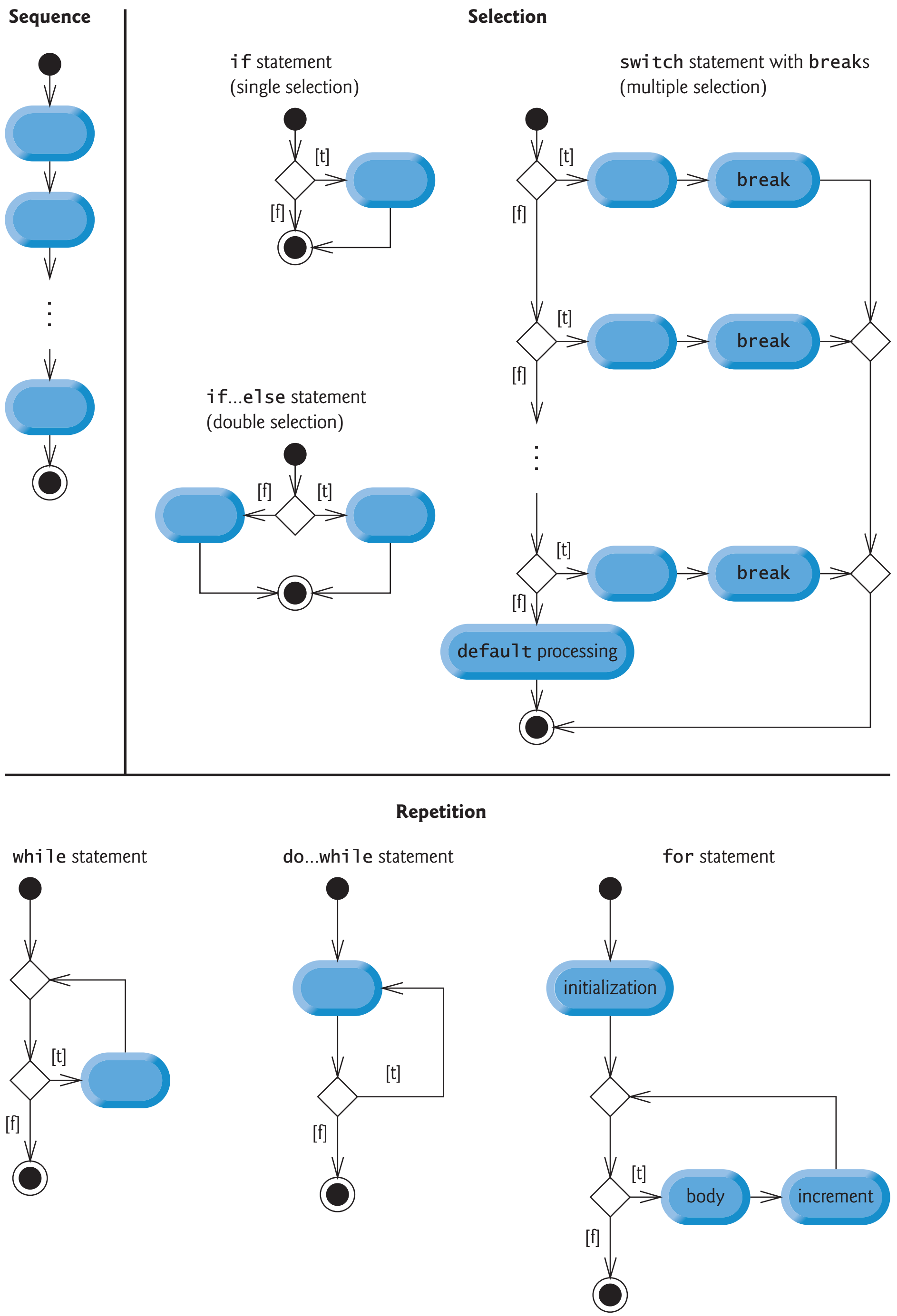


Fig. 5.20 | C++'s single-entry/single-exit sequence, selection and repetition statements.

Rules for forming structured programs	
1)	Begin with the "simplest activity diagram" (Fig. 5.22).
2)	Any action state can be replaced by two action states in sequence.
3)	Any action state can be replaced by any control statement (sequence, `if`, `if…else`, `switch`, `while`, `do…while` or `for`).
4)	Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 5.21 | Rules for forming structured programs.

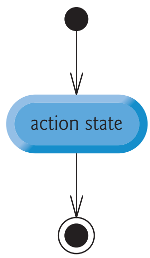


Fig. 5.22 | Simplest activity diagram.

activity diagram results in an activity diagram containing many action states in sequence (Fig. 5.23). Rule 2 generates a stack of control statements, so let's call Rule 2 the **stacking rule**. The vertical dashed lines in Fig. 5.23 are not part of the UML. We use them to separate the four activity diagrams that demonstrate Rule 2 of Fig. 5.21 being applied.

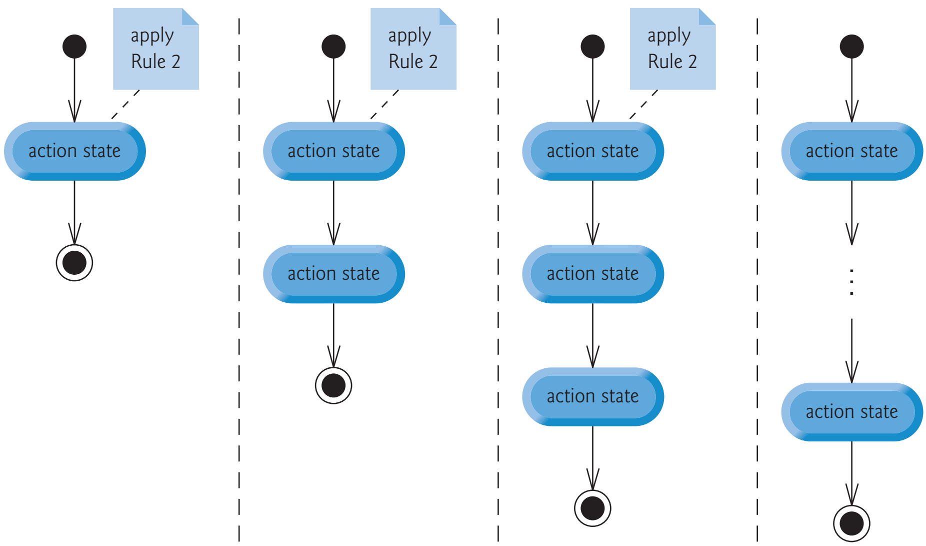


Fig. 5.23 | Repeatedly applying Rule 2 of Fig. 5.21 to the simplest activity diagram.

Rule 3 is the **nesting rule**. Repeatedly applying Rule 3 to the simplest activity diagram results in one with neatly nested control statements. For example, in Fig. 5.24, the action state in the simplest activity diagram is replaced with a double-selection (`if...else`) statement. Then Rule 3 is applied again to the action states in the double-selection statement, replacing each with a double-selection statement. The dashed action-state symbols around each of the double-selection statements represent an action state that was replaced in the preceding activity diagram. [*Note:* The dashed arrows and dashed action state symbols shown in Fig. 5.24 are not part of the UML. They're used here as pedagogic devices to illustrate that any action state may be replaced with a control statement.]

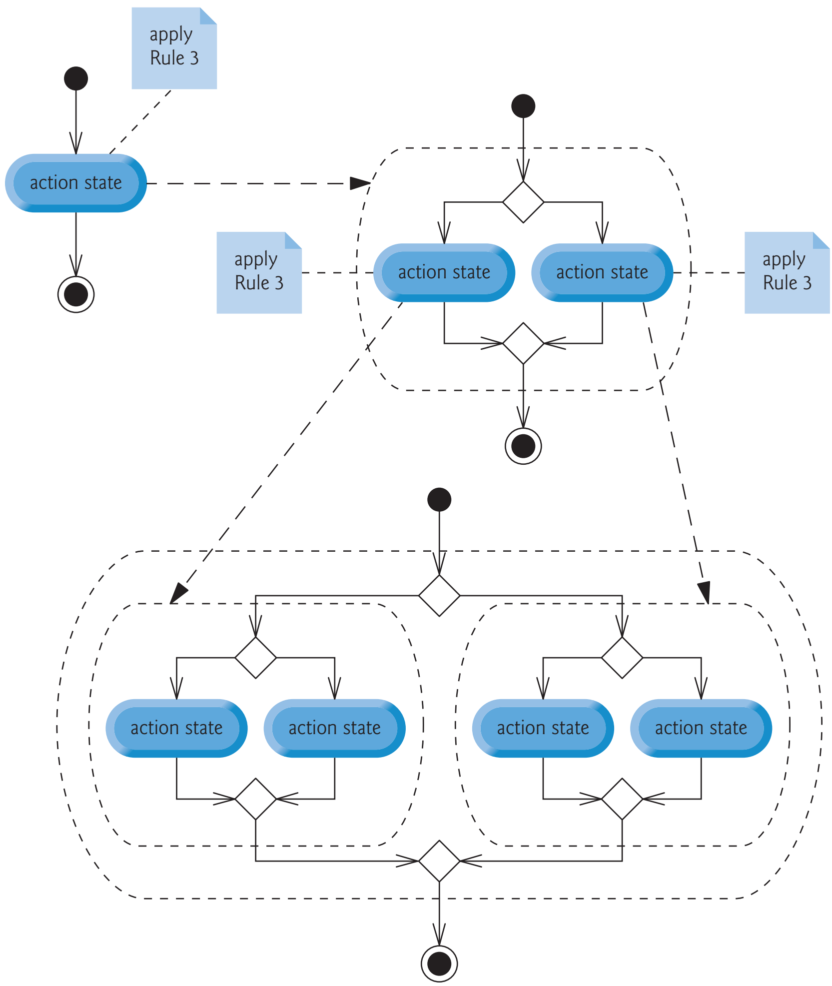


Fig. 5.24 | Applying Rule 3 of Fig. 5.21 to the simplest activity diagram several times.

Rule 4 generates larger, more involved and more deeply *nested* statements. The diagrams that emerge from applying the rules in Fig. 5.21 constitute the set of all possible activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use only *seven* simple single-entry/single-exit control statements and assemble them in only *two* simple ways.

If the rules in Fig. 5.21 are followed, an activity diagram with illegal syntax (such as that in Fig. 5.25) cannot be created. If you're uncertain about whether a particular diagram is legal, apply the rules of Fig. 5.21 in reverse to reduce the diagram to the simplest activity diagram. If it's reducible to the simplest activity diagram, the original diagram is structured; otherwise, it isn't.

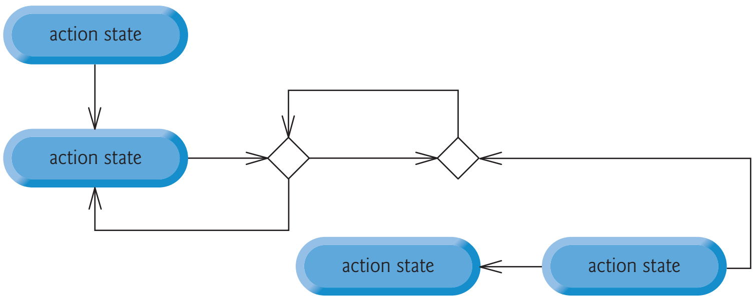


Fig. 5.25 | Activity diagram with illegal syntax.

Structured programming promotes simplicity. Böhm and Jacopini have given us the result that only *three* forms of control are needed:

- Sequence
- Selection
- Repetition

The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute.

Selection is implemented in one of three ways:

- `if` statement (single selection)
- `if...else` statement (double selection)
- `switch` statement (multiple selection)

It's straightforward to prove that the *simple `if` statement is sufficient* to provide any form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented (although perhaps not as clearly and efficiently) by combining `if` statements.

Repetition is implemented in one of three ways:

- `while` statement
- `do...while` statement
- `for` statement

It's straightforward to prove that the *`while` statement is sufficient* to provide any form of repetition. Everything that can be done with the `do...while` statement and the `for` statement can be done (although perhaps not as smoothly) with the `while` statement.

Combining these results illustrates that any form of control ever needed in a C++ program can be expressed in terms of the following:

- sequence
- `if` statement (selection)
- `while` statement (repetition)

and that these control statements can be combined in only two ways—stacking and nesting. Indeed, structured programming promotes simplicity.

5.11 Wrap-Up

We've now completed our introduction to control statements, which enable you to control the flow of execution in functions. Chapter 4 discussed the `if`, `if…else` and `while` statements. This chapter demonstrated the `for`, `do…while` and `switch` statements. We showed that any algorithm can be developed using combinations of the sequence structure, the three types of selection statements—`if`, `if…else` and `switch`—and the three types of repetition statements—`while`, `do…while` and `for`. We discussed how you can combine these building blocks to utilize proven program construction and problem-solving techniques. You used the `break` and `continue` statements to alter a repetition statement's flow of control. This chapter also introduced logical operators, which enable you to use more complex conditional expressions in control statements. Finally, we examined the common errors of confusing the equality and assignment operators and provided suggestions for avoiding these errors. In Chapter 6, we examine functions in greater depth.

Summary

Section 5.2 Essentials of Counter-Controlled Repetition

- In C++, it's more precise to call a declaration that also reserves memory a definition (p. 153).

Section 5.3 `for` Repetition Statement

- The `for` repetition statement (p. 155) handles all the details of counter-controlled repetition.
- The general format of the `for` statement is

  `for` (*initialization*; *loopContinuationCondition*; *increment*)
      statement

  where *initialization* initializes the control variable, *loopContinuationCondition* determines whether the loop should continue executing and *increment* increments or decrements the control variable.
- Typically, `for` statements are used for counter-controlled repetition and `while` statements are used for sentinel-controlled repetition.
- The scope of a variable (p. 156) specifies where it can be used in a program.
- The comma operator (p. 156) has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list.
- The initialization, loop-continuation condition and increment expressions of a `for` statement can contain arithmetic expressions. Also, the increment of a `for` statement can be negative.
- If the loop-continuation condition in a `for` header is initially `false`, the body of the `for` statement is not performed. Instead, execution proceeds with the statement following the `for`.

Section 5.4 Examples Using the `for` Statement

- Standard library function `pow(x, y)` (p. 160) calculates the value of `x` raised to the `y`th power. Function `pow` takes two arguments of type `double` and returns a `double` value.
- Parameterized stream manipulator `setw` (p. 162) specifies the field width in which the next value output should appear, right justified by default. If the value is larger than the field width, the field width is extended to accommodate the entire value. Stream manipulator `left` (p. 162) causes a value to be left justified and `right` can be used to restore right justification.
- Sticky settings are those output-formatting settings that remain in effect until they're changed.

Section 5.5 `do…while` Repetition Statement

- The `do…while` repetition statement tests the loop-continuation condition at the end of the loop, so the body of the loop will be executed at least once. The format for the `do…while` statement is

```
do
{
   statement
} while ( condition );
```

Section 5.6 `switch` Multiple-Selection Statement

- The `switch` multiple-selection statement (p. 164) performs different actions based on its controlling expression's value.
- A `switch` statement consists of a series of `case` labels (p. 169) and an optional `default` case (p. 169).
- Function `cin.get()` reads one character from the keyboard. Characters normally are stored in variables of type `char` (p. 168). A character can be treated either as an integer or as a character.
- The expression in the parentheses following `switch` is called the controlling expression (p. 169). The `switch` statement compares the value of the controlling expression with each `case` label.
- Consecutive `case`s with no statements between the perform the same set of statements.
- Each `case` label can specify only one constant integral expression.
- Each `case` can have multiple statements. The `switch` selection statement differs from other control statements in that it does not require braces around multiple statements in each `case`.
- C++ provides several data types to represent integers—`int`, `char`, `short` and `long`. The range of integer values for each type depends on the particular computer's hardware.

Section 5.7 `break` and `continue` Statements

- The `break` statement (p. 173), when executed in one of the repetition statements (`for`, `while` and `do…while`), causes immediate exit from the statement.
- The `continue` statement (p. 173), when executed in a repetition statement, skips any remaining statements in the loop's body and proceeds with the next iteration of the loop. In a `while` or `do…while` statement, execution continues with the next evaluation of the condition. In a `for` statement, execution continues with the increment expression in the `for` statement header.

Section 5.8 Logical Operators

- Logical operators (p. 175) enable you to form complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical negation).
- The `&&` (logical AND, p. 175) operator ensures that two conditions are *both* `true`.
- The `||` (logical OR, p. 176) operator ensures that either *or* both of two conditions are `true`.
- An expression containing `&&` or `||` operators evaluates only until the truth or falsehood of the expression is known. This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation (p. 176).

- The `!` (logical NOT, also called logical negation; p. 176) operator enables a programmer to "reverse" the meaning of a condition. The unary logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`. In most cases, you can avoid using logical negation by expressing the condition with an appropriate relational or equality operator.
- When used as a condition, any nonzero value implicitly converts to `true`; 0 (zero) implicitly converts to `false`.
- By default, `bool` values `true` and `false` are displayed by `cout` as `1` and `0`, respectively. Stream manipulator `boolalpha` (p. 177) specifies that the value of each `bool` expression should be displayed as either the word "true" or the word "false."

Section 5.9 Confusing the Equality (==) and Assignment (=) Operators

- Any expression that produces a value can be used in the decision portion of any control statement. If the value of the expression is zero, it's treated as `false`, and if the value is nonzero, it's treated as `true`.
- An assignment produces a value—namely, the value assigned to the variable on the left side of the assignment operator.

Section 5.10 Structured Programming Summary

- Any form of control can be expressed in terms of sequence, selection and repetition statements, and these can be combined in only two ways—stacking and nesting.

Self-Review Exercises

5.1 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The `default` case is required in the `switch` selection statement.
b) The `break` statement is required in the default case of a `switch` selection statement to exit the `switch` properly.
c) The expression `( x > y && a < b )` is `true` if either the expression `x > y` is `true` or the expression `a < b` is `true`.
d) An expression containing the `||` operator is `true` if either or both of its operands are `true`.

5.2 Write a C++ statement or a set of C++ statements to accomplish each of the following:

a) Sum the odd integers between 1 and 99 using a `for` statement. Assume the integer variables `sum` and `count` have been declared.
b) Print the value `333.546372` in a 15-character field with precisions of `1`, `2` and `3`. Print each number on the same line. Left-justify each number in its field. What three values print?
c) Calculate the value of `2.5` raised to the power `3` using function `pow`. Print the result with a precision of `2` in a field width of `10` positions. What prints?
d) Print the integers from 1 to 20 using a `while` loop and the counter variable `x`. Assume that the variable `x` has been declared, but not initialized. Print only 5 integers per line. [*Hint:* When `x % 5` is 0, print a newline character; otherwise, print a tab character.]
e) Repeat Exercise 5.2(d) using a `for` statement.

5.3 Find the errors in each of the following code segments and explain how to correct them.

a)
```
x = 1;
while ( x <= 10 );
   ++x;
}
```

b)
```
for ( y = .1; y != 1.0; y += .1 )
   cout << y << endl;
```
c)
```
switch ( n )
{
   case 1:
      cout << "The number is 1"  << endl;
   case 2:
      cout << "The number is 2" << endl;
      break;
   default:
      cout << "The number is not 1 or 2" << endl;
      break;
}
```
d) The following code should print the values 1 to 10.
```
n = 1;
while ( n < 10 )
   cout << n++ << endl;
```

Answers to Self-Review Exercises

5.1 a) False. The `default` case is optional. Nevertheless, it's considered good software engineering to always provide a `default` case.

b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required when the `default` case is the last case. Nor will the `break` statement be required if having control proceed with the next case makes sense.

c) False. When using the `&&` operator, both of the relational expressions must be `true` for the entire expression to be `true`.

d) True.

5.2 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
  sum += count;
```
b)
```
cout << fixed << left
   << setprecision( 1 ) << setw( 15 ) << 333.546372
   << setprecision( 2 ) << setw( 15 ) << 333.546372
   << setprecision( 3 ) << setw( 15 ) << 333.546372
   << endl;
```
Output is:
```
333.5          333.55         333.546
```
c)
```
cout << fixed << setprecision( 2 ) << setw( 10 ) << pow( 2.5, 3 ) << endl;
```
Output is:
```
     15.63
```
d)
```
x = 1;
while ( x <= 20 )
{
   if ( x % 5 == 0 )
      cout << x << endl;
   else
      cout << x << '\t';
   ++x;
}
```

e)
```
for ( x = 1; x <= 20; ++x )
{
   if ( x % 5 == 0 )
      cout << x << endl;
   else
      cout << x << '\t';
}
```

5.3 a) *Error:* The semicolon after the `while` header causes an infinite loop.
Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.

b) *Error:* Using a floating-point number to control a `for` repetition statement.
Correction: Use an `int` and perform the proper calculation to get the values you desire.

```
for ( y = 1; y != 10; ++y )
   cout << ( static_cast< double >( y ) / 10 ) << endl;
```

c) *Error:* Missing `break` statement in the first `case`.
Correction: Add a `break` statement at the end of the first `case`. This is not an error if you want the statement of `case 2:` to execute every time the `case 1:` statement executes.

d) *Error:* Improper relational operator used in the loop-continuation condition.
Correction: Use `<=` rather than `<`, or change `10` to `11`.

Exercises

5.4 *(Find the Code Errors)* Find the error(s), if any, in each of the following:

a)
```
For ( x = 100, x >= 1, ++x )
   cout << x << endl;
```

b) The following code should print whether integer `value` is odd or even:

```
switch ( value % 2 )
{
   case 0:
      cout << "Even integer" << endl;
   case 1:
      cout << "Odd integer" << endl;
}
```

c) The following code should output the odd integers from 19 to 1:

```
for ( x = 19; x >= 1; x += 2 )
   cout << x << endl;
```

d) The following code should output the even integers from 2 to 100:

```
counter = 2;

do
{
   cout << counter << endl;
   counter += 2;
} While ( counter < 100 );
```

5.5 *(Summing Integers)* Write a program that uses a `for` statement to sum a sequence of integers. Assume that the first integer read specifies the number of values remaining to be entered. Your program should read only one value per input statement. A typical input sequence might be

```
5 100 200 300 400 500
```

where the `5` indicates that the subsequent `5` values are to be summed.

5.6 *(Averaging Integers)* Write a program that uses a `for` statement to calculate the average of several integers. Assume the last value read is the sentinel `9999`. For example, the sequence `10 8 11 7 9 9999` indicates that the program should calculate the average of all the values preceding `9999`.

5.7 *(What Does This Program Do?)* What does the following program do?

```cpp
// Exercise 5.7: ex05_07.cpp
// What does this program print?
#include <iostream>
using namespace std;

int main()
{
   int x; // declare x
   int y; // declare y

   // prompt user for input
   cout << "Enter two integers in the range 1-20: ";
   cin >> x >> y;  // read values for x and y

   for ( int i = 1; i <= y; ++i ) // count from 1 to y
   {
      for ( int j = 1; j <= x; ++j ) // count from 1 to x
         cout << '@'; // output @

      cout << endl; // begin new line
   } // end outer for
} // end main
```

5.8 *(Find the Smallest Integer)* Write a program that uses a `for` statement to find the smallest of several integers. Assume that the first value read specifies the number of values remaining.

5.9 *(Product of Odd Integers)* Write a program that uses a `for` statement to calculate and print the product of the odd integers from 1 to 15.

5.10 *(Factorials)* The factorial function is used frequently in probability problems. Using the definition of factorial in Exercise 4.34, write a program that uses a `for` statement to evaluate the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

5.11 *(Compound Interest)* Modify the compound interest program of Section 5.4 to repeat its steps for the interest rates 5%, 6%, 7%, 8%, 9% and 10%. Use a `for` statement to vary the interest rate.

5.12 *(Drawing Patterns with Nested `for` Loops)* Write a program that uses `for` statements to print the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form `cout << '*';` (this causes the asterisks to print side by side). [*Hint:* The last two patterns require that each line begin with an appropriate number of blanks. *Extra credit:* Combine your code from the four separate problems into a single program that prints all four patterns side by side by making clever use of nested `for` loops.]

```
(a)             (b)             (c)             (d)
*               **********      **********               *
**              *********        *********              **
***             ********          ********             ***
****            *******            *******            ****
*****           ******              ******           *****
******          *****                *****          ******
*******         ****                  ****         *******
********        ***                    ***        ********
*********       **                      **       *********
**********      *                        *      **********
```

5.13 *(Bar Chart)* One interesting application of computers is drawing graphs and bar charts. Write a program that reads five numbers (each between 1 and 30). Assume that the user enters only valid values. For each number that is read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number 7, it should print `*******`.

5.14 *(Calculating Total Sales)* A mail order house sells five different products whose retail prices are: product 1 — $2.98, product 2—$4.50, product 3—$9.98, product 4—$4.49 and product 5—$6.87. Write a program that reads a series of pairs of numbers as follows:

a) product number
b) quantity sold

Your program should use a `switch` statement to determine the retail price for each product. Your program should calculate and display the total retail value of all products sold. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

5.15 *(`GradeBook` Modification)* Modify the `GradeBook` program of Fig. 5.9–Fig. 5.11 to calculate the grade-point average. A grade of A is worth 4 points, B is worth 3 points, and so on.

5.16 *(Compound Interest Calculation)* Modify Fig. 5.6 so it uses only integers to calculate the compound interest. [*Hint:* Treat all monetary amounts as numbers of pennies. Then "break" the result into its dollar and cents portions by using the division and modulus operations. Insert a period.]

5.17 *(What Prints?)* Assume `i = 1`, `j = 2`, `k = 3` and `m = 2`. What does each statement print?

a) `cout << ( i == 1 ) << endl;`
b) `cout << ( j == 3 ) << endl;`
c) `cout << ( i >= 1 && j < 4 ) << endl;`
d) `cout << ( m <= 99 && k < m ) << endl;`
e) `cout << ( j >= i || k == m ) << endl;`
f) `cout << ( k + m < j || 3 - j >= k ) << endl;`
g) `cout << ( !m ) << endl;`
h) `cout << ( !( j - m ) ) << endl;`
i) `cout << ( !( k > m ) ) << endl;`

5.18 *(Number Systems Table)* Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers in the range 1–256. If you are not familiar with these number systems, read Appendix D, first. [*Hint:* You can use the stream manipulators `dec`, `oct` and `hex` to display integers in decimal, octal and hexadecimal formats, respectively.]

5.19 *(Calculating π)* Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the approximate value of π after each of the first 1000 terms of this series.

5.20 *(Pythagorean Triples)* A right triangle can have sides that are all integers. A set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for `side1`, `side2` and `hypotenuse` all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This is an example of **brute force** computing. You'll learn in more advanced computer science courses that there are many interesting problems for which there's no known algorithmic approach other than sheer brute force.

5.21 *(Calculating Salaries)* A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and "time-and-a-half"—1.5 times their hourly wage—for overtime hours worked), commission workers (who receive $250 plus 5.7 percent of their gross weekly sales), or pieceworkers (who receive a fixed amount of money per item for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute the weekly pay for each employee. You do not know the number of employees in advance. Each type of employee has its own pay code: Managers have code 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a `switch` to compute each employee's pay according to that employee's paycode. Within the `switch`, prompt the user (i.e., the payroll clerk) to enter the appropriate facts your program needs to calculate each employee's pay according to that employee's paycode.

5.22 *(De Morgan's Laws)* In this chapter, we discussed the logical operators `&&`, `||` and `!`. De Morgan's laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(` *condition1* `&&` *condition2* `)` is logically equivalent to the expression `( !`*condition1* `||` `!`*condition2* `)`. Also, the expression `!(` *condition1* `||` *condition2* `)` is logically equivalent to the expression `( !`*condition1* `&&` `!`*condition2* `)`. Use De Morgan's laws to write equivalent expressions for each of the following, then write a program to show that the original expression and the new expression in each case are equivalent:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.23 *(Diamond of Asterisks)* Write a program that prints the following diamond shape. You may use output statements that print a single asterisk (`*`), a single blank or a single newline. Maximize your use of repetition (with nested `for` statements) and minimize the number of output statements.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

5.24 *(Diamond of Asterisks Modification)* Modify Exercise 5.23 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond, then display a diamond of the appropriate size.

5.25 *(Removing `break` and `continue`)* A criticism of the `break` and `continue` statements is that each is unstructured. These statements can always be replaced by structured statements. Describe in general how you'd remove any `break` statement from a loop in a program and replace it with some structured equivalent. [*Hint:* The `break` statement leaves a loop from within the body of the loop. Another way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you developed here to remove the `break` statement from the program of Fig. 5.13.

5.26 What does the following program segment do?

```
for ( int i = 1; i <= 5; ++i )
{
   for ( int j = 1; j <= 3; ++j )
   {
      for ( int k = 1; k <= 4; ++k )
         cout << '*';

      cout << endl;
   } // end inner for

   cout << endl;
} // end outer for
```

5.27 *(Removing the `continue` Statement)* Describe in general how you'd remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you developed here to remove the `continue` statement from the program of Fig. 5.14.

5.28 *("The Twelve Days of Christmas" Song)* Write a program that uses repetition and `switch` statements to print the song "The Twelve Days of Christmas." One `switch` statement should be used to print the day (i.e., "first," "second," etc.). A separate `switch` statement should be used to print the remainder of each verse. Visit the website `www.12days.com/library/carols/12daysofxmas.htm` for the complete lyrics to the song.

5.29 *(Peter Minuit Problem)* Legend has it that, in 1626, Peter Minuit purchased Manhattan Island for $24.00 in barter. Did he make a good investment? To answer this question, modify the compound interest program of Fig. 5.6 to begin with a principal of $24.00 and to calculate the amount of interest on deposit if that money had been kept on deposit until this year (e.g., 384 years through 2010). Place the `for` loop that performs the compound interest calculation in an outer `for` loop that varies the interest rate from 5% to 10% to observe the wonders of compound interest.

Making a Difference

5.30 *(Global Warming Facts Quiz)* The controversial issue of global warming has been widely publicized by the film *An Inconvenient Truth*, featuring former Vice President Al Gore. Mr. Gore and a U.N. network of scientists, the Intergovernmental Panel on Climate Change, shared the 2007 Nobel Peace Prize in recognition of "their efforts to build up and disseminate greater knowledge about man-made climate change." Research *both* sides of the global warming issue online (you might want to search for phrases like "global warming skeptics"). Create a five-question multiple-choice quiz on global warming, each question having four possible answers (numbered 1–4). Be objective and try to fairly represent both sides of the issue. Next, write an application that administers the quiz, calculates the number of correct answers (zero through five) and returns a message to the user. If the user correctly answers five questions, print "Excellent"; if four, print "Very good"; if three or fewer, print "Time to brush up on your knowledge of global warming," and include a list of the websites where you found your facts.

5.31 *(Tax Plan Alternatives; The "FairTax")* There are many proposals to make taxation fairer. Check out the FairTax initiative in the United States at

```
www.fairtax.org/site/PageServer?pagename=calculator
```

Research how the proposed FairTax works. One suggestion is to eliminate income taxes and most other taxes in favor of a 23% consumption tax on all products and services that you buy. Some FairTax opponents question the 23% figure and say that because of the way the tax is calculated, it would be more accurate to say the rate is 30%—check this carefully. Write a program that prompts the user to enter expenses in various expense categories they have (e.g., housing, food, clothing, transportation, education, health care, vacations), then prints the estimated FairTax that person would pay.

5.32 *(Facebook User Base Growth)* According to CNNMoney.com, Facebook hit 500 million users in July of 2010 and its user base has been growing at a rate of 5% per month. Using the compound-growth technique you learned in Fig. 5.6 and assuming this growth rate continues, how many months will it take for Facebook to grow its user base to one billion users? How many months will it take for Facebook to grow its user base to two billion users (which, at the time of this writing, was the total number of people on the Internet)?

6 Functions and an Introduction to Recursion

Form ever follows function.
—Louis Henri Sullivan

E pluribus unum.
(One composed of many.)
—Virgil

O! call back yesterday, bid time return.
—William Shakespeare

Answer me in one word.
—William Shakespeare

There is a point at which methods devour themselves.
—Frantz Fanon

Objectives

In this chapter you'll learn:

- To construct programs modularly from functions.
- To use common math library functions.
- The mechanisms for passing data to functions and returning results.
- How the function call/return mechanism is supported by the function call stack and activation records.
- To use random number generation to implement game-playing applications.
- How the visibility of identifiers is limited to specific regions of programs.
- To write and use recursive functions.

Outline

6.1 Introduction

Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters of this book. Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or components. This technique is called **divide and conquer**.

We'll overview a portion of the C++ Standard Library's math functions. Next, you'll learn how to declare a function with more than one parameter. We'll also present additional information about function prototypes and how the compiler uses them to convert the type of an argument in a function call to the type specified in a function's parameter list, if necessary.

Next, we'll take a brief diversion into simulation techniques with random number generation and develop a version of the casino dice game called craps that uses most of the programming techniques you've learned.

We then present C++'s storage classes and scope rules. These determine the period during which an object exists in memory and where its identifier can be referenced in a program. You'll learn how C++ keeps track of which function is currently executing, how parameters and other local variables of functions are maintained in memory and how a function knows where to return after it completes execution. We discuss topics that help improve program performance—inline functions that can eliminate the overhead of a function call and reference parameters that can be used to pass large data items to functions efficiently.

Many of the applications you develop will have more than one function of the same name. This technique, called function overloading, is used to implement functions that perform similar tasks for arguments of different types or possibly for different numbers of arguments. We consider function templates—a mechanism for defining a family of overloaded functions. The chapter concludes with a discussion of functions that call themselves, either directly, or indirectly (through another function)—a topic called recursion.

6.2 Program Components in C++

As you've seen, C++ programs are typically written by combining new functions and classes you write with "prepackaged" functions and classes available in the C++ Standard Library. The C++ Standard Library provides a rich collection of functions for common mathematical calculations, string manipulations, character manipulations, input/output, error checking and many other useful operations.

Functions allow you to modularize a program by separating its tasks into self-contained units. You've used a combination of library functions and your own functions in every program you've written. Functions you write are referred to as **user-defined functions** or **programmer-defined functions.** The statements in function bodies are written only once, are reused from perhaps several locations in a program and are hidden from other functions.

There are several motivations for modularizing a program with functions. One is the divide-and-conquer approach. Another is software reuse. For example, in earlier programs, we did not have to define how to read a line of text from the keyboard—C++ provides this capability via the `getline` function of the `<string>` header. A third motivation is to avoid repeating code. Also, dividing a program into meaningful functions makes the program easier to debug and maintain.

Software Engineering Observation 6.1

To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively.

As you know, a function is invoked by a function call, and when the called function completes its task, it either returns a result or simply returns control to the caller. An analogy to this program structure is the hierarchical form of management (Figure 6.1). A boss (similar to the calling function) asks a worker (similar to the called function) to perform a task and report back (i.e., return) the results after completing the task. The boss function does *not* know how the worker function performs its designated tasks. The worker may also call other worker functions, unbeknownst to the boss. This *hiding of implementation details* promotes good software engineering. Figure 6.1 shows the `boss` function communicating with several worker functions. The `boss` function divides the responsibilities among the `worker` functions, and `worker1` acts as a "boss function" to `worker4` and `worker5`.

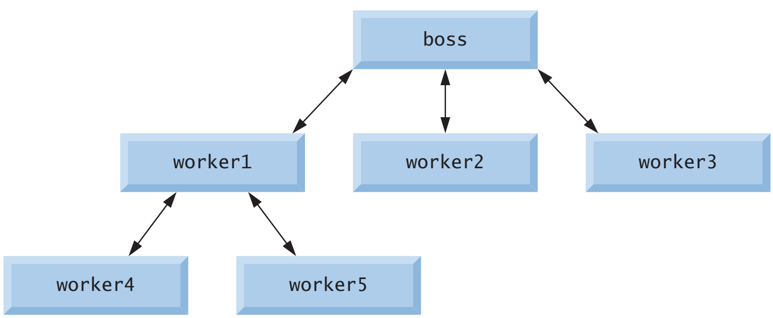


Fig. 6.1 | Hierarchical boss function/worker function relationship.

6.3 Math Library Functions

Sometimes functions, such as `main`, are *not* members of a class. Such functions are called **global functions**. Like a class's member functions, the function prototypes for global functions are placed in headers, so that the global functions can be reused in any program that includes the header and that can link to the function's object code. For example, recall that we used function `pow` of the `<cmath>` header to raise a value to a power in Figure 5.6. We introduce various functions from the `<cmath>` header here to present the concept of global functions that do not belong to a particular class.

The `<cmath>` header provides a collection of functions that enable you to perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the function call

```
sqrt( 900.0 )
```

The preceding expression evaluates to `30.0`. Function `sqrt` takes an argument of type `double` and returns a `double` result. There's no need to create any objects before calling function `sqrt`. Also, *all* functions in the `<cmath>` header are global functions—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments.

Function arguments may be constants, variables or more complex expressions. If `c = 13.0`, `d = 3.0` and `f = 4.0`, then the statement

```
cout << sqrt( c + d * f ) << endl;
```

displays the square root of `13.0 + 3.0 * 4.0 = 25.0`—namely, `5.0`. Some math library functions are summarized in Fig. 6.2. In the figure, the variables `x` and `y` are of type `double`.

Function	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential function e^x	`exp( 1.0 )` is `2.718282` `exp( 2.0 )` is `7.389056`

`fabs( x )`	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 2.6, 1.2 )` is `0.2`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0` `log10( 100.0 )` is `2.0`

Fig. 6.2 | Math library functions. (Part 1 of 2.)

Function	Description	Example
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x* (where *x* is a nonnegative value)	`sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Fig. 6.2 | Math library functions. (Part 2 of 2.)

6.4 Function Definitions with Multiple Parameters

Let's consider functions with *multiple* parameters. Figures 6.3–6.5 modify class `GradeBook` by including a user-defined function called `maximum` that determines and returns the largest of three `int` grades. When the application executes, the `main` function (lines 5–13 of Fig. 6.5) creates one `GradeBook` object (line 8) and calls its `inputGrades` member function (line 11) to read three integer grades from the user. In class `GradeBook`'s implementation file (Fig. 6.4), lines 52–53 of member function `inputGrades` prompt the user to enter three integer values and read them from the user. Line 56 calls member function `maximum` (defined in lines 60–73). Function `maximum` determines the largest value, then the `return` statement (line 72) returns that value to the point at which function `inputGrades` invoked `maximum` (line 56). Member function `inputGrades` then stores `maximum`'s return value in data member `maximumGrade`. This value is then output by calling function `displayGradeReport` (line 12 of Fig. 6.5). [*Note:* We named this function `displayGradeReport` because subsequent versions of class `GradeBook` will use this function to display a complete grade report, including the maximum and minimum grades.] In Chapter 7, Arrays and Vectors, we'll enhance class `GradeBook` to process an arbitrary number of grades.

```
// Fig. 6.3: GradeBook.h
// Definition of class GradeBook that finds the maximum of three grades.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor initializes course name
   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void inputGrades(); // input three grades from user
```

Fig. 6.3 | GradeBook header. (Part 1 of 2.)

```
   void displayGradeReport(); // display a report based on the grades
   int maximum( int, int, int ); // determine max of 3 values
private:
   string courseName; // course name for this GradeBook
   int maximumGrade; // maximum of three grades
}; // end class GradeBook
```

Fig. 6.3 | GradeBook header. (Part 2 of 2.)

```
// Fig. 6.4: GradeBook.cpp
// Member-function definitions for class GradeBook that
// determines the maximum of three grades.
#include <iostream>
using namespace std;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument;
// initializes maximumGrade to 0
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
   maximumGrade = 0; // this value will be replaced by the maximum grade
} // end GradeBook constructor

// function to set the course name; limits name to 25 or fewer characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage
```

Fig. 6.4 | GradeBook class defines function `maximum`. (Part 1 of 2.)

```

// input three grades from user; determine maximum
void GradeBook::inputGrades()
{
   int grade1; // first grade entered by user
   int grade2; // second grade entered by user
   int grade3; // third grade entered by user

   cout << "Enter three integer grades: ";
   cin >> grade1 >> grade2 >> grade3;

   // store maximum in member maximumGrade
   maximumGrade = maximum( grade1, grade2, grade3 );
} // end function inputGrades

// returns the maximum of its three integer parameters
int GradeBook::maximum( int x, int y, int z )
{
   int maximumValue = x; // assume x is the largest to start

   // determine whether y is greater than maximumValue
   if ( y > maximumValue )
      maximumValue = y; // make y the new maximumValue

   // determine whether z is greater than maximumValue
   if ( z > maximumValue )
      maximumValue = z; // make z the new maximumValue

   return maximumValue;
} // end function maximum

// display a report based on the grades entered by user
void GradeBook::displayGradeReport()
{
   // output maximum of grades entered
   cout << "Maximum of grades entered: " << maximumGrade << endl;
} // end function displayGradeReport
```

Fig. 6.4 | GradeBook class defines function `maximum`. (Part 2 of 2.)

```
// Fig. 6.5: fig06_05.cpp
// Create GradeBook object, input grades and display grade report.
#include "GradeBook.h" // include definition of class GradeBook

int main()
{
   // create GradeBook object
   GradeBook myGradeBook( "CS101 C++ Programming" );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.inputGrades(); // read grades from user
```

Fig. 6.5 | Demonstrating function `maximum`. (Part 1 of 2.)

```
    myGradeBook.displayGradeReport(); // display report based on grades
} // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 86 67 75
Maximum of grades entered: 86
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 86 75
Maximum of grades entered: 86
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86
```

Fig. 6.5 | Demonstrating function `maximum`. (Part 2 of 2.)

Software Engineering Observation 6.2

The commas used in line 56 of Fig. 6.4 to separate the arguments to function `maximum` *are* not *comma operators as discussed in Section 5.3. The comma operator guarantees that its operands are evaluated left to right. The order of evaluation of a function's arguments, however, is* not *specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders. The C++ standard* does *guarantee that all arguments in a function call are evaluated* before *the called function executes.*

Portability Tip 6.1

Sometimes when a function's arguments are expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes between compilers, the argument values passed to the function could vary, causing subtle logic errors.

Error-Prevention Tip 6.1

If you have doubts about the order of evaluation of a function's arguments and whether the order would affect the values passed to the function, evaluate the arguments in separate assignment statements before the function call, assign the result of each expression to a local variable, then pass those variables as arguments to the function.

Member function `maximum`'s prototype (Fig. 6.3, line 17) indicates that the function returns an integer value, has the name `maximum` and requires three integer parameters to perform its task. The function header (Fig. 6.4, line 60) matches the function prototype and indicates that the parameter names are `x`, `y` and `z`. When `maximum` is called (Fig. 6.4, line 56), the parameter `x` is initialized with the value of the argument `grade1`, the parameter `y` is initialized with the value of the argument `grade2` and the parameter `z` is initialized with the value of the argument `grade3`. There must be one argument in the function call for each parameter (also called a **formal parameter**) in the function definition.

Notice that multiple parameters are specified in both the function prototype and the function header as a comma-separated list. The compiler refers to the function prototype to check that calls to `maximum` contain the correct number and types of arguments and that the types of the arguments are in the correct order. In addition, the compiler uses the prototype to ensure that the value returned by the function can be used correctly in the expression that called the function (e.g., a function call that returns `void` *cannot* be used as the right side of an assignment statement). Each argument must be *consistent* with the type of the corresponding parameter. For example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not a string like `"hello"`. If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types. Section 6.5 discusses this conversion.

Common Programming Error 6.1

Declaring function parameters of the same type as `double x, y` *instead of* `double x, double y` *is a syntax error—a type is required for each parameter in the parameter list.*

Common Programming Error 6.2

Compilation errors occur if the function prototype, header and calls do not all agree in the number, type and order of arguments and parameters, and in the return type. Linker errors and other types of errors can occur as well as you'll see later in the book.

Software Engineering Observation 6.3

A function that has many parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. Limit the function header to one line if possible.

To determine the maximum value (lines 60–73 of Fig. 6.4), we begin with the assumption that parameter `x` contains the largest value, so line 62 of function `maximum` declares local variable `maximumValue` and initializes it with the value of parameter `x`. Of course, it's possible that parameter `y` or `z` contains the actual largest value, so we must compare each of these values with `maximumValue`. The `if` statement in lines 65–66 determines whether `y` is greater than `maximumValue` and, if so, assigns `y` to `maximumValue`. The `if` statement in lines 69–70 determines whether `z` is greater than `maximumValue` and, if so, assigns `z` to `maximumValue`. At this point the largest of the three values is in `maximumValue`, so line 72 returns that value to the call in line 56. When program control returns to the point in the program where `maximum` was called, `maximum`'s parameters `x`, `y` and `z` are no longer accessible to the program.

There are three ways to return control to the point at which a function was invoked. If the function does *not* return a result (i.e., it has a `void` return type), control returns when the program reaches the function-ending right brace, or by execution of the statement

```
return;
```

If the function *does* return a result, the statement

```
return expression;
```

evaluates *expression* and returns the value of *expression* to the caller.

6.5 Function Prototypes and Argument Coercion

A function prototype (also called a **function declaration**) tells the compiler the name of a function, the type of data it returns, the number of parameters it expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

Software Engineering Observation 6.4

Function prototypes are required. Use #include *preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the headers of the appropriate libraries (e.g., the prototype for* sqrt *is in header* <cmath>*; a partial list of C++ Standard Library headers appears in Section 6.6). Also use* #include *to obtain headers containing function prototypes written by you or other programmers.*

Common Programming Error 6.3

If a function is defined before *it's invoked, then its definition also serves as the function's prototype, so a separate prototype is unnecessary. If a function is invoked* before *it's defined, and that function does not have a function prototype, a compilation error occurs.*

Software Engineering Observation 6.5

Always provide function prototypes, even though it's possible to omit them when functions are defined before *they're used (in which case the function header acts as the function prototype as well). Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).*

Function Signatures

The portion of a function prototype that includes the *name of the function* and the *types of its arguments* is called the **function signature** or simply the **signature**. The function signature does not specify the function's return type. *Functions in the same scope must have unique signatures*. The scope of a function is the region of a program in which the function is known and accessible. We'll say more about scope in Section 6.10.

Common Programming Error 6.4

It's a compilation error if two functions in the same scope have the same signature but different return types.

In Fig. 6.3, if the function prototype in line 17 had been written

```
void maximum( int, int, int );
```

the compiler would report an error, because the `void` return type in the function prototype would differ from the `int` return type in the function header. Similarly, such a prototype would cause the statement

```
cout << maximum( 6, 7, 0 );
```

to generate a compilation error, because that statement depends on `maximum` to return a value to be displayed.

Argument Coercion

An important feature of function prototypes is **argument coercion**—i.e., forcing arguments to the appropriate types specified by the parameter declarations. For example, a program can call a function with an integer argument, even though the function prototype specifies a `double` argument—the function will still work correctly.

Argument Promotion Rules

Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called. These conversions occur as specified by C++'s **promotion rules**. The promotion rules indicate the implicit conversions that the compiler can perform between fundamental types. An `int` can be converted to a `double`. However, a `double` converted to an `int` *truncates* the fractional part of the `double` value. Keep in mind that `double` variables can hold numbers of much greater magnitude than `int` variables, so the loss of data may be considerable. Values may also be modified when converting large integer types to small integer types (e.g., `long` to `short`), signed to unsigned or unsigned to signed. Unsigned integers range from 0 to approximately twice the positive range of the corresponding signed type.

The promotion rules apply to expressions containing values of two or more data types; such expressions are also referred to as **mixed-type expressions**. The type of each value in a mixed-type expression is promoted to the "highest" type in the expression (actually a *temporary* version of each value is created and used for the expression—the original values remain unchanged). Promotion also occurs when the type of a function argument does not match the parameter type specified in the function definition or prototype. Figure 6.6 lists the fundamental data types in order from "highest type" to "lowest type."

Data types	
`long double`	
`double`	
`float`	
`unsigned long long int`	(synonymous with `unsigned long long`; in the new standard)
`long long int`	(synonymous with `long long`; in the new standard)
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char`	
`bool`	

Fig. 6.6 | Promotion hierarchy for fundamental data types.

Converting values to lower fundamental types can result in incorrect values. Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type (some compilers will issue a warning in this case) or by using a *cast operator* (see Section 4.9). Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types. If a `square` function that uses an integer parameter is called with a floating-point argument, the argument is converted to `int` (a lower type), and `square` could return an incorrect value. For example, `square(4.5)` returns `16`, not `20.25`.

Common Programming Error 6.5

Converting from a higher data type in the promotion hierarchy to a lower type, or between signed and unsigned, can corrupt the data value, causing a loss of information.

Common Programming Error 6.6

It's a compilation error if the arguments in a function call do not match the number and types of the parameters declared in the corresponding function prototype. It's also an error if the number of arguments in the call matches, but the arguments cannot be implicitly converted to the expected types.

6.6 C++ Standard Library Headers

The C++ Standard Library is divided into many portions, each with its own header. The headers contain the function prototypes for the related functions that form each portion of the library. The headers also contain definitions of various class types and functions, as well as constants needed by those functions. A header "instructs" the compiler on how to interface with library and user-written components.

Figure 6.7 lists some common C++ Standard Library headers, most of which are discussed later in the book. The term "macro" that's used several times in Fig. 6.7 is discussed in detail in Appendix E, Preprocessor. Header names ending in `.h` are "old-style" headers that have been superseded by the C++ Standard Library headers. We use only the C++ Standard Library versions of each header in this book to ensure that our examples will work on most standard C++ compilers.

Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 16, Exception Handling: A Deeper Look; Chapter 21, Bits, Characters, C Strings and `struct`s; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.

Fig. 6.7 | C++ Standard Library headers. (Part 1 of 3.)

Standard Library header	Explanation
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these headers in Chapter 22, Standard Template Library (STL).
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 21, Bits, Characters, C Strings and `struct`s.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 11, Operator Overloading; Class `string`.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 17, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 22.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 22.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 22.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).

Fig. 6.7 | C++ Standard Library headers. (Part 2 of 3.)

Standard Library header	Explanation
<limits>	Contains classes for defining the numerical data type limits on each computer platform.
<utility>	Contains classes and functions that are used by many C++ Standard Library headers.

Fig. 6.7 | C++ Standard Library headers. (Part 3 of 3.)

6.7 Case Study: Random Number Generation

We now take a brief and hopefully entertaining diversion into a popular programming application, namely simulation and game playing. In this and the next section, we develop a game-playing program that includes multiple functions.

The element of chance can be introduced into computer applications by using the C++ Standard Library function **rand**. Consider the following statement:

```
i = rand();
```

The function rand generates an unsigned integer between 0 and RAND_MAX (a symbolic constant defined in the <cstdlib> header). You can determine the value of RAND_MAX for your system simply by displaying the constant. If rand truly produces integers at random, every number between 0 and RAND_MAX has an equal *chance* (or probability) of being chosen each time rand is called.

The range of values produced directly by the function rand often is different than what a specific application requires. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates rolling a six-sided die would require random integers in the range 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1 through 4.

Rolling a Six-Sided Die

To demonstrate rand, Fig. 6.8 simulates 20 rolls of a six-sided die and displays the value of each roll. The function prototype for the rand function is in <cstdlib>. To produce integers in the range 0 to 5, we use the modulus operator (%) with rand as follows:

```
rand() % 6
```

This is called **scaling**. The number 6 is called the **scaling factor**. We then **shift** the range of numbers produced by adding 1 to our previous result. Figure 6.8 confirms that the results are in the range 1 to 6.

```
// Fig. 6.8: fig06_08.cpp
// Shifted and scaled random integers.
#include <iostream>
#include <iomanip>
```

Fig. 6.8 | Shifted, scaled integers produced by 1 + rand() % 6. (Part 1 of 2.)

```cpp
#include <cstdlib> // contains function prototype for rand
using namespace std;

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20; ++counter )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for
} // end main
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Fig. 6.8 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 2 of 2.)

Rolling a Six-Sided Die 6,000,000 Times

To show that the numbers produced by `rand` occur with approximately equal likelihood, Fig. 6.9 simulates 6,000,000 rolls of a die. Each integer in the range 1 to 6 should appear approximately 1,000,000 times. This is confirmed by the program's output.

```cpp
// Fig. 6.9: fig06_09.cpp
// Rolling a six-sided die 6,000,000 times.
#include <iostream>
#include <iomanip>
#include <cstdlib> // contains function prototype for rand
using namespace std;

int main()
{
   int frequency1 = 0; // count of 1s rolled
   int frequency2 = 0; // count of 2s rolled
   int frequency3 = 0; // count of 3s rolled
   int frequency4 = 0; // count of 4s rolled
   int frequency5 = 0; // count of 5s rolled
   int frequency6 = 0; // count of 6s rolled

   int face; // stores most recently rolled value

   // summarize results of 6,000,000 rolls of a die
   for ( int roll = 1; roll <= 6000000; ++roll )
   {
```

Fig. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 1 of 2.)

```
22         face = 1 + rand() % 6; // random number from 1 to 6
23
24         // determine roll value 1-6 and increment appropriate counter
25         switch ( face )
26         {
27            case 1:
28               ++frequency1; // increment the 1s counter
29               break;
30            case 2:
31               ++frequency2; // increment the 2s counter
32               break;
33            case 3:
34               ++frequency3; // increment the 3s counter
35               break;
36            case 4:
37               ++frequency4; // increment the 4s counter
38               break;
39            case 5:
40               ++frequency5; // increment the 5s counter
41               break;
42            case 6:
43               ++frequency6; // increment the 6s counter
44               break;
45            default: // invalid value
46               cout << "Program should never get here!";
47         } // end switch
48      } // end for
49
50      cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
51      cout << "   1" << setw( 13 ) << frequency1
52         << "\n   2" << setw( 13 ) << frequency2
53         << "\n   3" << setw( 13 ) << frequency3
54         << "\n   4" << setw( 13 ) << frequency4
55         << "\n   5" << setw( 13 ) << frequency5
56         << "\n   6" << setw( 13 ) << frequency6 << endl;
57   } // end main
```

```
Face    Frequency
   1       999702
   2      1000823
   3       999378
   4       998898
   5      1000777
   6      1000422
```

Fig. 6.9 | Rolling a six-sided die 6,000,000 times. (Part 2 of 2.)

As the output shows, we can simulate the rolling of a six-sided die by scaling and shifting the values produced by `rand`. The program should *never* get to the `default` case (lines 45–46) in the `switch` structure, because the `switch`'s controlling expression (`face`) always has values in the range 1–6; however, we provide the `default` case as a matter of good practice. After we study arrays in Chapter 7, we show how to replace the entire `switch` structure in Fig. 6.9 elegantly with a single-line statement.

Error-Prevention Tip 6.2

Provide a `default` *case in a* `switch` *to catch errors even if you are absolutely, positively certain that you have no bugs!*

Randomizing the Random Number Generator

Executing the program of Fig. 6.8 again produces

```
        6         6         5         5         6
        5         1         1         5         3
        6         6         2         4         2
        6         2         3         4         1
```

The program prints exactly the *same* sequence of values shown in Fig. 6.8. How can these be random numbers? *When debugging a simulation program, this repeatability is essential for proving that corrections to the program work properly.*

Function `rand` actually generates **pseudorandom numbers**. Repeatedly calling `rand` produces a sequence of numbers that appears to be random. However, the sequence *repeats* itself each time the program executes. Once a program has been thoroughly debugged, it can be conditioned to produce a *different* sequence of random numbers for each execution. This is called **randomizing** and is accomplished with the C++ Standard Library function **`srand`**. Function `srand` takes an `unsigned` integer argument and **seeds** the `rand` function to produce a different sequence of random numbers for each execution. The new C++ standard provides additional random number capabilities that can produce **nondeterministic random numbers**—a set of random numbers that can't be predicted. Such random number generators are used in simulations and security scenarios where predictability is undesirable.

Using Function `srand`

Figure 6.10 demonstrates function `srand`. The program uses the data type `unsigned`, which is short for `unsigned int`. An `int` is stored in at least two bytes of memory (typically four bytes on 32-bit systems and as much as eight bytes on 64-bit systems) and can have positive and negative values. A variable of type `unsigned int` is also stored in at least two bytes of memory. A two-byte `unsigned int` can have only *nonnegative* values in the range 0–65535. A four-byte `unsigned int` can have only *nonnegative* values in the range 0–4294967295. Function `srand` takes an `unsigned int` value as an argument. The function prototype for the `srand` function is in header `<cstdlib>`.

```cpp
// Fig. 6.10: fig06_10.cpp
// Randomizing the die-rolling program.
#include <iostream>
#include <iomanip>
#include <cstdlib> // contains prototypes for functions srand and rand
using namespace std;

int main()
{
   unsigned seed; // stores the seed entered by the user
```

Fig. 6.10 | Randomizing the die-rolling program. (Part 1 of 2.)

```
   cout << "Enter seed: ";
   cin >> seed;
   srand( seed ); // seed random number generator

   // loop 10 times
   for ( int counter = 1; counter <= 10; ++counter )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for
} // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

Fig. 6.10 | Randomizing the die-rolling program. (Part 2 of 2.)

Let's run the program several times and observe the results. Notice that the program produces a *different* sequence of random numbers each time it executes, provided that the user enters a different seed. We used the *same* seed in the first and third sample outputs, so the *same* series of 10 numbers is displayed in each of those outputs.

To randomize without having to enter a seed each time, we may use a statement like

```
srand( time( 0 ) );
```

This causes the computer to read its clock to obtain the value for the seed. Function **`time`** (with the argument `0` as written in the preceding statement) typically returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT). This value is converted to an `unsigned` integer and used as the seed to the random number generator. The function prototype for `time` is in `<ctime>`.

Generalized Scaling and Shifting of Random Numbers

Previously, we simulated the rolling of a six-sided die with the statement

```
face = 1 + rand() % 6;
```

which always assigns an integer (at random) to variable `face` in the range 1 ≤ `face` ≤ 6. The width of this range (i.e., the number of consecutive integers in the range) is 6 and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale `rand` with the modulus operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that is added to the expression `rand % 6`. We can generalize this result as

```
number = shiftingValue + rand() % scalingFactor;
```

where ***shiftingValue*** is equal to the first number in the desired range of consecutive integers and ***scalingFactor*** is equal to the width of the desired range of consecutive integers.

6.8 Case Study: Game of Chance; Introducing enum

One of the most popular games of chance is a dice game known as "craps," which is played in casinos and back alleys worldwide. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, the player wins. If the sum is 2, 3 or 12 on the first roll (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.*

The program in Fig. 6.11 simulates the game. In the rules, notice that the player must roll two dice on the first roll and on all subsequent rolls. We define function `rollDice` (lines 63–75) to roll the dice and compute and print their sum. The function is defined once, but called from lines 21 and 45. The function takes no arguments and returns the sum of the two dice, so empty parentheses and the return type `int` are indicated in the function prototype (line 8) and function header (line 63).

```
1  // Fig. 6.11: fig06_11.cpp
2  // Craps simulation.
3  #include <iostream>
4  #include <cstdlib> // contains prototypes for functions srand and rand
5  #include <ctime> // contains prototype for function time
6  using namespace std;
7
8  int rollDice(); // rolls dice, calculates and displays sum
9
10 int main()
11 {
12    // enumeration with constants that represent the game status
13    enum Status { CONTINUE, WON, LOST }; // all caps in constants
14
15    int myPoint; // point if no win or loss on first roll
16    Status gameStatus; // can contain CONTINUE, WON or LOST
17
18    // randomize random number generator using current time
19    srand( time( 0 ) );
20
21    int sumOfDice = rollDice(); // first roll of the dice
22
```

Fig. 6.11 | Craps simulation. (Part 1 of 3.)

```
   // determine game status and point (if needed) based on first roll
   switch ( sumOfDice )
   {
      case 7: // win with 7 on first roll
      case 11: // win with 11 on first roll
         gameStatus = WON;
         break;
      case 2: // lose with 2 on first roll
      case 3: // lose with 3 on first roll
      case 12: // lose with 12 on first roll
         gameStatus = LOST;
         break;
      default: // did not win or lose, so remember point
         gameStatus = CONTINUE; // game is not over
         myPoint = sumOfDice; // remember the point
         cout << "Point is " << myPoint << endl;
         break; // optional at end of switch
   } // end switch

   // while game is not complete
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again

      // determine game status
      if ( sumOfDice == myPoint ) // win by making point
         gameStatus = WON;
      else
         if ( sumOfDice == 7 ) // lose by rolling 7 before point
            gameStatus = LOST;
   } // end while

   // display won or lost message
   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;
} // end main

// roll dice, calculate sum and display results
int rollDice()
{
   // pick random die values
   int die1 = 1 + rand() % 6; // first die roll
   int die2 = 1 + rand() % 6; // second die roll

   int sum = die1 + die2; // compute sum of die values

   // display results of this roll
   cout << "Player rolled " << die1 << " + " << die2
      << " = " << sum << endl;
   return sum; // end function rollDice
} // end function rollDice
```

Fig. 6.11 | Craps simulation. (Part 2 of 3.)

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

Fig. 6.11 | Craps simulation. (Part 3 of 3.)

The game is reasonably involved. The player may win or lose on the first roll or on any subsequent roll. The program uses variable `gameStatus` to keep track of this. Variable `gameStatus` is declared to be of new type `Status`. Line 13 declares a user-defined type called an **enumeration**. An enumeration, introduced by the keyword **`enum`** and followed by a **type name** (in this case, `Status`), is a set of integer constants represented by identifiers. The values of these **enumeration constants** start at `0`, unless specified otherwise, and increment by `1`. In the preceding enumeration, the constant `CONTINUE` has the value 0, `WON` has the value 1 and `LOST` has the value 2. The identifiers in an `enum` must be unique, but separate enumeration constants *can* have the same integer value.

Good Programming Practice 6.1

Capitalize the first letter of an identifier used as a user-defined type name.

Good Programming Practice 6.2

Use only uppercase letters in enumeration constant names. This makes these constants stand out in a program and reminds you that enumeration constants are not *variables.*

Variables of user-defined type `Status` can be assigned only one of the three values declared in the enumeration. When the game is won, the program sets variable `gameStatus` to `WON` (lines 28 and 49). When the game is lost, the program sets variable gameStatus to LOST (lines 33 and 52). Otherwise, the program sets variable gameStatus to CONTINUE (line 36) to indicate that the dice must be rolled again.

Another popular enumeration is

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
   SEP, OCT, NOV, DEC };
```

which creates user-defined type Months with enumeration constants representing the months of the year. The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12. Any enumeration constant can be assigned an integer value in the enumeration definition, and subsequent enumeration constants each have a value 1 higher than the preceding constant in the list until the next explicit setting.

After the first roll, if the game is won or lost, the program skips the body of the while statement (lines 43–53) because gameStatus is not equal to CONTINUE. The program proceeds to the if…else statement in lines 56–59, which prints "Player wins" if gameStatus is equal to WON and "Player loses" if gameStatus is equal to LOST.

After the first roll, if the game is not over, the program saves the sum in myPoint (line 37). Execution proceeds with the while statement, because gameStatus is equal to CONTINUE. During each iteration of the while, the program calls rollDice to produce a new sum. If sum matches myPoint, the program sets gameStatus to WON (line 49), the while-test fails, the if…else statement prints "Player wins" and execution terminates. If sum is equal to 7, the program sets gameStatus to LOST (line 52), the while-test fails, the if…else statement prints "Player loses" and execution terminates.

The craps program uses two functions—main and rollDice—and the switch, while, if…else, nested if…else and nested if statements. In the exercises, we further investigate of the game of craps.

Common Programming Error 6.7

Assigning the integer equivalent of an enumeration constant (rather than the enumeration constant, itself) to a variable of the enumeration type is a compilation error.

6.9 Storage Classes

The programs you've seen so far use identifiers for variable names. The attributes of variables include *name*, *type*, *size* and *value*. This chapter also uses identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including **storage class**, scope and **linkage**.

C++ provides five **storage-class specifiers**: **auto**, **register**, **extern**, **mutable** and **static**. This section discusses storage-class specifiers auto, register, extern and static; mutable (discussed in Chapter 24, Other Topics) is used exclusively with classes.

Storage Class, Scope and Linkage

An identifier's *storage class* determines the period during which that identifier exists in memory. Some exist briefly, some are repeatedly created and destroyed and others exist for the entire execution of a program. First we discuss the storage classes **static** and **automatic**.

An identifier's *scope* is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. Section 6.10 discusses the scope of identifiers.

An identifier's linkage determines whether it's known only in the source file where it's declared or across multiple files that are compiled, then linked together. An identifier's storage-class specifier helps determine its storage class and linkage.

Storage Class Categories

The storage-class specifiers can be split into two storage classes: automatic storage class and static storage class. Keywords `auto` and `register` are used to declare variables of the automatic storage class. Such variables are created when program execution enters the block in which they're defined, they exist while the block is active and they're destroyed when the program exits the block.

Local Variables

Only local variables of a function can be of automatic storage class. A function's local variables and parameters normally are of automatic storage class. The storage class specifier `auto` explicitly declares variables of automatic storage class. For example, the following declaration indicates that `double` variable `x` is a local variable of automatic storage class—it exists only in the *nearest enclosing pair of curly braces* within the body of the function in which the definition appears:

```
auto double x;
```

Local variables are of automatic storage class by *default*, so keyword `auto` rarely is used. For this reason, the new C++ standard gives `auto` a new meaning which we discuss in Section 23.9. For the remainder of the text, we refer to variables of automatic storage class simply as automatic variables.

Performance Tip 6.1

Automatic storage is a means of conserving memory, because automatic storage class variables exist in memory only when the block in which they're defined is executing.

Software Engineering Observation 6.6

Automatic storage is an example of the ***principle of least privilege****. In the context of an application, the principle states that code should be granted* only *the amount of privilege and access that it needs to accomplish its designated task,* but no more. *Why should we have variables stored in memory and accessible when they're* not *needed?*

Register Variables

Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.

Performance Tip 6.2

The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers rather than in memory. If intensely used variables such as counters or totals are kept in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.

The compiler might ignore `register` declarations. For example, there might not be a sufficient number of registers available. The following definition *suggests* that the integer variable `counter` be placed in one of the computer's registers; regardless of whether the compiler does this, `counter` is initialized to 1:

```
register int counter = 1;
```

The `register` keyword can be used only with local variables and function parameters.

Performance Tip 6.3

Often, `register` is unnecessary. Optimizing compilers can recognize frequently used variables and may place them in registers without needing a `register` declaration.

Static Storage Class

Keywords `extern` and `static` declare identifiers for variables of the static storage class and for functions. Static-storage-class variables exist in memory from the point at which the program begins execution and last for the duration of the program. Such a variable is initialized once when its declaration is encountered. For functions, the name of the function exists when the program begins execution, just as for all other functions. However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program. Storage class and scope (where a name can be used) are separate issues, as we'll see in Section 6.10.

Identifiers with Static Storage Class

There are two types of identifiers with static storage class—external identifiers (such as **global variables**) and local variables declared with the storage-class specifier `static`. Global variables are created by placing variable declarations *outside* any class or function definition. Global variables retain their values throughout the execution of the program. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file.

Software Engineering Observation 6.7

Declaring a variable as global rather than local allows unintended side effects *to occur when a function that does not need access to the variable accidentally or maliciously modifies it. This is another example of the principle of least privilege. In general, except for truly global resources such as `cin` and `cout`, the use of global variables should be avoided unless there are unique performance requirements.*

Software Engineering Observation 6.8

Variables used only in a particular function should be declared as local variables in that function rather than as global variables.

Local variables declared `static` are still known only in the function in which they're declared, but, unlike automatic variables, `static` local variables retain their values when the function returns to its caller. The next time the function is called, the `static` local variables contain the values they had when the function last completed execution. The following statement declares local variable `count` to be `static` and to be initialized to 1:

```
static int count = 1;
```

All numeric variables of the static storage class are initialized to zero by default, but it's nevertheless a good practice to explicitly initialize all variables.

Storage-class specifiers `extern` and `static` have special meaning when they're applied explicitly to external identifiers such as global variables and global function names. In Appendix F, C Legacy Code Topics, we discuss using `extern` and `static` with external identifiers and multiple-source-file programs.

6.10 Scope Rules

The portion of the program where an identifier can be used is known as its scope. For example, when we declare a local variable in a block, it can be referenced only in that block and in blocks nested within that block. This section discusses four scopes for an identifier—**function scope**, **global namespace scope**, **local scope** and **function-prototype scope**. Later we'll see two other scopes—**class scope** (Chapter 9) and **namespace scope** (Chapter 24).

An identifier declared *outside* any function or class has global namespace scope. Such an identifier is "known" in all functions from the point at which it's declared until the end of the file. Global variables, function definitions and function prototypes placed outside a function all have global namespace scope.

Labels (identifiers followed by a colon such as `start:`) are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced *outside* the function body. Labels are used in `goto` statements (Appendix F).

Identifiers declared inside a block have local scope. Local scope begins at the identifier's declaration and ends at the terminating right brace (`}`) of the block in which the identifier is declared. Local variables have local scope, as do function parameters, which are also local variables of the function. Any block can contain variable declarations. When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates. The inner block "sees" the value of its own local identifier and not the value of the identically named identifier in the enclosing block. Local variables declared `static` still have local scope, even though they exist from the time the program begins execution. Storage duration does not affect the scope of an identifier.

The only identifiers with function prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do *not* require names in the parameter list—only types are required. Names appearing in the parameter list of a function prototype are *ignored* by the compiler. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity. In a single prototype, a particular identifier can be used only once.

Common Programming Error 6.8

Accidentally using the same name for an identifier in an inner block that is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block, is typically a logic error.

Error-Prevention Tip 6.3

Avoid variable names that hide names in outer scopes.

The program of Fig. 6.12 demonstrates scoping issues with global variables, automatic local variables and `static` local variables. Line 10 declares and initializes global variable `x` to 1. This global variable is hidden in any block (or function) that declares a variable named `x`. In `main`, line 14 displays the value of global variable `x`. Line 16 declares a local variable `x` and initializes it to 5. Line 19 outputs this variable to show that the global `x` is hidden in `main`. Next, lines 20–24 define a new block in `main` in which another local variable `x` is initialized to 7 (line 21). Line 23 outputs this variable to show that it *hides* `x` in the outer block of `main`. When the block exits, the variable `x` with value 7 is destroyed automatically. Next, line 26 outputs the local variable `x` in the outer block of `main` to show that it's *no longer hidden*.

```
1  // Fig. 6.12: fig06_12.cpp
2  // Scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x = 1; // global variable
11
12 int main()
13 {
14    cout << "global x in main is " << x << endl;
15
16    int x = 5; // local variable to main
17
18    cout << "local x in main's outer scope is " << x << endl;
19
20    { // start new scope
21       int x = 7; // hides both x in outer scope and global x
22
23       cout << "local x in main's inner scope is " << x << endl;
24    } // end new scope
25
26    cout << "local x in main's outer scope is " << x << endl;
27
28    useLocal(); // useLocal has local x
29    useStaticLocal(); // useStaticLocal has static local x
30    useGlobal(); // useGlobal uses global x
31    useLocal(); // useLocal reinitializes its local x
32    useStaticLocal(); // static local x retains its prior value
33    useGlobal(); // global x also retains its prior value
34
35    cout << "\nlocal x in main is " << x << endl;
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal()
40 {
```

Fig. 6.12 | Scoping example. (Part 1 of 2.)

```
   int x = 25; // initialized each time useLocal is called

   cout << "\nlocal x is " << x << " on entering useLocal" << endl;
   ++x;
   cout << "local x is " << x << " on exiting useLocal" << endl;
} // end function useLocal

// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal()
{
   static int x = 50; // initialized first time useStaticLocal is called

   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
      << endl;
   ++x;
   cout << "local static x is " << x << " on exiting useStaticLocal"
      << endl;
} // end function useStaticLocal

// useGlobal modifies global variable x during each call
void useGlobal()
{
   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting useGlobal" << endl;
} // end function useGlobal
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Fig. 6.12 | Scoping example. (Part 2 of 2.)

To demonstrate other scopes, the program defines three functions, each of which takes no arguments and returns nothing. Function `useLocal` (lines 39–46) declares automatic variable `x` (line 41) and initializes it to 25. When the program calls `useLocal`, the function prints the variable, increments it and prints it again before the function returns program control to its caller. Each time the program calls this function, the function *recreates* automatic variable `x` and *reinitializes* it to 25.

Function `useStaticLocal` (lines 51–60) declares `static` variable `x` and initializes it to 50. Local variables declared as `static` retain their values even when they're out of scope (i.e., the function in which they're declared is not executing). When the program calls `useStaticLocal`, the function prints `x`, increments it and prints it again before the function returns program control to its caller. In the next call to this function, `static` local variable `x` contains the value 51. The *initialization* in line 53 *occurs only once*—the first time `useStaticLocal` is called.

Function `useGlobal` (lines 63–68) does not declare any variables. Therefore, when it refers to variable `x`, the global `x` (line 10, preceding `main`) is used. When the program calls `useGlobal`, the function prints the global variable `x`, multiplies it by 10 and prints it again before the function returns program control to its caller. The next time the program calls `useGlobal`, the global variable has its modified value, 10. After executing functions `useLocal`, `useStaticLocal` and `useGlobal` twice each, the program prints the local variable `x` in `main` again to show that none of the function calls modified the value of `x` in `main`, because the functions all referred to variables in other scopes.

6.11 Function Call Stack and Activation Records

To understand how C++ performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. Think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's normally placed at the top (referred to as **pushing** the dish onto the stack). Similarly, when a dish is removed from the pile, it's normally removed from the top (referred to as **popping** the dish off the stack). Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

One of the most important mechanisms for computer science students to understand is the **function call stack** (sometimes referred to as the **program execution stack**). This data structure—working "behind the scenes"—supports the function call/return mechanism. It also supports the creation, maintenance and destruction of each called function's automatic variables. We explained the last-in, first-out (LIFO) behavior of stacks with our dish-stacking example. As we'll see in Figs. 6.14–6.16, this LIFO behavior is *exactly* what a function does when returning to the function that called it.

As each function is called, it may, in turn, call other functions, which may, in turn, call other functions—all *before* any of the functions returns. Each function eventually must return control to the function that called it. So, somehow, we must keep track of the return addresses that each function needs to return control to the function that called it. The function call stack is the perfect data structure for handling this information. Each time a function calls another function, an entry is pushed onto the stack. This entry, called a **stack frame** or an **activation record**, contains the return address that the called function needs in order to return to the calling function. It also contains some additional information we'll soon discuss. If the called function returns, instead of calling another function before returning, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.

The beauty of the call stack is that each called function always finds the information it needs to return to its caller at the top of the call stack. And, if a function makes a call to another function, a stack frame for the new function call is simply pushed onto the call stack. Thus, the return address required by the newly called function to return to its caller is now located at the top of the stack.

The stack frames have another important responsibility. Most functions have automatic variables—parameters and any local variables the function declares. Automatic variables need to exist while a function is executing. They need to remain active if the function makes calls to other functions. But when a called function returns to its caller, the called function's automatic variables need to "go away." The called function's stack frame is a perfect place to reserve the memory for the called function's automatic variables. That stack frame exists as long as the called function is active. When that function returns—and no longer needs its local automatic variables—its stack frame is popped from the stack, and those local automatic variables are no longer known to the program.

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function call stack. If more function calls occur than can have their activation records stored on the function call stack, an error known as **stack overflow** occurs.

Function Call Stack in Action

Now let's consider how the call stack supports the operation of a `square` function called by `main` (lines 9–14 of Fig. 6.13). First the operating system calls `main`—this pushes an activation record onto the stack (shown in Fig. 6.14). The activation record tells `main` how to return to the operating system (i.e., transfer to return address `R1`) and contains the space for `main`'s automatic variable (i.e., `a`, which is initialized to `10`).

```
 1  // Fig. 6.13: fig06_13.cpp
 2  // square function used to demonstrate the function
 3  // call stack and activation records.
 4  #include <iostream>
 5  using namespace std;
 6
 7  int square( int ); // prototype for function square
 8
 9  int main()
10  {
11     int a = 10; // value to square (local automatic variable in main)
12
13     cout << a << " squared: " << square( a ) << endl; // display a squared
14  } // end main
15
16  // returns the square of an integer
17  int square( int x ) // x is a local variable
18  {
19     return x * x; // calculate square and return result
20  } // end function square
```

Fig. 6.13 | Demonstrating the function call stack and activation records. (Part 1 of 2.)

```
10 squared: 100
```

Fig. 6.13 | Demonstrating the function call stack and activation records. (Part 2 of 2.)

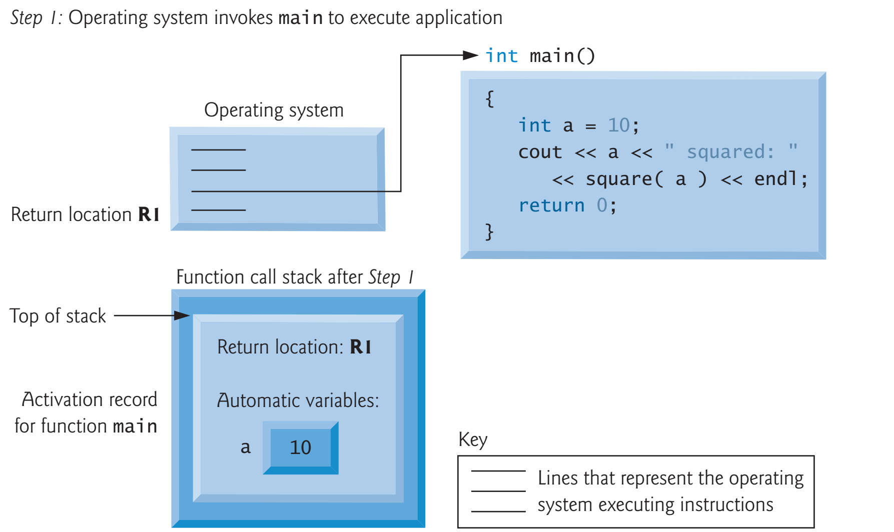


Fig. 6.14 | Function call stack after the operating system invokes `main` to execute the program.

Function `main`—before returning to the operating system—now calls function `square` in line 13 of Fig. 6.13. This causes a stack frame for `square` (lines 17–20) to be pushed onto the function call stack (Fig. 6.15). This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s automatic variable (i.e., `x`).

After `square` calculates the square of its argument, it needs to return to `main`—and no longer needs the memory for its automatic variable `x`. So the stack is popped—giving `square` the return location in `main` (i.e., `R2`) and losing `square`'s automatic variable. Figure 6.16 shows the function call stack *after* `square`'s activation record has been popped.

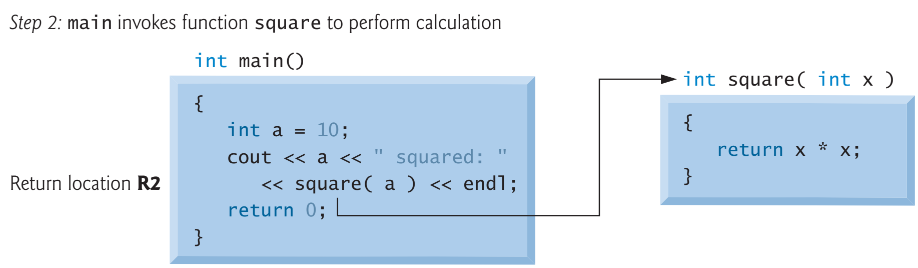


Fig. 6.15 | Function call stack after `main` invokes `square` to perform the calculation. (Part 1 of 2.)

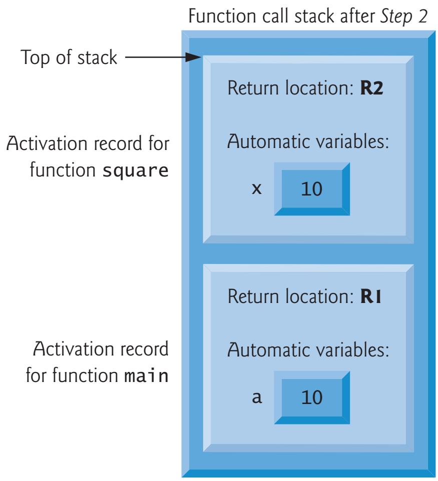


Fig. 6.15 | Function call stack after `main` invokes `square` to perform the calculation. (Part 2 of 2.)

Function `main` now displays the result of calling `square` (line 13). Reaching the closing right brace of `main` causes its activation record to be popped from the stack, gives `main` the address it needs to return to the operating system (i.e., `R1` in Fig. 6.14) and causes the memory for `main`'s automatic variable (i.e., `a`) to become unavailable.

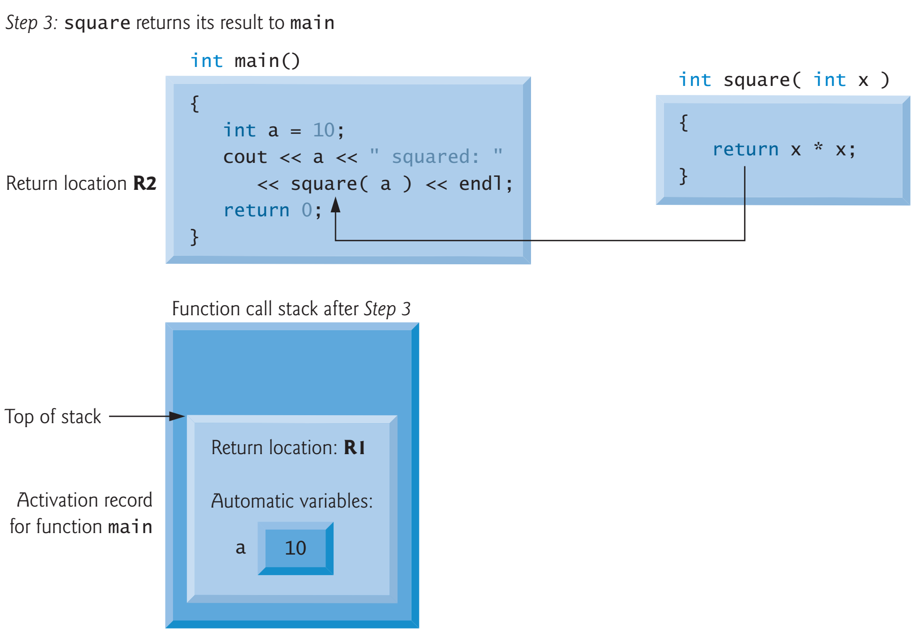


Fig. 6.16 | Function call stack after function `square` returns to `main`.

You've now seen how valuable the stack data structure is in implementing a key mechanism that supports program execution. Data structures have many important applications in computer science. We discuss stacks, queues, lists, trees and other data structures in Chapter 20, Custom Templatized Data Structures, and Chapter 22, Standard Template Library (STL).

6.12 Functions with Empty Parameter Lists

In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses. The prototype

```
void print();
```

specifies that function `print` does not take arguments and does not return a value. Figure 6.17 shows both ways to declare and use functions with empty parameter lists.

```
// Fig. 6.17: fig06_17.cpp
// Functions that take no arguments.
#include <iostream>
using namespace std;

void function1(); // function that takes no arguments
void function2( void ); // function that takes no arguments

int main()
{
   function1(); // call function1 with no arguments
   function2(); // call function2 with no arguments
} // end main

// function1 uses an empty parameter list to specify that
// the function receives no arguments
void function1()
{
   cout << "function1 takes no arguments" << endl;
} // end function1

// function2 uses a void parameter list to specify that
// the function receives no arguments
void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
} // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

Fig. 6.17 | Functions that take no arguments.

6.13 Inline Functions

Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead. C++ provides **inline functions** to help reduce function call overhead—especially for small functions. Placing the qualifier **inline** before a function's return type in the function definition "advises" the compiler to generate a copy of the function's body code in place (when appropriate) to avoid a function call. The trade-off is that multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called. The compiler can *ignore* the `inline` qualifier and typically does so for all but the smallest functions.

Software Engineering Observation 6.9

Any change to an `inline` function requires all clients of the function to be recompiled.

Figure 6.18 uses `inline` function `cube` (lines 9–12) to calculate the volume of a cube. Keyword **const** in function `cube`'s parameter list (line 9) tells the compiler that the function does *not* modify variable `side`. This ensures that `side`'s value is not changed by the function during the calculation. (Keyword `const` is discussed in detail in Chapters 7, 8 and 10.) Notice that the complete definition of function `cube` appears *before* it's used in the program. This is required so that the compiler knows how to expand a `cube` function call into its inlined code. For this reason, reusable inline functions are typically placed in headers, so that their definitions can be included in each source file that uses them.

Software Engineering Observation 6.10

The `const` qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition acts as the prototype.
9  inline double cube( const double side )
10 {
11    return side * side * side; // calculate cube
12 } // end function cube
13
14 int main()
15 {
16    double sideValue; // stores value entered by user
17    cout << "Enter the side length of your cube: ";
18    cin >> sideValue; // read value from user
19
20    // calculate cube of sideValue and display result
21    cout << "Volume of cube with side "
22       << sideValue << " is " << cube( sideValue ) << endl;
23 } // end main
```

Fig. 6.18 | `inline` function that calculates the volume of a cube. (Part 1 of 2.)

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 6.18 | `inline` function that calculates the volume of a cube. (Part 2 of 2.)

6.14 References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed by value, a *copy* of the argument's value is made and passed (on the function call stack) to the called function. Changes to the copy do *not* affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument in this chapter has been passed by value.

Performance Tip 6.4

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

Reference Parameters

This section introduces **reference parameters**—the first of the two means C++ provides for performing pass-by-reference. With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data.

Performance Tip 6.5

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Software Engineering Observation 6.11

Pass-by-reference can weaken security; the called function can corrupt the caller's data.

Later, we'll show how to achieve the performance advantage of pass-by-reference while *simultaneously* achieving the software engineering advantage of protecting the caller's data from corruption.

A reference parameter is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header. For example, the following declaration in a function header

```
int &count
```

when read from *right to left* is pronounced "`count` is a reference to an `int`." In the function call, simply mention the variable by name to pass it by reference. Then, mentioning the variable by its parameter name in the body of the called function actually refers to the original variable in the calling function, and the original variable can be modified directly by the called function. As always, the function prototype and header must agree.

Passing Arguments by Value and by Reference

Figure 6.19 compares pass-by-value and pass-by-reference with reference parameters. The "styles" of the arguments in the calls to function `squareByValue` and function `squareByReference` are identical—both variables are simply mentioned by name in the function calls. *Without checking the function prototypes or function definitions, it isn't possible to tell from the calls alone whether either function can modify its arguments.* Because function prototypes are mandatory, the compiler has no trouble resolving the ambiguity.

Common Programming Error 6.9

Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original variables are changed by the function.

```cpp
// Fig. 6.19: fig06_19.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using namespace std;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
} // end main

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

Fig. 6.19 | Passing arguments by value and by reference. (Part 1 of 2.)

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 6.19 | Passing arguments by value and by reference. (Part 2 of 2.)

Chapter 8 discusses pointers; pointers enable an alternate form of pass-by-reference in which the style of the call clearly indicates pass-by-reference (and the potential for modifying the caller's arguments).

Performance Tip 6.6

For passing large objects, use a constant *reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.*

To specify a reference to a constant, place the `const` qualifier *before* the type specifier in the parameter declaration. Note the placement of `&` in function `squareByReference`'s parameter list (line 35, Fig. 6.19). Some C++ programmers prefer to write the equivalent form `int& numberRef`.

References as Aliases within a Function

References can also be used as aliases for other variables *within* a function (although they typically are used with functions as shown in Fig. 6.19). For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
++cRef; // increment count (using its alias cRef)
```

increments variable `count` by using its alias `cRef`. Reference variables *must* be initialized in their declarations (see Fig. 6.20 and Fig. 6.21) and cannot be reassigned as aliases to other variables. Once a reference is declared as an alias for another variable, all operations supposedly performed on the *alias* (i.e., the reference) are actually performed on the *original* variable. The alias is simply another name for the original variable. Unless it's a reference to a constant, a reference argument must be an *lvalue* (e.g., a variable name), not a constant or expression that returns an *rvalue* (e.g., the result of a calculation). See Section 5.9 for definitions of the terms *lvalue* and *rvalue*.

```
// Fig. 6.20: fig06_20.cpp
// Initializing and using a reference.
#include <iostream>
using namespace std;

int main()
{
   int x = 3;
   int &y = x; // y refers to (is an alias for) x
```

Fig. 6.20 | Initializing and using a reference. (Part 1 of 2.)

```

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7; // actually modifies x
   cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 6.20 | Initializing and using a reference. (Part 2 of 2.)

```
// Fig. 6.21: fig06_21.cpp
// References must be initialized.
#include <iostream>
using namespace std;

int main()
{
   int x = 3;
   int &y; // Error: y must be initialized

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7;
   cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch06\Fig06_21\fig06_21.cpp(9) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp:9: error: 'y' declared as a reference but not initialized
```

Fig. 6.21 | Uninitialized reference causes a syntax error.

Returning a Reference from a Function

Functions can return references, but this can be dangerous. When returning a reference to a variable declared in the called function, unless that variable is declared `static`, the reference refers to an automatic variable that's discarded when the function terminates. Such a variable is said to be "undefined," and the program's behavior is unpredictable. References to undefined variables are called **dangling references**.

Common Programming Error 6.10

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.

Error Messages for Uninitialized References

The C++ standard does *not* specify the error messages that compilers use to indicate particular errors. For this reason, Fig. 6.21 shows the error messages produced by the Microsoft Visual C++ 2008 compiler and GNU C++ compiler when a reference is *not* initialized.

6.15 Default Arguments

It's common for a program to invoke a function repeatedly with the *same* argument value for a particular parameter. In such cases, you can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter. When a program *omits* an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument.

Default arguments *must* be the rightmost (trailing) arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, then all arguments to the right of that argument also *must* be omitted. Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype. If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header. Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used with `inline` functions.

Figure 6.22 demonstrates using default arguments to calculate a box's volume. The function prototype for `boxVolume` (line 7) specifies that all three parameters have been given default values of `1`. We provided variable names in the function prototype for readability. As always, variable names are *not* required in function prototypes.

```
1  // Fig. 6.22: fig06_22.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  int boxVolume( int length = 1, int width = 1, int height = 1 );
8
9  int main()
10 {
11    // no arguments--use default values for all dimensions
12    cout << "The default box volume is: " << boxVolume();
13
14    // specify length; default width and height
15    cout << "\n\nThe volume of a box with length 10,\n"
16       << "width 1 and height 1 is: " << boxVolume( 10 );
17
18    // specify length and width; default height
19    cout << "\n\nThe volume of a box with length 10,\n"
20       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
21
```

Fig. 6.22 | Default arguments to a function. (Part 1 of 2.)

```
22       // specify all arguments
23       cout << "\n\nThe volume of a box with length 10,\n"
24          << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
25          << endl;
26    } // end main
27
28    // function boxVolume calculates the volume of a box
29    int boxVolume( int length, int width, int height )
30    {
31       return length * width * height;
32    } // end function boxVolume
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 6.22 | Default arguments to a function. (Part 2 of 2.)

The first call to `boxVolume` (line 12) specifies no arguments, thus using all three default values of 1. The second call (line 16) passes only a `length` argument, thus using default values of 1 for the `width` and `height` arguments. The third call (line 20) passes arguments for only `length` and `width`, thus using a default value of 1 for the `height` argument. The last call (line 24) passes arguments for `length`, `width` and `height`, thus using no default values. Any arguments passed to the function explicitly are assigned to the function's parameters from left to right. Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list). When `boxVolume` receives two arguments, the function assigns the values of those arguments to its `length` and `width` parameters in that order. Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to its `length`, `width` and `height` parameters, respectively.

Good Programming Practice 6.3

Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

Software Engineering Observation 6.12

If the default values for a function change, all client code using the function must be recompiled.

6.16 Unary Scope Resolution Operator

It's possible to declare local and global variables of the same name. C++ provides the **unary scope resolution operator (`::`)** to access a global variable when a local variable of the same name is in scope. The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Figure 6.23 shows the unary scope resolution operator with local and global variables of the same name (lines 6 and 10). To emphasize that the local and global versions of variable `number` are distinct, the program declares one variable `int` and the other `double`.

```
1  // Fig. 6.23: fig06_23.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  int number = 7; // global variable named number
7
8  int main()
9  {
10    double number = 10.5; // local variable named number
11
12    // display values of local and global variables
13    cout << "Local double value of number = " << number
14       << "\nGlobal int value of number = " << ::number << endl;
15 } // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```

Fig. 6.23 | Unary scope resolution operator.

Using the unary scope resolution operator (`::`) with a given variable name is optional when the only variable with that name is a global variable.

Good Programming Practice 6.4

Always using the unary scope resolution operator (`::`) to refer to global variables makes programs easier to read and understand, because it makes it clear that you're intending to access a global variable rather than a nonglobal variable.

Software Engineering Observation 6.13

Always using the unary scope resolution operator (`::`) to refer to global variables makes programs easier to modify by reducing the risk of name collisions with nonglobal variables.

Error-Prevention Tip 6.4

Always using the unary scope resolution operator (`::`) to refer to a global variable eliminates logic errors that might occur if a nonglobal variable hides the global variable.

Error-Prevention Tip 6.5

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.

6.17 Function Overloading

C++ enables several functions of the same name to be defined, as long as they have different signatures. This is called **function overloading**. The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call. Function overloading is used to create several functions of the *same* name that perform similar tasks, but on *different* data types. For example, many functions in the math library are overloaded for different numeric types—the C++ standard requires `float`, `double` and `long double` overloaded versions of the math library functions discussed in Section 6.3.

> **Good Programming Practice 6.5**
> *Overloading functions that perform closely related tasks can make programs more readable and understandable.*

Overloaded `square` *Functions*

Figure 6.24 uses overloaded `square` functions to calculate the square of an `int` (lines 7–11) and the square of a `double` (lines 14–18). Line 22 invokes the `int` version of function `square` by passing the literal value `7`. C++ treats whole number literal values as type `int`. Similarly, line 24 invokes the `double` version of function `square` by passing the literal value `7.5`, which C++ treats as a `double` value. In each case the compiler chooses the proper function to call, based on the type of the argument. The last two lines of the output window confirm that the proper function was called in each case.

```
1  // Fig. 6.24: fig06_24.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square( int x )
8  {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 } // end function square with int argument
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 } // end function square with double argument
19
20 int main()
21 {
22    cout << square( 7 ); // calls int version
23    cout << endl;
24    cout << square( 7.5 ); // calls double version
25    cout << endl;
26 } // end main
```

Fig. 6.24 | Overloaded `square` functions. (Part 1 of 2.)

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 6.24 | Overloaded `square` functions. (Part 2 of 2.)

How the Compiler Differentiates Overloaded Functions

Overloaded functions are distinguished by their signatures. A signature is a combination of a function's name and its parameter types (in order). The compiler encodes each function identifier with the number and types of its parameters (sometimes referred to as **name mangling** or **name decoration**) to enable **type-safe linkage**. Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.

Figure 6.25 was compiled with GNU C++. Rather than showing the execution output of the program (as we normally would), we show the mangled function names produced in assembly language by GNU C++. Each mangled name (other than `main`) begins with two underscores (__) followed by the letter `Z`, a number and the function name. The number that follows `Z` specifies how many characters are in the function's name. For example, function `square` has 6 characters in its name, so its mangled name is prefixed with `__Z6`. The function name is then followed by an encoding of its parameter list. In the parameter list for function `nothing2` (line 25; see the fourth output line), `c` represents a `char`, `i` represents an `int`, `Rf` represents a `float &` (i.e., a reference to a `float`) and `Rd` represents a `double &` (i.e., a reference to a `double`). In the parameter list for function `nothing1`, `i` represents an `int`, `f` represents a `float`, `c` represents a `char` and `Ri` represents an `int &`. The two `square` functions are distinguished by their parameter lists; one specifies `d` for `double` and the other specifies `i` for `int`. The return types of the functions are *not* specified in the mangled names. *Overloaded functions can have different return types, but if they do, they must also have different parameter lists.* Again, you *cannot* have two functions with the *same* signature and *different* return types. Function-name mangling is compiler specific. Also, function `main` is *not* mangled, because it *cannot* be overloaded.

> **Common Programming Error 6.11**
> *Creating overloaded functions with identical parameter lists and different return types is a compilation error.*

```
1  // Fig. 6.25: fig06_25.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
```

Fig. 6.25 | Name mangling to enable type-safe linkage. (Part 1 of 2.)

```
   return y * y;
} // end function square

// function that receives arguments of types
// int, float, char and int &
void nothing1( int a, float b, char c, int &d )
{
   // empty function body
} // end function nothing1

// function that receives arguments of types
// char, int, float & and double &
int nothing2( char a, int b, float &c, double &d )
{
   return 0;
} // end function nothing2

int main()
{
} // end main
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Fig. 6.25 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

The compiler uses only the parameter lists to distinguish between overloaded functions. Such functions need *not* have the same number of parameters. Use caution when overloading functions with default parameters, because this may cause ambiguity.

Common Programming Error 6.12

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains both *a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler cannot determine which version of the function to choose.*

Overloaded Operators

In Chapter 11, we discuss how to overload operators to define how they should operate on objects of user-defined data types. (In fact, we've been using many overloaded operators to this point, including the stream insertion << and the stream extraction >> operators, which are overloaded for *all* the fundamental types. We say more about overloading << and >> to be able to handle objects of user-defined types in Chapter 11.)

6.18 Function Templates

Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations are *identical* for each data type, overloading may be performed more compactly and conveniently by using **function templates**. You write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate **function template specializations** to handle each type of call appropriately. Thus, defining a single function template essentially defines a whole family of overloaded functions.

Figure 6.26 defines a `maximum` function template (lines 3–17) that determines the largest of three values. All function template definitions begin with the **`template` keyword** (line 3) followed by a **template parameter list** to the function template enclosed in angle brackets (< and >). Every parameter in the template parameter list (often referred to as a **formal type parameter**) is preceded by keyword `typename` or keyword `class` (they are synonyms in this context). The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders, in this case, `T`, are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 6). A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3  template < typename T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Fig. 6.26 | Function template `maximum` header.

The function template declares a single formal type parameter `T` (line 3) as a placeholder for the type of the data to be tested by function `maximum`. The name of a type parameter must be unique in the template parameter list for a particular template definition. When the compiler detects a `maximum` invocation in the program source code, the *type of* the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type—all three must have the same type, since we use only one type parameter in this example. Then the newly created function is compiled. Thus, templates are a means of code generation.

Figure 6.27 uses the `maximum` function template to determine the largest of three `int` values, three `double` values and three `char` values, respectively (lines 17, 27 and 37). Separate functions are created as a result of the calls in lines 17, 27 and 37—expecting three `int` values, three `double` values and three `char` values, respectively. The function template specialization created for type `int` replaces each occurrence of `T` with `int` as follows:

```
int maximum( int value1, int value2, int value3 )
{
   int maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

```
// Fig. 6.27: fig06_27.cpp
// Function template maximum test program.
#include <iostream>
#include "maximum.h" // include definition of function template maximum
using namespace std;

int main()
{
   // demonstrate maximum with int values
   int int1, int2, int3;

   cout << "Input three integer values: ";
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
      << maximum( int1, int2, int3 );

   // demonstrate maximum with double values
   double double1, double2, double3;

   cout << "\n\nInput three double values: ";
   cin >> double1 >> double2 >> double3;

   // invoke double version of maximum
   cout << "The maximum double value is: "
      << maximum( double1, double2, double3 );

   // demonstrate maximum with char values
   char char1, char2, char3;

   cout << "\n\nInput three characters: ";
   cin >> char1 >> char2 >> char3;

   // invoke char version of maximum
   cout << "The maximum character value is: "
      << maximum( char1, char2, char3 ) << endl;
} // end main
```

Fig. 6.27 | Demonstrating function template `maximum`. (Part 1 of 2.)

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 6.27 | Demonstrating function template `maximum`. (Part 2 of 2.)

6.19 Recursion

For some problems, it's useful to have functions *call themselves*. A **recursive function** is a function that calls itself, either directly, or indirectly (through another function). [*Note:* Although many compilers allow function `main` to call itself, Section 3.6.1, paragraph 3, and Section 5.2.2, paragraph 9, of the C++ standard document indicate that `main` should not be called within a program or recursively. Its sole purpose is to be the starting point for program execution.] Recursion is an important topic discussed at length in upper-level computer science courses. This section and the next present simple examples of recursion. Figure 6.33 (at the end of Section 6.21) summarizes the extensive recursion examples and exercises in the book.

We first consider recursion conceptually, then examine two programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function knows how to solve only the *simplest case(s)*, or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece *must* resemble the original problem, but be a slightly simpler or smaller version. This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step often includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.

The recursion step executes while the original call to the function is still "open," i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces. In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues up the line until the original call eventually returns the final result to `main`. This sounds quite exotic compared to the kind of problem solving we've been using to this point. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer n, written $n!$ (and pronounced "n factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

with 1! equal to 1, and 0! defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, `number`, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) by using a `for` statement as follows:

```
factorial = 1;

for ( int counter = number; counter >= 1; --counter )
   factorial *= counter;
```

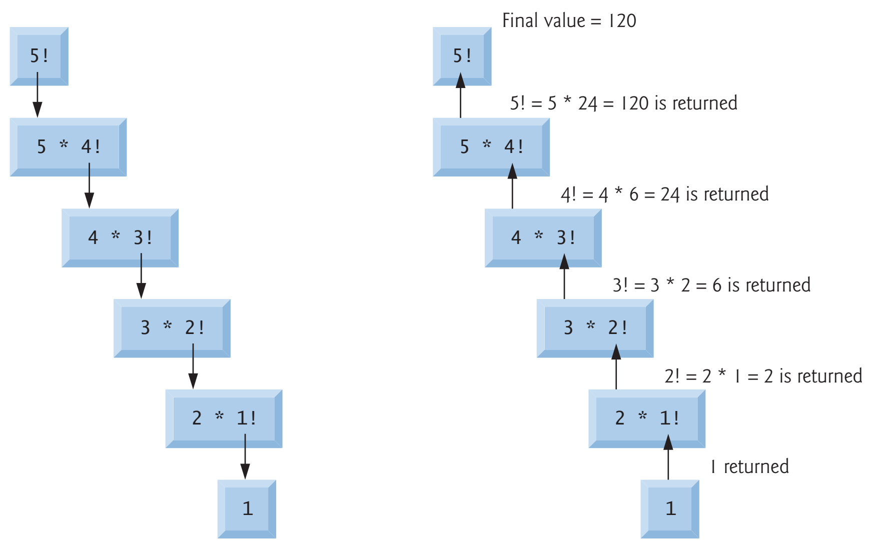
A *recursive* definition of the factorial function is arrived at by observing the following algebraic relationship:

$$n! = n \cdot (n-1)!$$

For example, 5! is clearly equal to 5 * 4! as is shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 6.28, which illustrates how the succession of recursive calls proceeds until 1! is evaluated to be 1, terminating the recursion. Figure 6.28(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.



(a) Procession of recursive calls (b) Values returned from each recursive call

Fig. 6.28 | Recursive evaluation of 5!.

Figure 6.29 uses recursion to calculate and print the factorials of the integers 0–10. (The choice of the data type `unsigned long` is explained momentarily.) The recursive function `factorial` (lines 18–24) first determines whether the terminating condition `number <= 1` (line 20) is true. If `number` is less than or equal to 1, the `factorial` function returns 1 (line 21), no further recursion is necessary and the function terminates. If `number` is greater than 1, line 23 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation `factorial(number)`.

```
1  // Fig. 6.29: fig06_29.cpp
2  // Demonstrating the recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; ++counter )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
17 // recursive definition of function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    if ( number <= 1 ) // test for base case
21       return 1; // base cases: 0! = 1 and 1! = 1
22    else // recursion step
23       return number * factorial( number - 1 );
24 } // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 6.29 | Demonstrating the recursive function `factorial`.

Function `factorial` has been declared to receive a parameter of type `unsigned long` and return a result of type `unsigned long`. This is shorthand notation for `unsigned long int`. The C++ standard requires that a variable of type `unsigned long int` be at least as big as an `int`. Typically, an `unsigned long int` is stored in at least four bytes (32 bits); such a variable can hold a value in the range 0 to at least 4294967295. (The data type `long int` is also stored in at least four bytes and can hold a value at least in the range –2147483648 to 2147483647.) As can be seen in Fig. 6.29, factorial values become large quickly. We chose the data type `unsigned long` so that the program can calculate factorials greater than 7! on computers with small (such as two-byte) integers. Unfortunately, the function `factorial` produces large values so quickly that even `unsigned long` does not help us compute many factorial values before even the size of an `unsigned long` variable is exceeded.

Variables of type `double` could be used to calculate factorials of larger numbers. This points to a weakness in many programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications. As we'll see when we discuss object-oriented programming in more depth, C++ is an *extensible* language that allows us to create classes that can represent arbitrarily large integers if we wish. Such classes already are available in popular class libraries,[1] and we work on similar classes of our own in Exercise 9.14 and Exercise 11.9.

Common Programming Error 6.13

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes "infinite" recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

6.20 Example Using Recursion: Fibonacci Series

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and, in particular, describes a form of spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number, too, frequently occurs in nature and has been called the **golden ratio** or the **golden mean.** Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio.

The Fibonacci series can be defined recursively as follows:

$$\text{fibonacci}(0) = 0$$
$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

The program of Fig. 6.30 calculates the *n*th Fibonacci number recursively by using function `fibonacci`. Fibonacci numbers tend to become large quickly, although slower than factorials do. Therefore, we chose the data type `unsigned long` for the parameter type and the return type in function `fibonacci`. Figure 6.30 shows the execution of the program, which displays the Fibonacci values for several numbers.

1. Such classes can be found at `www.trumphurst.com/cpplibs/datapage.phtml?category='intro'`.

```cpp
1 // Fig. 6.30: fig06_30.cpp
2 // Testing the recursive fibonacci function.
3 #include <iostream>
4 using namespace std;
5
6 unsigned long fibonacci( unsigned long ); // function prototype
7
8 int main()
9 {
10    // calculate the fibonacci values of 0 through 10
11    for ( int counter = 0; counter <= 10; ++counter )
12       cout << "fibonacci( " << counter << " ) = "
13          << fibonacci( counter ) << endl;
14
15    // display higher fibonacci values
16    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // end main
20
21 // recursive function fibonacci
22 unsigned long fibonacci( unsigned long number )
23 {
24    if ( ( number == 0 ) || ( number == 1 ) ) // base cases
25       return number;
26    else // recursion step
27       return fibonacci( number - 1 ) + fibonacci( number - 2 );
28 } // end function fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

Fig. 6.30 | Demonstrating function `fibonacci`.

The application begins with a `for` statement that calculates and displays the Fibonacci values for the integers 0–10 and is followed by three calls to calculate the Fibonacci values of the integers 20, 30 and 35 (lines 16–18). The calls to `fibonacci` (lines 13 and 16–18) from `main` are not recursive calls, but the calls from line 27 of `fibonacci` are recursive. Each time the program invokes `fibonacci` (lines 22–28), the function immediately tests the base case to determine whether `number` is equal to `0` or `1` (line 24). If this is true, line 25 returns `number`. Interestingly, if `number` is greater than 1, the recursion step (line 27) generates *two* recursive calls, each for a slightly smaller problem than the original call to `fibonacci`.

Figure 6.31 shows how function `fibonacci` would evaluate `fibonacci(3)`. This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators. This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity. Figure 6.31 shows that evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`. In what order are these calls made?

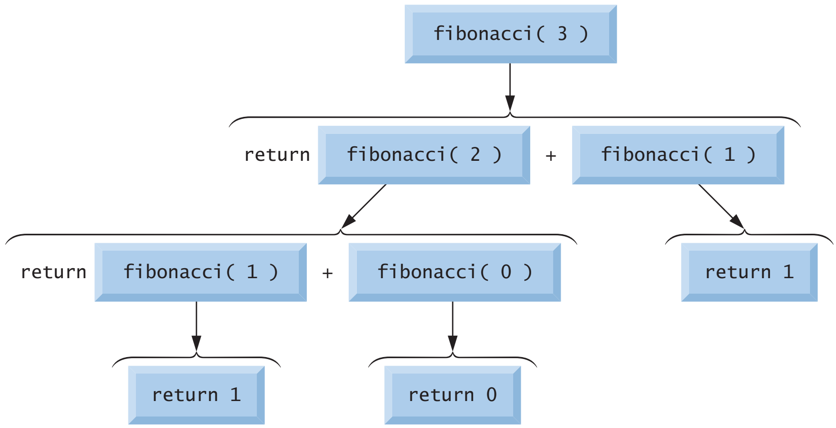


Fig. 6.31 | Set of recursive calls to function `fibonacci`.

Order of Evaluation of Operands

Most programmers simply assume that the operands are evaluated left to right. C++ does *not* specify the order in which the operands of most operators (including +) are to be evaluated. Therefore, you must make no assumption about the order in which these calls execute. The calls could in fact execute `fibonacci(2)` first, then `fibonacci(1)`, or they could execute in the reverse order: `fibonacci(1)`, then `fibonacci(2)`. In this program and in most others, it turns out that the final result would be the same. However, in some programs the evaluation of an operand can have **side effects** (changes to data values) that could affect the final result of the expression.

C++ specifies the order of evaluation of the operands of only *four* operators—`&&`, `||`, comma (`,`) and `?:`. The first three are binary operators whose two operands are guaranteed to be evaluated left to right. The last operator is C++'s only *ternary* operator—its leftmost operand is always evaluated first; if it evaluates to nonzero (true), the middle operand evaluates next and the last operand is ignored; if the leftmost operand evaluates to zero (false), the third operand evaluates next and the middle operand is ignored.

Portability Tip 6.2

Programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:` *and the comma (*`,`*) operator can function differently with different compilers.*

Common Programming Error 6.14

Writing programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:` *and the comma (*`,`*) operator can lead to logic errors.*

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in function `fibonacci` has a *doubling* effect on the number of function calls; i.e., the number of recursive calls that are required to calculate the *n*th Fibonacci number is on the order of 2^n. This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on. Computer scientists refer to this as **exponential complexity**. Problems of this nature humble even the world's most powerful computers! Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science course generally called "Algorithms."

Performance Tip 6.7

Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

6.21 Recursion vs. Iteration

In the two previous sections, we studied two functions that easily can be implemented recursively or iteratively. This section compares the two approaches and discusses why you might choose one approach over the other in a particular situation.

Both iteration and recursion are *based on a control statement*: Iteration uses a *repetition* structure; recursion uses a *selection* structure. Both iteration and recursion involve *repetition*: Iteration explicitly uses a *repetition structure*; recursion achieves repetition through *repeated function calls*. Iteration and recursion both involve a *termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion terminates when a *base case is recognized*. Iteration with counter-controlled repetition and recursion both *gradually approach termination*: Iteration *modifies a counter* until the counter assumes a value that makes the loop-continuation condition fail; recursion produces *simpler versions of the original problem* until the base case is reached. Both iteration and recursion *can occur infinitely*: An *infinite loop* occurs with iteration if the loop-continuation test never becomes false; *infinite recursion* occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

To illustrate the differences between iteration and recursion, let's examine an iterative solution to the factorial problem (Fig. 6.32). A repetition statement is used (lines 23–24 of Fig. 6.32) rather than the selection statement of the recursive solution (lines 20–23 of Fig. 6.29). Both solutions use a termination test. In the recursive solution, line 20 tests for the base case. In the iterative solution, line 23 tests the loop-continuation condition—if the test fails, the loop terminates. Finally, instead of producing simpler versions of the original problem, the iterative solution uses a counter that is modified until the loop-continuation condition becomes false.

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
```

Fig. 6.32 | Iterative factorial solution. (Part 1 of 2.)

```

unsigned long factorial( unsigned long ); // function prototype

int main()
{
   // calculate the factorials of 0 through 10
   for ( int counter = 0; counter <= 10; ++counter )
      cout << setw( 2 ) << counter << "! = " << factorial( counter )
         << endl;
} // end main

// iterative function factorial
unsigned long factorial( unsigned long number )
{
   unsigned long result = 1;

   // iterative factorial calculation
   for ( unsigned long i = number; i >= 1; --i )
      result *= i;

   return result;
} // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 6.32 | Iterative factorial solution. (Part 2 of 2.)

Recursion has negatives. It repeatedly invokes the mechanism, and consequently the *overhead, of function calls*. This can be expensive in both processor time and memory space. Each recursive call causes *another copy of the function* (actually only the function's variables) to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

Software Engineering Observation 6.14

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

Performance Tip 6.8

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 6.15

Accidentally having a nonrecursive function call itself, either directly or indirectly (through another function), is a logic error.

Figure 6.33 summarizes the recursion examples and exercises in the text.

Location in text	Recursion examples and exercises
Chapter 6	
Section 6.19, Fig. 6.29	Factorial function
Section 6.20, Fig. 6.30	Fibonacci function
Exercise 6.36	Raising an integer to an integer power
Exercise 6.38	Towers of Hanoi
Exercise 6.40	Visualizing recursion
Exercise 6.41	Greatest common divisor
Exercise 6.45, Exercise 6.46	Mystery "What does this program do?" exercise
Chapter 7	
Exercise 7.18	Mystery "What does this program do?" exercise
Exercise 7.21	Mystery "What does this program do?" exercise
Exercise 7.31	Selection sort
Exercise 7.32	Determine whether a string is a palindrome
Exercise 7.33	Linear search
Exercise 7.34	Eight Queens
Exercise 7.35	Print an array
Exercise 7.36	Print a string backward
Exercise 7.37	Minimum value in an array
Chapter 8	
Exercise 8.15	Quicksort
Exercise 8.16	Maze traversal
Exercise 8.17	Generating mazes randomly
Chapter 19	
Section 19.3.3, Figs. 19.5–19.7	Mergesort
Exercise 19.8	Linear search
Exercise 19.9	Binary search
Exercise 19.10	Quicksort
Chapter 20	
Section 20.7, Figs. 20.20–20.22	Binary tree insert
Section 20.7, Figs. 20.20–20.22	Preorder traversal of a binary tree
Section 20.7, Figs. 20.20–20.22	Inorder traversal of a binary tree

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 1 of 2.)

Location in text	Recursion examples and exercises
Section 20.7, Figs. 20.20–20.22	Postorder traversal of a binary tree
Exercise 20.20	Print a linked list backward
Exercise 20.21	Search a linked list
Exercise 20.22	Binary tree delete
Exercise 20.23	Binary tree search
Exercise 20.24	Level order traversal of a binary tree
Exercise 20.25	Printing tree

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 2 of 2.)

6.22 Wrap-Up

In this chapter, you learned more about function declarations, including function prototypes, function signatures, function headers and function bodies. We overviewed the math library functions. You learned about argument coercion, or the forcing of arguments to the appropriate types specified by the parameter declarations of a function. We demonstrated how to use functions `rand` and `srand` to generate sets of random numbers that can be used for simulations. We showed how to define sets of constants with `enum`s. You also learned about the scope of variables and storage classes. Two different ways to pass arguments to functions were covered—pass-by-value and pass-by-reference. For pass-by-reference, references are used as an alias to a variable. We showed how to implement inline functions and functions that receive default arguments. You learned that multiple functions in one class can be overloaded by providing functions with the same name and different signatures. Such functions can be used to perform the same or similar tasks, using different types or different numbers of parameters. We demonstrated a simpler way of overloading functions using function templates, where a function is defined once but can be used for several different types. You then studied recursion, where a function calls itself to solve a problem.

In Chapter 7, you'll learn how to maintain lists and tables of data in arrays and object-oriented `vector`s. You'll see a more elegant array-based implementation of the dice-rolling application and two enhanced versions of the `GradeBook` case study we presented in Chapters 3–6 that will use arrays to store the actual grades entered.

Summary

Section 6.1 Introduction

- Experience has shown that the best way to develop and maintain a large program is to construct it from small, simple pieces, or components. This technique is called divide and conquer (p. 195).

Section 6.2 Program Components in C++

- C++ programs are typically written by combining new functions and classes you write with "prepackaged" functions and classes available in the C++ Standard Library.
- Functions allow you to modularize a program by separating its tasks into self-contained units.

- The statements in the function bodies are written only once, are reused from perhaps several locations in a program and are hidden from other functions.

Section 6.3 Math Library Functions

- Sometimes functions are not members of a class. These are called global functions (p. 197).
- The prototypes for global functions are placed in headers, so that they can be reused in any program that includes the header and that can link to the function's object code.

Section 6.4 Function Definitions with Multiple Parameters

- The compiler refers to the function prototype to check that calls to a function contain the correct number and types of arguments, that the types of the arguments are in the correct order and that the value returned by the function can be used correctly in the expression that called the function.
- If a function does not return a result, control returns when the program reaches the function-ending right brace, or by execution of the statement

  ```
  return;
  ```

  If a function does return a result, the statement

  `return` *expression*;

  evaluates *expression* and returns the value of *expression* to the caller.

Section 6.5 Function Prototypes and Argument Coercion

- The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature (p. 203) or simply the signature.
- An important feature of function prototypes is argument coercion (p. 203)—i.e., forcing arguments to the appropriate types specified by the parameter declarations.
- Arguments can be promoted by the compiler to the parameter types as specified by C++'s promotion rules (p. 204). The promotion rules indicate the implicit conversions that the compiler can perform between fundamental types.

Section 6.6 C++ Standard Library Headers

- The C++ Standard Library is divided into many portions, each with its own header. The headers also contain definitions of various class types, functions and constants.
- A header "instructs" the compiler on how to interface with library components.

Section 6.7 Case Study: Random Number Generation

- Calling `rand` (p. 207) repeatedly produces a sequence of pseudorandom numbers (p. 210). The sequence repeats itself each time the program executes.
- To randomize the numbers produced by `rand` pass an `unsigned` integer argument (typically from function `time`; p. 211) to function `srand` (p. 210), which seeds the `rand` function.
- Random numbers in a range can be generated with

  number `=` *shiftingValue* `+ rand() %` *scalingFactor*;

  where *shiftingValue* (p. 207) is equal to the first number in the desired range of consecutive integers and *scalingFactor* (p. 207) is equal to the width of the desired range of consecutive integers.

Section 6.8 Case Study: Game of Chance; Introducing `enum`

- An enumeration, introduced by the keyword `enum` and followed by a type name (p. 214), is a set of named integer constants (p. 214) that start at `0`, unless specified otherwise, and increment by `1`.

Section 6.9 Storage Classes

- An identifier's storage class (p. 215) determines the period during which it exists in memory.
- An identifier's scope is where the identifier can be referenced in a program.
- An identifier's linkage (p. 215) determines whether it's known only in the source file where it's declared or across multiple files that are compiled, then linked together.
- Keywords `auto` (p. 215) and `register` (p. 215) declare variables of the automatic storage class (p. 216). Such variables are created when program execution enters the block in which they're defined, exist while the block is active and are destroyed when the program exits the block.
- Only local variables of a function can be of automatic storage class.
- The storage-class specifier `auto` (p. 215) explicitly declares variables of automatic storage class. Local variables are of automatic storage class by default, so keyword `auto` is rarely used.
- Keywords `extern` (p. 215) and `static` declare identifiers for variables of the static storage class (p. 215) and for functions. Static-storage-class variables exist from the point at which the program begins execution and last for the duration of the program.
- A static-storage-class variable's storage is allocated when the program begins execution. Such a variable is initialized once when its declaration is encountered. For functions, the name of the function exists when the program begins execution as for all other functions.
- External identifiers (such as global variables) and local variables declared with the storage class-specifier `static` have static storage class (p. 215).
- Global variables (p. 217) declarations are placed outside any class or function definition. Global variables retain their values throughout the program's execution. Global variables and functions can be referenced by any function that follows their declarations or definitions.

Section 6.10 Scope Rules

- Unlike automatic variables, `static` local variables retain their values when the function in which they're declared returns to its caller.
- An identifier declared outside any function or class has global namespace scope (p. 218).
- Labels are the only identifiers with function scope (p. 218). Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.
- Identifiers declared inside a block have local scope (p. 218), which begins at the identifier's declaration and ends at the terminating right brace (`}`) of the block in which the identifier is declared.
- Identifiers in the parameter list of a function prototype have function-prototype scope (p. 218).

Section 6.11 Function Call Stack and Activation Records

- Stacks (p. 221) are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted; p. 221) on the stack is the first item popped (removed; p. 221) from the stack.
- The function call stack (p. 221) supports the function call/return mechanism and the creation, maintenance and destruction of each called function's automatic variables.
- Each time a function calls another function, a stack frame or an activation record (p. 221) is pushed onto the stack containing the return address that the called function needs to return to the calling function, and the function call's automatic variables and parameters.
- The stack frame (p. 221) exists as long as the called function is active. When the called function returns, its stack frame is popped from the stack, and its local automatic variables no longer exist.

Section 6.12 Functions with Empty Parameter Lists

- In C++, an empty parameter list is specified by writing either `void` or nothing in parentheses.

Section 6.13 Inline Functions

- C++ provides inline functions (p. 225) to help reduce function call overhead—especially for small functions. Placing the qualifier `inline` before a function's return type in the function definition "advises" the compiler to generate a copy of the function's code in place to avoid a function call.

Section 6.14 References and Reference Parameters

- When an argument is passed by value (p. 227), a *copy* of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.
- With pass-by-reference (p. 227), the caller gives the called function the ability to access the caller's data directly and to modify it if the called function chooses to do so.
- A reference parameter (p. 227) is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, follow the parameter's type in the function prototype and header by an ampersand (`&`).
- All operations performed on a reference are actually performed on the original variable.

Section 6.15 Default Arguments

- When a function is called repeatedly with the same argument for a particular parameter, you can specify that such a parameter has a default argument (p. 231).
- When a program omits an argument for a parameter with a default argument, the compiler inserts the default value of that argument to be passed to the function call.
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list.
- Default arguments are specified in the function prototype.

Section 6.16 Unary Scope Resolution Operator

- C++ provides the unary scope resolution operator (p. 232), `::`, to access a global variable when a local variable of the same name is in scope.

Section 6.17 Function Overloading

- C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading (p. 234).
- When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.
- Overloaded functions are distinguished by their signatures.
- The compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage (p. 235). Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.

Section 6.18 Function Templates

- Overloaded functions typically perform similar operations that involve different program logic on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently using function templates (p. 237).
- Given the argument types provided in calls to a function template, C++ automatically generates separate function template specializations (p. 237) to handle each type of call appropriately.
- All function template definitions begin with the `template` keyword (p. 237) followed by a template parameter list (p. 237) to the function template enclosed in angle brackets (`<` and `>`).
- The formal type parameters (p. 237) are preceded by keyword `typename` and are placeholders for fundamental types or user-defined types. These placeholders are used to specify the types of the

function's parameters, to specify the function's return type and to declare variables within the body of the function definition.

Section 6.19 Recursion

- A recursive function (p. 239) calls itself, either directly or indirectly.
- A recursive function knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case (p. 239), the function simply returns a result.
- If the function is called with a more complex problem, the function typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of it.
- For recursion to terminate, the sequence of recursive calls (p. 239) must converge on the base case.

Section 6.20 Example Using Recursion: Fibonacci Series

- The ratio of successive Fibonacci numbers converges on a constant value of 1.618…. This number frequently occurs in nature and has been called the golden ratio or the golden mean (p. 242).

Section 6.21 Recursion vs. Iteration

- Iteration (p. 240) and recursion are similar: both are based on a control statement, involve repetition, involve a termination test, gradually approach termination and can occur infinitely.
- Recursion repeatedly invokes the mechanism, and overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call (p. 239) causes another copy of the function's variables to be created; this can consume considerable memory.

Self-Review Exercises

6.1 Answer each of the following:

a) Program components in C++ are called ________ and________.
b) A function is invoked with a(n) ________.
c) A variable known only within the function in which it's defined is called a(n) ________.
d) The ________ statement in a called function passes the value of an expression back to the calling function.
e) The keyword ________ is used in a function header to indicate that a function does not return a value or to indicate that a function contains no parameters.
f) An identifier's ________ is the portion of the program in which the identifier can be used.
g) The three ways to return control from a called function to a caller are ________, ________ and ________.
h) A(n) ________ allows the compiler to check the number, types and order of the arguments passed to a function.
i) Function ________ is used to produce random numbers.
j) Function ________ is used to set the random number seed to randomize the number sequence generated by function `rand`.
k) The storage-class specifiers are `mutable`, ________, ________, ________ and ________.
l) Variables declared in a block or in the parameter list of a function are assumed to be of storage class ________ unless specified otherwise.
m) Storage-class specifier ________ is a recommendation to the compiler to store a variable in one of the computer's registers.
n) A variable declared outside any block or function is a(n) ________ variable.
o) For a local variable in a function to retain its value between calls to the function, it must be declared with the ________ storage-class specifier.

p) The six possible scopes of an identifier are ________, ________, ________, ________, ________ and ________.
q) A function that calls itself either directly or indirectly (i.e., through another function) is a(n) ________ function.
r) A recursive function typically has two components—one that provides a means for the recursion to terminate by testing for a(n) ________ case and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.
s) It's possible to have various functions with the same name that operate on different types or numbers of arguments. This is called function ________.
t) The ________ enables access to a global variable with the same name as a variable in the current scope.
u) The ________ qualifier is used to declare read-only variables.
v) A function ________ enables a single function to be defined to perform a task on many different data types.

6.2 For the program in Fig. 6.34, state the scope (either function scope, global namespace scope, local scope or function-prototype scope) of each of the following elements:
a) The variable `x` in `main`.
b) The variable `y` in `cube`.
c) The function `cube`.
d) The function `main`.
e) The function prototype for `cube`.
f) The identifier `y` in the function prototype for `cube`.

```
// Exercise 6.2: Ex06_02.cpp
#include <iostream>
using namespace std;

int cube( int y ); // function prototype

int main()
{
   int x;

   for ( x = 1; x <= 10; x++ ) // loop 10 times
      cout << cube( x ) << endl; // calculate cube of x and output results
} // end main

// definition of function cube
int cube( int y )
{
   return y * y * y;
} // end function cube
```

Fig. 6.34 | Program for Exercise 6.2.

6.3 Write a program that tests whether the examples of the math library function calls shown in Fig. 6.2 actually produce the indicated results.

6.4 Give the function header for each of the following functions:
a) Function `hypotenuse` that takes two double-precision, floating-point arguments, `side1` and `side2`, and returns a double-precision, floating-point result.
b) Function `smallest` that takes three integers, `x`, `y` and `z`, and returns an integer.
c) Function `instructions` that does not receive any arguments and does not return a value. [*Note:* Such functions are commonly used to display instructions to a user.]

d) Function `intToDouble` that takes an integer argument, `number`, and returns a double-precision, floating-point result.

6.5 Give the function prototype (without parameter names) for each of the following:

a) The function described in Exercise 6.4(a).
b) The function described in Exercise 6.4(b).
c) The function described in Exercise 6.4(c).
d) The function described in Exercise 6.4(d).

6.6 Write a declaration for each of the following:

a) Integer `count` that should be maintained in a register. Initialize `count` to `0`.
b) Double-precision, floating-point variable `lastVal` that is to retain its value between calls to the function in which it's defined.

6.7 Find the error(s) in each of the following program segments, and explain how the error(s) can be corrected (see also Exercise 6.48):

a)
```
int g()
{
   cout << "Inside function g" << endl;
   int h()
   {
      cout << "Inside function h" << endl;
   }
}
```

b)
```
int sum( int x, int y )
{
   int result;

   result = x + y;
}
```

c)
```
int sum( int n )
{
   if ( n == 0 )
      return 0;
   else
      n + sum( n - 1 );
}
```

d)
```
void f( double a );
{
   float a;
   cout << a << endl;
}
```

e)
```
void product()
{
   int a;
   int b;
   int c;
   int result;
   cout << "Enter three integers: ";
   cin >> a >> b >> c;
   result = a * b * c;
   cout << "Result is " << result;
   return result;
}
```

6.8 Why would a function prototype contain a parameter type declaration such as `double &`?

6.9 (True/False) All arguments to function calls in C++ are passed by value.

6.10 Write a complete program that prompts the user for the radius of a sphere, and calculates and prints the volume of that sphere. Use an `inline` function `sphereVolume` that returns the result of the following expression: `(4.0 / 3.0 * 3.14159 * pow(radius, 3))`.

Answers to Self-Review Exercises

6.1 a) functions, classes. b) function call. c) local variable. d) `return`. e) `void`. f) scope. g) `return;`, `return` *expression*; or encounter the closing right brace of a function. h) function prototype. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) `register`. n) global. o) `static`. p) function scope, global namespace scope, local scope, function-prototype scope, class scope, namespace scope. q) recursive. r) base. s) overloading. t) unary scope resolution operator (`::`). u) `const`. v) template.

6.2 a) local scope. b) local scope. c) global namespace scope. d) global namespace scope. e) global namespace scope. f) function-prototype scope.

6.3 See the following program:

```cpp
// Exercise 6.3: Ex06_03.cpp
// Testing the math library functions.
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
   cout << fixed << setprecision( 1 );

   cout << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
      << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 );
   cout << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
      << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
      << ") = " << setprecision( 6 ) << exp( 2.0 );
   cout << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
      << log( 2.718282 )
      << "\nlog(" << setprecision( 6 ) << 7.389056 << ") = "
      << setprecision( 1 ) << log( 7.389056 );
   cout << "\nlog10(" << 1.0 << ") = " << log10( 1.0 )
      << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
      << "\nlog10(" << 100.0 << ") = " << log10( 100.0 ) ;
   cout << "\nfabs(" << 5.1 << ") = " << fabs( 5.1 )
      << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
      << "\nfabs(" << -8.76 << ") = " << fabs( -8.76 );
   cout << "\nceil(" << 9.2 << ") = " << ceil( 9.2 )
      << "\nceil(" << -9.8 << ") = " << ceil( -9.8 );
   cout << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
      << "\nfloor(" << -9.8 << ") = " << floor( -9.8 );
   cout << "\npow(" << 2.0 << ", " << 7.0 << ") = "
      << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
      << 0.5 << ") = " << pow( 9.0, 0.5 );
   cout << setprecision(3) << "\nfmod("
      << 2.6 << ", " << 1.2 << ") = "
      << fmod( 2.6, 1.2 ) << setprecision( 1 );
   cout << "\nsin(" << 0.0 << ") = " << sin( 0.0 );
```

```
    cout << "\ncos(" << 0.0 << ") = " << cos( 0.0 );
    cout << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
} // end main
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(5.1) = 5.1
fabs(0.0) = 0.0
fabs(-8.8) = 8.8
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(2.600, 1.200) = 0.200
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

6.4 a) `double hypotenuse( double side1, double side2 )`
b) `int smallest( int x, int y, int z )`
c) `void instructions()`
d) `double intToDouble( int number )`

6.5 a) `double hypotenuse( double, double );`
b) `int smallest( int, int, int );`
c) `void instructions();`
d) `double intToDouble( int );`

6.6 a) `register int count = 0;`
b) `static double lastVal;`

6.7 a) *Error:* Function `h` is defined in function `g`.
Correction: Move the definition of `h` out of the definition of `g`.
b) *Error:* The function is supposed to return an integer, but does not.
Correction: Delete variable `result` and place the following statement in the function:

```
return x + y;
```

c) *Error:* The result of `n + sum( n - 1 )` is not returned; `sum` returns an improper result.
Correction: Rewrite the statement in the `else` clause as

```
return n + sum( n - 1 );
```

d) *Errors:* Semicolon after the right parenthesis that encloses the parameter list, and redefining the parameter `a` in the function definition.
Corrections: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration `float a;`.
e) *Error:* The function returns a value when it isn't supposed to.
Correction: Eliminate the `return` statement or change the return type.

6.8 This creates a reference parameter of type "reference to `double`" that enables the function to modify the original variable in the calling function.

6.9 False. C++ enables pass-by-reference using reference parameters (and pointers, as we discuss in Chapter 8).

6.10 See the following program:

```cpp
// Exercise 6.10 Solution: Ex06_10.cpp
// Inline function that calculates the volume of a sphere.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159; // define global constant PI

// calculates volume of a sphere
inline double sphereVolume( const double radius )
{
   return 4.0 / 3.0 * PI * pow( radius, 3 );
} // end inline function sphereVolume

int main()
{
   double radiusValue;

   // prompt user for radius
   cout << "Enter the length of the radius of your sphere: ";
   cin >> radiusValue; // input radius

   // use radiusValue to calculate volume of sphere and display result
   cout << "Volume of sphere with radius " << radiusValue
      << " is " << sphereVolume( radiusValue ) << endl;
} // end main
```

Exercises

6.11 Show the value of `x` after each of the following statements is performed:

a) `x = fabs( 7.5 )`
b) `x = floor( 7.5 )`
c) `x = fabs( 0.0 )`
d) `x = ceil( 0.0 )`
e) `x = fabs( -6.4 )`
f) `x = ceil( -6.4 )`
g) `x = ceil( -fabs( -8 + floor( -5.5 ) ) )`

6.12 *(Parking Charges)* A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write a program that calculates and prints the parking charges for each of three customers who parked their cars in this garage yesterday. You should enter the hours parked for each customer. Your program should print the results in a neat tabular format and should calculate and print the total of yesterday's receipts. The program should use the function `calculateCharges` to determine the charge for each customer. Your outputs should appear in the following format:

```
Car        Hours        Charge
1            1.5          2.00
2            4.0          2.50
3           24.0         10.00
TOTAL       29.5         14.50
```

6.13 *(Rounding Numbers)* An application of function `floor` is rounding a value to the nearest integer. The statement

```
y = floor( x + .5 );
```

rounds the number `x` to the nearest integer and assigns the result to `y`. Write a program that reads several numbers and uses the preceding statement to round each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

6.14 *(Rounding Numbers)* Function `floor` can be used to round a number to a specific decimal place. The statement

```
y = floor( x * 10 + .5 ) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = floor( x * 100 + .5 ) / 100;
```

rounds `x` to the hundredths position (the second position to the right of the decimal point). Write a program that defines four functions to round a number `x` in various ways:

a) `roundToInteger( number )`
b) `roundToTenths( number )`
c) `roundToHundredths( number )`
d) `roundToThousandths( number )`

For each value read, your program should print the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

6.15 *(Short Answer Questions)* Answer each of the following questions:

a) What does it mean to choose numbers "at random?"
b) Why is the `rand` function useful for simulating games of chance?
c) Why would you randomize a program by using `srand`? Under what circumstances is it desirable not to randomize?
d) Why is it often necessary to scale or shift the values produced by `rand`?
e) Why is computerized simulation of real-world situations a useful technique?

6.16 *(Random Numbers)* Write statements that assign random integers to the variable n in the following ranges:

a) $1 \le n \le 2$
b) $1 \le n \le 100$
c) $0 \le n \le 9$
d) $1000 \le n \le 1112$
e) $-1 \le n \le 1$
f) $-3 \le n \le 11$

6.17 *(Random Numbers)* Write a single statement that prints a number at random from each of the following sets:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.18 *(Exponentiation)* Write a function `integerPower(`*base*`,` *exponent*`)` that returns the value of

$$base^{exponent}$$

For example, `integerPower(3, 4) = 3 * 3 * 3 * 3`. Assume that *exponent* is a positive, nonzero integer and that *base* is an integer. Do not use any math library functions.

6.19 *(Hypotenuse Calculations)* Define a function `hypotenuse` that calculates the hypotenuse of a right triangle when the other two sides are given. The function should take two `double` arguments and return the hypotenuse as a `double`. Use this function in a program to determine the hypotenuse for each of the triangles shown below.

Triangle	Side 1	Side 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

6.20 *(Multiples)* Write a function `multiple` that determines for a pair of integers whether the second is a multiple of the first. The function should take two integer arguments and return `true` if the second is a multiple of the first, `false` otherwise. Use this function in a program that inputs a series of pairs of integers.

6.21 *(Even Numbers)* Write a program that inputs a series of integers and passes them one at a time to function `isEven`, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise.

6.22 *(Square of Asterisks)* Write a function that displays at the left margin of the screen a solid square of asterisks whose side is specified in integer parameter `side`. For example, if `side` is `4`, the function displays the following:

```
****
****
****
****
```

6.23 *(Square of Any Character)* Modify the function created in Exercise 6.22 to form the square out of whatever character is contained in character parameter `fillCharacter`. Thus, if `side` is `5` and `fillCharacter` is `#`, then this function should print the following:

```
#####
#####
#####
#####
#####
```

6.24 *(Separating Digits)* Write program segments that accomplish each of the following:

a) Calculate the integer part of the quotient when integer `a` is divided by integer `b`.
b) Calculate the integer remainder when integer `a` is divided by integer `b`.
c) Use the program pieces developed in (a) and (b) to write a function that inputs an integer between `1` and `32767` and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer `4562` should print as follows:

```
4  5  6  2
```

6.25 *(Calculating Number of Seconds)* Write a function that takes the time as three integer arguments (hours, minutes and seconds) and returns the number of seconds since the last time the clock "struck 12." Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

6.26 *(Celsius and Fahrenheit Temperatures)* Implement the following integer functions:

a) Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature.
b) Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature.
c) Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

6.27 *(Find the Minimum)* Write a program that inputs three double-precision, floating-point numbers and passes them to a function that returns the smallest number.

6.28 *(Perfect Numbers)* An integer is said to be a *perfect number* if the sum of its divisors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a function `isPerfect` that determines whether parameter `number` is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the divisors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

6.29 *(Prime Numbers)* An integer is said to be *prime* if it's divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

a) Write a function that determines whether a number is prime.
b) Use this function in a program that determines and prints all the prime numbers between 2 and 10,000. How many of these numbers do you really have to test before being sure that you've found all the primes?
c) Initially, you might think that $n/2$ is the upper limit for which you must test to see whether a number is prime, but you need only go as high as the square root of n. Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

6.30 *(Reverse Digits)* Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

6.31 *(Greatest Common Divisor)* The *greatest common divisor (GCD)* of two integers is the largest integer that evenly divides each of the numbers. Write a function `gcd` that returns the greatest common divisor of two integers.

6.32 *(Quality Points for Numeric Grades)* Write a function `qualityPoints` that inputs a student's average and returns 4 if a student's average is 90–100, 3 if the average is 80–89, 2 if the average is 70–79, 1 if the average is 60–69 and 0 if the average is lower than 60.

6.33 *(Coin Tossing)* Write a program that simulates coin tossing. For each toss of the coin, the program should print `Heads` or `Tails`. Let the program toss the coin 100 times and count the number of times each side of the coin appears. Print the results. The program should call a separate function `flip` that takes no arguments and returns `0` for tails and `1` for heads. [*Note:* If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time.]

6.34 *(Guess-the-Number Game)* Write a program that plays the game of "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then displays the following:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

The player then types a first guess. The program responds with one of the following:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

If the player's guess is incorrect, your program should loop until the player finally gets the number right. Your program should keep telling the player `Too high` or `Too low` to help the player "zero in" on the correct answer.

6.35 *(Guess-the-Number Game Modification)* Modify the program of Exercise 6.34 to count the number of guesses the player makes. If the number is 10 or fewer, print `"Either you know the secret or you got lucky!"` If the player guesses the number in 10 tries, then print `"Ahah! You know the secret!"` If the player makes more than 10 guesses, then print `"You should be able to do better!"` Why should it take no more than 10 guesses? Well, with each "good guess" the player should be able to eliminate half of the numbers. Now show why any number from 1 to 1000 can be guessed in 10 or fewer tries.

6.36 *(Recursive Exponentiation)* Write a recursive function `power( base, exponent )` that, when invoked, returns

$$base^{\text{exponent}}$$

For example, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that `exponent` is an integer greater than or equal to 1. *Hint:* The recursion step would use the relationship

$$base^{\text{exponent}} = base \cdot base^{\text{exponent} - 1}$$

and the terminating condition occurs when `exponent` is equal to `1`, because

$$base^1 = base$$

6.37 *(Fibonacci Series)* The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms. (a) Write a *nonrecursive* function `fibonacci(n)` that uses type `int` to calculate the nth Fibonacci number. (b) Determine the largest `int` Fibonacci number that can be printed on your system. Modify the program of part (a) to use `double` instead of `int` to calculate and return Fibonacci numbers, and use this modified program to repeat part (b).

6.38 *(Towers of Hanoi)* In this chapter, you studied functions that can be easily implemented both recursively and iteratively. In this exercise, we present a problem whose recursive solution demonstrates the elegance of recursion, and whose iterative solution may not be as apparent.

The **Towers of Hanoi** is one of the most famous classic problems every budding computer scientist must grapple with. Legend has it that in a temple in the Far East, priests are attempting to move a stack of golden disks from one diamond peg to another (Fig. 6.35). The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from one peg to another under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. Three pegs are provided, one being used for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that prints the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, attacking this problem with recursion in mind allows the steps to be simple. Moving n disks can be viewed in terms of moving only n – 1 disks (hence, the recursion), as follows:

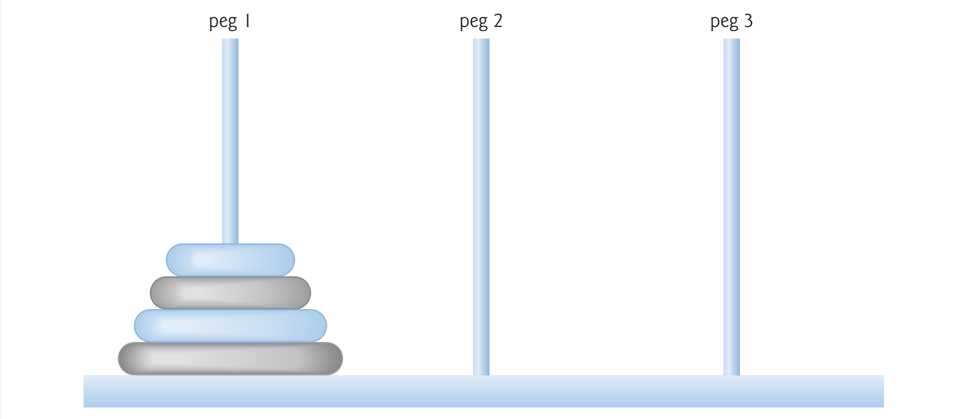


Fig. 6.35 | Towers of Hanoi for the case with four disks.

a) Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area. Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved
b) The peg on which these disks are initially threaded
c) The peg to which this stack of disks is to be moved
d) The peg to be used as a temporary holding area

Display the precise instructions for moving the disks from the starting peg to the destination peg. To move a stack of three disks from peg 1 to peg 3, the program displays the following moves:

1 → 3 (This means move one disk from peg 1 to peg 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

6.39 *(Towers of Hanoi: Iterative Version)* Any program that can be implemented recursively can be implemented iteratively, although sometimes with more difficulty and less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version developed in Exercise 6.38. Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

6.40 *(Visualizing Recursion)* It's interesting to watch recursion "in action." Modify the factorial function of Fig. 6.29 to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

6.41 *(Recursive Greatest Common Divisor)* The greatest common divisor of integers `x` and `y` is the largest integer that evenly divides both `x` and `y`. Write a recursive function `gcd` that returns the greatest common divisor of `x` and `y`, defined recursively as follows: If `y` is equal to `0`, then `gcd(x, y)` is `x`; otherwise, `gcd(x, y)` is `gcd(y, x % y)`, where `%` is the modulus operator. [*Note:* For this algorithm, `x` must be larger than `y`.]

6.42 *(Recursive `main`)* Can `main` be called recursively on your system? Write a program containing a function `main`. Include `static` local variable `count` and initialize it to 1. Postincrement and print the value of `count` each time `main` is called. Compile your program. What happens?

6.43 *(Distance Between Points)* Write function `distance` that calculates the distance between two points *(x1, y1)* and *(x2, y2)*. All numbers and return values should be of type `double`.

6.44 What's wrong with the following program?

```cpp
// Exercise 6.44: ex06_44.cpp
// What is wrong with this program?
#include <iostream>
using namespace std;

int main()
{
   int c;

   if ( ( c = cin.get() ) != EOF )
   {
      main();
      cout << c;
   } // end if
} // end main
```

6.45 What does the following program do?

```cpp
// Exercise 6.45: ex06_45.cpp
// What does this program do?
#include <iostream>
using namespace std;

int mystery( int, int ); // function prototype

int main()
{
   int x, y;

   cout << "Enter two integers: ";
   cin >> x >> y;
   cout << "The result is " << mystery( x, y ) << endl;
} // end main

// Parameter b must be a positive integer to prevent infinite recursion
int mystery( int a, int b )
{
   if ( b == 1 ) // base case
      return a;
   else // recursion step
      return a + mystery( a, b - 1 );
} // end function mystery
```

6.46 After you determine what the program of Exercise 6.45 does, modify the program to function properly after removing the restriction that the second argument be nonnegative.

6.47 *(Math Library Functions)* Write a program that tests as many of the math library functions in Fig. 6.2 as you can. Exercise each of these functions by having your program print out tables of return values for a diversity of argument values.

6.48 *(Find the Error)* Find the error in each of the following program segments and explain how to correct it:

a)
```
float cube( float ); // function prototype

cube( float number ) // function definition
{
   return number * number * number;
}
```
b)
```
register auto int x = 7;
```
c)
```
int randomNumber = srand();
```
d)
```
float y = 123.45678;
int x;

x = y;
cout << static_cast< float >( x ) << endl;
```
e)
```
double square( double number )
{
   double number;
   return number * number;
}
```
f)
```
int sum( int n )
{
   if ( n == 0 )
      return 0;
   else
      return n + sum( n );
}
```

6.49 *(Craps Game Modification)* Modify the craps program of Fig. 6.11 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable `bankBalance` to 1000 dollars. Prompt the player to enter a `wager`. Use a `while` loop to check that `wager` is less than or equal to `bankBalance` and, if not, prompt the user to reenter `wager` until a valid `wager` is entered. After a correct `wager` is entered, run one game of craps. If the player wins, increase `bankBalance` by `wager` and print the new `bankBalance`. If the player loses, decrease `bankBalance` by `wager`, print the new `bankBalance`, check on whether `bankBalance` has become zero and, if so, print the message `"Sorry. You busted!"` As the game progresses, print various messages to create some “chatter” such as `"Oh, you're going for broke, huh?"`, `"Aw cmon, take a chance!"` or `"You're up big. Now's the time to cash in your chips!"`.

6.50 *(Circle Area)* Write a C++ program that prompts the user for the radius of a circle, then calls `inline` function `circleArea` to calculate the area of that circle.

6.51 *(Pass-by-Value vs. Pass-by-Reference)* Write a complete C++ program with the two alternate functions specified below, each of which simply triples the variable `count` defined in `main`. Then compare and contrast the two approaches. These two functions are

a) function `tripleByValue` that passes a copy of `count` by value, triples the copy and returns the new value and
a) function `tripleByReference` that passes `count` by reference via a reference parameter and triples the original value of `count` through its alias (i.e., the reference parameter).

6.52 What's the purpose of the unary scope resolution operator?

6.53 *(Function Template `minimum`)* Write a program that uses a function template called `minimum` to determine the smaller of two arguments. Test the program using integer, character and floating-point number arguments.

6.54 *(Function Template `maximum`)* Write a program that uses a function template called `maximum` to determine the larger of two arguments. Test the program using integer, character and floating-point number arguments.

6.55 *(Find the Error)* Determine whether the following program segments contain errors. For each error, explain how it can be corrected. [*Note:* For a particular program segment, it's possible that no errors are present in the segment.]

a)
```
template < class A >
int sum( int num1, int num2, int num3 )
{
   return num1 + num2 + num3;
}
```
b)
```
void printResults( int x, int y )
{
   cout << "The sum is " << x + y << '\n';
   return x + y;
}
```
c)
```
template < A >
A product( A num1, A num2, A num3 )
{
   return num1 * num2 * num3;
}
```
d)
```
double cube( int );
int cube( int );
```

Making a Difference

As computer costs decline, it becomes feasible for every student, regardless of economic circumstance, to have a computer and use it in school. This creates exciting possibilities for improving the educational experience of all students worldwide as suggested by the next five exercises. [*Note:* Check out initiatives such as the One Laptop Per Child Project (`www.laptop.org`). Also, research "green" laptops—and note the key "going green" characteristics of these devices? Look into the Electronic Product Environmental Assessment Tool (`www.epeat.net`) which can help you assess the "greenness" of desktops, notebooks and monitors to help you decide which products to purchase.]

6.56 *(Computer-Assisted Instruction)* The use of computers in education is referred to as *computer-assisted instruction* (*CAI*). Write a program that will help an elementary school student learn multiplication. Use the `rand` function to produce two positive one-digit integers. The program should then prompt the user with a question, such as

```
How much is 6 times 7?
```

The student then inputs the answer. Next, the program checks the student's answer. If it's correct, display the message `"Very good!"` and ask another multiplication question. If the answer is wrong, display the message `"No. Please try again."` and let the student try the same question repeatedly until the student finally gets it right. A separate function should be used to generate each new question. This function should be called once when the application begins execution and each time the user answers the question correctly.

6.57 *(Computer-Assisted Instruction: Reducing Student Fatigue)* One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise 6.56 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Possible responses to an incorrect answer:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a `switch` statement to issue the responses.

6.58 *(Computer-Assisted Instruction: Monitoring Student Performance)* More sophisticated computer-assisted instruction systems monitor the student's performance over a period of time. The decision to begin a new topic is often based on the student's success with previous topics. Modify the program of Exercise 6.57 to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage that are correct. If the percentage is lower than 75%, display `"Please ask your teacher for extra help."`, then reset the program so another student can try it. If the percentage is 75% or higher, display `"Congratulations, you are ready to go to the next level!"`, then reset the program so another student can try it.

6.59 *(Computer-Assisted Instruction: Difficulty Levels)* Exercises 6.56–6.58 developed a computer-assisted instruction program to help teach an elementary school student multiplication. Modify the program to allow the user to enter a difficulty level. At a difficulty level of 1, the program should use only single-digit numbers in the problems; at a difficulty level of 2, numbers as large as two digits, and so on.

6.60 *(Computer-Assisted Instruction: Varying the Types of Problems)* Modify the program of Exercise 6.59 to allow the user to pick a type of arithmetic problem to study. An option of `1` means addition problems only, `2` means subtraction problems only, `3` means multiplication problems only, `4` means division problems only and `5` means a random mixture of all these types.

7

Arrays and Vectors

Now go, write it before them in a table, and note it in a book.
—Isaiah 30:8

Begin at the beginning, … and go on till you come to the end: then stop.
—Lewis Carroll

To go beyond is as wrong as to fall short.
—Confucius

Objectives

In this chapter you'll learn:

- To use the array data structure to represent a set of related data items.
- To use arrays to store, sort and search lists and tables of values.
- To declare arrays, initialize arrays and refer to the individual elements of arrays.
- To pass arrays to functions.
- Basic searching and sorting techniques.
- To declare and manipulate multidimensional arrays.
- To use C++ Standard Library class template `vector`.

Outline

7.1 Introduction

This chapter introduces the topic of **data structures**—collections of related data items. **Arrays** are data structures consisting of related data items of the same type. You learned about classes in Chapter 3. In Chapter 21 we discuss the notion of **structures**. Structures and classes can each hold related data items of possibly different types. Arrays, structures and classes are "static" entities in that they remain the same size throughout program execution. (They may, of course, be of automatic storage class and hence be created and destroyed each time the blocks in which they're defined are entered and exited.)

After discussing how arrays are declared, created and initialized, we present a series of practical examples that demonstrate several common array manipulations. We present an example of searching arrays to find particular elements. The chapter also introduces one of the most important computing applications—sorting data (i.e., putting the data in some particular order). Two sections of the chapter enhance the `GradeBook` class by using arrays to enable the class to maintain a set of grades in memory and analyze student grades from multiple exams in a semester.

The style of arrays we use throughout most of this chapter are C-style pointer-based arrays. (We'll study pointers in Chapter 8.) In Section 7.11, and in Chapter 22, Standard Template Library (STL), we'll cover arrays as full-fledged objects called vectors. We'll discover that these object-based arrays are safer and more versatile than the C-style, pointer-based arrays that we discuss in the early part of this chapter. As part of the vector example, we introduce the exception-handling mechanism and use it to allow a program to continue executing when the program attempts to access a vector element that does not exist.

7.2 Arrays

An array is a consecutive group of memory locations that all have the *same* type. To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element in the array.

Figure 7.1 shows an integer array called `c` that contains 12 **elements**. You refer to any one of these elements by giving the array name followed by the particular element's position number in square brackets (`[]`). The position number is more formally called a **subscript** or **index** (this number specifies the number of elements from the beginning of the array). The first element has **subscript 0** (**zero**) and is sometimes called the **zeroth element**. Thus, the elements of array `c` are `c[0]` (pronounced "c sub zero"), `c[1]`, `c[2]` and so on. The highest subscript in array `c` is 11, which is 1 less than the number of elements in the array (12). Array names follow the same conventions as other variable names.

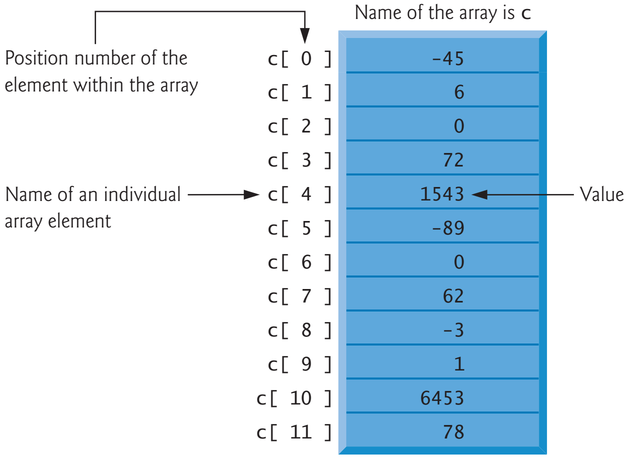


Fig. 7.1 | Array of 12 elements.

A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable `a` is equal to `5` and that variable `b` is equal to `6`, then the statement

```
c[ a + b ] += 2;
```

adds 2 to array element `c[11]`. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment, just as nonarray variable names can.

Let's examine array `c` in Fig. 7.1 more closely. The **name** of the entire array is `c`. Its 12 elements are referred to as `c[0]` to `c[11]`. The **value** of `c[0]` is `-45`, the value of `c[1]` is `6`, the value of `c[2]` is `0`, the value of `c[7]` is `62`, and the value of `c[11]` is `78`. To print the sum of the values contained in the first three elements of array `c`, we'd write

```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

To divide the value of `c[6]` by `2` and assign the result to the variable `x`, we would write

```
x = c[ 6 ] / 2;
```

Common Programming Error 7.1

Note the difference between the "seventh element of the array" and "array element 7." Subscripts begin at 0, so the "seventh element of the array" has a subscript of 6, while "array element 7" has a subscript of 7 and is actually the eighth *element of the array. This distinction is a frequent source of* ***off-by-one errors****. To avoid such errors, we refer to specific array elements explicitly by their array name and subscript number (e.g.,* `c[6]` *or* `c[7]`*).*

The brackets that enclose a subscript are actually an operator that has the same precedence as parentheses. Figure 7.2 shows the precedence and associativity of the operators introduced so far. The operators are shown top to bottom in decreasing order of precedence with their associativity and type.

Operators	Associativity	Type
`::` `()`	*[See parentheses caution in Fig. 2.10]*	scope resolution
`()` `[]`	left to right	function call/array access
`++` `--` `static_cast<type>(operand)`	left to right	unary (postfix)
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.2 | Operator precedence and associativity.

7.3 Declaring Arrays

Arrays occupy space in memory. To specify the type of the elements and the number of elements required by an array use a declaration of the form:

```
type arrayName[ arraySize ];
```

The compiler reserves the appropriate amount of memory. (Recall that a declaration which reserves memory is more properly known as a *definition*.) The *arraySize* must be an integer constant greater than zero. For example, to tell the compiler to reserve 12 elements for integer array `c`, use the declaration

```
int c[ 12 ]; // c is an array of 12 integers
```

Arrays can be declared to contain values of any nonreference data type. For example, an array of type `string` can be used to store character strings.

7.4 Examples Using Arrays

This section presents many examples that demonstrate how to declare, initialize and manipulate arrays.

7.4.1 Declaring an Array and Using a Loop to Initialize the Array's Elements

The program in Fig. 7.3 declares 10-element integer array `n` (line 9). Lines 12–13 use a `for` statement to initialize the array elements to zeros. Like other automatic variables, automatic arrays are *not* implicitly initialized to zero although `static` arrays are. The first output statement (line 15) displays the column headings for the columns printed in the subsequent `for` statement (lines 18–19), which prints the array in tabular format. Remember that `setw` specifies the field width in which only the *next* value is to be output.

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array's elements to zeros and printing the array.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int n[ 10 ]; // n is an array of 10 integers
10
11    // initialize elements of array n to 0
12    for ( int i = 0; i < 10; ++i )
13       n[ i ] = 0; // set element at location i to 0
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
16
17    // output each array element's value
18    for ( int j = 0; j < 10; ++j )
19       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
20 } // end main
```

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Fig. 7.3 | Initializing an array's elements to zeros and printing the array.

7.4.2 Initializing an Array in a Declaration with an Initializer List

The elements of an array also can be initialized in the array declaration by following the array name with an equals sign and a brace-delimited comma-separated list of **initializers**. The program in Fig. 7.4 uses an **initializer list** to initialize an integer array with 10 values (line 10) and prints the array in tabular format (lines 12–16).

```
1  // Fig. 7.4: fig07_04.cpp
2  // Initializing an array in a declaration.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // use initializer list to initialize array n
10    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12    cout << "Element" << setw( 13 ) << "Value" << endl;
13
14    // output each array element's value
15    for ( int i = 0; i < 10; ++i )
16       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17 } // end main
```

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Fig. 7.4 | Initializing an array in a declaration.

If there are fewer initializers than array elements, the remaining array elements are initialized to zero. For example, the elements of array `n` in Fig. 7.3 could have been initialized to zero with the declaration

```
int n[ 10 ] = {}; // initialize elements of array n to 0
```

which initializes the elements to zero, because there are fewer initializers (none in this case) than array elements. This technique can be used only in the array's declaration, whereas the initialization technique shown in Fig. 7.3 can be used repeatedly during program execution to "reinitialize" an array's elements.

If the array size is omitted from a declaration with an initializer list, the compiler sizes the array to the number of elements in the initializer list. For example,

```
int n[] = { 1, 2, 3, 4, 5 };
```

creates a five-element array.

If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. The array declaration

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

causes a compilation error, because there are six initializers and only five array elements.

7.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations

Figure 7.5 sets the elements of a 10-element array `s` to the even integers `2`, `4`, `6`, …, `20` (lines 14–15) and prints the array in tabular format (lines 17–21). These numbers are generated (line 15) by multiplying each successive value of the loop counter by `2` and adding `2`.

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 20.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // constant variable can be used to specify array size
10    const int arraySize = 10;
11
12    int s[ arraySize ]; // array s has 10 elements
13
14    for ( int i = 0; i < arraySize; ++i ) // set the values
15       s[ i ] = 2 + 2 * i;
16
17    cout << "Element" << setw( 13 ) << "Value" << endl;
18
19    // output contents of array s in tabular format
20    for ( int j = 0; j < arraySize; ++j )
21       cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
22 } // end main
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

Fig. 7.5 | Generating values to be placed into elements of an array.

Line 10 uses the **const qualifier** to declare a so-called **constant variable** `arraySize` with the value `10`. Constant variables *must* be initialized with a constant expression when they're declared and *cannot* be modified thereafter (as shown in Fig. 7.6 and Fig. 7.7). Constant variables are also called **named constants** or **read-only variables**.

Common Programming Error 7.2

Not initializing a constant variable when it's declared is a compilation error.

Common Programming Error 7.3

Assigning a value to a constant variable in an executable statement is a compilation error.

```
// Fig. 7.6: fig07_06.cpp
// Using a properly initialized constant variable.
#include <iostream>
using namespace std;

int main()
{
   const int x = 7; // initialized constant variable

   cout << "The value of constant variable x is: " << x << endl;
} // end main
```

```
The value of constant variable x is: 7
```

Fig. 7.6 | Using a properly initialized constant variable.

```
// Fig. 7.7: fig07_07.cpp
// A const variable must be initialized.

int main()
{
   const int x; // Error: x must be initialized

   x = 7; // Error: cannot modify a const variable
} // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
   must be initialized if not extern
C:\cpphtp8_examples\ch07\fig07_07.cpp(8) : error C3892: 'x' : you cannot
   assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_07.cpp:6: error: uninitialized const 'x'
fig07_07.cpp:8: error: assignment of read-only variable 'x'
```

Fig. 7.7 | A `const` variable must be initialized.

In Fig. 7.7, the compilation error produced by Microsoft Visual C++ refers to the `int` variable `x` as a "`const` object." The ISO/IEC C++ standard defines an "object" as any "region of storage." Like objects of classes, fundamental-type variables also occupy space in memory, so they're often referred to as "objects."

Constant variables can be placed anywhere a constant expression is expected. In Fig. 7.5, constant variable `arraySize` specifies the size of array `s` in line 12.

Common Programming Error 7.4

Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.

Using constant variables to specify array sizes makes programs more **scalable**. In Fig. 7.5, the first `for` statement could fill a 1000-element array by simply changing the value of `arraySize` in its declaration from `10` to `1000`. If the constant variable `arraySize` had not been used, we would have to change lines 12, 14 and 20 of the program to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clearer, easier-to-modify programs.

Software Engineering Observation 7.1

Defining the size of each array as a constant variable instead of a literal constant can make programs more scalable.

Good Programming Practice 7.1

*Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called **magic numbers**. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can be confusing when the program includes other 10s that have nothing to do with the array size.*

7.4.4 Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam.

The program in Fig. 7.8 sums the values contained in the 10-element integer array `a`. The program declares, creates and initializes the array in line 9. The `for` statement (lines 13–14) performs the calculations. The values being supplied as initializers for array `a` also could be read into the program from the user at the keyboard, or from a file on disk (see Chapter 17, File Processing). For example, the `for` statement

```
for ( int j = 0; j < arraySize; ++j )
   cin >> a[ j ];
```

reads one value at a time from the keyboard and stores the value in element `a[j]`.

```
1 // Fig. 7.8: fig07_08.cpp
2 // Compute the sum of the elements of the array.
3 #include <iostream>
```

Fig. 7.8 | Computing the sum of the elements of an array. (Part 1 of 2.)

```
using namespace std;

int main()
{
   const int arraySize = 10; // constant variable indicating size of array
   int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
   int total = 0;

   // sum contents of array a
   for ( int i = 0; i < arraySize; ++i )
      total += a[ i ];

   cout << "Total of array elements: " << total << endl;
} // end main
```

```
Total of array elements: 849
```

Fig. 7.8 | Computing the sum of the elements of an array. (Part 2 of 2.)

7.4.5 Using Bar Charts to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).

Professors often like to examine the distribution of grades on an exam. A professor might graph the number of grades in each of several categories to visualize the grade distribution. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. There was one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next program (Fig. 7.9) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, `n[0]` indicates the number of grades in the range 0–9, `n[7]` indicates the number of grades in the range 70–79 and `n[10]` indicates the number of grades of 100. The two versions of class `GradeBook` later in the chapter (Figs. 7.15–7.16 and Figs. 7.22–7.23) contain code that calculates these grade frequencies based on a set of grades. For now, we manually create the array by looking at the set of grades.

```
// Fig. 7.9: fig07_09.cpp
// Bar chart printing program.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int arraySize = 11;
   int n[ arraySize ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

```

Fig. 7.9 | Bar chart printing program. (Part 1 of 2.)

```
12     cout << "Grade distribution:" << endl;
13
14     // for each element of array n, output a bar of the chart
15     for ( int i = 0; i < arraySize; ++i )
16     {
17        // output bar labels ("0-9:", ..., "90-99:", "100:" )
18        if ( i == 0 )
19           cout << "  0-9: ";
20        else if ( i == 10 )
21           cout << "  100: ";
22        else
23           cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
24
25        // print bar of asterisks
26        for ( int stars = 0; stars < n[ i ]; ++stars )
27           cout << '*';
28
29        cout << endl; // start a new line of output
30     } // end outer for
31  } // end main
```

```
Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.9 | Bar chart printing program. (Part 2 of 2.)

The program reads the numbers from the array and graphs the information as a bar chart, displaying each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 18–23 output a grade range (e.g., `"70-79: "`) based on the current value of counter variable `i`. The nested `for` statement (lines 26–27) outputs the bars. Note the loop-continuation condition in line 26 (`stars < n[i]`). Each time the program reaches the inner `for`, the loop counts from `0` up to `n[i]`, thus using a value in array `n` to determine the number of asterisks to display. In this example, `n[0]`–`n[5]` contain zeros because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges.

7.4.6 Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. In Fig. 6.9, we used separate counters in our die-rolling program to track the number of occurrences of each side of a die as the program rolled the die 6,000,000 times. An array version of this program is shown in Fig. 7.10.

```
1  // Fig. 7.10: fig07_10.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib>
6  #include <ctime>
7  using namespace std;
8
9  int main()
10 {
11    const int arraySize = 7; // ignore element zero
12    int frequency[ arraySize ] = {}; // initialize elements to 0
13
14    srand( time( 0 ) ); // seed random number generator
15
16    // roll die 6,000,000 times; use die value as frequency index
17    for ( int roll = 1; roll <= 6000000; ++roll )
18       ++frequency[ 1 + rand() % 6 ];
19
20    cout << "Face" << setw( 13 ) << "Frequency" << endl;
21
22    // output each array element's value
23    for ( int face = 1; face < arraySize; ++face )
24       cout << setw( 4 ) << face << setw( 13 ) << frequency[ face ]
25          << endl;
26 } // end main
```

```
Face    Frequency
   1      1000167
   2      1000149
   3      1000152
   4       998748
   5       999626
   6      1001158
```

Fig. 7.10 | Die-rolling program using an array instead of `switch`.

Figure 7.10 uses the array `frequency` (line 12) to count the occurrences of each side of the die. *The single statement in line 18 of this program replaces the `switch` statement in lines 25–47 of Fig. 6.9.* Line 18 uses a random value to determine which `frequency` element to increment during each iteration of the loop. The calculation in line 18 produces a random subscript from 1 to 6, so array `frequency` must be large enough to store six counters. However, we use a seven-element array in which we ignore `frequency[0]`—it's more logical to have the die face value 1 increment `frequency[1]` than `frequency[0]`. Thus, each face value is used as a subscript for array `frequency`. We also replace lines 51–56 of Fig. 6.9 by looping through array `frequency` to output the results (lines 23–25).

7.4.7 Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:

> *Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.*

This is a typical array-processing application (Fig. 7.11). We wish to summarize the number of responses of each type (that is, 1–5). The array `responses` (lines 14–15) is a 20-element integer array of the students' responses to the survey. The array `responses` is declared `const`, as its values do not (and should not) change. We use a six-element array `frequency` (line 18) to count the number of occurrences of each response. Each element of the array is used as a counter for one of the survey responses and is initialized to zero. As in Fig. 7.10, we ignore `frequency[0]`.

The first `for` statement (lines 22–23) takes the responses one at a time from the array `responses` and increments one of the five counters in the `frequency` array (`frequency[1]`

```
1  // Fig. 7.11: fig07_11.cpp
2  // Poll analysis program.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // define array sizes
10    const int responseSize = 20; // size of array responses
11    const int frequencySize = 6; // size of array frequency
12
13    // place survey responses in array responses
14    const int responses[ responseSize ] = { 1, 2, 5, 4, 3, 5, 2, 1, 3,
15       1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5 };
16
17    // initialize frequency counters to 0
18    int frequency[ frequencySize ] = {};
19
20    // for each answer, select responses element and use that value
21    // as frequency subscript to determine element to increment
22    for ( int answer = 0; answer < responseSize; ++answer )
23       ++frequency[ responses[ answer ] ];
24
25    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
26
27    // output each array element's value
28    for ( int rating = 1; rating < frequencySize; ++rating )
29       cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
30          << endl;
31 } // end main
```

```
Rating          Frequency
     1                  3
     2                  5
     3                  7
     4                  2
     5                  3
```

Fig. 7.11 | Poll analysis program.

to `frequency[5]`). The key statement in the loop is line 23, which increments the appropriate `frequency` counter, depending on the value of `responses[answer]`.

Let's consider several iterations of the `for` loop. When control variable `answer` is `0`, the value of `responses[answer]` is the value of `responses[0]` (i.e., `1` in line 14), so the program interprets `++frequency[responses[answer]]` as

```
++frequency[ 1 ]
```

which increments the value in array element 1. To evaluate the expression, start with the value in the *innermost* set of square brackets (`answer`). Once you know `answer`'s value (which is the value of the loop control variable in line 23), plug it into the expression and evaluate the next outer set of square brackets (i.e., `responses[answer]`, which is a value selected from the `responses` array in lines 14–17). Then use the resulting value as the subscript for the `frequency` array to specify which counter to increment.

When `answer` is `1`, `responses[answer]` is the value of `responses[1]`, which is `2`, so the program interprets `++frequency[responses[answer]]` as

```
++frequency[ 2 ]
```

which increments array element 2.

When `answer` is `2`, `responses[answer]` is the value of `responses[2]`, which is `5`, so the program interprets `++frequency[responses[answer]]` as

```
++frequency[ 5 ]
```

which increments array element 5, and so on. Regardless of the number of responses processed in the survey, the program requires *only* an six-element array (ignoring element zero) to summarize the results, because all the response values are between 1 and 5 and the subscript values for an six-element array are 0 through 5.

If the data in `responses` contained an invalid value, such as 13, the program would have attempted to add `1` to `frequency[13]`, which is outside the bounds of the array. *C++ has no array* ***bounds checking*** *to prevent the computer from referring to an element that does not exist.* Thus, an executing program can "walk off" either end of an array without warning. You should ensure that all array references remain within the bounds of the array.

Common Programming Error 7.5

Referring to an element outside the array bounds is an execution-time *logic error. It isn't a syntax error.*

Error-Prevention Tip 7.1

When looping through an array, the index should never go below 0 and should always be less than the total number of array elements (one less than the size of the array). Make sure that the loop-termination condition prevents accessing elements outside this range.

C++ is an extensible language. Section 7.11 presents C++ Standard Library class template `vector`, which enables you to perform many operations that are not available for built-in arrays. For example, we'll be able to compare `vector`s directly and assign one `vector` to another. In Chapter 11, we extend C++ further by implementing an array as a class of our own. This new array definition will enable us to input and output entire arrays with `cin` and `cout`, initialize arrays when they're created and prevent access to out-of-range array elements. We'll even be able to use noninteger subscripts.

Error-Prevention Tip 7.2

In Chapter 11, we'll see how to develop a class representing a "smart array," which checks that all subscript references are in bounds at runtime. Using such smart data types helps eliminate bugs.

7.4.8 Static Local Arrays and Automatic Local Arrays

Chapter 6 discussed the storage-class specifier `static`. A `static` local variable in a function definition exists for the program's duration but is visible only in the function's body.

Performance Tip 7.1

We can apply `static` to a local array declaration so that it not created and initialized each time the program calls the function and is not destroyed each time the function terminates. This can improve performance, especially when using large arrays.

A program initializes `static` local arrays when their declarations are first encountered. If a `static` array is not initialized explicitly by you, each element of that array is initialized to zero by the compiler when the array is created. Recall that C++ does *not* perform such default initialization for automatic variables.

Figure 7.12 demonstrates function `staticArrayInit` (lines 23–39) with a `static` local array (line 26) and function `automaticArrayInit` (lines 42–58) with an automatic local array (line 45).

```
1  // Fig. 7.12: fig07_12.cpp
2  // Static arrays are initialized to zero.
3  #include <iostream>
4  using namespace std;
5
6  void staticArrayInit( void ); // function prototype
7  void automaticArrayInit( void ); // function prototype
8  const int arraySize = 3;
9
10 int main()
11 {
12    cout << "First call to each function:\n";
13    staticArrayInit();
14    automaticArrayInit();
15
16    cout << "\n\nSecond call to each function:\n";
17    staticArrayInit();
18    automaticArrayInit();
19    cout << endl;
20 } // end main
21
22 // function to demonstrate a static local array
23 void staticArrayInit( void )
24 {
25    // initializes elements to 0 first time function is called
26    static int array1[ arraySize ]; // static local array
27
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 1 of 2.)

```
   cout << "\nValues on entering staticArrayInit:\n";

   // output contents of array1
   for ( int i = 0; i < arraySize; ++i )
      cout << "array1[" << i << "] = " << array1[ i ] << "  ";

   cout << "\nValues on exiting staticArrayInit:\n";

   // modify and output contents of array1
   for ( int j = 0; j < arraySize; ++j )
      cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "  ";
} // end function staticArrayInit

// function to demonstrate an automatic local array
void automaticArrayInit( void )
{
   // initializes elements each time function is called
   int array2[ arraySize ] = { 1, 2, 3 }; // automatic local array

   cout << "\n\nValues on entering automaticArrayInit:\n";

   // output contents of array2
   for ( int i = 0; i < arraySize; ++i )
      cout << "array2[" << i << "] = " << array2[ i ] << "  ";

   cout << "\nValues on exiting automaticArrayInit:\n";

   // modify and output contents of array2
   for ( int j = 0; j < arraySize; ++j )
      cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "  ";
} // end function automaticArrayInit
```

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Fig. 7.12 | `static` array initialization and automatic array initialization. (Part 2 of 2.)

Function `staticArrayInit` is called twice (lines 13 and 17). The `static` local array is initialized to zero by the compiler the first time the function is called. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the `static` array contains the modified values stored during the first function call. Function `automaticArrayInit` also is called twice (lines 14 and 18). The elements of the automatic local array are initialized (line 45) with the values 1, 2 and 3. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are reinitialized to 1, 2 and 3. The array has automatic storage class, so the array is recreated and reinitialized during each call to `automaticArrayInit`.

Common Programming Error 7.6

Assuming that elements of a function's local `static` array are initialized every time the function is called can lead to logic errors in a program.

7.5 Passing Arrays to Functions

To pass an array argument to a function, specify the name of the array without any brackets. For example, if array `hourlyTemperatures` has been declared as

```
int hourlyTemperatures[ 24 ];
```

the function call

```
modifyArray( hourlyTemperatures, 24 );
```

passes array `hourlyTemperatures` and its size to function `modifyArray`. When passing an array to a function, the array size is normally passed as well, so the function can process the specific number of elements in the array. Otherwise, we would need to build this knowledge into the called function itself or, worse yet, place the array size in a global variable. In Section 7.11, when we present C++ Standard Library class template `vector` to represent a more robust type of array, you'll see that the size of a `vector` is built in—every `vector` object "knows" its own size, which can be obtained by invoking the `vector` object's `size` member function. Thus, when we pass a `vector` *object* into a function, we won't have to pass the size of the `vector` as an argument.

C++ passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays. *The value of the name of the array is the address in the computer's memory of the first element of the array.* Because the starting address of the array is passed, the called function knows precisely where the array is stored in memory. Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their original memory locations.

Performance Tip 7.2

Passing arrays by reference makes sense for performance reasons. Passing by value would require copying each element. For large, frequently passed arrays, this would be time consuming and would require considerable storage for the copies of the array elements.

Software Engineering Observation 7.2

It's possible to pass an array by value by simply embedding it as a data member of a class and passing an object of the class, which defaults to pass-by-value.

Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call. In Chapter 6, we showed how to pass individual variables and array elements by reference with references—in Chapter 8, we show how to pass them by reference with pointers.

For a function to receive an array through a function call, the function's parameter list must specify that the function expects to receive an array. For example, the function header for function `modifyArray` might be written as

```
void modifyArray( int b[], int arraySize )
```

indicating that `modifyArray` expects to receive the address of an array of integers in parameter `b` and the number of array elements in parameter `arraySize`. The array's size is *not* required in the array brackets. If it's included, the compiler ignores it; thus, arrays of any size can be passed to the function. C++ passes arrays to functions by reference—when the called function uses the array name `b`, it refers to the actual array in the caller (i.e., array `hourlyTemperatures` discussed at the beginning of this section).

Note the strange appearance of the function prototype for `modifyArray`

```
void modifyArray( int [], int );
```

This prototype could have been written (for documentation purposes)

```
void modifyArray( int anyArrayName[], int anyVariableName );
```

but, as we learned in Chapter 3, C++ compilers *ignore* variable names in prototypes. Remember, the prototype tells the compiler the number of arguments and the type of each argument (in the order in which the arguments are expected to appear).

The program in Fig. 7.13 demonstrates the difference between passing an entire array and passing an array element. Lines 19–20 print the five original elements of integer array `a`. Line 25 passes `a` and its size to function `modifyArray` (lines 40–45), which multiplies each of `a`'s elements by 2 (through parameter `b`). Then, lines 29–30 print array `a` again in `main`. As the output shows, the elements of `a` are indeed modified by `modifyArray`. Next, line 33 prints the value of `a[3]`, then line 35 passes element `a[3]` to function `modifyElement` (lines 49–53), which multiplies its parameter by 2 and prints the new value. When line 36 prints `a[3]` again in `main`, the value has not been modified, because individual array elements are passed by value.

```
1  // Fig. 7.13: fig07_13.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  void modifyArray( int [], int ); // appears strange; array and size
8  void modifyElement( int ); // receive array element value
9
10 int main()
11 {
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 1 of 2.)

```
   const int arraySize = 5; // size of array a
   int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a

   cout << "Effects of passing entire array by reference:"
      << "\nThe values of the original array are:\n";

   // output original array elements
   for ( int i = 0; i < arraySize; ++i )
      cout << setw( 3 ) << a[ i ];

   cout << endl;

   // pass array a to modifyArray by reference
   modifyArray( a, arraySize );
   cout << "The values of the modified array are:\n";

   // output modified array elements
   for ( int j = 0; j < arraySize; ++j )
      cout << setw( 3 ) << a[ j ];

   cout << "\n\nEffects of passing array element by value:"
      << "\na[3] before modifyElement: " << a[ 3 ] << endl;

   modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
   cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
} // end main

// in function modifyArray, "b" points to the original array "a" in memory
void modifyArray( int b[], int sizeOfArray )
{
   // multiply each array element by 2
   for ( int k = 0; k < sizeOfArray; ++k )
      b[ k ] *= 2;
} // end function modifyArray

// in function modifyElement, "e" is a local copy of
// array element a[ 3 ] passed from main
void modifyElement( int e )
{
   // multiply parameter by 2
   cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
} // end function modifyElement
```

```
Effects of passing entire array by reference:
The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element by value:
a[3] before modifyElement: 6
Value of element in modifyElement: 12
a[3] after modifyElement: 6
```

Fig. 7.13 | Passing arrays and individual array elements to functions. (Part 2 of 2.)

There may be situations in your programs in which a function should *not* be allowed to modify array elements. The type qualifier `const` can be used to prevent modification of array values in the caller by code in a called function. When a function specifies an array parameter that's preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error. This enables you to prevent accidental modification of array elements in the function's body.

Figure 7.14 demonstrates the `const` qualifier applied to an array parameter. Function `tryToModifyArray` (lines 18–21) is defined with parameter `const int b[]`, which specifies that array `b` is constant and cannot be modified. The attempt by the function to modify array `b`'s element 0 (line 20) results in a compilation error. Some compilers, for example, produce an error like "`Cannot modify a const object`." This message indicates that using a `const` object (`b[0]`) as an *lvalue* is an error—you *cannot* assign a new value to a `const` object.

Software Engineering Observation 7.3

Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it's absolutely necessary.

```
1  // Fig. 7.14: fig07_14.cpp
2  // Demonstrating the const type qualifier.
3  #include <iostream>
4  using namespace std;
5
6  void tryToModifyArray( const int [] ); // function prototype
7
8  int main()
9  {
10    int a[] = { 10, 20, 30 };
11
12    tryToModifyArray( a );
13    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
14 } // end main
15
16 // In function tryToModifyArray, "b" cannot be used
17 // to modify the original array "a" in main.
18 void tryToModifyArray( const int b[] )
19 {
20    b[ 0 ] /= 2; // compilation error
21 } // end function tryToModifyArray
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch07\fig07_14\fig07_14.cpp(20) : error C3892: 'b' : you
   cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_14.cpp:20: error: assignment of read-only location
```

Fig. 7.14 | `const` type qualifier applied to an array parameter.

7.6 Case Study: Class GradeBook Using an Array to Store Grades

This section further evolves class `GradeBook`, introduced in Chapter 3 and expanded in Chapters 4–6. Recall that this class represents a grade book used by a professor to store and analyze student grades. Previous versions of the class process grades entered by the user, but *do not* maintain the individual grade values in the class's data members. Thus, repeat calculations require the user to reenter the grades. One way to solve this problem would be to store each grade entered in an individual data member of the class. For example, we could create data members `grade1`, `grade2`, …, `grade10` in class `GradeBook` to store 10 student grades. However, the code to total the grades and determine the class average would be cumbersome. In this section, we solve this problem by storing grades in an array.

Storing Student Grades in an Array in Class `GradeBook`

The version of class `GradeBook` (Figs. 7.15–7.16) presented here uses an array of integers to store the grades of several students on a single exam. This eliminates the need to repeatedly input the same set of grades. Array `grades` is declared as a data member in line 28 of Fig. 7.15—therefore, each `GradeBook` object maintains its own set of grades.

```
1  // Fig. 7.15: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ Standard Library string class
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    // constant -- number of students who took the test
12    static const int students = 10; // note public data
13
14    // constructor initializes course name and array of grades
15    GradeBook( string, const int [] );
16
17    void setCourseName( string ); // function to set the course name
18    string getCourseName(); // function to retrieve the course name
19    void displayMessage(); // display a welcome message
20    void processGrades(); // perform various operations on the grade data
21    int getMinimum(); // find the minimum grade for the test
22    int getMaximum(); // find the maximum grade for the test
23    double getAverage(); // determine the average grade for the test
24    void outputBarChart(); // output bar chart of grade distribution
25    void outputGrades(); // output the contents of the grades array
26 private:
27    string courseName; // course name for this grade book
28    int grades[ students ]; // array of student grades
29 }; // end class GradeBook
```

Fig. 7.15 | Definition of class `GradeBook` that uses an array to store test grades.

```
// Fig. 7.16: GradeBook.cpp
// Member-function definitions for class GradeBook that
// uses an array to store test grades.
#include <iostream>
#include <iomanip>
#include "GradeBook.h" // GradeBook class definition
using namespace std;

// constructor initializes courseName and grades array
GradeBook::GradeBook( string name, const int gradesArray[] )
{
   setCourseName( name ); // initialize courseName

   // copy grades from gradesArray to grades data member
   for ( int grade = 0; grade < students; ++grade )
      grades[ grade ] = gradesArray[ grade ];
} // end GradeBook constructor

// function to set the course name
void GradeBook::setCourseName( string name )
{
   courseName = name; // store the course name
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!"
      << endl;
} // end function displayMessage

// perform various operations on the data
void GradeBook::processGrades()
{
   outputGrades(); // output grades array

   // display average of all grades and minimum and maximum grades
   cout << "\nClass average is " << setprecision( 2 ) << fixed <<
      getAverage() << "\nLowest grade is " << getMinimum() <<
      "\nHighest grade is " << getMaximum() << endl;

   outputBarChart(); // print grade distribution chart
} // end function processGrades

```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 1 of 3.)

```cpp
// find minimum grade
int GradeBook::getMinimum()
{
   int lowGrade = 100; // assume lowest grade is 100

   // loop through grades array
   for ( int grade = 0; grade < students; ++grade )
   {
      // if current grade lower than lowGrade, assign it to lowGrade
      if ( grades[ grade ] < lowGrade )
         lowGrade = grades[ grade ]; // new lowest grade
   } // end for

   return lowGrade; // return lowest grade
} // end function getMinimum

// find maximum grade
int GradeBook::getMaximum()
{
   int highGrade = 0; // assume highest grade is 0

   // loop through grades array
   for ( int grade = 0; grade < students; ++grade )
   {
      // if current grade higher than highGrade, assign it to highGrade
      if ( grades[ grade ] > highGrade )
         highGrade = grades[ grade ]; // new highest grade
   } // end for

   return highGrade; // return highest grade
} // end function getMaximum

// determine average grade for test
double GradeBook::getAverage()
{
   int total = 0; // initialize total

   // sum grades in array
   for ( int grade = 0; grade < students; ++grade )
      total += grades[ grade ];

   // return average of grades
   return static_cast< double >( total ) / students;
} // end function getAverage

// output bar chart displaying grade distribution
void GradeBook::outputBarChart()
{
   cout << "\nGrade distribution:" << endl;

   // stores frequency of grades in each range of 10 grades
   const int frequencySize = 11;
   int frequency[ frequencySize ] = {}; // initialize elements to 0
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 2 of 3.)

```

   // for each grade, increment the appropriate frequency
   for ( int grade = 0; grade < students; ++grade )
      ++frequency[ grades[ grade ] / students ];

   // for each grade frequency, print bar in chart
   for ( int count = 0; count < frequencySize; ++count )
   {
      // output bar labels ("0-9:", ..., "90-99:", "100:" )
      if ( count == 0 )
         cout << "  0-9: ";
      else if ( count == 10 )
         cout << "  100: ";
      else
         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";

      // print bar of asterisks
      for ( int stars = 0; stars < frequency[ count ]; ++stars )
         cout << '*';

      cout << endl; // start a new line of output
   } // end outer for
} // end function outputBarChart

// output the contents of the grades array
void GradeBook::outputGrades()
{
   cout << "\nThe grades are:\n\n";

   // output each student's grade
   for ( int student = 0; student < students; ++student )
      cout << "Student " << setw( 2 ) << student + 1 << ": " << setw( 3 )
         << grades[ student ] << endl;
} // end function outputGrades
```

Fig. 7.16 | GradeBook class member functions manipulating an array of grades. (Part 3 of 3.)

The size of the array in line 28 of Fig. 7.15 is specified by `public static const` data member `students` (declared in line 12), which is `public` so that it's accessible to the class's clients. We'll soon see an example of a client program using this constant. Declaring `students` with the `const` qualifier indicates that this data member is constant—its value cannot be changed after being initialized. Keyword `static` in this variable declaration indicates that the data member is shared by all objects of the class—all `GradeBook` objects store grades for the same number of students. Recall from Section 3.4 that when each object of a class maintains its own copy of an attribute, the variable that represents the attribute is known as a data member—each object (instance) of the class has a separate copy of the variable in memory. There are variables for which each object of a class does not have a separate copy. That is the case with **`static` data members**, which are also known as **class variables**. When objects of a class containing `static` data members are created, all the objects share one copy of the class's `static` data members. A `static` data member can be accessed within the class definition and the member-function definitions like any other data member. As you'll soon see, a `public` `static` data member can also be accessed outside of the class, *even when no objects of the class* *exist*, using the class name followed by the scope resolution operator (`::`) and the name of the data member. You'll learn more about `static` data members in Chapter 10.

Constructor

The class's constructor (declared in line 15 of Fig. 7.15 and defined in lines 10–17 of Fig. 7.16) has two parameters—the course name and an array of grades. When a program creates a `GradeBook` object (e.g., lines 12–13 of `fig07_17.cpp`), the program passes an existing `int` array to the constructor, which copies the array's values into the data member `grades` (lines 15–16 of Fig. 7.16). The grade values in the passed array could have been input from a user or read from a file on disk (as we discuss in Chapter 17, File Processing). In our test program, we simply initialize an array with a set of grade values (Fig. 7.17, lines 9–10). Once the grades are stored in data member `grades` of class `GradeBook`, all the class's member functions can access the `grades` array as needed to perform various calculations.

Member Function `processGrades`

Member function `processGrades` (declared in line 20 of Fig. 7.15 and defined in lines 41–51 of Fig. 7.16) contains a series of member function calls that output a report summarizing the grades. Line 43 calls member function `outputGrades` to print the contents of the array `grades`. Lines 136–138 in member function `outputGrades` use a `for` statement to output each student's grade. Although array indices start at 0, a professor would typically number students starting at 1. Thus, lines 137–138 output `student + 1` as the student number to produce grade labels `"Student 1: "`, `"Student 2: "`, and so on.

Member Function `getAverage`

Member function `processGrades` next calls member function `getAverage` (line 47) to obtain the average of the grades. Member function `getAverage` (declared in line 23 of Fig. 7.15 and defined in lines 86–96 of Fig. 7.16) totals the values in array `grades` before calculating the average. The averaging calculation in line 95 uses `static const` data member `students` to determine the number of grades being averaged.

Member Functions `getMinimum` *and* `getMaximum`

Lines 47–48 in `processGrades` call member functions `getMinimum` and `getMaximum` to determine the lowest and highest grades of any student on the exam, respectively. Let's examine how member function `getMinimum` finds the *lowest* grade. Because the highest grade allowed is 100, we begin by assuming that 100 is the lowest grade (line 56). Then, we compare each of the elements in the array to the lowest grade, looking for smaller values. Lines 59–64 in member function `getMinimum` loop through the array, and line 62 compares each grade to `lowGrade`. If a grade is less than `lowGrade`, `lowGrade` is set to that grade. When line 66 executes, `lowGrade` contains the lowest grade in the array. Member function `getMaximum` (lines 70–83) works similarly to member function `getMinimum`.

Member Function `outputBarChart`

Finally, line 50 in member function `processGrades` calls member function `outputBarChart` to print a distribution chart of the grade data using a technique similar to that in Fig. 7.9. In that example, we manually calculated the number of grades in each category (i.e., 0–9, 10–19, …, 90–99 and 100) by simply looking at a set of grades. In this example, lines 108–109 use a technique similar to that in Fig. 7.10 and Fig. 7.11 to calculate the frequency of grades in each category. Line 105 declares and creates array `frequency` of 11 `int`s to store the frequency of grades in each grade category. For each `grade` in array `grades`, lines 108–109 increment the appropriate element of the `frequency` array. To determine which element to increment, line 109 divides the current `grade` by 10 using integer division. For example, if `grade` is `85`, line 109 increments `frequency[8]` to update the count of grades in the range 80–89. Lines 112–127 next print the bar chart (see Fig. 7.17) based on the values in array `frequency`. Like lines 26–27 of Fig. 7.9, lines 123–124 of Fig. 7.16 use a value in array `frequency` to determine the number of asterisks to display in each bar.

Testing Class **GradeBook**

The program of Fig. 7.17 creates an object of class `GradeBook` (Figs. 7.15–7.16) using the `int` array `gradesArray` (declared and initialized in lines 9–10). The scope resolution operator (`::`) is used in the expression "`GradeBook::students`" (line 9) to access class `GradeBook`'s `static` constant `students`. We use this constant here to create an array that is the same size as array `grades` stored as a data member in class `GradeBook`. Lines 12–13 pass a course name and `gradesArray` to the `GradeBook` constructor. Line 14 displays a welcome message, and line 15 invokes the `GradeBook` object's `processGrades` member function. The output reveals the summary of the 10 grades in `myGradeBook`.

```
1  // Fig. 7.17: fig07_17.cpp
2  // Creates GradeBook object using an array of grades.
3  #include "GradeBook.h" // GradeBook class definition
4
5  // function main begins program execution
6  int main()
7  {
8     // array of student grades
9     int gradesArray[ GradeBook::students ] =
10       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12    GradeBook myGradeBook(
13       "CS101 Introduction to C++ Programming", gradesArray );
14    myGradeBook.displayMessage();
15    myGradeBook.processGrades();
16 } // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87
```

Fig. 7.17 | Creates a `GradeBook` object using an array of grades, then invokes member function `processGrades` to analyze them. (Part 1 of 2.)

```
Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.17 | Creates a `GradeBook` object using an array of grades, then invokes member function `processGrades` to analyze them. (Part 2 of 2.)

7.7 Searching Arrays with Linear Search

Often it may be necessary to determine whether an array contains a value that matches a certain **key value**. The process of finding a particular element of an array is called **searching**. In this section we discuss the simple linear search. Exercise 7.33 at the end of this chapter asks you to implement a recursive version of the linear search. In Chapter 19, Searching and Sorting, we present the more complex, yet more efficient, binary search.

Linear Search

The **linear search** (Fig. 7.18, lines 33–40) compares each element of an array with a **search key** (line 36). Because the array is not in any particular order, it's just as likely that the value will be found in the first element as the last. On average, therefore, the program must compare the search key with half the elements of the array. To determine that a value is not in the array, the program must compare the search key to every element of the array.

```
1  // Fig. 7.18: fig07_18.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using namespace std;
5
6  int linearSearch( const int [], int, int ); // prototype
7
8  int main()
9  {
10    const int arraySize = 100; // size of array a
11    int a[ arraySize ]; // create array a
12    int searchKey; // value to locate in array a
13
14    for ( int i = 0; i < arraySize; ++i )
15       a[ i ] = 2 * i; // create some data
```

Fig. 7.18 | Linear search of an array. (Part 1 of 2.)

```
   cout << "Enter integer search key: ";
   cin >> searchKey;

   // attempt to locate searchKey in array a
   int element = linearSearch( a, searchKey, arraySize );

   // display results
   if ( element != -1 )
      cout << "Found value in element " << element << endl;
   else
      cout << "Value not found" << endl;
} // end main

// compare key to every element of array until location is
// found or until end of array is reached; return subscript of
// element if key is found or -1 if key not found
int linearSearch( const int array[], int key, int sizeOfArray )
{
   for ( int j = 0; j < sizeOfArray; ++j )
      if ( array[ j ] == key ) // if found,
         return j; // return location of key

   return -1; // key not found
} // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

Fig. 7.18 | Linear search of an array. (Part 2 of 2.)

The linear searching method works well for small arrays or for unsorted arrays (i.e., arrays whose elements are in no particular order). However, for large arrays, linear searching is inefficient. If the array is sorted (e.g., its elements are in ascending order), you can use the high-speed binary search technique that you'll learn about in Chapter 19.

7.8 Sorting Arrays with Insertion Sort

Sorting data (i.e., placing the data into some particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their phone directories by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data and, in many cases, massive amounts of it. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science. In this chapter, we discuss a simple sorting scheme. In Chapter 19, we investigate more complex schemes that yield superior performance, and we introduce Big O (pronounced "Big Oh") notation for characterizing how hard each scheme must work to accomplish its task.

Insertion Sort

The program in Fig. 7.19 sorts the values of the 10-element array `data` into ascending order. The technique we use is called **insertion sort**—a simple, but inefficient, sorting algorithm. The first iteration of this algorithm takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the program *inserts* the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the i^{th} iteration of this algorithm, the first i elements in the original array will be sorted.

Line 10 of Fig. 7.19 declares and initializes array `data` with the following values:

```
34   56   4   10   77   51   93   30   5   52
```

The program first looks at `data[0]` and `data[1]`, whose values are `34` and `56`, respectively. These two elements are already in order, so the program continues—if they were out of order, the program would swap them.

```
1  // Fig. 7.19: fig07_19.cpp
2  // This program sorts an array's values into ascending order.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int arraySize = 10; // size of array a
10    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
11    int insert; // temporary variable to hold element to insert
12
13    cout << "Unsorted array:\n";
14
15    // output original array
16    for ( int i = 0; i < arraySize; ++i )
17       cout << setw( 4 ) << data[ i ];
18
19    // insertion sort
20    // loop over the elements of the array
21    for ( int next = 1; next < arraySize; ++next )
22    {
23       insert = data[ next ]; // store the value in the current element
24
25       int moveItem = next; // initialize location to place element
26
27       // search for the location in which to put the current element
28       while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
29       {
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 1 of 2.)

```
30          // shift element one slot to the right
31          data[ moveItem ] = data[ moveItem - 1 ];
32          moveItem--;
33       } // end while
34
35       data[ moveItem ] = insert; // place inserted element into the array
36    } // end for
37
38    cout << "\nSorted array:\n";
39
40    // output sorted array
41    for ( int i = 0; i < arraySize; ++i )
42       cout << setw( 4 ) << data[ i ];
43
44    cout << endl;
45 } // end main
```

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Fig. 7.19 | Sorting an array with insertion sort. (Part 2 of 2.)

In the second iteration, the program looks at the value of `data[2]`, `4`. This value is less than `56`, so the program stores `4` in a temporary variable and moves `56` one element to the right. The program then checks and determines that `4` is less than `34`, so it moves `34` one element to the right. The program has now reached the beginning of the array, so it places `4` in `data[0]`. The array now is

```
      4    34    56    10    77    51    93    30    5    52
```

In the third iteration, the program stores the value of `data[3]`, `10`, in a temporary variable. Then the program compares `10` to `56` and moves `56` one element to the right because it's larger than `10`. The program then compares `10` to `34`, moving `34` right one element. When the program compares `10` to `4`, it observes that `10` is larger than `4` and places `10` in `data[1]`. The array now is

```
      4    10    34    56    77    51    93    30    5    52
```

Using this algorithm, at the i^{th} iteration, the first i elements of the original array are sorted. They may not be in their final locations, however, because smaller values may be located later in the array.

The sorting is performed by the `for` statement in lines 21–36 that loops over the elements of the array. In each iteration, line 23 temporarily stores in variable `insert` (declared in line 11) the value of the element that will be inserted into the sorted portion of the array. Line 25 declares and initializes the variable `moveItem`, which keeps track of where to insert the element. Lines 28–33 loop to locate the correct position where the element should be inserted. The loop terminates either when the program reaches the front of the array or when it reaches an element that's less than the value to be inserted. Line 31 moves an element to the right, and line 32 decrements the position at which to insert the next element. After the `while` loop ends, line 35 inserts the element into place. When the `for` statement in lines 21–36 terminates, the elements of the array are sorted.

The chief virtue of the insertion sort is that it's easy to program; however, it runs slowly. This becomes apparent when sorting large arrays. In the exercises, we'll investigate some alternate algorithms for sorting an array. We investigate sorting and searching in greater depth in Chapter 19.

7.9 Multidimensional Arrays

Arrays with two dimensions (i.e., subscripts) often represent **tables of values** consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify two subscripts. By convention, the first identifies the element's row and the second identifies the element's column. Arrays that require two subscripts to identify a particular element are called **two-dimensional arrays** or **2-D arrays**. Arrays with two or more dimensions are known as **multidimensional arrays** and can have more than two dimensions. Figure 7.20 illustrates a two-dimensional array, `a`. The array contains three rows and four columns, so it's said to be a 3-by-4 array. In general, an array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

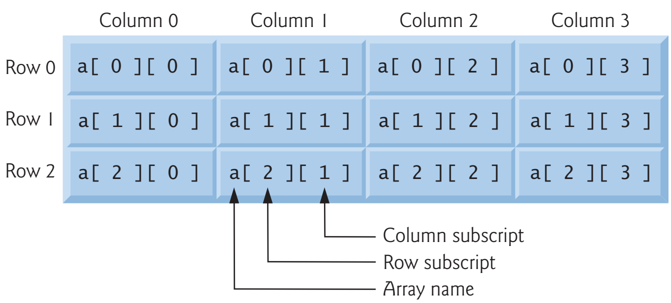


Fig. 7.20 | Two-dimensional array with three rows and four columns.

Every element in array `a` is identified in Fig. 7.20 by an element name of the form `a[i][j]`, where `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`. Notice that the names of the elements in row 0 all have a first subscript of `0`; the names of the elements in column 3 all have a second subscript of `3`.

Common Programming Error 7.7

Referencing a two-dimensional array element `a[x][y]` *incorrectly as* `a[x, y]` *is an error. Actually,* `a[x, y]` *is treated as* `a[y]`*, because C++ evaluates the expression* `x, y` *(containing a comma operator) simply as* `y` *(the* last *of the comma-separated expressions).*

A multidimensional array can be initialized in its declaration much like a one-dimensional array. For example, a two-dimensional array `b` with values 1 and 2 in its row 0 elements and values 3 and 4 in its row 1 elements could be declared and initialized with

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

The values are grouped by row in braces. So, `1` and `2` initialize `b[0][0]` and `b[0][1]`, respectively, and `3` and `4` initialize `b[1][0]` and `b[1][1]`, respectively. If there are not enough initializers for a given row, the remaining elements of that row are initialized to `0`. Thus, the declaration

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

initializes `b[0][0]` to `1`, `b[0][1]` to `0`, `b[1][0]` to `3` and `b[1][1]` to `4`.

Figure 7.21 demonstrates initializing two-dimensional arrays in declarations. Lines 12–14 declare three arrays, each with two rows and three columns. The declaration of `array1` (line 12) provides six initializers in two sublists. The first sublist initializes row 0 of the array to the values 1, 2 and 3; and the second sublist initializes row 1 of the array to

```
1  // Fig. 7.21: fig07_21.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using namespace std;
5
6  void printArray( const int [][ 3 ] ); // prototype
7  const int rows = 2;
8  const int columns = 3;
9
10 int main()
11 {
12    int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13    int array2[ rows ][ columns ] = { 1, 2, 3, 4, 5 };
14    int array3[ rows ][ columns ] = { { 1, 2 }, { 4 } };
15
16    cout << "Values in array1 by row are:" << endl;
17    printArray( array1 );
18
19    cout << "\nValues in array2 by row are:" << endl;
20    printArray( array2 );
21
22    cout << "\nValues in array3 by row are:" << endl;
23    printArray( array3 );
24 } // end main
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ columns ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < rows; ++i )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < columns; ++j )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 1 of 2.)

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

Fig. 7.21 | Initializing multidimensional arrays. (Part 2 of 2.)

the values 4, 5 and 6. If the braces around each sublist are removed from the `array1` initializer list, the compiler initializes the elements of row 0 followed by the elements of row 1, yielding the same result.

The declaration of `array2` (line 13) provides only five initializers. The initializers are assigned to row 0, then row 1. Any elements that do not have an explicit initializer are initialized to zero, so `array2[1][2]` is initialized to zero.

The declaration of `array3` (line 14) provides three initializers in two sublists. The sublist for row 0 *explicitly* initializes the first two elements of row 0 to 1 and 2; the third element is *implicitly* initialized to zero. The sublist for row 1 *explicitly* initializes the first element to 4 and *implicitly* initializes the last two elements to zero.

The program calls function `printArray` to output each array's elements. Notice that the function prototype (line 6) and definition (lines 27–38) specify the parameter `const int a[][columns]`. When a function receives a one-dimensional array as an argument, the array brackets are empty in the function's parameter list. The size of a two-dimensional array's first dimension (i.e., the number of rows) is not required either, but all subsequent dimension sizes are required. The compiler uses these sizes to determine the locations in memory of elements in multidimensional arrays. All array elements are stored consecutively in memory, regardless of the number of dimensions. In a two-dimensional array, row 0 is stored in memory followed by row 1. Each row is a one-dimensional array. To locate an element in a particular row, the function must know exactly how many elements are in each row so it can skip the proper number of memory locations when accessing the array. Thus, when accessing `a[1][2]`, the function knows to skip row 0's three elements in memory to get to row 1. Then, the function accesses element 2 of that row.

Many common array manipulations use `for statements`. For example, the following `for` statement sets all the elements in row 2 of array `a` in Fig. 7.20 to zero:

```
for ( int column = 0; column < 4; ++column )
   a[ 2 ][ column ] = 0;
```

The `for` statement varies only the second subscript (i.e., the column subscript). The preceding `for` statement is equivalent to the following assignment statements:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following nested `for` statement determines the total of *all* the elements in array `a`:

```
total = 0;
for ( int row = 0; row < 3; ++row )
   for ( int column = 0; column < 4; ++column )
      total += a[ row ][ column ];
```

The `for` statement totals the elements of the array one row at a time. The outer `for` statement begins by setting `row` (i.e., the row subscript) to `0`, so the elements of row 0 may be totaled by the inner `for` statement. The outer `for` statement then increments `row` to `1`, so the elements of row 1 can be totaled. Then, the outer `for` statement increments `row` to `2`, so the elements of row 2 can be totaled. When the nested `for` statement terminates, `total` contains the sum of all the array elements.

7.10 Case Study: Class GradeBook Using a Two-Dimensional Array

In Section 7.6, we presented class `GradeBook` (Figs. 7.15–7.16), which used a one-dimensional array to store student grades on a single exam. In most semesters, students take several exams. Professors are likely to want to analyze grades across the entire semester, both for a single student and for the class as a whole.

Storing Student Grades in a Two-Dimensional Array in Class **GradeBook**

Figures 7.22–7.23 contain a version of class `GradeBook` that uses a two-dimensional array `grades` to store the grades of a number of students on multiple exams. Each row of the array represents a single student's grades for the entire course, and each column represents all the grades the students earned for one particular exam. A client program, such as Fig. 7.24, passes the array as an argument to the `GradeBook` constructor. In this example, we use a ten-by-three array containing ten students' grades on three exams.

```
// Fig. 7.22: GradeBook.h
// Definition of class GradeBook that uses a
// two-dimensional array to store test grades.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ Standard Library string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   // constants
   static const int students = 10; // number of students
   static const int tests = 3; // number of tests

   // constructor initializes course name and array of grades
   GradeBook( string, const int [][ tests ] );
```

Fig. 7.22 | Definition of class `GradeBook` that uses a two-dimensional array to store test grades. (Part 1 of 2.)

```
   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void processGrades(); // perform various operations on the grade data
   int getMinimum(); // find the minimum grade in the grade book
   int getMaximum(); // find the maximum grade in the grade book
   double getAverage( const int [], const int ); // get student's average
   void outputBarChart(); // output bar chart of grade distribution
   void outputGrades(); // output the contents of the grades array
private:
   string courseName; // course name for this grade book
   int grades[ students ][ tests ]; // two-dimensional array of grades
}; // end class GradeBook
```

Fig. 7.22 | Definition of class **GradeBook** that uses a two-dimensional array to store test grades. (Part 2 of 2.)

```
// Fig. 7.23: GradeBook.cpp
// Member-function definitions for class GradeBook that
// uses a two-dimensional array to store grades.
#include <iostream>
#include <iomanip> // parameterized stream manipulators
using namespace std;

// include definition of class GradeBook from GradeBook.h
#include "GradeBook.h"

// two-argument constructor initializes courseName and grades array
GradeBook::GradeBook( string name, const int gradesArray[][ tests ] )
{
   setCourseName( name ); // initialize courseName

   // copy grades from gradeArray to grades
   for ( int student = 0; student < students; ++student )

      for ( int test = 0; test < tests; ++test )
         grades[ student ][ test ] = gradesArray[ student ][ test ];
} // end two-argument GradeBook constructor

// function to set the course name
void GradeBook::setCourseName( string name )
{
   courseName = name; // store the course name
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
```

Fig. 7.23 | Member-function definitions for class **GradeBook** that uses a two-dimensional array to store grades. (Part 1 of 4.)

```
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!"
      << endl;
} // end function displayMessage

// perform various operations on the data
void GradeBook::processGrades()
{
   outputGrades(); // output grades array

   // call functions getMinimum and getMaximum
   cout << "\nLowest grade in the grade book is " << getMinimum()
      << "\nHighest grade in the grade book is " << getMaximum() << endl;

   outputBarChart(); // display distribution chart of grades on all tests
} // end function processGrades

// find minimum grade in the entire gradebook
int GradeBook::getMinimum()
{
   int lowGrade = 100; // assume lowest grade is 100

   // loop through rows of grades array
   for ( int student = 0; student < students; ++student )
   {
      // loop through columns of current row
      for ( int test = 0; test < tests; ++test )
      {
         // if current grade less than lowGrade, assign it to lowGrade
         if ( grades[ student ][ test ] < lowGrade )
            lowGrade = grades[ student ][ test ]; // new lowest grade
      } // end inner for
   } // end outer for

   return lowGrade; // return lowest grade
} // end function getMinimum

// find maximum grade in the entire gradebook
int GradeBook::getMaximum()
{
   int highGrade = 0; // assume highest grade is 0

   // loop through rows of grades array
   for ( int student = 0; student < students; ++student )
   {
```

Fig. 7.23 | Member-function definitions for class `GradeBook` that uses a two-dimensional array to store grades. (Part 2 of 4.)

```
      // loop through columns of current row
      for ( int test = 0; test < tests; ++test )
      {
         // if current grade greater than highGrade, assign to highGrade
         if ( grades[ student ][ test ] > highGrade )
            highGrade = grades[ student ][ test ]; // new highest grade
      } // end inner for
   } // end outer for

   return highGrade; // return highest grade
} // end function getMaximum

// determine average grade for particular set of grades
double GradeBook::getAverage( const int setOfGrades[], const int grades )
{
   int total = 0; // initialize total

   // sum grades in array
   for ( int grade = 0; grade < grades; ++grade )
      total += setOfGrades[ grade ];

   // return average of grades
   return static_cast< double >( total ) / grades;
} // end function getAverage

// output bar chart displaying grade distribution
void GradeBook::outputBarChart()
{
   cout << "\nOverall grade distribution:" << endl;

   // stores frequency of grades in each range of 10 grades
   const int frequencySize = 11;
   int frequency[ frequencySize ] = {}; // initialize elements to 0

   // for each grade, increment the appropriate frequency
   for ( int student = 0; student < students; ++student )

      for ( int test = 0; test < tests; ++test )
         ++frequency[ grades[ student ][ test ] / 10 ];

   // for each grade frequency, print bar in chart
   for ( int count = 0; count < frequencySize; ++count )
   {
      // output bar label ("0-9:", ..., "90-99:", "100:" )
      if ( count == 0 )
         cout << "  0-9: ";
      else if ( count == 10 )
         cout << "  100: ";
      else
         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";

```

Fig. 7.23 | Member-function definitions for class GradeBook that uses a two-dimensional array to store grades. (Part 3 of 4.)

```
135          // print bar of asterisks
136          for ( int stars = 0; stars < frequency[ count ]; ++stars )
137             cout << '*';
138
139          cout << endl; // start a new line of output
140       } // end outer for
141    } // end function outputBarChart
142
143    // output the contents of the grades array
144    void GradeBook::outputGrades()
145    {
146       cout << "\nThe grades are:\n\n";
147       cout << "            "; // align column heads
148
149       // create a column heading for each of the tests
150       for ( int test = 0; test < tests; ++test )
151          cout << "Test " << test + 1 << "  ";
152
153       cout << "Average" << endl; // student average column heading
154
155       // create rows/columns of text representing array grades
156       for ( int student = 0; student < students; ++student )
157       {
158          cout << "Student " << setw( 2 ) << student + 1;
159
160          // output student's grades
161          for ( int test = 0; test < tests; ++test )
162             cout << setw( 8 ) << grades[ student ][ test ];
163
164          // call member function getAverage to calculate student's average;
165          // pass row of grades and the value of tests as the arguments
166          double average = getAverage( grades[ student ], tests );
167          cout << setw( 9 ) << setprecision( 2 ) << fixed << average << endl;
168       } // end outer for
169    } // end function outputGrades
```

Fig. 7.23 | Member-function definitions for class `GradeBook` that uses a two-dimensional array to store grades. (Part 4 of 4.)

Five member functions (declared in lines 23–27 of Fig. 7.22) perform array manipulations to process the grades. Each of these member functions is similar to its counterpart in the earlier one-dimensional array version of class `GradeBook` (Figs. 7.15–7.16). Member function `getMinimum` (defined in lines 57–74 of Fig. 7.23) determines the lowest grade of all students for the semester. Member function `getMaximum` (defined in lines 77–94 of Fig. 7.23) determines the highest grade of all students for the semester. Member function `getAverage` (lines 97–107 of Fig. 7.23) determines a particular student's semester average. Member function `outputBarChart` (lines 110–141 of Fig. 7.23) outputs a bar chart of the distribution of all student grades for the semester. Member function `outputGrades` (lines 144–169 of Fig. 7.23) outputs the two-dimensional array in a tabular format, along with each student's semester average.

Member functions `getMinimum`, `getMaximum`, `outputBarChart` and `outputGrades` each loop through array `grades` by using nested `for` statements. For example, consider the nested `for` statement in member function `getMinimum` (lines 62–71). The outer `for` statement begins by setting `student` (i.e., the row subscript) to `0`, so the elements of row 0 can be compared with variable `lowGrade` in the body of the inner `for` statement. The inner `for` statement loops through the grades of a particular row and compares each grade with `lowGrade`. If a grade is less than `lowGrade`, `lowGrade` is set to that grade. The outer `for` statement then increments the row subscript to `1`. The elements of row 1 are compared with variable `lowGrade`. The outer `for` statement then increments the row subscript to `2`, and the elements of row 2 are compared with variable `lowGrade`. This repeats until all rows of `grades` have been traversed. When execution of the nested statement is complete, `lowGrade` contains the smallest grade in the two-dimensional array. Member function `getMaximum` works similarly to member function `getMinimum`.

Member function `outputBarChart` in Fig. 7.23 is nearly identical to the one in Fig. 7.16. However, to output the overall grade distribution for a whole semester, the function uses a nested `for` statement (lines 119–122) to create the one-dimensional array `frequency` based on all the grades in the two-dimensional array. The rest of the code in each of the two `outputBarChart` member functions that displays the chart is identical.

Member function `outputGrades` (lines 144–169) also uses nested `for` statements to output values of the array `grades`, in addition to each student's semester average. The output in Fig. 7.24 shows the result, which resembles the tabular format of a professor's physical grade book. Lines 150–151 print the column headings for each test. We use a counter-controlled `for` statement so that we can identify each test with a number. Similarly, the `for` statement in lines 156–168 first outputs a row label using a counter variable to identify each student (line 158). Although array indices start at 0, lines 151 and 158 output `test + 1` and `student + 1`, respectively, to produce test and student numbers starting at 1 (see Fig. 7.24). The inner `for` statement in lines 161–162 uses the outer `for` statement's counter variable `student` to loop through a specific row of array `grades` and output each student's test grade. Finally, line 166 obtains each student's semester average by passing the current row of `grades` (i.e., `grades[student]`) to member function `getAverage`.

Member function `getAverage` (lines 97–107) takes two arguments—a one-dimensional array of test results for a particular student and the number of test results in the array. When line 166 calls `getAverage`, the first argument is `grades[student]`, which specifies that a particular row of the two-dimensional array `grades` should be passed to `getAverage`. For example, based on the array created in Fig. 7.24, the argument `grades[1]` represents the three values (a one-dimensional array of grades) stored in row `1` of the two-dimensional array `grades`. A two-dimensional array can be considered an array whose elements are one-dimensional arrays. Member function `getAverage` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result as a `double` value (line 106).

Testing Class **GradeBook**

The program in Fig. 7.24 creates an object of class `GradeBook` (Figs. 7.22–7.23) using the two-dimensional array of `int`s named `gradesArray` (declared and initialized in lines 10–20). Line 10 accesses class `GradeBook`'s `static` constants `students` and `tests` to indicate the size of each dimension of array `gradesArray`. Lines 22–23 pass a course name and `gradesArray` to the `GradeBook` constructor. Lines 24–25 then invoke `myGradeBook`'s `displayMessage` and `processGrades` member functions to display a welcome message and obtain a report summarizing the students' grades for the semester, respectively.

```
// Fig. 7.24: fig07_24.cpp
// Creates GradeBook object using a two-dimensional array of grades.

#include "GradeBook.h" // GradeBook class definition

// function main begins program execution
int main()
{
   // two-dimensional array of student grades
   int gradesArray[ GradeBook::students ][ GradeBook::tests ] =
      { { 87, 96, 70 },
        { 68, 87, 90 },
        { 94, 100, 90 },
        { 100, 81, 82 },
        { 83, 65, 85 },
        { 78, 87, 65 },
        { 85, 75, 83 },
        { 91, 94, 100 },
        { 76, 72, 84 },
        { 87, 93, 73 } };

   GradeBook myGradeBook(
      "CS101 Introduction to C++ Programming", gradesArray );
   myGradeBook.displayMessage();
   myGradeBook.processGrades();
} // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
```

Fig. 7.24 | Creates a `GradeBook` object using a two-dimensional array of grades, then invokes member function `processGrades` to analyze them. (Part 1 of 2.)

```
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.24 | Creates a `GradeBook` object using a two-dimensional array of grades, then invokes member function `processGrades` to analyze them. (Part 2 of 2.)

7.11 Introduction to C++ Standard Library Class Template vector

We now introduce C++ Standard Library class template **`vector`**, which represents a more robust type of array featuring many additional capabilities. As you'll see in later chapters, C-style pointer-based arrays (i.e., the type of arrays presented thus far) have great potential for errors. For example, as mentioned earlier, a program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array. Two arrays cannot be meaningfully compared with equality operators or relational operators. As you'll learn in Chapter 8, pointer variables (known more commonly as pointers) contain memory addresses as their values. Array names are simply *pointers* to where the arrays begin in memory, and, of course, two arrays will always be at different memory locations. When an array is passed to a general-purpose function designed to handle arrays of any size, the size of the array must be passed as an additional argument. Furthermore, one array cannot be assigned to another with the assignment operator(s)—array names are `const` pointers, and, as you'll learn in Chapter 8, a constant pointer cannot be used on the left side of an assignment operator. These and other capabilities certainly seem like "naturals" for dealing with arrays, but C++ does not provide such capabilities. However, the C++ Standard Library provides class template `vector` to allow you to create a more powerful and less error-prone alternative to arrays. In Chapter 11, we present the means to implement such array capabilities as those provided by `vector`. You'll learn how to customize operators for use with your own classes (a technique known as *operator overloading*).

The `vector` class template is available to anyone building applications with C++. The notations that the `vector` example uses might be unfamiliar to you, because `vector`s use template notation. Recall that Section 6.18 discussed function templates. In Chapter 14, we discuss class templates. For now, you should feel comfortable using class template `vector` by mimicking the syntax in the example we show in this section. You'll deepen your understanding as we study class templates in Chapter 14. Chapter 22 presents class template `vector` (and several other standard C++ container classes) in detail.

The program of Fig. 7.25 demonstrates capabilities provided by C++ Standard Library class template `vector` that are not available for C-style pointer-based arrays. Standard class template `vector` provides many of the same features as the `Array` class that we construct in Chapter 11. Standard class template `vector` is defined in header `<vector>` (line 5) and belongs to namespace `std`. Chapter 22 discusses the full functionality of `vector`. At the end of this section, we'll demonstrate class vector's bounds checking capabilities and introduce C++'s exception-handling mechanism, which can be used to detect and handle an out-of-bounds `vector` index.

```cpp
// Fig. 7.25: fig07_25.cpp
// Demonstrating C++ Standard Library class template vector.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

void outputVector( const vector< int > & ); // display the vector
void inputVector( vector< int > & ); // input values into the vector

int main()
{
   vector< int > integers1( 7 ); // 7-element vector< int >
   vector< int > integers2( 10 ); // 10-element vector< int >

   // print integers1 size and contents
   cout << "Size of vector integers1 is " << integers1.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers1 );

   // print integers2 size and contents
   cout << "\nSize of vector integers2 is " << integers2.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers2 );

   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   inputVector( integers1 );
   inputVector( integers2 );

   cout << "\nAfter input, the vectors contain:\n"
      << "integers1:" << endl;
   outputVector( integers1 );
   cout << "integers2:" << endl;
   outputVector( integers2 );

   // use inequality (!=) operator with vector objects
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create vector integers3 using integers1 as an
   // initializer; print size and contents
   vector< int > integers3( integers1 ); // copy constructor

   cout << "\nSize of vector integers3 is " << integers3.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers3 );
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 1 of 4.)

```

   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:" << endl;
   integers1 = integers2; // assign integers2 to integers1

   cout << "integers1:" << endl;
   outputVector( integers1 );
   cout << "integers2:" << endl;
   outputVector( integers2 );

   // use equality (==) operator with vector objects
   cout << "\nEvaluating: integers1 == integers2" << endl;

   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal" << endl;

   // use square brackets to create rvalue
   cout << "\nintegers1[5] is " << integers1[ 5 ];

   // use square brackets to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]" << endl;
   integers1[ 5 ] = 1000;
   cout << "integers1:" << endl;
   outputVector( integers1 );

   // attempt to use out-of-range subscript
   try
   {
      cout << "\nAttempt to display integers1.at( 15 )" << endl;
      cout << integers1.at( 15 ) << endl; // ERROR: out of range
   } // end try
   catch ( out_of_range &ex )
   {
      cout << "An exception occurred: " << ex.what() << endl;
   } // end catch
} // end main

// output vector contents
void outputVector( const vector< int > &array )
{
   size_t i; // declare control variable

   for ( i = 0; i < array.size(); ++i )
   {
      cout << setw( 12 ) << array[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         cout << endl;
   } // end for

   if ( i % 4 != 0 )
      cout << endl;
} // end function outputVector
```

Fig. 7.25 | Demonstrating C++ Standard Library class template vector. (Part 2 of 4.)

```
// input vector contents
void inputVector( vector< int > &array )
{
   for ( size_t i = 0; i < array.size(); ++i )
      cin >> array[ i ];
} // end function inputVector
```

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 3 of 4.)

```
Assigning 1000 to integers1[5]
integers1:
          8           9          10          11
         12        1000          14          15
         16          17

Attempt to display integers1.at( 15 )
An exception occurred: invalid vector<T> subscript
```

Fig. 7.25 | Demonstrating C++ Standard Library class template `vector`. (Part 4 of 4.)

Creating `vector` *Objects*

Lines 13–14 create two `vector` objects that store values of type `int`—`integers1` contains seven elements, and `integers2` contains 10 elements. By default, all the elements of each `vector` object are set to `0`. Note that `vector`s can be defined to store *any* data type, by replacing `int` in `vector<int>` with the appropriate data type. This notation, which specifies the type stored in the `vector`, is similar to the template notation that Section 6.18 introduced with function templates.

`vector` *Member Function* `size`*; Function* `outputVector`

Line 17 uses `vector` member function **`size`** to obtain the size (i.e., the number of elements) of `integers1`. Line 19 passes `integers1` to function `outputVector` (lines 88–102), which uses square brackets, `[]` (line 94), to obtain the value in each element of the `vector` for output. Note the resemblance of this notation to that used to access the value of an array element. Lines 22 and 24 perform the same tasks for `integers2`.

Member function `size` of class template `vector` returns the number of elements in a `vector` as a value of type `size_t` (which represents the type `unsigned int` on many systems). As a result, line 90 declares the control variable `i` to be of type `size_t`, too. On some compilers, declaring `i` as an `int` causes the compiler to issue a warning message, since the loop-continuation condition (line 92) would compare a `signed` value (i.e., `int i`) and an `unsigned` value (i.e., a value of type `size_t` returned by function `size`).

Function `inputVector`

Lines 28–29 pass `integers1` and `integers2` to function `inputVector` (lines 105–109) to read values for each `vector`'s elements from the user. The function uses square brackets (`[]`) to form *lvalues* that are used to store the input values in each `vector` element.

Comparing `vector` *Objects for Inequality*

Line 40 demonstrates that `vector` objects can be compared with one another using the `!=` operator. If the contents of two `vector`s are not equal, the operator returns `true`; otherwise, it returns `false`.

Initializing One `vector` *with the Contents of Another*

The C++ Standard Library class template `vector` allows you to create a new `vector` object that is initialized with the contents of an existing `vector`. Line 45 creates a `vector` object `integers3` and initializes it with a copy of `integers1`. This invokes `vector`'s so-called copy constructor to perform the copy operation. You'll learn about copy constructors in detail in Chapter 11. Lines 47–49 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

Assigning `vector`s and Comparing `vector`s for Equality

Line 53 assigns `integers2` to `integers1`, demonstrating that the assignment (=) operator can be used with `vector` objects. Lines 55–58 output the contents of both objects to show that they now contain identical values. Line 63 then compares `integers1` to `integers2` with the equality (==) operator to determine whether the contents of the two objects are equal after the assignment in line 53 (which they are).

Using the `[]` Operator to Access and Modify `vector` Elements

Lines 67 and 71 use square brackets (`[]`) to obtain a `vector` element as an *rvalue* and as an *lvalue*, respectively. Recall from Section 5.9 that an *rvalue* cannot be modified, but an *lvalue* can. As is the case with C-style pointer-based arrays, *C++ does not perform any bounds checking when `vector` elements are accessed with square brackets*. Therefore, you must ensure that operations using `[]` do not accidentally attempt to manipulate elements outside the bounds of the `vector`. Standard class template `vector` does, however, provide bounds checking in its member function `at`, which we use at line 79 and discuss shortly.

Exception Handling: Processing an Out-of-Range Subscript

An **exception** indicates a problem that occurs while a program executes. The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule." **Exception handling** enables you to create **fault-tolerant programs** that can resolve (or handle) exceptions. In many cases, this allows a program to continue executing as if no problems were encountered. For example, Fig. 7.25 still runs to completion, even though an attempt was made to access an out-of-range subscript. More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate. When a function detects a problem, such as an invalid array subscript or an invalid argument, it **throws** an exception—that is, an exception occurs. Here we introduce exception handling briefly. We'll discuss it in detail in Chapter 16, Exception Handling: A Deeper Look.

The `try` Statement

To handle an exception, place any code that might throw an exception in a **`try` statement** (lines 76–84). The **`try` block** (lines 76–80) contains the code that might *throw* an exception, and the **`catch` block** (lines 81–84) contains the code that *handles* the exception if one occurs. You can have many `catch` blocks to handle different types of exceptions that might be thrown in the corresponding `try` block. If the code in the `try` block executes successfully, lines 81–84 are ignored. The braces that delimit `try` and `catch` blocks' bodies are required.

The `vector` member function **`at`** provides bounds checking and throws an exception if its argument is an invalid subscript. By default, this causes a C++ program to terminate. If the subscript is valid, function `at` returns the element at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue*, depending on the context in which the call appears. An unmodifiable *lvalue* is an expression that identifies an object in memory (such as an element in a `vector`), but cannot be used to modify that object.

Executing the `catch` Block

When the program calls `vector` member function `at` with the argument `15` (line 79), the function attempts to access the element at location 15, which is *outside* the `vector`'s bounds—`integers1` has only 10 elements at this point. Because bounds checking is performed at execution time, `vector` member function `at` generates an exception—specifically line 79 throws an **`out_of_range`** exception (from header `<stdexcept>`) to notify the program of this problem. At this point, the `try` block terminates immediately and the `catch` block begins executing—if you declared any variables in the `try` block, they're now out of scope and are not accessible in the `catch` block. [*Note:* To avoid compilation errors with GNU C++, you may need to include header `<stdexcept>` to use class `out_of_range`.]

The `catch` block declares a type (`out_of_range`) and an exception parameter (`ex`) that it receives as a reference. The `catch` block can handle exceptions of the specified type. Inside the block, you can use the parameter's identifier to interact with a caught exception object.

`what` *Member Function of the Exception Parameter*

When lines 81–84 *catch* the exception, the program displays a message indicating the problem that occurred. Line 83 calls the exception object's **`what`** member function to get the error message that is stored in the exception object and display it. Once the message is displayed in this example, the exception is considered handled and the program continues with the next statement after the `catch` block's closing brace. In this example, the end of the program is reached, so the program terminates. We use exception handling again in Chapters 9–13 and Chapter 16 presents a deeper look at exception handling.

Summary of This Example

In this section, we demonstrated the C++ Standard Library class template `vector`, a robust, reusable class that can replace C-style pointer-based arrays. In Chapter 11, you'll see that `vector` achieves many of its capabilities by "overloading" C++'s built-in operators, and you'll learn how to customize operators for use with your own classes in similar ways. For example, we create an `Array` class that, like class template `vector`, improves upon basic array capabilities. Our `Array` class also provides additional features, such as the ability to input and output entire arrays with operators >> and <<, respectively.

7.12 Wrap-Up

This chapter began our introduction to data structures, exploring the use of arrays and `vector`s to store data in and retrieve data from lists and tables of values. The chapter examples demonstrated how to declare an array, initialize an array and refer to individual elements of an array. We also illustrated how to pass arrays to functions and how to use the `const` qualifier to enforce the principle of least privilege. Chapter examples also presented basic searching and sorting techniques. You learned how to declare and manipulate multidimensional arrays. Finally, we demonstrated the capabilities of C++ Standard Library class template `vector`, which provides a more robust alternative to arrays. In that example, we also discussed basic exception-handling concepts.

We continue our coverage of data structures in Chapter 14, Templates, where we build a stack class template and in Chapter 20, Custom Templatized Data Structures, which introduces other dynamic data structures, such as lists, queues, stacks and trees, that can grow and shrink as programs execute. Chapter 22 introduces several of the C++ Standard Library's predefined data structures, which you can use instead of building their own. Chapter 22 presents the full functionality of class template `vector` and discusses many additional data structure classes, including `list` and `deque`, which are array-like data structures that can grow and shrink in response to a program's changing storage requirements.

We've now introduced the basic concepts of classes, objects, control statements, functions and arrays. In Chapter 8, we present one of C++'s most powerful features—the pointer. Pointers keep track of where data and functions are stored in memory, which allows us to manipulate those items in interesting ways. After introducing basic pointer concepts, we examine in detail the close relationship among arrays, pointers and strings.

Summary

Section 7.1 Introduction

- Data structures (p. 268) are collections of related data items. Arrays (p. 268) are data structures consisting of related data items of the same type. Arrays are "static" entities in that they remain the same size throughout program execution. (They may, of course, be of automatic storage class and hence be created and destroyed each time the blocks in which they're defined are entered and exited.)

Section 7.2 Arrays

- An array is a consecutive group of memory locations that share the same type.
- To refer to a particular location or element in an array, we specify the name of the array (p. 269) and the position number of the particular element in the array.
- A program refers to any one of an array's elements by giving the name of the array followed by the index (p. 269) of the particular element in square brackets (`[]`).
- The first element in every array has index zero (p. 269) and is sometimes called the zeroth element.
- An index must be an integer or integer expression (using any integral type).
- The brackets used to enclose the index are an operator with the same precedence as parentheses.

Section 7.3 Declaring Arrays

- Arrays occupy space in memory. You specify the type of each element and the number of elements required by an array as follows:

  type arrayName`[` *arraySize* `];`

  and the compiler reserves the appropriate amount of memory.
- Arrays can be declared to contain any data type. For example, an array of type `char` can be used to store a character string.

Section 7.4 Examples Using Arrays

- The elements of an array can be initialized in the array declaration by following the array name with an equals sign and an initializer list (p. 272)—a comma-separated list (enclosed in braces) of constant initializers (p. 272). When initializing an array with an initializer list, if there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list.
- If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size.
- Constants must be initialized with a constant expression when they're declared and cannot be modified thereafter. Constants can be placed anywhere a constant expression is expected.
- C++ has no array bounds checking (p. 280). You should ensure that all array references remain within the bounds of the array.

- A `static` local variable in a function definition exists for the duration of the program but is visible only in the function body.
- A program initializes `static` local arrays when their declarations are first encountered. If a `static` array is not initialized explicitly by you, each element of that array is initialized to zero by the compiler when the array is created.

Section 7.5 Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call.
- Arrays are passed to functions by reference—the called functions can modify the element values in the callers' original arrays. The value of the name of the array is the address in the computer's memory of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored in memory.
- Individual array elements are passed by value exactly as simple variables are.
- To receive an array argument, a function's parameter list must specify that the function expects to receive an array. The size of the array is not required between the array brackets.
- The type qualifier `const` (p. 273) can be used to prevent modification of array values in the caller by code in a called function. When an array parameter is preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error.

*Section 7.6 Case Study: Class **GradeBook** Using an Array to Store Grades*

- Class variables (`static` data members; p. 290) are shared by all objects of the class in which the variables are declared.
- A `static` data member can be accessed within the class definition and the member-function definitions like any other data member.
- A `public static` data member can also be accessed outside of the class, even when no objects of the class exist, using the class name followed by the scope resolution operator (`::`) and the name of the data member.

Section 7.7 Searching Arrays with Linear Search

- The linear search (p. 293) compares each array element with a search key (p. 293). Because the array is not in any particular order, it's just as likely that the value will be found in the first element as the last. On average, a program must compare the search key with half the array elements. To determine that a value is not in the array, the program must compare the search key to every element in the array.

Section 7.8 Sorting Arrays with Insertion Sort

- An array can be sorted using insertion sort (p. 295). The first iteration of this algorithm takes the second element and, if it's less than the first element, swaps it with the first element (i.e., the program *inserts* the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the i^{th} iteration of this algorithm, the first i elements in the original array will be sorted. For small arrays, the insertion sort is acceptable, but for larger arrays it's inefficient compared to other more sophisticated sorting algorithms.

Section 7.9 Multidimensional Arrays

- Multidimensional arrays (p. 297) with two dimensions are often used to represent tables of values (p. 297) consisting of information arranged in rows and columns.

- Arrays that require two subscripts to identify a particular element are called two-dimensional arrays (p. 297). An array with *m* rows and *n* columns is called an *m*-by-*n* array (p. 297).

Section 7.11 Introduction to C++ Standard Library Class Template `vector`

- C++ Standard Library class template `vector` (p. 307) represents a more robust alternative to arrays featuring many capabilities that are not provided for C-style pointer-based arrays.
- By default, all the elements of an integer `vector` object are set to 0.
- A `vector` can be defined to store any data type using a declaration of the form

  `vector<` *type* `>` *name*`(` *size* `);`

- Member function `size` (p. 311) of class template `vector` returns the number of elements in the `vector` on which it's invoked.
- The value of an element of a `vector` can be accessed or modified using square brackets (`[]`).
- Objects of standard class template `vector` can be compared directly with the equality (`==`) and inequality (`!=`) operators. The assignment (`=`) operator can also be used with `vector` objects.
- An unmodifiable *lvalue* is an expression that identifies an object in memory (such as an element in a vector), but cannot be used to modify that object. A modifiable *lvalue* also identifies an object in memory, but can be used to modify the object.
- An exception (p. 312) indicates a problem that occurs while a program executes. The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule."
- Exception handling (p. 312) enables you to create fault-tolerant programs (p. 312) that can resolve exceptions.
- To handle an exception, place any code that might throw an exception (p. 312) in a `try` statement.
- The `try` block (p. 312) contains the code that might throw an exception, and the `catch` block (p. 312) contains the code that handles the exception if one occurs.
- When a `try` block terminates any variables declared in the `try` block go out of scope.
- A `catch` block (p. 312) declares a type and an exception parameter. Inside the `catch` block, you can use the parameter's identifier to interact with a caught exception object.
- An exception object's `what` method (p. 313) returns the exception's error message.

Self-Review Exercises

7.1 *(Fill in the Blanks)* Answer each of the following:

a) Lists and tables of values can be stored in ________ or ________.
b) The elements of an array are related by the fact that they have the same ________ and ________.
c) The number used to refer to a particular element of an array is called its ________.
d) A(n) ________ should be used to declare the size of an array, because it makes the program more scalable.
e) The process of placing the elements of an array in order is called ________ the array.
f) The process of determining if an array contains a particular key value is called ________ the array.
g) An array that uses two subscripts is referred to as a(n) ________ array.

7.2 *(True or False)* State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) An array can store many different types of values.
b) An array subscript should normally be of data type `float`.

c) If there are fewer initializers in an initializer list than the number of elements in the array, the remaining elements are initialized to the last value in the initializer list.
d) It's an error if an initializer list has more initializers than there are elements in the array.
e) An individual array element that is passed to a function and modified in that function will contain the modified value when the called function completes execution.

7.3 *(Write C++ Statements)* Write one or more statements that perform the following tasks for an array called `fractions`:
a) Define a constant integer variable `arraySize` initialized to `10`.
b) Declare an array with `arraySize` elements of type `double`, and initialize the elements to `0`.
c) Name the fourth element of the array.
d) Refer to array element 4.
e) Assign the value `1.667` to array element 9.
f) Assign the value `3.333` to the seventh element of the array.
g) Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that is actually displayed on the screen.
h) Print all the array elements using a `for` statement. Define the integer variable `i` as a control variable for the loop. Show the output.

7.4 *(Double Array Questions)* Answer the following questions regarding an array called `table`:
a) Declare the array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable `arraySize` has been defined to be 3.
b) How many elements does the array contain?
c) Use a `for` statement to initialize each element of the array to the sum of its subscripts. Assume that the integer variables `i` and `j` are declared as control variables.
d) Write a program segment to print the values of each element of array `table` in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration

```
int table[ arraySize ][ arraySize ] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

and the integer variables `i` and `j` are declared as control variables. Show the output.

7.5 *(Find the Error)* Find the error in each of the following program segments and correct the error:
a) `#include <iostream>;`
b) `arraySize = 10; // arraySize was declared const`
c) Assume that `int b[ 10 ] = {};`

```
for ( int i = 0; i <= 10; ++i )
    b[ i ] = 1;
```

d) Assume that `int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`

```
a[ 1, 1 ] = 5;
```

Answers to Self-Review Exercises

7.1 a) arrays, `vector`s. b) array name, type. c) subscript or index. d) constant variable. e) sorting. f) searching. g) two-dimensional.

7.2 a) False. An array can store only values of the same type.
b) False. An array subscript should be an integer or an integer expression.
c) False. The remaining elements are initialized to zero.
d) True.
e) False. Individual elements of an array are passed by value. If the entire array is passed to a function, then any modifications to the elements will be reflected in the original.

7.3 a) `const int arraySize = 10;`
b) `double fractions[ arraySize ] = { 0.0 };`
c) `fractions[ 3 ]`
d) `fractions[ 4 ]`
e) `fractions[ 9 ] = 1.667;`
f) `fractions[ 6 ] = 3.333;`
g)
```
cout << fixed << setprecision( 2 );
cout << fractions[ 6 ] << ' ' << fractions[ 9 ] << endl;
```
Output: `3.33 1.67`.
h)
```
for ( int i = 0; i < arraySize; ++i )
   cout << "fractions[" << i << "] = " << fractions[ i ] << endl;
```
Output:
```
fractions[ 0 ] = 0.0
fractions[ 1 ] = 0.0
fractions[ 2 ] = 0.0
fractions[ 3 ] = 0.0
fractions[ 4 ] = 0.0
fractions[ 5 ] = 0.0
fractions[ 6 ] = 3.333
fractions[ 7 ] = 0.0
fractions[ 8 ] = 0.0
fractions[ 9 ] = 1.667
```

7.4 a) `int table[ arraySize ][ arraySize ];`
b) Nine.
c)
```
for ( i = 0; i < arraySize; ++i )
   for ( j = 0; j < arraySize; ++j )
      table[ i ][ j ] = i + j;
```
d)
```
cout << "     [0]  [1]  [2]" << endl;

for ( int i = 0; i < arraySize; ++i ) {
   cout << '[' << i << "] ";

   for ( int j = 0; j < arraySize; ++j )
      cout << setw( 3 ) << table[ i ][ j ] << "  ";
   cout << endl;
}
```
Output:
```
     [0]  [1]  [2]
[0]   1    8    0
[1]   2    4    6
[2]   5    0    0
```

7.5 a) *Error:* Semicolon at end of `#include` preprocessor directive.
Correction: Eliminate semicolon.
b) *Error:* Assigning a value to a constant variable using an assignment statement.
Correction: Initialize the constant variable in a `const int arraySize` declaration.
c) *Error:* Referencing an array element outside the bounds of the array (`b[10]`).
Correction: Change the final value of the control variable to `9` or change `<=` to `<`.
d) *Error:* Array subscripting done incorrectly.
Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`

Exercises

7.6 *(Fill in the Blanks)* Fill in the blanks in each of the following:

a) The names of the four elements of array `p` (`int p[4];`) are ________, ________, ________ and ________.

b) Naming an array, stating its type and specifying the number of elements in the array is called ________ the array.

c) By convention, the first subscript in a two-dimensional array identifies an element's ________ and the second subscript identifies an element's ________.

d) An *m*-by-*n* array contains ________ rows, ________ columns and ________ elements.

e) The name of the element in row 3 and column 5 of array `d` is ________.

7.7 *(True or False)* Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.

b) An array definition reserves space for an array.

c) To indicate reserve 100 locations for integer array `p`, you write the declaration

```
p[ 100 ];
```

d) A `for` statement must be used to initialize the elements of a 15-element array to zero.

e) Nested `for` statements must be used to total the elements of a two-dimensional array.

7.8 *(Write C++ Statements)* Write C++ statements to accomplish each of the following:

a) Display the value of element 6 of character array `f`.

b) Input a value into element 4 of one-dimensional floating-point array `b`.

c) Initialize each of the 5 elements of one-dimensional integer array `g` to `8`.

d) Total and print the elements of floating-point array `c` of 100 elements.

e) Copy array `a` into the first portion of array `b`. Assume `double a[ 11 ], b[ 34 ];`

f) Determine and print the smallest and largest values contained in 99-element floating-point array `w`.

7.9 *(Double Array Questions)* Consider a 2-by-3 integer array `t`.

a) Write a declaration for `t`.

b) How many rows does `t` have?

c) How many columns does `t` have?

d) How many elements does `t` have?

e) Write the names of all the elements in row 1 of `t`.

f) Write the names of all the elements in column 2 of `t`.

g) Write a statement that sets the element of `t` in the first row and second column to zero.

h) Write a series of statements that initialize each element of `t` to zero. Do not use a loop.

i) Write a nested `for` statement that initializes each element of `t` to zero.

j) Write a statement that inputs the values for the elements of `t` from the keyboard.

k) Write a series of statements that determine and print the smallest value in array `t`.

l) Write a statement that displays the elements in row 0 of `t`.

m) Write a statement that totals the elements in column 3 of `t`.

n) Write a series of statements that prints the array `t` in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

7.10 *(Salesperson Salary Ranges)* Use a one-dimensional array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople each receive \$200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses \$5000 in sales in a week receives \$200 plus 9 percent of \$5000, or a total of \$650. Write a program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a) $200–299
b) $300–399
c) $400–499
d) $500–599
e) $600–699
f) $700–799
g) $800–899
h) $900–999
i) $1000 and over

7.11 *(Bubble Sort)* In the **bubble sort algorithm**, smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom. The bubble sort makes several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array. The comparisons on each pass proceed as follows—the 0th element value is compared to the 1st, the 1st is compared to the 2nd, the 2nd is compared to the third, ..., the second-to-last element is compared to the last element. Write a program that sorts an array of 10 integers using bubble sort.

7.12 *(Bubble Sort Enhancements)* The bubble sort described in Exercise 7.11 is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort:

a) After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are “in place,” and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass, and so on.
b) The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass if any swaps have been made. If none have been made, then the data must already be in the proper order, so the program should terminate. If swaps have been made, then at least one more pass is needed.

7.13 *(Single Array Questions)* Write single statements that perform the following one-dimensional array operations:

a) Initialize the 10 elements of integer array `counts` to zero.
b) Add 1 to each of the 15 elements of integer array `bonus`.
c) Read 12 values for `double` array `monthlyTemperatures` from the keyboard.
d) Print the 5 values of integer array `bestScores` in column format.

7.14 *(Find the Errors)* Find the error(s) in each of the following statements:

a) Assume that: `int a[ 3 ];`

```
cout << a[ 1 ] << " " << a[ 2 ] << " " << a[ 3 ] << endl;
```

b) `double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`
c) Assume that: `double d[ 2 ][ 10 ];`

```
d[ 1, 9 ] = 2.345;
```

7.15 *(Duplicate Elimination)* Use a one-dimensional array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, validate it and store it in the array only if it isn’t a duplicate of a number already read. After reading all the values, display only the unique values that the user entered. Provide for the “worst case” in which all 20 numbers are different. Use the smallest possible array to solve this problem.

7.16 *(Double Array Initialization)* Label the elements of a 3-by-5 two-dimensional array `sales` to indicate the order in which they’re set to zero by the following program segment:

```
for ( row = 0; row < 3; ++row )

   for ( column = 0; column < 5; ++column )
      sales[ row ][ column ] = 0;
```

7.17 *(Dice Rolling)* Write a program that simulates the rolling of two dice. The program should use `rand` to roll the first die and should use `rand` again to roll the second die. The sum of the two values should then be calculated. [*Note:* Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] Figure 7.26 shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7).

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.26 | The 36 possible outcomes of rolling two dice.

7.18 *(What Does This Code Do?)* What does the following program do?

```
// Ex. 7.18: Ex07_18.cpp
// What does this program do?
#include <iostream>
using namespace std;

int whatIsThis( int [], int ); // function prototype

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int result = whatIsThis( a, arraySize );

   cout << "Result is " << result << endl;
} // end main

// What does this function do?
int whatIsThis( int b[], int size )
{
   if ( size == 1 ) // base case
      return b[ 0 ];
   else // recursive step
      return b[ size - 1 ] + whatIsThis( b, size - 1 );
} // end function whatIsThis
```

7.19 *(Craps Game Modification)* Modify the program of Fig. 6.11 to play 1000 games of craps. The program should keep track of the statistics and answer the following questions:

a) How many games are won on the 1^{st} roll, 2^{nd} roll, …, 20^{th} roll, and after the 20^{th} roll?
b) How many games are lost on the 1^{st} roll, 2^{nd} roll, …, 20^{th} roll, and after the 20^{th} roll?
c) What are the chances of winning at craps? [*Note:* You should discover that craps is one of the fairest casino games. What do you suppose this means?]
d) What's the average length of a game of craps?
e) Do the chances of winning improve with the length of the game?

7.20 (*Airline Reservations System*) A small airline has just purchased a computer for its new automated reservations system. You've been asked to program the new system. You are to write a program to assign seats on each flight of the airline's only plane (capacity: 10 seats).

Your program should display the following menu of alternatives—`Please type 1 for "First Class"` and `Please type 2 for "Economy"`. If the person types `1`, your program should assign a seat in the first class section (seats 1–5). If the person types `2`, your program should assign a seat in the economy section (seats 6–10). Your program should print a boarding pass indicating the person's seat number and whether it's in the first class or economy section of the plane.

Use a one-dimensional array to represent the seating chart of the plane. Initialize all the elements of the array to `false` to indicate that all seats are empty. As each seat is assigned, set the corresponding elements of the array to `true` to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first class section is full, your program should ask the person if it's acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message `"Next flight leaves in 3 hours."`

7.21 *(What Does This Code Do?)* What does the following program do?

```cpp
// Ex. 7.21: Ex07_21.cpp
// What does this program do?
#include <iostream>
using namespace std;

void someFunction( int [], int, int ); // function prototype

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   cout << "The values in the array are:" << endl;
   someFunction( a, 0, arraySize );
   cout << endl;
} // end main

// What does this function do?
void someFunction( int b[], int current, int size )
{
   if ( current < size )
   {
      someFunction( b, current + 1, size );
      cout << b[ current ] << "  ";
   } // end if
} // end function someFunction
```

7.22 *(Sales Summary)* Use a two-dimensional array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains the following:

a) The salesperson number
b) The product number
c) The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that will read all this information for last month's sales (one salesperson's data at a time) and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array `sales`. After processing all the information for last month, print the results in tabular format with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

7.23 (*Turtle Graphics*) The Logo language, which is popular among elementary school children, made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a C++ program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem, you'll simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that is initialized to `false`. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor with its pen up. The set of turtle commands your program must process are shown in Fig. 7.27.

Command	Meaning
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5,10	Move forward 10 spaces (or a number other than 10)
6	Print the 20-by-20 array
9	End of data (sentinel)

Fig. 7.27 | Turtle graphics commands.

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square and end with the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
```

```
5,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to `true`. When the `6` command (print) is given, wherever there is a `true` in the array, display an asterisk or some other character you choose. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

7.24 (***Knight's Tour***) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth in this exercise.

The knight makes L-shaped moves (over two in one direction then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 7.28.

a) Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a `1` in the first square you move to, a `2` in the second square, a `3` in the third, etc. Before starting the tour, estimate how far you think you'll get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

b) Now let's develop a program that will move the knight around a chessboard. The board is represented by an 8-by-8 two-dimensional array `board`. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0, as shown in Fig. 7.28, consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

```
horizontal[ 0 ] = 2      vertical[ 0 ] = -1
horizontal[ 1 ] = 1      vertical[ 1 ] = -2
horizontal[ 2 ] = -1     vertical[ 2 ] = -2
horizontal[ 3 ] = -2     vertical[ 3 ] = -1
horizontal[ 4 ] = -2     vertical[ 4 ] = 1
horizontal[ 5 ] = -1     vertical[ 5 ] = 2
horizontal[ 6 ] = 1      vertical[ 6 ] = 2
horizontal[ 7 ] = 2      vertical[ 7 ] = 1
```

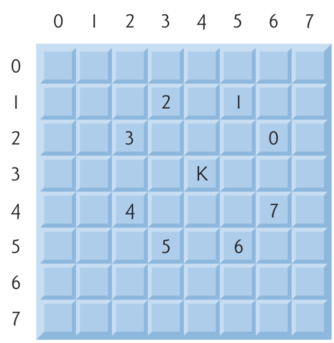


Fig. 7.28 | The eight possible moves of the knight.

Let the variables `currentRow` and `currentColumn` indicate the row and column of the knight's current position. To make a move of type `moveNumber`, where `moveNumber` is between 0 and 7, your program uses the statements

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Keep a counter that varies from `1` to `64`. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square, and, of course, test every potential move to make sure that the knight does not land off the chessboard. Now write a program to move the knight around the chessboard. Run the program. How many moves did the knight make?

c) After attempting to write and run a Knight's Tour program, you've probably developed some valuable insights. We'll use these to develop a **heuristic** (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so when the board gets congested near the end of the tour, there will be a greater chance of success.

We may develop an "accessibility heuristic" by classifying each square according to how accessible it's then always moving the knight to the square (within the knight's L-shaped moves, of course) that is most inaccessible. We label a two-dimensional array `accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each center square is rated as `8`, each corner square is rated as `2` and the other squares have accessibility numbers of `3`, `4` or `6` as follows:

```
2  3  4  4  4  4  3  2
3  4  6  6  6  6  4  3
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
4  6  8  8  8  8  6  4
3  4  6  6  6  6  4  3
2  3  4  4  4  4  3  2
```

Now write a version of the Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [*Note:* As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

d) Write a version of the Knight's Tour program which, when encountering a tie between two or more squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

7.25 (***Knight's Tour: Brute Force Approaches***) In Exercise 7.24, we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue increasing in power, we'll be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. This is the "brute force" approach to problem solving.

a) Use random number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b) Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c) Most likely, the preceding program gave you some "respectable" tours, but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. [*Caution:* This version of the program could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length, and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d) Compare the brute force version of the Knight's Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute force problem solving in general.

7.26 (***Eight Queens***) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is "attacking" any other, i.e., no two queens are in the same row, the same column, or along the same diagonal? Use the thinking developed in Exercise 7.24 to formulate a heuristic for solving the Eight Queens problem. Run your program. [*Hint:* It's possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as in Fig. 7.29. Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?]

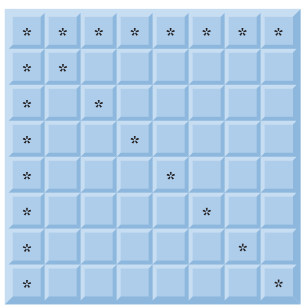


Fig. 7.29 | The 22 squares eliminated by placing a queen in the upper-left corner.

7.27 (***Eight Queens: Brute Force Approaches***) In this exercise, you'll develop several brute-force approaches to solving the Eight Queens problem introduced in Exercise 7.26.

a) Solve the Eight Queens exercise, using the random brute force technique developed in Exercise 7.25.

b) Use an exhaustive technique, i.e., try all possible combinations of eight queens on the chessboard.
c) Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight's Tour problem?
d) Compare and contrast the random brute force and exhaustive brute force approaches in general.

7.28 (***Knight's Tour: Closed-Tour Test***) In the Knight's Tour, a full tour occurs when the knight makes 64 moves touching each square of the chess board once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight's Tour program you wrote in Exercise 7.24 to test for a closed tour if a full tour has occurred.

7.29 (***The Sieve of Eratosthenes***) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:
a) Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain 1. All other array elements will eventually be set to zero. You'll ignore elements 0 and 1 in this exercise.
b) Starting with array subscript 2, every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to one indicate that the subscript is a prime number. These subscripts can then be printed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 2 and 999. Ignore element 0 of the array.

7.30 (***Bucket Sort***) A **bucket sort** begins with a one-dimensional array of positive integers to be sorted and a two-dimensional array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to $n-1$, where n is the number of values in the array to be sorted. Each row of the two-dimensional array is referred to as a bucket. Write a function `bucketSort` that takes an integer array and the array size as arguments and performs as follows:
a) Place each value of the one-dimensional array into a row of the bucket array based on the value's ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This is called a "distribution pass."
b) Loop through the bucket array row by row, and copy the values back to the original array. This is called a "gathering pass." The new order of the preceding values in the one-dimensional array is 100, 3 and 97.
c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 is placed in row zero and 97 is placed in row zero (after the 3). After the last gathering pass, the original array is now in sorted order.

Note that the two-dimensional array of buckets is 10 times the size of the integer array being sorted. This sorting technique provides better performance than an insertion sort, but requires much more memory. The insertion sort requires space for only one additional element of data. This is an example of the space-time trade-off: The bucket sort uses more memory than the insertion sort, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly swap the data between the two bucket arrays.

Recursion Exercises

7.31 (*Selection Sort*) A **selection sort** searches an array looking for the smallest element. Then, the smallest element is swapped with the first element of the array. The process is repeated for the subarray beginning with the second element of the array. Each pass of the array results in one element being placed in its proper location. This sort performs comparably to the insertion sort—for an array of n elements, $n - 1$ passes must be made, and for each subarray, $n - 1$ comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write recursive function `selectionSort` to perform this algorithm.

7.32 (*Palindromes*) A palindrome is a string that is spelled the same way forward and backward. Examples of palindromes include "radar" and "able was i ere i saw elba." Write a recursive function `testPalindrome` that returns `true` if a `string` is a palindrome, and `false` otherwise. Note that like an array, the square brackets (`[]`) operator can be used to iterate through the characters in a `string`.

7.33 (*Linear Search)* Modify the program in Fig. 7.18 to use recursive function `linearSearch` to perform a linear search of the array. The function should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return –1.

7.34 (*Eight Queens)* Modify the Eight Queens program you created in Exercise 7.26 to solve the problem recursively.

7.35 (*Print an Array*) Write a recursive function `printArray` that takes an array, a starting subscript and an ending subscript as arguments, returns nothing and prints the array. The function should stop processing and return when the starting subscript equals the ending subscript.

7.36 (*Print a String Backward*) Write a recursive function `stringReverse` that takes a `string` and a starting subscript as arguments, prints the string backward and returns nothing. The function should stop processing and return when the end of the string is encountered. Note that like an array the square brackets (`[]`) operator can be used to iterate through the characters in a `string`.

7.37 (*Find the Minimum Value in an Array*) Write a recursive function `recursiveMinimum` that takes an integer array, a starting subscript and an ending subscript as arguments, and returns the smallest element of the array. The function should stop processing and return when the starting subscript equals the ending subscript.

vector Exercises

7.38 *(Salesperson Salary Ranges with* **`vector`***)* Use a `vector` of integers to solve the problem described in Exercise 7.10.

7.39 *(Dice Rolling with* **`vector`***)* Modify the dice-rolling program you created in Exercise 7.17 to use a `vector` to store the numbers of times each possible sum of the two dice appears.

7.40 (*Find the Minimum Value in a* **`vector`**) Modify your solution to Exercise 7.37 to find the minimum value in a `vector` instead of an array.

Making a Difference

7.41 *(Polling)* The Internet and the web are enabling more people to network, join a cause, voice opinions, and so on. The presidential candidates in 2008 used the Internet intensively to get out their messages and raise money for their campaigns. In this exercise, you'll write a simple polling program that allows users to rate five social-consciousness issues from 1 (least important) to 10 (most important). Pick five causes that are important to you (e.g., political issues, global environmental issues). Use a one-dimensional array `topics` (of type `string`) to store the five causes. To summarize the survey responses, use a 5-row, 10-column two-dimensional array `responses` (of type `int`), each row corresponding to an element in the `topics` array. When the program runs, it should ask the user to rate each issue. Have your friends and family respond to the survey. Then have the program display a summary of the results, including:

a) A tabular report with the five topics down the left side and the 10 ratings across the top, listing in each column the number of ratings received for each topic.
b) To the right of each row, show the average of the ratings for that issue.
c) Which issue received the highest point total? Display both the issue and the point total.
d) Which issue received the lowest point total? Display both the issue and the point total.

8 Pointers

Addresses are given to us to conceal our whereabouts.
—Saki (H. H. Munro)

By indirection find direction out.
—William Shakespeare

Many things, having full reference
To one consent, may work contrariously.
—William Shakespeare

You will find it a very good practice always to verify your references, sir!
—Dr. Routh

Objectives

In this chapter you'll learn:

- What pointers are.
- The similarities and differences between pointers and references, and when to use each.
- To use pointers to pass arguments to functions by reference.
- The close relationships between pointers and arrays.
- To use arrays of pointers.
- Basic pointer-based string processing.
- To use pointers to functions.

Outline

8.1 Introduction

This chapter discusses one of the most powerful features of the C++ programming language, the pointer. In Chapter 6, we saw that references can be used to perform pass-by-reference. Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures that can grow and shrink, such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts and reinforces the intimate relationship among arrays and pointers. The view of arrays as pointers derives from the C programming language. As we saw in Chapter 7, the C++ Standard Library class `vector` provides an implementation of arrays as full-fledged objects.

Similarly, C++ actually offers two types of strings—`string` class objects (which we've been using since Chapter 3) and C-style, pointer-based strings. This chapter on pointers briefly introduces C strings to deepen your knowledge of pointers. C strings are widely used in legacy C and C++ systems. We discuss C strings in depth in Chapter 21. In new software development projects, you should favor `string` class objects.

We'll examine the use of pointers with classes in Chapter 13, Object-Oriented Programming: Polymorphism, where we'll see that the so-called "polymorphic processing" associated with object-oriented programming is performed with pointers and references. Chapter 20, Custom Templatized Data Structures, presents examples of creating and using dynamic data structures that are implemented with pointers.

8.2 Pointer Variable Declarations and Initialization

Pointer variables contain *memory addresses* as their values. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value** (Fig. 8.1). Referencing a value through a pointer is called **indirection**. Diagrams typically represent a pointer as an arrow from the *variable that contains an address* to the *variable located at that address* in memory.

Pointers, like any other variables, *must* be declared before they can be used. For example, for the pointer in Fig. 8.1, the declaration

```
int *countPtr, count;
```

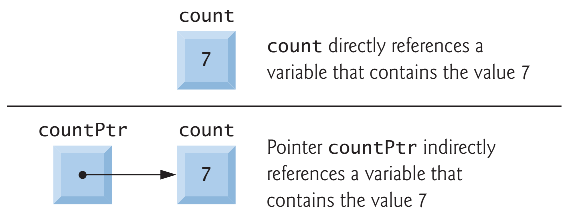


Fig. 8.1 | Directly and indirectly referencing a variable.

declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read (right to left), "`countPtr` is a pointer to `int`." Also, variable `count` in the preceding declaration is declared to be an `int`, *not* a pointer to an `int`. The `*` in the declaration applies only to `countPtr`. Each variable being declared as a pointer *must* be preceded by an asterisk (`*`). For example, the declaration

```
double *xPtr, *yPtr;
```

indicates that both `xPtr` and `yPtr` are pointers to `double` values. When `*` appears in a declaration, it isn't an operator; rather, it indicates that the variable being declared is a pointer. Pointers can be declared to point to objects of any data type.

Common Programming Error 8.1

Assuming that the `*` used to declare a pointer distributes to all names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the `*` prefixed to the name (with or without spaces in between). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.

Good Programming Practice 8.1

Although it isn't a requirement, including the letters `Ptr` in a pointer variable name makes it clear that the variable is a pointer and that it must be handled accordingly.

Pointers should be initialized to `0`, `NULL` or an address of the corresponding type either when they're declared or in an assignment. A pointer with the value `0` or `NULL` "points to nothing" and is known as a **null pointer**. Symbolic constant `NULL` is defined in several standard library headers to represent the value `0`. Initializing a pointer to `NULL` is equivalent to initializing a pointer to `0`, but in C++, `0` is used by convention. When `0` is assigned, it's converted to a pointer of the appropriate type. The value `0` is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type. [*Note:* In the new standard, you should use the constant **`nullptr`** to initialize a pointer instead of `0` or `NULL`. Several C++ compilers already implement this constant.]

Error-Prevention Tip 8.1

Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

8.3 Pointer Operators

The **address operator (`&`)** is a unary operator that *obtains the memory address of its operand*. For example, assuming the declarations

```
int y = 5; // declare variable y
int *yPtr; // declare pointer variable yPtr
```

the statement

```
yPtr = &y; // assign address of y to yPtr
```

assigns the address of the variable `y` to pointer variable `yPtr`. Then variable `yPtr` is said to "point to" `y`. Now, `yPtr` *indirectly* references variable `y`'s value. The use of the `&` in the preceding statement is *not* the same as the use of the `&` in a reference variable declaration, which is always preceded by a data-type name. When declaring a reference, the `&` is part of the type. In an expression like `&y`, the `&` is the address operator.

Figure 8.2 shows a schematic representation of memory after the preceding assignment. The "pointing relationship" is indicated by drawing an arrow from the box that represents the pointer `yPtr` in memory to the box that represents the variable `y` in memory.

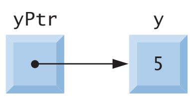


Fig. 8.2 | Graphical representation of a pointer pointing to a variable in memory.

Figure 8.3 shows another pointer representation in memory with integer variable `y` stored at memory location `600000` and pointer variable `yPtr` stored at memory location `500000`. The operand of the address operator must be an *lvalue*; the address operator cannot be applied to constants or to expressions that do not result in references.

Fig. 8.3 | Representation of `y` and `yPtr` in memory.

The **`*` operator**, commonly referred to as the **indirection operator** or **dereferencing operator**, *returns a synonym (i.e., an alias or a nickname) for the object to which its pointer operand points*. For example (referring again to Fig. 8.2), the statement

```
cout << *yPtr << endl;
```

prints the value of variable `y`, namely, `5`, just as the statement

```
cout << y << endl;
```

would. Using `*` in this manner is called **dereferencing a pointer**. A dereferenced pointer may also be used on the *left* side of an assignment statement, as in

```
*yPtr = 9;
```

which would assign `9` to `y` in Fig. 8.3. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

which places the input value in `y`. The dereferenced pointer is an *lvalue*.

Common Programming Error 8.2

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.

Common Programming Error 8.4

Dereferencing a null pointer is often a fatal execution-time error.

The program in Fig. 8.4 demonstrates the `&` and `*` pointer operators. Memory locations are output by `<<` in this example as *hexadecimal* (i.e., base-16) integers. (See Appendix D, Number Systems, for more information on hexadecimal integers.) The hexadecimal memory addresses output by this program are compiler and operating-system dependent, so you may get different results when you run the program.

```cpp
// Fig. 8.4: fig08_04.cpp
// Pointer operators & and *.
#include <iostream>
using namespace std;

int main()
{
   int a; // a is an integer
   int *aPtr; // aPtr is an int * which is a pointer to an integer

   a = 7; // assigned 7 to a
   aPtr = &a; // assign the address of a to aPtr

   cout << "The address of a is " << &a
      << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is " << a
      << "\nThe value of *aPtr is " << *aPtr;
   cout << "\n\nShowing that * and & are inverses of "
      << "each other.\n&*aPtr = " << &*aPtr
      << "\n*&aPtr = " << *&aPtr << endl;
} // end main
```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Fig. 8.4 | Pointer operators `&` and `*`.

The address of a (line 14) and the value of aPtr (line 15) are identical in the output, confirming that the address of a is indeed assigned to the pointer variable aPtr. The & and * operators are *inverses* of one another—when they're applied consecutively to aPtr in either order, they "cancel one another out" yielding the same result (the value in aPtr).

Figure 8.5 lists the precedence and associativity of the operators introduced to this point. The address (&) and dereferencing operator (*) are unary operators on the third level.

Operators	Associativity	Type
:: ()	*[See caution in Fig. 2.10]*	highest
() []	left to right	function call/array access
++ -- static_cast<*type*>(*operand*)	left to right	unary (postfix)
++ -- + - ! & *	right to left	unary (prefix)
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 8.5 | Operator precedence and associativity.

8.4 Pass-by-Reference with Pointers

There are three ways in C++ to pass arguments to a function—pass-by-value, **pass-by-reference with reference arguments** and **pass-by-reference with pointer arguments**. Chapter 6 compared and contrasted pass-by-value and pass-by-reference with reference arguments. In this section, we explain pass-by-reference with pointer arguments.

As we saw in Chapter 6, return can be used to return one value from a called function to a caller (or to simply return control). We also saw that arguments can be passed to a function using reference arguments. Such arguments enable the called function to *modify the original values of the arguments in the caller*. Reference arguments also enable programs to pass large data objects to a function and avoid the overhead of passing the objects by value (which, of course, requires making a copy of the object). Pointers, like references, also can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.

You can use pointers and the indirection operator (*) to accomplish pass-by-reference (exactly as pass-by-reference is done in C programs—C does not have references). When calling a function with an argument that should be modified, the *address* of the argument is passed. This is normally accomplished by applying the address operator (&) to the name of the variable whose value will be modified.

As we saw in Chapter 7, arrays are *not* passed using operator `&`, because the name of the array is the starting location in memory of the array (i.e., an array name is already a pointer). The name of the array `arrayName` is equivalent to `&arrayName[0]`. When the address of a variable is passed to a function, the indirection operator (*) can be used in the function to form a synonym for the name of the variable (i.e., an *lvalue*)—this in turn can be used to modify the variable's value at that location in the caller's memory.

Figure 8.6 and Fig. 8.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`. Figure 8.6 passes variable `number` by value to function `cubeByValue` (line 14). Function `cubeByValue` (lines 19–22) cubes its argument and passes the new value back to `main` using a `return` statement (line 21). The new value is assigned to `number` (line 14) in `main`. The calling function has the opportunity to examine the function call's result *before* modifying variable `number`'s value. For example, we could have stored the result of `cubeByValue` in another variable, examined its value and assigned the result to `number` only after determining that the returned value was reasonable.

```
1  // Fig. 8.6: fig08_06.cpp
2  // Pass-by-value used to cube a variable's value.
3  #include <iostream>
4  using namespace std;
5
6  int cubeByValue( int ); // prototype
7
8  int main()
9  {
10    int number = 5;
11
12    cout << "The original value of number is " << number;
13
14    number = cubeByValue( number ); // pass number by value to cubeByValue
15    cout << "\nThe new value of number is " << number << endl;
16 } // end main
17
18 // calculate and return cube of integer argument
19 int cubeByValue( int n )
20 {
21    return n * n * n; // cube local variable n and return result
22 } // end function cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.6 | Pass-by-value used to cube a variable's value.

Figure 8.7 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument (line 15)—the address of `number` is passed to the function. Function `cubeByReference` (lines 21–24) specifies parameter `nPtr` (a pointer to `int`) to receive its argument. The function dereferences the pointer and cubes the value to which `nPtr` points (line 23). This *directly* changes the value of `number` in `main`.

Common Programming Error 8.5

Not dereferencing a pointer when it's necessary to do so to obtain the value to which the pointer points is an error.

```
1  // Fig. 8.7: fig08_07.cpp
2  // Pass-by-reference with a pointer argument used to cube a
3  // variable's value.
4  #include <iostream>
5  using namespace std;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18 } // end main
19
20 // calculate cube of *nPtr; modifies variable number in main
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, the header for function `cubeByReference` (line 21) specifies that `cubeByReference` receives the address of an `int` variable (i.e., a pointer to an `int`) as an argument, stores the address locally in `nPtr` and does not return a value.

The function prototype for `cubeByReference` (line 7) contains `int *` in parentheses. As with other variable types, it isn't necessary to include the *names* of pointer parameters in function prototypes. Parameter names included for documentation purposes are ignored by the compiler.

Figures 8.8–8.9 analyze graphically the execution of the programs in Fig. 8.6 and Fig. 8.7, respectively.

Software Engineering Observation 8.1

Use pass-by-value to pass arguments to a function unless the caller explicitly *requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.*

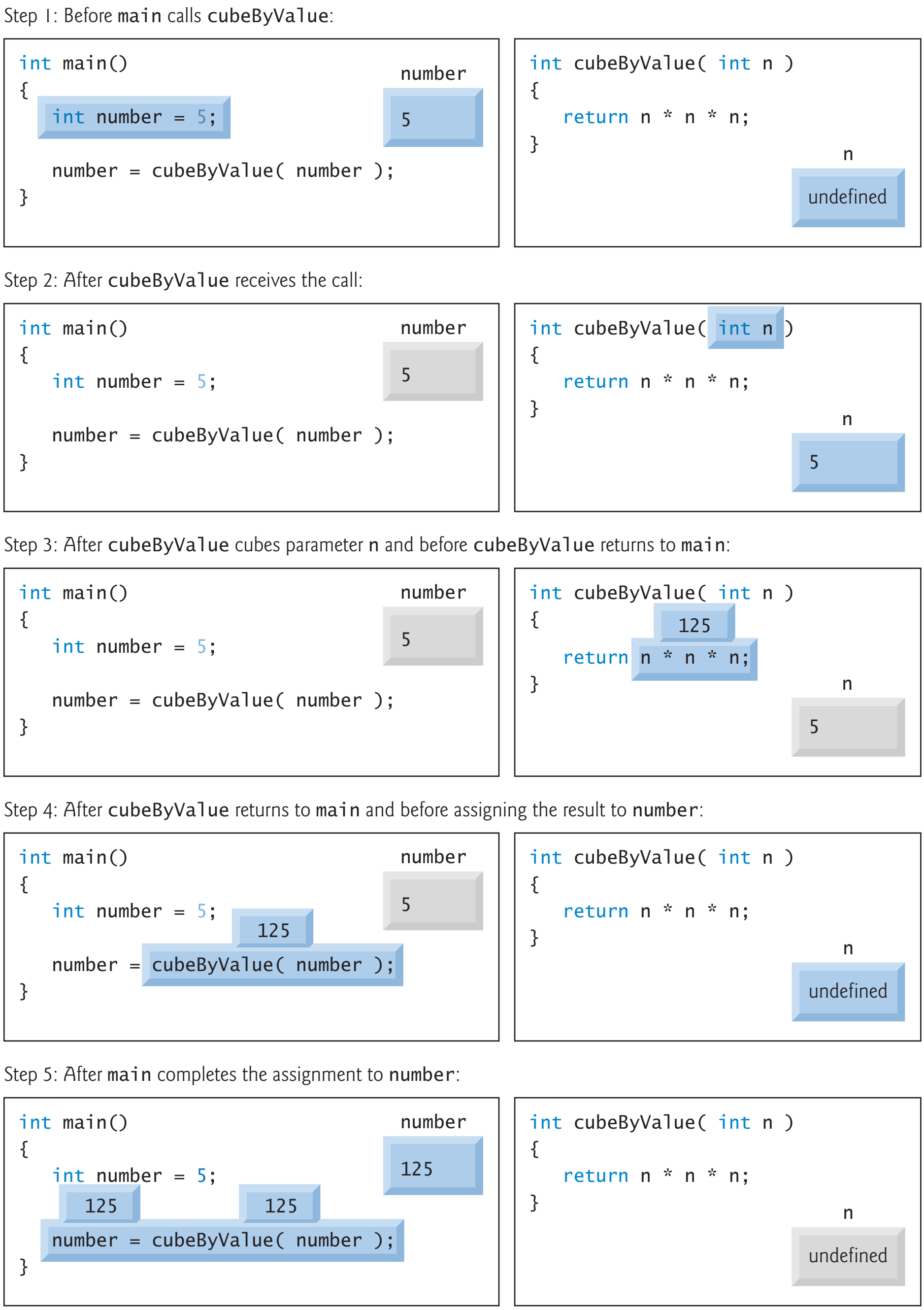


Fig. 8.8 | Pass-by-value analysis of the program of Fig. 8.6.

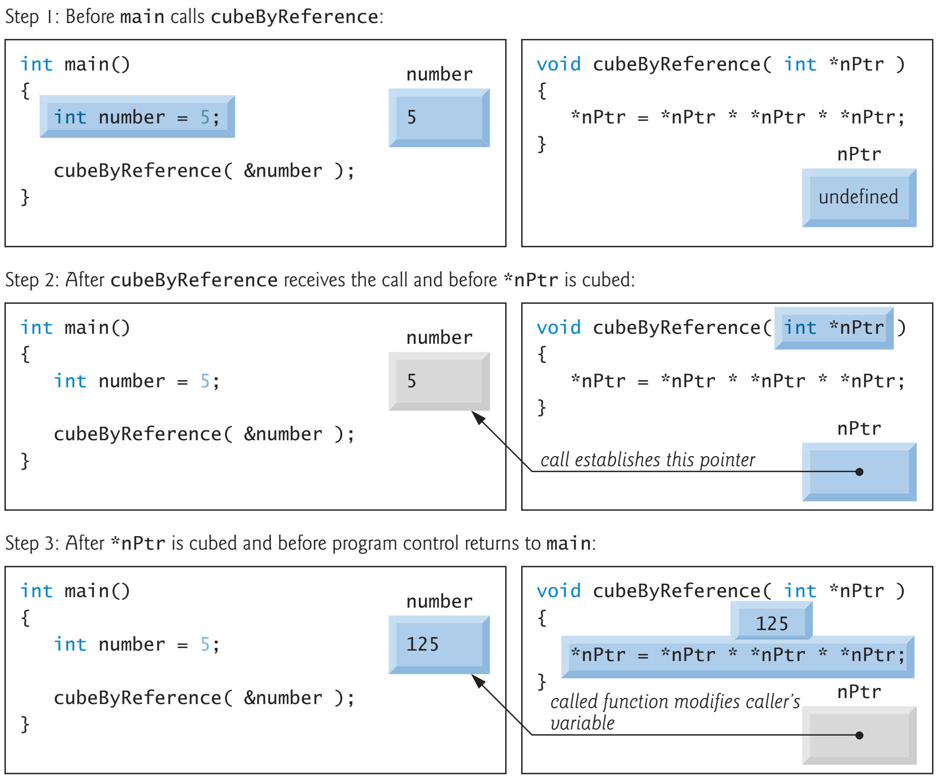


Fig. 8.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7.

In the function header and in the prototype for a function that expects a one-dimensional array as an argument, the pointer notation in the parameter list of `cubeByReference` may be used. *The compiler does not differentiate between a function that receives a pointer and a function that receives a one-dimensional array.* This, of course, means that the function must "know" when it's receiving an array or simply a single variable which is being passed by reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b` (that is, "`b` is a pointer to an integer"). Both forms of declaring a function parameter as a one-dimensional array are interchangeable.

8.5 Using `const` with Pointers

Recall that `const` enables you to inform the compiler that the value of a particular variable should *not* be modified. Many possibilities exist for using (or not using) `const` with function parameters. How do you choose the most appropriate of these possibilities? Let the principle of least privilege be your guide. Always give a function enough access to the data in its parameters to accomplish its specified task, *but no more*. This section discusses how to combine `const` with pointer declarations to enforce the principle of least privilege.

Chapter 6 explained that when an argument is passed by value, a *copy* of the argument in the function call is made and passed to the function. If the copy is *modified* in the function, the original value in the caller *does not change*. In many cases, a value passed to a function is modified in *that* function. However, in some instances, the value should *not* be altered in the called function, even though the called function manipulates only a copy of the original value.

Consider a function that takes a one-dimensional array and its size as arguments and subsequently prints the array. Such a function should loop through the array and output each element individually. The size of the array is used in the function body to determine the array's highest subscript so the loop can terminate when the printing completes. The array's size does not need to change in the function body, so it should be declared `const` to ensure that it will not change. Because the array is only being printed, it, too, should be declared `const`. This is especially important because arrays are *always* passed by reference and could easily be changed in the called function. If an attempt is made to modify a `const` value, an error occurs.

Software Engineering Observation 8.2

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared `const`.

Error-Prevention Tip 8.2

Before using a function, check its function prototype to determine the parameters that it can and cannot modify.

There are four ways to pass a pointer to a function: a nonconstant pointer to nonconstant data, a nonconstant pointer to constant data (Fig. 8.10), a constant pointer to nonconstant data (Fig. 8.11) and a constant pointer to constant data (Fig. 8.12). Each combination provides a different level of access privilege.

Nonconstant Pointer to Nonconstant Data

The highest access is granted by a **nonconstant pointer to nonconstant data**—the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data. Such a pointer's declaration (e.g., `int *countPtr`) does not include `const`.

Nonconstant Pointer to Constant Data

A **nonconstant pointer to constant data** is a pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer. Such a pointer might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data. Any attempt to modify the data in the function results in a compilation error. The declaration for such a pointer places `const` to the left of the pointer's type, as in

```
const int *countPtr;
```

The declaration is read from *right to left* as "`countPtr` is a pointer to an integer constant."

Figure 8.10 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data.

```
// Fig. 8.10: fig08_10.cpp
// Attempting to modify data through a
// nonconstant pointer to constant data.

void f( const int * ); // prototype

int main()
{
   int y;

   f( &y ); // f attempts illegal modification
} // end main

// xPtr cannot modify the value of constant variable to which it points
void f( const int *xPtr )
{
   *xPtr = 100; // error: cannot modify a const object
} // end function f
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch08\Fig08_10\fig08_10.cpp(17) :
   error C3892: 'xPtr' : you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_10.cpp: In function `void f(const int*)':
fig08_10.cpp:17: error: assignment of read-only location
```

Fig. 8.10 | Attempting to modify data through a nonconstant pointer to constant data.

As you know, arrays are aggregate data types that store related data items of the same type under one name. When a function is called with an array as an argument, the array is passed to the function by reference. However, *by default, objects are passed by value—a copy of the entire object is passed.* This requires the execution-time overhead of making a copy of each data item in the object and storing it on the function call stack. When a pointer to an object is passed, only a copy of the address of the object must be made—the object itself is not copied.

Performance Tip 8.1

If they do not need to be modified by the called function, pass large objects using pointers to constant data or references to constant data, to obtain the performance benefits of pass-by-reference.

Software Engineering Observation 8.3

Pass large objects using pointers to constant data, or references to constant data, to obtain the security of pass-by-value.

Constant Pointer to Nonconstant Data

A **constant pointer to nonconstant data** is a pointer that always points to the same memory location; the data at that location *can* be modified through the pointer. An example of such a pointer is an array name, which is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation. Pointers that are declared `const` must be initialized when they're declared, but if the pointer is a function parameter, it's initialized with the pointer that's passed to the function.

Common Programming Error 8.6

Not initializing a pointer that's declared `const` *is a compilation error.*

The program of Fig. 8.11 attempts to modify a constant pointer. Line 11 declares pointer `ptr` to be of type `int * const`. The declaration is read from right to left as "`ptr` is a constant pointer to a nonconstant integer." The pointer is initialized with the address of integer variable `x`. Line 14 attempts to assign the address of `y` to `ptr`, but the compiler generates an error message. No error occurs when line 13 assigns the value `7` to `*ptr`—the nonconstant value to which `ptr` points *can* be modified using the dereferenced `ptr`, even though `ptr` itself has been declared `const`.

```
1  // Fig. 8.11: fig08_11.cpp
2  // Attempting to modify a constant pointer to nonconstant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch08\Fig08_11\fig08_11.cpp(14) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_11.cpp: In function `int main()':
fig08_11.cpp:14: error: assignment of read-only variable `ptr'
```

Fig. 8.11 | Attempting to modify a constant pointer to nonconstant data.

Constant Pointer to Constant Data

The minimum access privilege is granted by a **constant pointer to constant data**. Such a pointer *always* points to the *same* memory location, and the data at that location *cannot* be modified via the pointer. This is how an array should be passed to a function that *only reads* the array, using array subscript notation, and *does not modify* the array. The program of Fig. 8.12 declares pointer variable `ptr` to be of type `const int * const` (line 13). This declaration is read from *right to left* as "`ptr` is a constant pointer to an integer constant." The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points (line 17) and when an attempt is made to modify the address stored in the pointer variable (line 18). No errors occur when the program attempts to dereference `ptr` (line 15), or when the program attempts to output the value to which `ptr` points, because neither the pointer nor the data it points to is being modified in this statement.

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch08\Fig08_12\fig08_12.cpp(17) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
c:\cpphtp8_examples\ch08\Fig08_12\fig08_12.cpp(18) : error C3892: 'ptr' :
   you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig08_12.cpp: In function `int main()':
fig08_12.cpp:17: error: assignment of read-only location
fig08_12.cpp:18: error: assignment of read-only variable `ptr'
```

Fig. 8.12 | Attempting to modify a constant pointer to constant data.

8.6 Selection Sort Using Pass-by-Reference

In this section, we define a sorting program to demonstrate passing arrays and individual array elements by reference. We use the **selection sort** algorithm, which is an easy-to-program, but unfortunately inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. After the i^{th} iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

As an example, consider the array

```
34   56   4   10   77   51   93   30   5   52
```

A program that implements the selection sort first determines the smallest value (4) in the array, which is contained in element 2. The program swaps the 4 with the value in element 0 (34), resulting in

```
4   56   34   10   77   51   93   30   5   52
```

The program then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8. The program swaps the 5 with the 56 in element 1, resulting in

```
4   5   34   10   77   51   93   30   56   52
```

On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
4   5   10   34   77   51   93   30   56   52
```

The process continues until the array is fully sorted.

```
4   5   10   30   34   51   52   56   77   93
```

After the first iteration, the smallest element is in the first position. After the second iteration, the two smallest elements are in order in the first two positions. After the third iteration, the three smallest elements are in order in the first three positions.

Figure 8.13 implements selection sort using functions `selectionSort` and `swap`. Function `selectionSort` (lines 32–49) sorts the array. Line 34 declares the variable `smallest`, which will store the index of the smallest element in the remaining array. Lines 37–48 loop `size - 1` times. Line 39 sets the smallest element's index to the current index. Lines 42–45 loop over the remaining array elements. For each element, line 44 compares its value to the value of the smallest element. If the current element is smaller than the smallest element, line 45 assigns the current element's index to `smallest`. When this loop finishes, `smallest` will contain the index of the smallest element in the remaining array. Line 47 calls function `swap` (lines 53–58) to place the smallest remaining element in the next spot in the array (i.e., exchange the array elements `array[i]` and `array[smallest]`).

```
1  // Fig. 8.13: fig08_13.cpp
2  // Selection sort with pass-by-reference. This program puts values into an
3  // array, sorts them into ascending order and prints the resulting array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
```

Fig. 8.13 | Selection sort with pass-by-reference. (Part 1 of 3.)

```
void selectionSort( int * const, const int ); // prototype
void swap( int * const, int * const ); // prototype

int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   cout << "Data items in original order\n";

   for ( int i = 0; i < arraySize; ++i )
      cout << setw( 4 ) << a[ i ];

   selectionSort( a, arraySize ); // sort the array

   cout << "\nData items in ascending order\n";

   for ( int j = 0; j < arraySize; ++j )
      cout << setw( 4 ) << a[ j ];

   cout << endl;
} // end main

// function to sort an array
void selectionSort( int * const array, const int size )
{
   int smallest; // index of smallest element

   // loop over size - 1 elements
   for ( int i = 0; i < size - 1; ++i )
   {
      smallest = i; // first index of remaining array

      // loop to find index of smallest element
      for ( int index = i + 1; index < size; ++index )

         if ( array[ index ] < array[ smallest ] )
            smallest = index;

      swap( &array[ i ], &array[ smallest ] );
   } // end if
} // end function selectionSort

// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap
```

Fig. 8.13 | Selection sort with pass-by-reference. (Part 2 of 3.)

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 8.13 | Selection sort with pass-by-reference. (Part 3 of 3.)

Let's now look more closely at function `swap`. Remember that *information hiding* is enforced between functions, so `swap` does not have access to individual array elements in `selectionSort`. Because `selectionSort` *wants* `swap` to have access to the array elements to be swapped, `selectionSort` passes each of these elements to `swap` by reference—the address of each array element is passed explicitly. Although entire arrays are passed by reference, individual array elements are ordinarily passed by value. Therefore, `selectionSort` uses the address operator (`&`) on each array element in the `swap` call (line 47) to effect pass-by-reference. Function `swap` (lines 53–58) receives `&array[ i ]` in pointer variable `element1Ptr`. Information hiding prevents `swap` from "knowing" the name `array[ i ]`, but `swap` can use `*element1Ptr` as a synonym for `array[ i ]`. Thus, when `swap` references `*element1Ptr`, it's actually referencing `array[ i ]` in `selectionSort`. Similarly, when `swap` references `*element2Ptr`, it's actually referencing `array[ smallest ]` in `selectionSort`.

Even though `swap` is not allowed to use the statements

```
hold = array[ i ];
array[ i ] = array[ smallest ];
array[ smallest ] = hold;
```

precisely the same effect is achieved by

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

in the `swap` function of Fig. 8.13.

Several features of function `selectionSort` should be noted. The function header (line 32) declares `array` as `int * const array`, rather than `int array[]`, to indicate that the function receives a one-dimensional array as an argument. Both parameter `array`'s pointer and the parameter `size` are declared `const` to enforce the principle of least privilege. Although parameter `size` receives a copy of a value in `main` and modifying the copy cannot change the value in `main`, `selectionSort` does *not* need to alter `size` to accomplish its task—the array size remains fixed during the execution of `selectionSort`. Therefore, `size` is declared `const` to ensure that it isn't modified. If the size of the array were to be modified during the sorting process, the sorting algorithm would not run correctly.

Function `selectionSort` receives the size of the array as a parameter, because the function must have that information to sort the array. When an array is passed to a function, *only* the memory address of the first element of the array is received by the function; the array size must be passed separately to the function.

By defining function `selectionSort` to receive the array size as a parameter, we enable the function to be used by *any* program that sorts one-dimensional `int` arrays of *arbitrary* size. The size of the array could have been programmed directly into the function, but this would restrict the function to processing an array of a specific size and reduce the function's *reusability*—only programs processing one-dimensional `int` arrays of the specific size "hard coded" into the function could use the function.

Software Engineering Observation 8.4

When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size)—this makes the function more reusable.

8.7 `sizeof` Operator

The compile time unary operator **`sizeof`** determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation. When applied to the *name* of an array, as in Fig. 8.14 (line 13), the `sizeof` operator returns the *total number of bytes in the array* as a value of type `size_t` (an unsigned integer type that is at least as big as `unsigned int`). This is different from the `size` of a `vector<int>`, for example, which is the number of integer elements in the `vector`. The computer we used to compile this program stores variables of type `double` in 8 bytes of memory, and `array` is declared to have 20 elements (line 11), so `array` uses 160 bytes in memory. When applied to a pointer parameter (line 22) in a function that receives an array as an argument, the `sizeof` operator returns the size of the pointer in bytes (4 on the system we used)—*not* the size of the array.

Common Programming Error 8.7

Using the `sizeof` operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.

```
1  // Fig. 8.14: fig08_14.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using namespace std;
6
7  size_t getSize( double * ); // prototype
8
9  int main()
10 {
11    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
12
13    cout << "The number of bytes in the array is " << sizeof( array );
14
15    cout << "\nThe number of bytes returned by getSize is "
16       << getSize( array ) << endl;
17 } // end main
18
19 // return size of ptr
20 size_t getSize( double *ptr )
21 {
22    return sizeof( ptr );
23 } // end function getSize
```

Fig. 8.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 1 of 2.)

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 8.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 2 of 2.)

The number of elements in an array also can be determined using the results of two `sizeof` operations. For example, consider the following array declaration:

```
double realArray[ 22 ];
```

If variables of data type `double` are stored in eight bytes of memory, array `realArray` contains a total of 176 bytes. To determine the number of *elements* in the array, the following expression (which is evaluated at *compile time*) can be used:

```
sizeof realArray / sizeof( realArray[ 0 ] )
```

The expression determines the number of bytes in array `realArray` (176) and divides that value by the number of bytes used in memory to store the array's first element (typically 8 for a `double` value)—the result is the number of elements in `realArray` (22).

Determining the Sizes of the Fundamental Types, an Array and a Pointer

Figure 8.15 uses `sizeof` to calculate the number of bytes used to store most of the standard data types. The output shows that the types `double` and `long double` have the same size. Types may have different sizes based on the platform running the program. On another system, for example, `double` and `long double` may be of different sizes.

```
// Fig. 8.15: fig08_15.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using namespace std;

int main()
{
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[ 20 ]; // array of int
   int *ptr = array; // variable of type int *

   cout << "sizeof c = " << sizeof c
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof s
      << "\tsizeof(short) = " << sizeof( short )
      << "\nsizeof i = " << sizeof i
```

Fig. 8.15 | `sizeof` operator used to determine standard data type sizes. (Part 1 of 2.)

```
      << "\tsizeof(int) = " << sizeof( int )
      << "\nsizeof l = " << sizeof l
      << "\tsizeof(long) = " << sizeof( long )
      << "\nsizeof f = " << sizeof f
      << "\tsizeof(float) = " << sizeof( float )
      << "\nsizeof d = " << sizeof d
      << "\tsizeof(double) = " << sizeof( double )
      << "\nsizeof ld = " << sizeof ld
      << "\tsizeof(long double) = " << sizeof( long double )
      << "\nsizeof array = " << sizeof array
      << "\nsizeof ptr = " << sizeof ptr << endl;
} // end main
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Fig. 8.15 | `sizeof` operator used to determine standard data type sizes. (Part 2 of 2.)

Portability Tip 8.1

The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.

Operator `sizeof` can be applied to any expression or type name. When `sizeof` is applied to a variable name (which is not an array name) or other expression, the number of bytes used to store the specific type of the expression's value is returned. The parentheses used with `sizeof` are required *only* if a type name (e.g., `int`) is supplied as its operand. The parentheses used with `sizeof` are not required when `sizeof`'s operand is an expression. Remember that `sizeof` is an operator, not a function, and that it has its effect at *compile time*, not execution time.

8.8 Pointer Expressions and Pointer Arithmetic

This section describes the operators that can have *pointers* as operands and how these operators are used with pointers. C++ enables **pointer arithmetic**—certain arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=), or one pointer may be subtracted from another of the same type.

Assume that array `int v[5]` has been declared and that its first element is at memory location `3000`. Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is `3000`). Figure 8.16 diagrams this situation for a machine with four-byte integers. Variable `vPtr` can be initialized to point to array `v` with either of the following statements (because the name of an array is equivalent to the address of its first element):

```
int *vPtr = v;
int *vPtr = &v[ 0 ];
```

Portability Tip 8.2

Most computers today have two-byte or four-byte integers. Some of the newer machines use eight-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

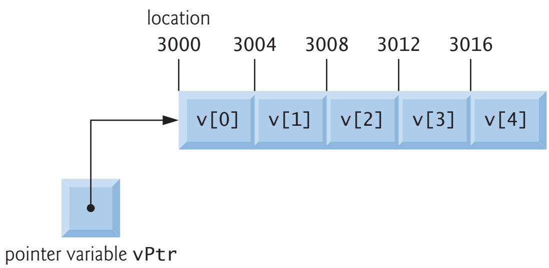


Fig. 8.16 | Array v and a pointer variable int *vPtr that points to v.

In conventional arithmetic, the addition 3000 + 2 yields the value 3002. This is normally *not* the case with pointer arithmetic. When an integer is added to, or subtracted from, a pointer, the pointer is *not* simply incremented or decremented by that integer, but by that integer *times* the size of the object to which the pointer refers. The number of bytes depends on the object's data type. For example, the statement

```
vPtr += 2;
```

would produce 3008 (from the calculation 3000 + 2 * 4), assuming that an int is stored in four bytes of memory. In the array v, vPtr would now point to v[2] (Fig. 8.17). If an integer is stored in two bytes of memory, then the preceding calculation would result in memory location 3004 (3000 + 2 * 2). If the array elements were of a different data type, the preceding statement would increment the pointer by twice the number of bytes it takes to store an object of that data type.

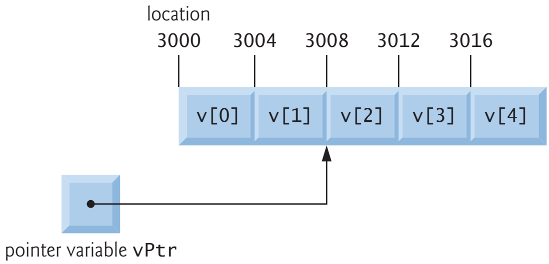


Fig. 8.17 | Pointer vPtr after pointer arithmetic.

If `vPtr` had been incremented to `3016`, which points to `v[4]`, the statement

```
vPtr -= 4;
```

would set `vPtr` back to `3000`—the beginning of the array. If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used. Each of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the *next* element of the array. Each of the statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the *previous* element of the array.

Pointer variables pointing to the *same* array may be subtracted from one another. For example, if `vPtr` contains the address `3000` and `v2Ptr` contains the address `3008`, the statement

```
x = v2Ptr - vPtr;
```

would assign to `x` the *number of array elements* from `vPtr` to `v2Ptr`—in this case, `2`. Pointer arithmetic is meaningless unless performed on a pointer that points to an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.

Common Programming Error 8.8

Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.

A pointer can be assigned to another pointer if both pointers are of the *same* type. Otherwise, a cast operator (normally a `reinterpret_cast`; discussed in Section 17.7) must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment. The exception to this rule is the **pointer to `void`** (i.e., **`void *`**), which is a generic pointer capable of representing *any* pointer type. All pointer types can be assigned to a pointer of type `void *` without casting. However, a pointer of type `void *` *cannot* be assigned directly to a pointer of another type—the pointer of type `void *` must first be cast to the proper pointer type.

Software Engineering Observation 8.5

Nonconstant pointer arguments can be passed to constant pointer parameters.

A `void *` pointer *cannot* be dereferenced. For example, the compiler "knows" that a pointer to `int` refers to four bytes of memory on a machine with four-byte integers, but a pointer to `void` simply contains a memory address for an unknown data type—the precise number of bytes to which the pointer refers and the type of the data are *not* known by the compiler. The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined.

Common Programming Error 8.9

Assigning a pointer of one type to a pointer of another (other than `void *`) without using a cast (normally a `reinterpret_cast`) is a compilation error.

Common Programming Error 8.10

All operations on a `void *` pointer are compilation errors, except comparing `void *` pointers with other pointers, casting `void *` pointers to valid pointer types and assigning addresses to `void *` pointers.

Pointers can be compared using equality and relational operators. Comparisons using relational operators are meaningless unless the pointers point to members of the *same* array. Pointer comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to the same array could show, for example, that one pointer points to a higher numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).

8.9 Relationship Between Pointers and Arrays

Arrays and pointers are intimately related in C++ and may be used *almost* interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operation involving array subscripting.

Assume the following declarations:

```
int b[ 5 ]; // create 5-element int array b
int *bPtr; // create int pointer bPtr
```

Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set `bPtr` to the address of the first element in array `b` with the statement

```
bPtr = b; // assign address of array b to bPtr
```

This is equivalent to assigning the address of the first element of the array as follows:

```
bPtr = &b[ 0 ]; // also assigns address of array b to bPtr
```

Array element `b[3]` can alternatively be referenced with the pointer expression

```
*( bPtr + 3 )
```

The `3` in the preceding expression is the **offset** to the pointer. When the pointer points to the beginning of an array, the offset indicates which array element should be referenced, and the offset value is identical to the subscript. This notation is referred to as **pointer/offset notation**. The parentheses are necessary, because the precedence of `*` is higher than that of `+`. Without the parentheses, the preceding expression would add `3` to a copy `*bPtr`'s value (i.e., `3` would be added to `b[0]`, assuming that `bPtr` points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
&b[ 3 ]
```

can be written with the pointer expression

```
bPtr + 3
```

The array name (which is implicitly `const`) can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
*( b + 3 )
```

also refers to the array element `b[3]`. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. The preceding expression does *not* modify the array name in any way; `b` still points to the first element in the array.

Pointers can be subscripted exactly as arrays can. For example, the expression

```
bPtr[ 1 ]
```

refers to the array element `b[1]`; this expression uses **pointer/subscript notation**.

Remember that an array name is a *constant* pointer; it always points to the beginning of the array. Thus, the expression

```
b += 3
```

causes a compilation error, because it attempts to *modify* the value of the array name (a constant) with pointer arithmetic.

Common Programming Error 8.11

Although array names are pointers to the beginning of the array, array names cannot be modified in arithmetic expressions, because array names are constant pointers.

Good Programming Practice 8.2

For clarity, use array notation instead of pointer notation when manipulating arrays.

Figure 8.18 uses the four notations discussed in this section for referring to array elements—array subscript notation, pointer/offset notation with the array name as a pointer, pointer subscript notation and pointer/offset notation with a pointer—to accomplish the same task, namely printing the four elements of the integer array `b`.

```
// Fig. 8.18: fig08_18.cpp
// Using subscripting and pointer notations with arrays.
#include <iostream>
using namespace std;

int main()
{
   int b[] = { 10, 20, 30, 40 }; // create 4-element array b
   int *bPtr = b; // set bPtr to point to array b

   // output array b using array subscript notation
   cout << "Array b printed with:\n\nArray subscript notation\n";

   for ( int i = 0; i < 4; ++i )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

```

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 1 of 2.)

```
   // output array b using the array name and pointer/offset notation
   cout << "\nPointer/offset notation where "
      << "the pointer is the array name\n";

   for ( int offset1 = 0; offset1 < 4; ++offset1 )
      cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';

   // output array b using bPtr and array subscript notation
   cout << "\nPointer subscript notation\n";

   for ( int j = 0; j < 4; ++j )
      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

   cout << "\nPointer/offset notation\n";

   // output array b using bPtr and pointer/offset notation
   for ( int offset2 = 0; offset2 < 4; ++offset2 )
      cout << "*(bPtr + " << offset2 << ") = "
         << *( bPtr + offset2 ) << '\n';
} // end main
```

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 8.18 | Referencing array elements with the array name and with pointers. (Part 2 of 2.)

8.10 Pointer-Based String Processing

We've already used the C++ Standard Library `string` class to represent strings as full-fledged objects. For example, the `GradeBook` class case study in Chapters 3–7 represents a course name using a `string` object. Chapter 18 presents class `string` in detail. This section introduces C-style, pointer-based strings. *C++'s `string` class is preferred for use in new programs, because it eliminates many of the security problems and bugs that can be caused by* *manipulating C strings*. We cover C strings here for a deeper understanding of arrays. Also, if you work with legacy C and C++ programs, you're likely to encounter these pointer-based strings. We cover C-style, pointer-based strings in detail in Chapter 21.

Characters and Character Constants

Characters are the fundamental building blocks of C++ source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the compiler as a series of instructions used to accomplish a task. A program may contain **character constants**. A character constant is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, `'z'` represents the integer value of `z` (122 in the ASCII character set; see Appendix B), and `'\n'` represents the integer value of newline (10 in the ASCII character set).

Strings

A string is a series of characters treated as a single unit. A string may include letters, digits and various **special characters** such as `+`, `-`, `*`, `/`and `$`. **String literals**, or **string constants**, in C++ are written in double quotation marks as follows:

`"John Q. Doe"`	(a name)
`"9999 Main Street"`	(a street address)
`"Maynard, Massachusetts"`	(a city and state)
`"(201) 555-1212"`	(a telephone number)

Pointer-Based Strings

A pointer-based string is an array of characters ending with a **null character (`'\0'`)**, which marks where the string terminates in memory. A string is accessed via a pointer to its first character. The value of a string literal is the address of its first character, but the `sizeof` a string literal is the length of the string *including* the terminating null character. Pointer-based strings are like arrays—an array name is also a pointer to its first element.

String Literals as Initializers

A string literal may be used as an initializer in the declaration of either a character array or a variable of type `char *`. The declarations

```
char color[] = "blue";
const char *colorPtr = "blue";
```

each initialize a variable to the string `"blue"`. The first declaration creates a *five-element* array `color` containing the characters `'b'`, `'l'`, `'u'`, `'e'` *and* `'\0'`. The second declaration creates pointer variable `colorPtr` that points to the letter `b` in the string `"blue"` (which ends in `'\0'`) somewhere in memory. String literals have `static` storage class (they exist for the duration of the program) and may or may not be shared if the same string literal is referenced from multiple locations in a program. The effect of modifying a string literal is *undefined*; thus, you should always declare a pointer to a string literal as `const char *`.

Character Constants as Initializers

The declaration `char color[] = "blue";` could also be written

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

which uses character constants in single quotes (`'`) as initializers for each element of the array. When declaring a character array to contain a string, the array must be large enough to store the string *and* its terminating null character. The compiler determines the size of the array in the preceding declaration, based on the number of initializers in the initializer list.

Common Programming Error 8.12

Not allocating sufficient space in a character array to store the null character that terminates a string is a logic error.

Common Programming Error 8.13

Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.

Error-Prevention Tip 8.3

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length. If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors and potential security breaches.

Accessing Characters in a C-String

Because a C-style string is an array of characters, we can access individual characters in a string directly with array subscript notation. For example, in the preceding declaration, `color[0]` is the character `'b'`, `color[2]` is `'u'` and `color[4]` is the null character.

Reading Strings into `char` Arrays with `cin`

A string can be read into a character array using stream extraction with `cin`. For example, the following statement reads a string into character array `word[20]`:

```
cin >> word;
```

The string entered by the user is stored in `word`. The preceding statement reads characters until a white-space character or end-of-file indicator is encountered. The string should be no longer than 19 characters to leave room for the terminating null character. The `setw` stream manipulator can be used to *ensure* that the string read into `word` *does not exceed the size of the array*. For example, the statement

```
cin >> setw( 20 ) >> word;
```

specifies that `cin` should read a maximum of 19 characters into array `word` and save the 20th location in the array to store the terminating null character for the string. The `setw` stream manipulator applies *only* to the next value being input. If more than 19 characters are entered, the remaining characters are not saved in `word`, but they will be in the input stream and can be read by the next input operation.

Reading Lines of Text into `char` Arrays with `cin.getline`

In some cases, it's desirable to input an *entire line of text* into a character array. For this purpose, the `cin` object provides the member function **`getline`**, which takes three arguments—a character array in which the line of text will be stored, a length and a delimiter character. For example, the statements

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

declare array `sentence` of 80 characters and read a line of text from the keyboard into the array. The function stops reading characters when the delimiter character `'\n'` is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument. The last character in the array is reserved for the terminating null character. If the delimiter character is encountered, it's read and discarded. The third argument to `cin.getline` has `'\n'` as a default value, so the preceding function call could have been written as:

```
cin.getline( sentence, 80 );
```

Chapter 15, Stream Input/Output, provides a detailed discussion of `cin.getline` and other input/output functions.

Displaying C-Style Strings

A character array representing a null-terminated string can be output with `cout` and `<<`. The statement

```
cout << sentence;
```

prints the array `sentence`. Note that `cout <<`, like `cin >>`, does not care how large the character array is. The characters of the string are output until a terminating null character is encountered; the null character is not printed. [*Note:* `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters; `cin` and `cout` do not provide similar input and output processing capabilities for other array types.]

8.11 Arrays of Pointers

Arrays may contain pointers. A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a **string array**. Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string. Consider the declaration of string array `suit` that might be useful in representing a deck of cards:

```
const char * const suit[ 4 ] =
   { "Hearts", "Diamonds", "Clubs", "Spades" };
```

The `suit[4]` portion of the declaration indicates an array of four elements. The `const char *` portion of the declaration indicates that each element of array `suit` is of type "pointer to `char` constant data." The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`. Each is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes. The four strings are seven, nine, six and seven characters long (including their terminating null characters), respectively. Although it appears as though these strings are being placed in the `suit` array, only *pointers* are actually stored in the array, as shown in Fig. 8.19. Each pointer points to the first character of its corresponding string. Thus, even though the `suit` array is fixed in size, it provides access to character strings of any length. This flexibility is one example of C++'s powerful data-structuring capabilities.

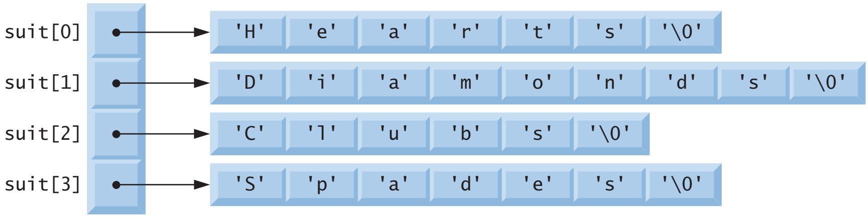


Fig. 8.19 | Graphical representation of the suit array.

The suit strings could be placed into a two-dimensional array, in which each row represents one suit and each column represents one of the letters of a suit name. Such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when we store a large number of strings, of which most are shorter than the longest string. We use arrays of strings to help represent a deck of cards in the next section.

String arrays are commonly used with **command-line arguments** that are passed to function `main` when a program begins execution. Such arguments follow the program name when a program is executed from the command line. A typical use of command-line arguments is to pass options to a program. For example, from the command line on a Windows computer, the user can type

```
dir /p
```

to list the contents of the current directory and pause after each screen of information. When the `dir` command executes, the option `/p` is passed to `dir` as a command-line argument. Such arguments are placed in a string array that `main` receives as an argument. We discuss command-line arguments in Appendix F, C Legacy Code Topics.

8.12 Function Pointers

A **pointer to a function** contains the function's address in memory. We know that an array's name is actually the address in memory of the first element. Similarly, a function's name is actually the starting address in memory of the code that performs the function's task. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned to other function pointers and used to call the underlying function.

Multipurpose Selection Sort Using Function Pointers

To illustrate the use of pointers to functions, Fig. 8.20 modifies the selection sort program of Fig. 8.13. Figure 8.20 consists of `main` (lines 13–50) and the functions `selectionSort` (lines 54–71), `swap` (lines 75–80), `ascending` (lines 84–87) and `descending` (lines 91–94). Function `selectionSort` receives a pointer to a function—either function `ascending` or function `descending`—as an argument in addition to the integer array to sort and the size of the array. Functions `ascending` and `descending` determine the sorting order. The program prompts the user to choose whether the array should be sorted in ascending order or in descending order (lines 20–22). If the user enters 1, a pointer to function `ascending` is passed to function `selectionSort` (line 33), causing the array to be sorted into increasing order. If the user enters 2, a pointer to function `descending` is passed to function `selectionSort` (line 41), causing the array to be sorted into decreasing order.

```
1  // Fig. 8.20: fig08_20.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // prototypes
8  void selectionSort( int [], const int, bool (*)( int, int ) );
9  void swap( int * const, int * const );
10 bool ascending( int, int ); // implements ascending order
11 bool descending( int, int ); // implements descending order
12
13 int main()
14 {
15    const int arraySize = 10;
16    int order; // 1 = ascending, 2 = descending
17    int counter; // array index
18    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19
20    cout << "Enter 1 to sort in ascending order,\n"
21       << "Enter 2 to sort in descending order: ";
22    cin >> order;
23    cout << "\nData items in original order\n";
24
25    // output original array
26    for ( counter = 0; counter < arraySize; ++counter )
27       cout << setw( 4 ) << a[ counter ];
28
29    // sort array in ascending order; pass function ascending
30    // as an argument to specify ascending sorting order
31    if ( order == 1 )
32    {
33       selectionSort( a, arraySize, ascending );
34       cout << "\nData items in ascending order\n";
35    } // end if
36
37    // sort array in descending order; pass function descending
38    // as an argument to specify descending sorting order
39    else
40    {
41       selectionSort( a, arraySize, descending );
42       cout << "\nData items in descending order\n";
43    } // end else part of if...else
44
45    // output sorted array
46    for ( counter = 0; counter < arraySize; ++counter )
47       cout << setw( 4 ) << a[ counter ];
48
49    cout << endl;
50 } // end main
```

Fig. 8.20 | Multipurpose sorting program using function pointers. (Part 1 of 3.)

```

// multipurpose selection sort; the parameter compare is a pointer to
// the comparison function that determines the sorting order
void selectionSort( int work[], const int size,
                    bool (*compare)( int, int ) )
{
   int smallestOrLargest; // index of smallest (or largest) element

   // loop over size - 1 elements
   for ( int i = 0; i < size - 1; ++i )
   {
      smallestOrLargest = i; // first index of remaining vector

      // loop to find index of smallest (or largest) element
      for ( int index = i + 1; index < size; ++index )
         if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
            smallestOrLargest = index;

      swap( &work[ smallestOrLargest ], &work[ i ] );
   } // end if
} // end function selectionSort

// swap values at memory locations to which
// element1Ptr and element2Ptr point
void swap( int * const element1Ptr, int * const element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
} // end function swap

// determine whether element a is less than
// element b for an ascending order sort
bool ascending( int a, int b )
{
   return a < b; // returns true if a is less than b
} // end function ascending

// determine whether element a is greater than
// element b for a descending order sort
bool descending( int a, int b )
{
   return a > b; // returns true if a is greater than b
} // end function descending
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 8.20 | Multipurpose sorting program using function pointers. (Part 2 of 3.)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Fig. 8.20 | Multipurpose sorting program using function pointers. (Part 3 of 3.)

The following parameter appears in line 55 of `selectionSort`'s function header:

```
bool ( *compare )( int, int )
```

This parameter specifies a pointer to a function. The keyword `bool` indicates that the function being pointed to returns a `bool` value. The text `(*compare)` indicates the name of the pointer to the function (the `*` indicates that parameter `compare` is a pointer). The text `(int, int)` indicates that the function pointed to by `compare` takes two integer arguments. Parentheses are needed around `*compare` to indicate that `compare` is a pointer to a function. If we had not included the parentheses, the declaration would have been

```
bool *compare( int, int )
```

which declares a function that receives two `int`s and returns a pointer to a `bool` value.

The corresponding parameter in the function prototype of `selectionSort` (line 8) is

```
bool (*)( int, int )
```

Only types have been included. As always, for documentation purposes, you can include names that the compiler will ignore.

The function passed to `selectionSort` is called in line 66 as follows:

```
( *compare )( work[ smallestOrLargest ], work[ index ] )
```

Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to execute the function. The parentheses around `*compare` are necessary—if they were left out, the `*` operator would attempt to dereference the value returned from the function call. The call to the function could have been made without dereferencing the pointer, as in

```
compare( work[ smallestOrLargest ], work[ index ] )
```

which uses the pointer directly as the function name. We prefer the first method of calling a function through a pointer, because it explicitly illustrates that `compare` is a pointer to a function that is dereferenced to call the function. The second method of calling a function through a pointer makes it appear as though `compare` is the name of an actual function in the program. This may be confusing to a user of the program who would like to see the definition of function `compare` and finds that it isn't defined in the file. Chapter 22, Standard Template Library (STL), presents many common uses of function pointers.

8.13 Wrap-Up

In this chapter we provided a detailed introduction to pointers—variables that contain memory addresses as their values. We began by demonstrating how to declare and initialize pointers. You saw how to use the address operator (&) to assign the address of a variable to a pointer and the indirection operator (*) to access the data stored in the variable indirectly referenced by a pointer. We discussed passing arguments by reference using both pointer arguments and reference arguments.

You learned how to use `const` with pointers to enforce the principle of least privilege. We demonstrated using nonconstant pointers to nonconstant data, nonconstant pointers to constant data, constant pointers to nonconstant data, and constant pointers to constant data. We then used selection sort to demonstrate passing arrays and individual array elements by reference. We discussed the compile-time `sizeof` operator, which can be used to determine the sizes of data types and variables in bytes during program compilation.

We demonstrated how to use pointers in arithmetic and comparison expressions. You saw that pointer arithmetic can be used to jump from one element of an array to another. You learned how to use arrays of pointers, and more specifically string arrays (arrays of strings). We discussed function pointers, which enable you to pass functions as parameters. We briefly introduced pointer-based strings.

In the next chapter, we begin our deeper treatment of classes. You'll learn about the scope of a class's members, and how to keep objects in a consistent state. You'll also learn about using special member functions called constructors and destructors, which execute when an object is created and destroyed, respectively, and we'll discuss when constructors and destructors are called. In addition, we'll demonstrate using default arguments with constructors and using default memberwise assignment to assign one object of a class to another object of the same class. We'll also discuss the danger of returning a reference to a `private` data member of a class.

Summary

Section 8.2 Pointer Variable Declarations and Initialization

- Pointers are variables that contain as their values memory addresses of other variables.
- The declaration

```
int *ptr;
```

  declares `ptr` to be a pointer to a variable of type `int` and is read, "`ptr` is a pointer to `int`." The `*` as used here in a declaration indicates that the variable is a pointer.
- There are three values that can be used to initialize a pointer: `0`, `NULL` or an address of an object of the same type. The new C++ standard also provides the `nullptr` constant, which is preferred.
- The only integer that can be assigned to a pointer without casting is zero.

Section 8.3 Pointer Operators

- The `&` (address) operator (p. 332) obtains the memory address of its operand.
- The operand of the address operator must be a variable name (or another *lvalue*); the address operator cannot be applied to constants or to expressions that do not return a reference.
- The `*` indirection (or dereferencing) operator (p. 333) returns a synonym for the name of the object that its operand points to in memory. This is called dereferencing the pointer (p. 333).

Section 8.4 Pass-by-Reference with Pointers

- When calling a function with an argument that the caller wants the called function to modify, the address of the argument may be passed. The called function then uses the indirection operator (*) to dereference the pointer and modify the value of the argument in the calling function.

- A function receiving an address as an argument must have a pointer as its corresponding parameter.

Section 8.5 Using `const` with Pointers

- The `const` qualifier enables you to inform the compiler that the value of a particular variable cannot be modified through the specified identifier.
- There are four ways to pass a pointer to a function—a nonconstant pointer to nonconstant data (p. 340), a nonconstant pointer to constant data (p. 340), a constant pointer to nonconstant data (p. 341), and a constant pointer to constant data (p. 341).
- The value of the array name is the address of the array's first element.
- To pass a single array element by reference using pointers, pass the element's address.

Section 8.6 Selection Sort Using Pass-by-Reference

- The selection sort algorithm (p. 343) is an easy-to-program, but inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. After the i^{th} iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

Section 8.7 `sizeof` Operator

- `sizeof` (p. 347) determines the size in bytes of a type, variable or constant at compile time.
- When applied to an array name, `sizeof` returns the total number of bytes in the array.

Section 8.8 Pointer Expressions and Pointer Arithmetic

- The arithmetic operations that may be performed on pointers are incrementing (`++`) a pointer, decrementing (`--`) a pointer, adding (`+` or `+=`) an integer to a pointer, subtracting (`-` or `-=`) an integer from a pointer and subtracting one pointer from another.
- When an integer is added or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Pointers can be assigned to one another if they are of the same type. Otherwise, a cast must be used. The exception to this is a `void *` pointer, which is a generic pointer type that can hold pointer values of any type.
- The only valid operations on a `void *` pointer are comparing `void *` pointers with other pointers, assigning addresses to `void *` pointers and casting `void *` pointers to valid pointer types.
- Pointers can be compared using the equality and relational operators. Comparisons using relational operators are meaningful only if the pointers point to members of the same array.

Section 8.9 Relationship Between Pointers and Arrays

- Pointers that point to arrays can be subscripted exactly as array names can (p. 353).
- In pointer/offset notation (p. 352), if the pointer points to the first element of the array, the offset is the same as an array subscript.
- All subscripted array expressions can be written with a pointer and an offset (p. 352), using either the name of the array as a pointer or using a separate pointer that points to the array.

Section 8.10 Pointer-Based String Processing

- A character constant (p. 355) is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set.

- A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, /and $.
- String literals, or string constants, in C++ are written in double quotation marks (p. 355).
- A pointer-based string is an array of characters ending with a null character (`'\0'`; p. 355), which marks where the string terminates in memory. A string is accessed via a pointer to its first character.
- The value of a string literal is the address of its first character, but the `sizeof` a string literal is the length of the string including the terminating null character.
- A string literal may be used as an initializer for a character array or a variable of type `char *`.
- String literals have static storage class and may or may not be shared if the same string literal is referenced from multiple locations in a program.
- The effect modifying a string literal is undefined; thus, you should always declare a pointer to a string literal as `const char *`.
- When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character.
- If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.
- You can access individual characters in a string directly with array subscript notation.
- A string can be read into a character array using stream extraction with `cin`.
- The `setw` stream manipulator can be used to ensure that the string read into a character array does not exceed the size of the array.
- The `cin` object provides the member function `getline` (p. 356) to input an entire line of text into a character array. The function takes three arguments—a character array in which the line of text will be stored, a length and a delimiter character. The third argument has `'\n'` as a default value.
- A character array representing a null-terminated string can be output with `cout` and `<<`. The characters of the string are output until a terminating null character is encountered.

Section 8.11 Arrays of Pointers

- Arrays may contain pointers.
- Such a data structure can be used to form an array of pointer-based strings, referred to as a string array (p. 357). Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string.
- String arrays are commonly used with command-line arguments (p. 358) that are passed to `main` when a program begins execution.

Section 8.12 Function Pointers

- A pointer to a function (p. 358) is the address where the code for the function resides.
- Pointers to functions can be used to call the functions they point to, passed to functions, returned from functions, stored in arrays, assigned to other pointers.

Self-Review Exercises

8.1 Answer each of the following:

a) A pointer is a variable that contains as its value the ________ of another variable.

b) The three values that can be used to initialize a pointer are ________, ________ and ________.

c) The only integer that can be assigned directly to a pointer is ________.

8.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The address operator `&` can be applied only to constants and to expressions.
b) A pointer that is declared to be of type `void *` can be dereferenced.
c) A pointer of one type can't be assigned to one of another type without a cast operation.

8.3 For each of the following, write C++ statements that perform the specified task. Assume that double-precision, floating-point numbers are stored in eight bytes and that the starting address of the array is at location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

a) Declare an array of type `double` called `numbers` with 10 elements, and initialize the elements to the values `0.0`, `1.1`, `2.2`, …, `9.9`. Assume that the symbolic constant `SIZE` has been defined as `10`.
b) Declare a pointer `nPtr` that points to a variable of type `double`.
c) Use a `for` statement to print the elements of array `numbers` using array subscript notation. Print each number with one position of precision to the right of the decimal point.
d) Write two separate statements that each assign the starting address of array `numbers` to the pointer variable `nPtr`.
e) Use a `for` statement to print the elements of array `numbers` using pointer/offset notation with pointer `nPtr`.
f) Use a `for` statement to print the elements of array `numbers` using pointer/offset notation with the array name as the pointer.
g) Use a `for` statement to print the elements of array `numbers` using pointer/subscript notation with pointer `nPtr`.
h) Refer to the fourth element of array `numbers` using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.
i) Assuming that `nPtr` points to the beginning of array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?
j) Assuming that `nPtr` points to `numbers[5]`, what address is referenced by `nPtr` after `nPtr -= 4` is executed? What's the value stored at that location?

8.4 For each of the following, write a single statement that performs the specified task. Assume that floating-point variables `number1` and `number2` have been declared and that `number1` has been initialized to `7.3`. Assume that variable `ptr` is of type `char *`. Assume that arrays `s1` and `s2` are each 100-element `char` arrays that are initialized with string literals.

a) Declare the variable `fPtr` to be a pointer to an object of type `double`.
b) Assign the address of variable `number1` to pointer variable `fPtr`.
c) Print the value of the object pointed to by `fPtr`.
d) Assign the value of the object pointed to by `fPtr` to variable `number2`.
e) Print the value of `number2`.
f) Print the address of `number1`.
g) Print the address stored in `fPtr`. Is the value printed the same as the address of `number1`?

8.5 Perform the task specified by each of the following statements:

a) Write the function header for a function called `exchange` that takes two pointers to double-precision, floating-point numbers `x` and `y` as parameters and does not return a value.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function called `evaluate` that returns an integer and that takes as parameters integer `x` and a pointer to function `poly`. Function `poly` takes an integer parameter and returns an integer.
d) Write the function prototype for the function in part (c).
e) Write two statements that each initialize character array `vowel` with the string of vowels, `"AEIOU"`.

8.6 Find the error in each of the following program segments. Assume the following declarations and statements:

```
int *zPtr; // zPtr will reference array z
void *sPtr = 0;
int number;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
```

a) `++zPtr;`

b)
```
// use pointer to get first value of array
number = zPtr;
```

c)
```
// assign array element 2 (the value 3) to number
number = *zPtr[ 2 ];
```

d)
```
// print entire array z
for ( int i = 0; i <= 5; ++i )
   cout << zPtr[ i ] << endl;
```

e)
```
// assign the value pointed to by sPtr to number
number = *sPtr;
```

f) `++z;`

Answers to Self-Review Exercises

8.1 a) address. b) `0`, `NULL`, an address. c) `0`.

8.2 a) False. The operand of the address operator must be an *lvalue*; the address operator cannot be applied to constants or to expressions that do not result in references.

b) False. A pointer to `void` cannot be dereferenced. Such a pointer does not have a type that enables the compiler to determine the number of bytes of memory to dereference and the type of the data to which the pointer points.

c) False. Pointers of any type can be assigned to `void` pointers. Pointers of type `void` can be assigned to pointers of other types only with an explicit type cast.

8.3 a) `double numbers[ SIZE ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`

b) `double *nPtr;`

c)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int i = 0; i < SIZE; ++i )
    cout << numbers[ i ] << ' ';
```

d)
```
nPtr = numbers;
nPtr = &numbers[ 0 ];
```

e)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int j = 0; j < SIZE; ++j )
   cout << *( nPtr + j ) << ' ';
```

f)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int k = 0; k < SIZE; ++k )
   cout << *( numbers + k ) << ' ';
```

g)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int m = 0; m < SIZE; ++m )
   cout << nPtr[ m ] << ' ';
```

h)
```
numbers[ 3 ]
*( numbers + 3 )
nPtr[ 3 ]
*( nPtr + 3 )
```

i) The address is `1002500 + 8 * 8 = 1002564`. The value is `8.8`.

j) The address of `numbers[ 5 ]` is `1002500 + 5 * 8 = 1002540`.
The address of `nPtr -= 4` is `1002540 - 4 * 8 = 1002508`.
The value at that location is `1.1`.

8.4 a) `double *fPtr;`
b) `fPtr = &number1;`
c) `cout << "The value of *fPtr is " << *fPtr << endl;`
d) `number2 = *fPtr;`
e) `cout << "The value of number2 is " << number2 << endl;`
f) `cout << "The address of number1 is " << &number1 << endl;`
g) `cout << "The address stored in fPtr is " << fPtr << endl;`
Yes, the value is the same.

8.5 a) `void exchange( double *x, double *y )`
b) `void exchange( double *, double * );`
c) `int evaluate( int x, int (*poly)( int ) )`
d) `int evaluate( int, int (*)( int ) );`
e) `char vowel[] = "AEIOU";`
`char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };`

8.6 a) *Error:* `zPtr` has not been initialized.
Correction: Initialize `zPtr` with `zPtr = z;`
b) *Error:* The pointer is not dereferenced.
Correction: Change the statement to `number = *zPtr;`
c) *Error:* `zPtr[ 2 ]` is not a pointer and should not be dereferenced.
Correction: Change `*zPtr[ 2 ]` to `zPtr[ 2 ]`.
d) *Error:* Referring to an array element outside the array bounds with pointer subscripting.
Correction: To prevent this, change the relational operator in the `for` statement to `<` or change the `5` to a `4`.
e) *Error:* Dereferencing a void pointer.
Correction: To dereference the void pointer, it must first be cast to an integer pointer. Change the statement to `number = *static_cast< int * >( sPtr );`
f) *Error:* Trying to modify an array name with pointer arithmetic.
Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

Exercises

8.7 *(True or False)* State whether the following are *true* or *false*. If *false*, explain why.
a) Two pointers that point to different arrays cannot be compared meaningfully.
b) Because the name of an array is a pointer to the first element of the array, array names can be manipulated in precisely the same manner as pointers.

8.8 *(Write C++ Statements)* For each of the following, write C++ statements that perform the specified task. Assume that unsigned integers are stored in two bytes and that the starting address of the array is at location 1002500 in memory.
a) Declare an array of type `unsigned int` called `values` with five elements, and initialize the elements to the even integers from 2 to 10. Assume that the symbolic constant `SIZE` has been defined as `5`.
b) Declare a pointer `vPtr` that points to an object of type `unsigned int`.
c) Use a `for` statement to print the elements of array `values` using array subscript notation.
d) Write two separate statements that assign the starting address of array `values` to pointer variable `vPtr`.
e) Use a `for` statement to print the elements of array `values` using pointer/offset notation.

f) Use a `for` statement to print the elements of array `values` using pointer/offset notation with the array name as the pointer.
g) Use a `for` statement to print the elements of array `values` by subscripting the pointer to the array.
h) Refer to the fifth element of `values` using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation and pointer/offset notation.
i) What address is referenced by `vPtr + 3`? What value is stored at that location?
j) Assuming that `vPtr` points to `values[ 4 ]`, what address is referenced by `vPtr -= 4`? What value is stored at that location?

8.9 *(Write C++ Statements)* For each of the following, write a single statement that performs the specified task. Assume that `long` variables `value1` and `value2` have been declared and `value1` has been initialized to `200000`.
a) Declare the variable `longPtr` to be a pointer to an object of type `long`.
b) Assign the address of variable `value1` to pointer variable `longPtr`.
c) Print the value of the object pointed to by `longPtr`.
d) Assign the value of the object pointed to by `longPtr` to variable `value2`.
e) Print the value of `value2`.
f) Print the address of `value1`.
g) Print the address stored in `longPtr`. Is the value printed the same as `value1`'s address?

8.10 *(Function Headers and Prototypes)* Perform the task specified by each of the following statements:
a) Write the function header for function `zero` that takes a long integer array parameter `bigIntegers` and does not return a value.
b) Write the function prototype for the function in part (a).
c) Write the function header for function `add1AndSum` that takes an integer array parameter `oneTooSmall` and returns an integer.
d) Write the function prototype for the function described in part (c).

8.11 *(Find the Code Errors)* Find the error in each of the following segments. If the error can be corrected, explain how.

a)
```
int *number;
cout << number << endl;
```
b)
```
double *realPtr;
long *integerPtr;
integerPtr = realPtr;
```
c)
```
int * x, y;
x = y;
```
d)
```
char s[] = "this is a character array";
for ( ; *s != '\0'; ++s)
   cout << *s << ' ';
```
e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```
f)
```
double x = 19.34;
double xPtr = &x;
cout << xPtr << endl;
```

8.12 (*Simulation: The Tortoise and the Hare*) In this exercise, you'll re-create the classic race of the tortoise and the hare. You'll use random number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your program should use function `moveTortoise` and `moveHare` to adjust the position of the animals according to the rules in Fig. 8.21. These functions should use pointer-based pass-by-reference to modify the position of the tortoise and the hare.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 8.21 | Rules for moving the tortoise and the hare.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in the preceding table by producing a random integer i in the range $1 \le i \le 10$. For the tortoise, perform a "fast plod" when $1 \le i \le 5$, a "slip" when $6 \le i \le 7$ or a "slow plod" when $8 \le i \le 10$. Use a similar technique to move the hare.

Begin the race by printing

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

For each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing the letter `T` in the tortoise's position and the letter `H` in the hare's position. Occasionally, the contenders land on the same square. In this case, the tortoise bites the hare and your program should print `OUCH!!!` beginning at that position. All print positions other than the `T`, the `H` or the `OUCH!!!` (in case of a tie) should be blank.

After printing each line, test whether either animal has reached or passed square 70. If so, print the winner and terminate the simulation. If the tortoise wins, print `TORTOISE WINS!!! YAY!!!` If the hare wins, print `Hare wins. Yuch.` If both animals win on the same clock tick, you may want to favor the tortoise (the "underdog"), or you may want to print `It's a tie`. If neither animal wins, perform the loop again to simulate the next tick of the clock.

8.13 *(What Does This Code Do?)* What does this program do?

```
// Ex. 8.13: ex08_13.cpp
// What does this program do?
#include <iostream>
```

```
using namespace std;

void mystery1( char *, const char * ); // prototype

int main()
{
   char string1[ 80 ];
   char string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   mystery1( string1, string2 );
   cout << string1 << endl;
} // end main

// What does this function do?
void mystery1( char *s1, const char *s2 )
{
   while ( *s1 != '\0' )
      ++s1;

   for ( ; *s1 = *s2; ++s1, ++s2 )
      ; // empty statement
} // end function mystery1
```

8.14 *(What Does This Code Do?)* What does this program do?

```
// Ex. 8.14: ex08_14.cpp
// What does this program do?
#include <iostream>
using namespace std;

int mystery2( const char * ); // prototype

int main()
{
   char string1[ 80 ];

   cout << "Enter a string: ";
   cin >> string1;
   cout << mystery2( string1 ) << endl;
} // end main

// What does this function do?
int mystery2( const char *s )
{
   int x;

   for ( x = 0; *s != '\0'; ++s )
      ++x;

   return x;
} // end function mystery2
```

8.15 (*Quicksort*) You've previously seen the sorting techniques of the bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

a) *Partitioning Step:* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.
b) *Recursive Step:* Perform *Step 1* on each unsorted subarray.

Each time *Step 1* is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, that subarray must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

```
37 2 6 4 89 8 10 12 68 45
```

a) Starting from the rightmost element of the array, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The values now reside in the array as follows:

```
12   2   6   4   89   8   10   37   68   45
```

Element 12 is in italics to indicate that it was just swapped with **37**.

b) Starting from the left of the array, but beginning with the element after 12, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The values now reside in the array as follows:

```
12   2   6   4   37   8   10   89   68   45
```

c) Starting from the right, but beginning with the element before 89, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The values now reside in the array as follows:

```
12   2   6   4   10   8   37   89   68   45
```

d) Starting from the left, but beginning with the element after 10, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive function `quickSort` to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function `partition` should be called by `quickSort` to perform the partitioning step.

8.16 (***Maze Traversal***) The grid of hashes (`#`) and dots (`.`) in Fig. 8.22 is a two-dimensional array representation of a maze. In the two-dimensional array, the hashes represent the walls of the maze and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming that there is an exit). If there is not an exit, you'll arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you'll arrive at the exit of the maze. There may be a shorter path than the one you've taken, but you are guaranteed to get out of the maze if you follow the algorithm.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Fig. 8.22 | Two-dimensional array representation of a maze.

Write recursive function `mazeTraverse` to walk through the maze. The function should receive arguments that include a 12-by-12 character array representing the maze and the starting location of the maze. As `mazeTraverse` attempts to locate the exit from the maze, it should place the character `X` in each square in the path. The function should display the maze after each move, so the user can watch as the maze is solved.

8.17 (***Generating Mazes Randomly***) Write a function `mazeGenerator` that randomly produces a maze. The function should take as arguments a two-dimensional 12-by-12 character array and pointers to the `int` variables that represent the row and column of the maze's entry point. Try your function `mazeTraverse` from Exercise 8.16, using several randomly generated mazes.

Special Section: Building Your Own Computer

In the next several problems, we take a temporary diversion away from the world of high-level-language programming. We "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (using software-based *simulation*) on which you can execute your machine-language programs!

8.18 (***Machine-Language Programming***) Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but, as we'll soon see, it's a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator*—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of *words.* A word is a signed four-digit decimal number, such as `+3364`, `-1293`, `+0007`, `-0001`, etc. The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers `00`, `01`, …, `99`.

Before running an SML program, we must *load,* or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location `00`. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory; thus, instructions are signed four-digit decimal numbers. Assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the *operation code* that specifies the operation to be performed. SML operation codes are shown in Fig. 8.23.

The last two digits of an SML instruction are the *operand*—the address of the memory location containing the word to which the operation applies.

Operation code	Meaning
Input/output operations	
`const int READ = 10;`	Read a word from the keyboard into a specific location in memory.
`const int WRITE = 11;`	Write a word from a specific location in memory to the screen.
Load and store operations	
`const int LOAD = 20;`	Load a word from a specific location in memory into the accumulator.
`const int STORE = 21;`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations	
`const int ADD = 30;`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`const int SUBTRACT = 31;`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
`const int DIVIDE = 32;`	Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
`const int MULTIPLY = 33;`	Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).
Transfer-of-control operations	
`const int BRANCH = 40;`	Branch to a specific location in memory.
`const int BRANCHNEG = 41;`	Branch to a specific location in memory if the accumulator is negative.
`const int BRANCHZERO = 42;`	Branch to a specific location in memory if the accumulator is zero.
`const int HALT = 43;`	Halt—the program has completed its task.

Fig. 8.23 | Simpletron Machine Language (SML) operation codes.

Now let's consider two simple SML programs. The first (Fig. 8.24) reads two numbers from the keyboard and computes and prints their sum. The instruction `+1007` reads the first number from the keyboard and places it into location `07` (which has been initialized to zero). Instruction `+1008` reads the next number into location `08`. The *load* instruction, `+2007`, places (copies) the first number into the accumulator, and the *add* instruction, `+3008`, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, `+2109`, places (copies) the result back into memory location `09`. Then the *write* instruction, `+1109`, takes the number and prints it (as a signed four-digit decimal number). The *halt* instruction, `+4300`, terminates execution.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 8.24 | SML Example 1.

The SML program in Fig. 8.25 reads two numbers from the keyboard, then determines and prints the larger value. Note the use of the instruction `+4107` as a conditional transfer of control, much the same as C++'s `if` statement.

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 8.25 | SML Example 2.

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read positive numbers and compute and print their sum. Terminate input when a negative number is entered.
b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.
c) Read a series of numbers, and determine and print the largest number. The first number read indicates how many numbers should be processed.

8.19 (*Computer Simulator*) It may at first seem outrageous, but in this problem you are going to build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you actually will be able to run, test and debug the SML programs you wrote in Exercise 8.18.

When you run your Simpletron simulator, it should begin by printing

```
*** Welcome to Simpletron! ***

*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Your program should simulate the Simpletron's memory with a single-subscripted, 100-element array `memory`. Now assume that the simulator is running, and let's examine the dialog as we enter the program of the second example of Exercise 8.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Program loading completed ***
*** Program execution begins  ***
```

The numbers to the right of each `?` in the preceding dialog represent the SML program instructions input by the user.

The SML program has now been placed (or loaded) into array `memory`. Now the Simpletron executes your SML program. Execution begins with the instruction in location `00` and, like C++, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use variable `accumulator` to represent the accumulator register. Use variable `instructionCounter` to keep track of the location in memory that contains the instruction being performed. Use variable `operationCode` to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use variable `operand` to indicate the memory location on which the current instruction operates. Thus, `operand` is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in `operationCode`, and "pick off" the right two digits and place them in `operand`. When Simpletron begins execution, the special registers are all initialized to zero.

Now let's "walk through" the execution of the first SML instruction, `+1009` in memory location `00`. This is called an *instruction execution cycle.*

The `instructionCounter` tells us the location of the next instruction to be performed. We *fetch* the contents of that location from `memory` by using the C++ statement

```
instructionRegister = memory[ instructionCounter ];
```

The operation code and operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now, the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, etc.). A `switch` differentiates among the 12 operations of SML. In the `switch` statement, the behavior of various SML instructions is simulated as shown in Fig. 8.26 (we leave the others to you).

read:	`cin >> memory[ operand ];`
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
branch:	We'll discuss the branch instructions shortly.
halt:	This instruction prints the message `*** Simpletron execution terminated ***`

Fig. 8.26 | Behavior of SML instructions.

The *halt* instruction also causes the Simpletron to print the name and contents of each register, as well as the complete contents of memory. Such a printout is often called a *register and memory dump*. To help you program your dump function, a sample dump format is shown in Fig. 8.27. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. To format numbers with their sign as shown in the dump, use stream manipulator **`showpos`**. To disable the display of the sign, use stream manipulator **`noshowpos`**. For numbers that have fewer than four digits, you can format numbers with leading zeros between the sign and the value by using the following statement before outputting the value:

```
cout << setfill( '0' ) << internal;
```

```
REGISTERS:
accumulator          +0000
instructionCounter      00
instructionRegister  +0000
operationCode           00
operand                 00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 8.27 | A sample register and memory dump.

Parameterized stream manipulator **`setfill`** (from header `<iomanip>`) specifies the fill character that will appear between the sign and the value when a number is displayed with a field width of five characters but does not have four digits. (One position in the field width is reserved for the sign.) Stream manipulator **`internal`** indicates that the fill characters should appear between the sign and the numeric value .

Let's proceed with the execution of our program's first instruction—`+1009` in location `00`. As we've indicated, the `switch` statement simulates this by performing the C++ statement

```
cin >> memory[ operand ];
```

A question mark (`?`) should be displayed on the screen before the `cin` statement executes to prompt the user for input. The Simpletron waits for the user to type a value and press the *Enter* key. The value is then read into location `09`.

At this point, simulation of the first instruction is complete. All that remains is to prepare the Simpletron to execute the next instruction. The instruction just performed was not a transfer of control, so we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to execute.

Now let's consider how to simulate the branching instructions (i.e., the transfers of control). All we need to do is adjust the value in the `instructionCounter` appropriately. Therefore, the unconditional branch instruction (`40`) is simulated in the `switch` as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 )
   instructionCounter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 8.18. The variables that represent the Simpletron simulator's memory and registers should be defined in `main` and passed to other functions by value or by reference as appropriate.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron's `memory` must be in the range `-9999` to `+9999`. Your simulator should use a `while` loop to test that each number entered is in this range and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, accumulator overflows (i.e., arithmetic operations resulting in values larger than `+9999` or smaller than `-9999`) and the like. Such serious errors are called **fatal errors**. When a fatal error is detected, your simulator should print an error message such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should print a full register and memory dump in the format we've discussed previously. This will help the user locate the error in the program.

8.20 (***Project: Modifications to the Simpletron Simulator***) In Exercise 8.19, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron Simulator. In Exercises 20.31–20.35, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to SML. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler. [*Note:* Some modifications may conflict with others and therefore must be done separately.]

a) Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.
b) Allow the simulator to perform modulus calculations. This requires an additional Simpletron Machine Language instruction.
c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.
e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.
f) Modify the simulator to process floating-point values in addition to integer values.
g) Modify the simulator to handle string input. [*Hint:* Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that inputs a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half-word.]
h) Modify the simulator to handle output of strings stored in the format of part (g). [*Hint:* Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]
i) Modify the simulator to include instruction `SML_DEBUG` that prints a memory dump after each instruction executes. Give `SML_DEBUG` an operation code of `44`. The word `+4401` turns on debug mode, and `+4400` turns off debug mode.

9

Classes: A Deeper Look, Part 1

My object all sublime
I shall achieve in time.
—W. S. Gilbert

Is it a world to hide virtues in?
—William Shakespeare

Don't be "consistent," but be simply true.
—Oliver Wendell Holmes, Jr.

Objectives

In this chapter you'll learn:

- How to use a preprocessor wrapper to prevent multiple definition errors.
- To understand class scope and accessing class members via the name of an object, a reference to an object or a pointer to an object.
- To define constructors with default arguments.
- How destructors are used to perform "termination housekeeping" on an object before it's destroyed.
- When constructors and destructors are called and the order in which they're called.
- The logic errors that may occur when a `public` member function returns a reference to `private` data.
- To assign the data members of one object to those of another object by default memberwise assignment.

Outline

9.1 Introduction

In the preceding chapters, we introduced many basic terms and concepts of C++ object-oriented programming. We also discussed our program development methodology: We selected appropriate attributes and behaviors for each class and specified the manner in which objects of our classes collaborated with objects of C++ Standard Library classes to accomplish each program's overall goals.

In this chapter, we take a deeper look at classes. We use an integrated `Time` class case study in both this chapter and Chapter 10, Classes: A Deeper Look, Part 2 to demonstrate several class construction capabilities. We begin with a `Time` class that reviews several of the features presented in the preceding chapters. The example also demonstrates an important C++ software engineering concept—using a "preprocessor wrapper" in headers to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple definition errors.

Next, we discuss class scope and the relationships among class members. We demonstrate how client code can access a class's `public` members via three types of "handles"—the name of an object, a reference to an object or a pointer to an object. As you'll see, object names and references can be used with the dot (`.`) member selection operator to access a `public` member, and pointers can be used with the arrow (`->`) member selection operator.

We discuss access functions that can read or display data in an object. A common use of access functions is to test the truth or falsity of conditions—such functions are known as *predicate functions*. We also demonstrate the notion of a utility function (also called a *helper function*)—a `private` member function that supports the operation of the class's `public` member functions, but is not intended for use by clients of the class.

In the second `Time` class case study example, we demonstrate how to pass arguments to constructors and show how default arguments can be used in a constructor to enable client code to initialize objects using a variety of arguments. Next, we discuss a special member function called a *destructor* that's part of every class and is used to perform "termination housekeeping" on an object before the object is destroyed. We then demonstrate the *order* in which constructors and destructors are called, because your programs' correctness depends on using properly initialized objects that have not yet been destroyed.

Our last example of the `Time` class case study in this chapter shows a dangerous programming practice in which a member function returns a reference to `private` data. We discuss how this *breaks the encapsulation of a class* and allows client code to directly access an object's data. This last example shows that objects of the same class can be assigned to one another using default memberwise assignment, which copies the data members in the object on the right side of the assignment into the corresponding data members of the object on the left side of the assignment. The chapter concludes with a discussion of software reusability.

9.2 Time Class Case Study

Our first example (Figs. 9.1–9.3) creates class `Time` and a driver program that tests the class. We demonstrate an important C++ software engineering concept—using a "preprocessor wrapper" in headers to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple-definition errors.

```
1  // Fig. 9.1: Time.h
2  // Time class definition.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Fig. 9.1 | Time class definition.

Time Class Definition

The class definition (Fig. 9.1) contains prototypes (lines 13–16) for member functions `Time`, `setTime`, `printUniversal` and `printStandard`, and includes `private` integer members `hour`, `minute` and `second` (lines 18–20). Class `Time`'s `private` data members can be accessed *only* by its four `public` member functions. Chapter 12 introduces a third access specifier, `protected`, as we study inheritance and the part it plays in object-oriented programming.

Good Programming Practice 9.1

For clarity and readability, use each access specifier only once in a class definition. Place `public` *members first, where they're easy to locate.*

Software Engineering Observation 9.1

Each element of a class should have `private` *visibility unless it can be proven that the element needs* `public` *visibility. This is another example of the principle of least privilege.*

In Fig. 9.1, the class definition is enclosed in the following **preprocessor wrapper** (lines 6, 7 and 23):

```
// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

When we build larger programs, other definitions and declarations will also be placed in headers. The preceding preprocessor wrapper prevents the code between **`#ifndef`** (which means "if not defined") and **`#endif`** from being included if the name `TIME_H` has been defined. If the header has *not* been included previously in a file, the name `TIME_H` is defined by the **`#define`** directive and the header statements *are* included. If the header has been included previously, `TIME_H` *is defined* already and the header is *not* included again. Attempts to include a header multiple times (inadvertently) typically occur in large programs with many headers that may themselves include other headers.

Error-Prevention Tip 9.1

Use `#ifndef`*,* `#define` *and* `#endif` *preprocessor directives to form a preprocessor wrapper that prevents headers from being included more than once in a program.*

Good Programming Practice 9.2

By convention, use the name of the header in upper case with the period replaced by an underscore in the `#ifndef` *and* `#define` *preprocessor directives of a header.*

`Time` Class Member Functions

In Fig. 9.2, the `Time` constructor (lines 11–14) initializes the data members to 0—the universal-time equivalent of 12 AM. Invalid values cannot be stored in the data members of a `Time` object, because the constructor is called when the `Time` object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function `setTime` (discussed shortly). Finally, it's important to note that you can define several *overloaded constructors* for a class.

```
1  // Fig. 9.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept> // for invalid_argument exception class
```

Fig. 9.2 | `Time` class member-function definitions. (Part 1 of 2.)

```cpp
#include "Time.h" // include definition of class Time from Time.h

using namespace std;

// Time constructor initializes each data member to zero.
Time::Time()
{
   hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time
void Time::setTime( int h, int m, int s )
{
   // validate hour, minute and second
   if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
      ( s >= 0 && s < 60 ) )
   {
      hour = h;
      minute = m;
      second = s;
   } // end if
   else
      throw invalid_argument(
         "hour, minute and/or second was out of range" );
} // end function setTime

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 9.2 | `Time` class member-function definitions. (Part 2 of 2.)

With the exception of `static const int` *data members (which you saw in Chapter 7), a class's data members cannot be initialized where they're declared in the class body*—though this *will* be allowed in the next version of the C++ standard. It's strongly recommended that these data members be initialized by the class's constructor as *there is no default initialization for fundamental-type data members*. Data members can also be assigned values by `Time`'s *set* functions.

Common Programming Error 9.1

Attempting to initialize a non-`static` *data member of a class explicitly in the class definition is a syntax error.*

`Time` Class Member Function `setTime` and Throwing Exceptions

Function `setTime` (lines 17–30) is a `public` function that declares three `int` parameters and uses them to set the time. Lines 20–21 test each argument to determine whether the value is in range, and, if so, lines 23–25 assign the values to the `hour`, `minute` and `second` data members. The `hour` value must be greater than or equal to `0` and less than `24`, because universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both `minute` and `second` must be greater than or equal to `0` and less than `60`. For values outside these ranges, `setTime` **throws an exception** of type **`invalid_argument`** (lines 28–29), which notifies the client code that an invalid argument was received. As you learned in Section 7.11, you can use `try...catch` to catch exceptions and attempt to recover from them, which we'll do in Fig. 9.3. The **`throw` statement** (lines 28–29) creates a new object of type `invalid_argument`. The parentheses following the class name indicate a call to the `invalid_argument` constructor that allows us to specify a custom error message string. After the exception object is created, the `throw` statement immediately terminates function `setTime` and the exception is returned to the code that attempted to set the time. [*Note:* To avoid compilation errors with GNU C++, you may need to include header <stdexcept> to use class `invalid_argument`.]

`Time` Class Member Function `printUniversal`

Function `printUniversal` (lines 33–37 of Fig. 9.2) takes no arguments and outputs the time in universal-time format, consisting of three colon-separated pairs of digits for the hour, minute and second. For example, if the time were 1:30:07 PM, function `printUniversal` would return `13:30:07`. Line 35 uses parameterized stream manipulator **`setfill`** to specify the **fill character** that's displayed when an integer is output in a field wider than the number of digits in the value. By default, the fill characters appear to the left of the digits in the number. In this example, if the `minute` value is 2, it will be displayed as 02, because the fill character is set to zero (`'0'`). If the number being output fills the specified field, the fill character will *not* be displayed. Once the fill character is specified with `setfill`, it applies for all subsequent values that are displayed in fields wider than the value being displayed (i.e., `setfill` is a "sticky" setting). This is in contrast to `setw`, which applies only to the next value displayed (`setw` is a "nonsticky" setting).

> **Error-Prevention Tip 9.2**
>
> *Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it's no longer needed. Failure to do so may result in incorrectly formatted output later in a program. Chapter 15, Stream Input/Output, discusses how to reset the fill character and precision.*

`Time` Class Member Function `printStandard`

Function `printStandard` (lines 40–45) takes no arguments and outputs the date in standard-time format, consisting of the `hour`, `minute` and `second` values separated by colons and followed by an AM or PM indicator (e.g., `1:27:06 PM`). Like function `printUniversal`, function `printStandard` uses `setfill('0')` to format the `minute` and `second` as two digit values with leading zeros if necessary. Line 42 uses the conditional operator (`?:`) to determine the value of `hour` to be displayed—if the `hour` is `0` or `12` (AM or PM), it appears as 12; otherwise, the `hour` appears as a value from 1 to 11. The conditional operator in line 44 determines whether AM or PM will be displayed.

Defining Member Functions Outside the Class Definitiogn; Class Scope

Even though a member function declared in a class definition may be defined outside that class definition (and "tied" to the class via the *scope resolution operator*), that member function is still within that **class's scope**—that is, its name is known to other members of the class unless. We'll say more about class scope shortly.

If a member function is defined in the body of a class definition, the member function is implicitly declared inline. Remember that the compiler reserves the right not to inline any function.

> **Performance Tip 9.1**
>
> *Defining a member function* inside *the class definition inlines the member function (if the compiler chooses to do so). This can improve performance.*

> **Software Engineering Observation 9.2**
>
> *Defining a small member function inside the class definition does* not *promote the best software engineering, because clients of the class will be able to see the implementation of the function, and the client code must be recompiled if the function definition changes.*

> **Software Engineering Observation 9.3**
>
> *Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.*

Member Functions vs. Global Functions

The `printUniversal` and `printStandard` member functions take no arguments, because these member functions implicitly know that they're to print the data members of the particular `Time` object for which they're invoked. This can make member function calls more concise than conventional function calls in procedural programming.

> **Software Engineering Observation 9.4**
>
> *Using an object-oriented programming approach often simplifies function calls by reducing the number of parameters. This benefit of object-oriented programming derives from the fact that encapsulating data members and member functions within an object gives the member functions the right to access the data members.*

> **Software Engineering Observation 9.5**
>
> *Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or fewer arguments than function calls in non-object-oriented languages. Thus, the calls, the function definitions and the function prototypes are shorter. This improves many aspects of program development.*

> **Error-Prevention Tip 9.3**
>
> *The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.*

Using Class `Time`

Once defined, `Time` can be used as a type in declarations as follows:

```
Time sunset; // object of type Time
Time arrayOfTimes[ 5 ]; // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime; // pointer to a Time object
```

Figure 9.3 uses class `Time`. Line 10 instantiates a single object of class `Time` called `t`. When the object is instantiated, the `Time` constructor is called to initialize each `private` data member to `0`. Then, lines 14 and 16 print the time in universal and standard formats, respectively, to confirm that the members were initialized properly. Line 18 sets a new time by calling member function `setTime`, and lines 22 and 24 print the time again in both formats.

```
1  // Fig. 9.3: fig09_03.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t; // instantiate object t of class Time
11
12    // output Time object t's initial values
13    cout << "The initial universal time is ";
14    t.printUniversal(); // 00:00:00
15    cout << "\nThe initial standard time is ";
16    t.printStandard(); // 12:00:00 AM
17
18    t.setTime( 13, 27, 6 ); // change time
19
20    // output Time object t's new values
21    cout << "\n\nUniversal time after setTime is ";
22    t.printUniversal(); // 13:27:06
23    cout << "\nStandard time after setTime is ";
24    t.printStandard(); // 1:27:06 PM
25
26    // attempt to set the time with invalid values
27    try
28    {
29       t.setTime( 99, 99, 99 ); // all values out of range
30    } // end try
31    catch ( invalid_argument &e )
32    {
33       cout << "Exception: " << e.what() << endl << endl;
34    } // end catch
35
36    // output t's values after specifying invalid values
37    cout << "\n\nAfter attempting invalid settings:"
38       << "\nUniversal time: ";
```

Fig. 9.3 | Program to test class `Time`. (Part 1 of 2.)

```
39      t.printUniversal(); // 00:00:00
40      cout << "\nStandard time: ";
41      t.printStandard(); // 12:00:00 AM
42      cout << endl;
43   } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. 9.3 | Program to test class `Time`. (Part 2 of 2.)

Calling `setTime` with Invalid Values

To illustrate that method `setTime` validates its arguments, line 29 calls `setTime` with invalid arguments of `99` for the `hour`, `minute` and `second`. This statement is placed in a `try` block (lines 27–30) in case `setTime` throws an `invalid_argument` exception, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 31–34 and line 33 displays the exception's error message by calling its `what` member function. Lines 37–41 output the time again in both formats to confirm that `setTime` did not change the time when invalid arguments were supplied.

Looking Ahead to Composition and Inheritance

Often, classes do not have to be created "from scratch." Rather, they can include objects of other classes as members or they may be **derived** from other classes that provide attributes and behaviors the new classes can use. Such software reuse can greatly enhance productivity and simplify code maintenance. Including class objects as members of other classes is called **composition** (or **aggregation**) and is discussed in Chapter 10. Deriving new classes from existing classes is called **inheritance** and is discussed in Chapter 12.

Object Size

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

Performance Tip 9.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share *this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.*

9.3 Class Scope and Accessing Class Members

A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope. Nonmember functions are defined at *global namespace scope*.

Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, `public` class members are referenced through one of the **handles** on an object—an object name, a reference to an object or a pointer to an object. The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client. [We'll see in Chapter 10 that an *implicit handle* is inserted by the compiler on every reference to a data member or member function from within an object.]

Member functions of a class can be overloaded, but only by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function. This also applies to the class's constructors.

Variables declared in a member function have *local scope* and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is *hidden* by the block-scope variable in the local scope. Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (`::`). Hidden global variables can be accessed with the scope resolution operator (see Chapter 6).

The dot member selection operator (`.`) is preceded by an object's name or with a reference to an object to access the object's members. The **arrow member selection operator (`->`)** is preceded by a pointer to an object to access the object's members.

Figure 9.4 uses a simple class called `Count` (lines 7–24) with `private` data member `x` of type `int` (line 23), `public` member function `setX` (lines 11–14) and `public` member function `print` (lines 17–20) to illustrate accessing class members with the member-selection operators. For simplicity, we've included this small class in the same file as `main`. Lines 28–30 create three variables related to type `Count`—`counter` (a `Count` object), `counterPtr` (a pointer to a `Count` object) and `counterRef` (a reference to a `Count` object). Variable `counterRef` *refers to* `counter`, and variable `counterPtr` *points to* `counter`. In lines 33–34 and 37–38, note that the program invokes member functions `setX` and `print` by using the dot (`.`) member selection operator preceded by either the name of the object (`counter`) or a reference to the object (`counterRef`, which is an alias for `counter`). Similarly, lines 41–42 demonstrate that the program can invoke member functions `setX` and `print` by using a pointer (`countPtr`) and the arrow (`->`) member-selection operator.

```
1  // Fig. 9.4: fig09_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using namespace std;
5
```

Fig. 9.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 1 of 2.)

```
// class Count definition
class Count
{
public: // public data is dangerous
   // sets the value of private data member x
   void setX( int value )
   {
      x = value;
   } // end function setX

   // prints the value of private data member x
   void print()
   {
      cout << x << endl;
   } // end function print

private:
   int x;
}; // end class Count

int main()
{
   Count counter; // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter; // create reference to counter

   cout << "Set x to 1 and print using the object's name: ";
   counter.setX( 1 ); // set data member x to 1
   counter.print(); // call member function print

   cout << "Set x to 2 and print using a reference to an object: ";
   counterRef.setX( 2 ); // set data member x to 2
   counterRef.print(); // call member function print

   cout << "Set x to 3 and print using a pointer to an object: ";
   counterPtr->setX( 3 ); // set data member x to 3
   counterPtr->print(); // call member function print
} // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

Fig. 9.4 | Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object. (Part 2 of 2.)

9.4 Separating Interface from Implementation

In Chapter 3, we began by including a class's definition and member-function definitions in *one* file. We then demonstrated separating this code into *two* files—a header for the class definition (i.e., the class's *interface*) and a source code file for the class's member-function definitions (i.e., the class's *implementation*). Recall that this makes it easier to modify programs—

as far as clients of a class are concerned, changes in the class's implementation do not affect the client as long as the class's interface originally provided to the client remains unchanged.

Software Engineering Observation 9.6

Clients of a class do not need access to the class's source code in order to use the class. The clients do, however, need to be able to link to the class's object code (i.e., the compiled version of the class). This encourages independent software vendors (ISVs) to provide class libraries for sale or license. The ISVs provide in their products only the headers and the object modules. No proprietary information is revealed—as would be the case if source code were provided. The C++ user community benefits by having more ISV-produced class libraries available.

Actually, things are not quite this rosy. Headers do contain some portions of the implementation and hints about others. Inline member functions, for example, should be in a header, so that when the compiler compiles a client, the client can include the `inline` function definition in place. A class's `private` members are listed in the class definition in the header, so these members are visible to clients even though the clients may not access the `private` members. In Chapter 10, we show how to use a *proxy class* to hide even the `private` data of a class from clients of the class.

Software Engineering Observation 9.7

Information important to the interface of a class should be included in the header. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

9.5 Access Functions and Utility Functions

Access functions can read or display data. Another common use for access functions is to test the truth or falsity of conditions—such functions are often called **predicate functions.** An example of a predicate function would be an `isEmpty` function for any container class—a class capable of holding many objects, like a `vector`. A program might test `isEmpty` before attempting to read another item from the container object. An `isFull` predicate function might test a container-class object to determine whether it has no additional room. Useful predicate functions for our `Time` class might be `isAM` and `isPM`.

The program of Figs. 9.5–9.7 demonstrates the notion of a **utility function** (also called a **helper function**). A utility function is *not* part of a class's `public` interface; rather, it's a `private` member function that supports the operation of the class's other member functions. Utility functions are not intended to be used by clients of a class (but can be used by *friends* of a class, as we'll see in Chapter 10).

Class `SalesPerson` (Fig. 9.5) declares an array of 12 monthly sales figures (line 17) and the prototypes for the class's constructor and member functions that manipulate the array.

```
1  // Fig. 9.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
```

Fig. 9.5 | `SalesPerson` class definition. (Part 1 of 2.)

```cpp
#ifndef SALESPERSON_H
#define SALESPERSON_H

class SalesPerson
{
public:
   static const int monthsPerYear = 12; // months in one year
   SalesPerson(); // constructor
   void getSalesFromUser(); // input sales from keyboard
   void setSales( int, double ); // set sales for a specific month
   void printAnnualSales(); // summarize and print sales
private:
   double totalAnnualSales(); // prototype for utility function
   double sales[ monthsPerYear ]; // 12 monthly sales figures
}; // end class SalesPerson

#endif
```

Fig. 9.5 | `SalesPerson` class definition. (Part 2 of 2.)

In Fig. 9.6, the `SalesPerson` constructor (lines 9–13) initializes array `sales` to zero. The `public` member function `setSales` (lines 30–37) sets the sales figure for one month in array `sales`. The `public` member function `printAnnualSales` (lines 40–45) prints the total sales for the last 12 months. The `private` utility function `totalAnnualSales` (lines 48–56) totals the 12 monthly sales figures for the benefit of `printAnnualSales`. Member function `printAnnualSales` edits the sales figures into monetary format.

```cpp
1  // Fig. 9.6: SalesPerson.cpp
2  // SalesPerson class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "SalesPerson.h" // include SalesPerson class definition
6  using namespace std;
7
8  // initialize elements of array sales to 0.0
9  SalesPerson::SalesPerson()
10 {
11    for ( int i = 0; i < monthsPerYear; ++i )
12       sales[ i ] = 0.0;
13 } // end SalesPerson constructor
14
15 // get 12 sales figures from the user at the keyboard
16 void SalesPerson::getSalesFromUser()
17 {
18    double salesFigure;
19
20    for ( int i = 1; i <= monthsPerYear; ++i )
21    {
22       cout << "Enter sales amount for month " << i << ": ";
23       cin >> salesFigure;
```

Fig. 9.6 | `SalesPerson` class member-function definitions. (Part 1 of 2.)

```
      setSales( i, salesFigure );
   } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
   // test for valid month and amount values
   if ( month >= 1 && month <= monthsPerYear && amount > 0 )
      sales[ month - 1 ] = amount; // adjust for subscripts 0-11
   else // invalid month or amount value
      cout << "Invalid month or sales figure" << endl;
} // end function setSales

// print total annual sales (with the help of utility function)
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales

// private utility function to total annual sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < monthsPerYear; ++i ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales
```

Fig. 9.6 | SalesPerson class member-function definitions. (Part 2 of 2.)

In Fig. 9.7, notice that the application's `main` function includes only a simple sequence of member-function calls—there are no control statements. The logic of manipulating the `sales` array is completely encapsulated in class `SalesPerson`'s member functions.

Software Engineering Observation 9.8

A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of member-function calls—few, if any, control statements are needed. By contrast, it's common to have control statements in the implementation of a class's member functions.

```
// Fig. 9.7: fig09_07.cpp
// Utility function demonstration.
// Compile this program with SalesPerson.cpp

```

Fig. 9.7 | Utility function demonstration. (Part 1 of 2.)

```
// include SalesPerson class definition from SalesPerson.h
#include "SalesPerson.h"

int main()
{
   SalesPerson s; // create SalesPerson object s

   s.getSalesFromUser(); // note simple sequential code; there are
   s.printAnnualSales(); // no control statements in main
} // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Fig. 9.7 | Utility function demonstration. (Part 2 of 2.)

9.6 Time Class Case Study: Constructors with Default Arguments

The program of Figs. 9.8–9.10 enhances class `Time` to demonstrate how arguments are implicitly passed to a constructor. The constructor defined in Fig. 9.2 initialized `hour`, `minute` and `second` to `0` (i.e., midnight in universal time). Like other functions, constructors can specify *default arguments*. Line 13 of Fig. 9.8 declares the `Time` constructor to include default arguments, specifying a default value of zero for each argument passed to the constructor. In Fig. 9.9, lines 10–13 define the new version of the `Time` constructor that receives values for parameters `hour`, `minute` and `second` that will be used to initialize `private` data members `hour`, `minute` and `second`, respectively. Class `Time` provides *set* and *get* functions for each data member. The `Time` constructor now calls `setTime`, which calls the `setHour`, `setMinute` and `setSecond` functions to validate and assign values to the data members. The default arguments to the constructor ensure that, even if no values are provided in a constructor call, the constructor still initializes the data members. *A constructor that defaults all its arguments is also a default constructor—that is, a constructor that can be invoked with no arguments. There can be at most one default constructor per class.*

Software Engineering Observation 9.9

Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

```
// Fig. 9.8: Time.h
// Time class containing a constructor with default arguments.
// Member functions defined in Time.cpp.

// prevent multiple inclusions of header
#ifndef TIME_H
#define TIME_H

// Time abstract data type definition
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions
   void setTime( int, int, int ); // set hour, minute, second
   void setHour( int ); // set hour (after validation)
   void setMinute( int ); // set minute (after validation)
   void setSecond( int ); // set second (after validation)

   // get functions
   int getHour(); // return hour
   int getMinute(); // return minute
   int getSecond(); // return second

   void printUniversal(); // output time in universal-time format
   void printStandard(); // output time in standard-time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 9.8 | `Time` class containing a constructor with default arguments.

In Fig. 9.9, line 12 of the constructor calls member function `setTime` with the values passed to the constructor (or the default values). Function `setTime` calls `setHour` to ensure that the value supplied for `hour` is in the range 0–23, then calls `setMinute` and `setSecond` to ensure that the values for `minute` and `second` are each in the range 0–59. Functions `setHour` (lines 24–30), `setMinute` (lines 33–39) and `setSecond` (lines 42–48) each throw an exception if an out-of-range argument is received.

```
1 // Fig. 9.9: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 #include <iomanip>
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 1 of 3.)

```
#include <stdexcept>
#include "Time.h" // include definition of class Time from Time.h
using namespace std;

// Time constructor initializes each data member to zero
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second ); // validate and set time
} // end Time constructor

// set new Time value using universal time
void Time::setTime( int h, int m, int s )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime

// set hour value
void Time::setHour( int h )
{
   if ( h >= 0 && h < 24 )
      hour = h;
   else
      throw invalid_argument( "hour must be 0-23" );
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   if ( m >= 0 && m < 60 )
      minute = m;
   else
      throw invalid_argument( "minute must be 0-59" );
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   if ( s >= 0 && s < 60 )
      second = s;
   else
      throw invalid_argument( "second must be 0-59" );
} // end function setSecond

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

```

Fig. 9.9 | Time class member-function definitions including a constructor that takes arguments. (Part 2 of 3.)

```
// return minute value
int Time::getMinute()
{
   return minute;
} // end function getMinute

// return second value
int Time::getSecond()
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
      << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
      << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 9.9 | `Time` class member-function definitions including a constructor that takes arguments. (Part 3 of 3.)

Function `main` in Fig. 9.10 initializes five `Time` objects—one with all three arguments defaulted in the implicit constructor call (line 10), one with one argument specified (line 11), one with two arguments specified (line 12), one with three arguments specified (line 13) and one with three invalid arguments specified (line 38). The program displays each object in universal-time and standard-time formats. For `Time` object `t5` (line 38), the program displays an error message because the constructor arguments are out of range.

```
1  // Fig. 9.10: fig09_10.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Time.h" // include definition of class Time from Time.h
6  using namespace std;
7
8  int main()
9  {
10    Time t1; // all arguments defaulted
11    Time t2( 2 ); // hour specified; minute and second defaulted
12    Time t3( 21, 34 ); // hour and minute specified; second defaulted
13    Time t4( 12, 25, 42 ); // hour, minute and second specified
```

Fig. 9.10 | Constructor with default arguments. (Part 1 of 2.)

```cpp

   cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
   t1.printUniversal(); // 00:00:00
   cout << "\n  ";
   t1.printStandard(); // 12:00:00 AM

   cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
   t2.printUniversal(); // 02:00:00
   cout << "\n  ";
   t2.printStandard(); // 2:00:00 AM

   cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
   t3.printUniversal(); // 21:34:00
   cout << "\n  ";
   t3.printStandard(); // 9:34:00 PM

   cout << "\n\nt4: hour, minute and second specified\n  ";
   t4.printUniversal(); // 12:25:42
   cout << "\n  ";
   t4.printStandard(); // 12:25:42 PM

   // attempt to initialize t6 with invalid values
   try
   {
      Time t5( 27, 74, 99 ); // all bad values specified
   } // end try
   catch ( invalid_argument &e )
   {
      cout << "\n\nException while initializing t5: " << e.what() << endl;
   } // end catch
} // end main
```

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

Exception while initializing t5: hour must be 0-23
```

Fig. 9.10 | Constructor with default arguments. (Part 2 of 2.)

Notes Regarding Class `Time`'s Set *and* Get *Functions and Constructor*

`Time`'s *set* and *get* functions are called throughout the class's body. In particular, function `setTime` (lines 17–22 of Fig. 9.9) calls functions `setHour`, `setMinute` and `setSecond`, and functions `printUniversal` and `printStandard` call functions `getHour`, `getMinute` and `getSecond` in line 72–73 and lines 79–81, respectively. In each case, these functions could have accessed the class's `private` data directly. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory). If we made such a change, only the bodies of the functions that access the `private` data directly would need to change—in particular, the individual *set* and *get* functions for the `hour`, `minute` and `second`. There would be no need to modify the bodies of functions `setTime`, `printUniversal` or `printStandard`, because they do *not* access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, the `Time` constructor could be written to include a copy of the appropriate statements from function `setTime`. Doing so may be slightly more efficient, because the extra constructor call and call to `setTime` are eliminated. However, duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult. Having the `Time` constructor call `setTime` and having `setTime` call `setHour`, `setMinute` and `setSecond` enables us to limit the changes to code that validates the `hour`, `minute` or `second` to the corresponding *set* function. This reduces the likelihood of errors when altering the class's implementation. Also, the performance of the `Time` constructor and `setTime` can (possibly) be enhanced by explicitly declaring them `inline` or by defining them in the class definition (which implicitly inlines the function definition).

Software Engineering Observation 9.10

If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

Common Programming Error 9.2

A constructor can call other member functions of the class, such as set *or* get *functions, but because the constructor is initializing the object, the data members may not yet be initialized. Using data members before they have been properly initialized can cause logic errors.*

9.7 Destructors

A **destructor** is another type of special member function. The name of the destructor for a class is the **tilde character (~)** followed by the class name. This naming convention has intuitive appeal, because as we'll see in a later chapter, the tilde operator is the bitwise complement operator, and, in a sense, the destructor is the complement of the constructor.

A class's destructor is called *implicitly* when an object is destroyed. This occurs, for example, as an automatic object is destroyed when program execution leaves the scope in which that object was instantiated. *The destructor itself does not actually release the object's* *memory*—it performs **termination housekeeping** before the object's memory is reclaimed, so the memory may be reused to hold new objects.

Common Programming Error 9.3

It's a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even `void` cannot be specified), to return values from a destructor or to overload a destructor.

Even though destructors have not been provided for the classes presented so far, *every class has a destructor*. If you do not explicitly provide a destructor, the compiler creates an "empty" destructor. [*Note:* We'll see that such an implicitly created destructor does, in fact, perform important operations on objects that are created through composition (Chapter 10) and inheritance (Chapter 12).] In Chapter 11, we'll build destructors appropriate for classes whose objects contain dynamically allocated memory (e.g., for arrays and strings) or use other system resources (e.g., files on disk, which we study in Chapter 17). We discuss how to dynamically allocate and deallocate memory in Chapter 10.

9.8 When Constructors and Destructors Are Called

Constructors and destructors are called *implicitly* by the compiler. The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated. Generally, destructor calls are made in the *reverse order* of the corresponding constructor calls, but as we'll see in Figs. 9.11–9.13, the storage classes of objects can alter the order in which destructors are called.

Constructors and Destructors for Objects in Global Scope

Constructors are called for objects defined in global scope *before* any other function (including `main`) in that file begins execution (although the order of execution of global object constructors between files is *not* guaranteed). The corresponding destructors are called when `main` terminates. Function **`exit`** forces a program to terminate immediately and does *not* execute the destructors of automatic objects. The function often is used to terminate a program when an error is detected in the input or if a file to be processed by the program cannot be opened. Function **`abort`** performs similarly to function `exit` but forces the program to terminate *immediately*, without allowing the destructors of any objects to be called. Function `abort` is usually used to indicate an abnormal termination of the program. (See Appendix F for more information on functions `exit` and `abort`.)

Constructors and Destructors for Local Automatic Objects

The constructor for an automatic local object is called when execution reaches the point where that object is defined—the corresponding destructor is called when execution leaves the object's scope (i.e., the block in which that object is defined has finished executing). Constructors and destructors for automatic objects are called each time execution enters and leaves the scope of the object. Destructors are not called for automatic objects if the program terminates with a call to function `exit` or function `abort`.

Constructors and Destructors for `static` Local Objects

The constructor for a `static` local object is called only *once*, when execution first reaches the point where the object is defined—the corresponding destructor is called when `main` terminates or the program calls function `exit`. Global and `static` objects are destroyed in the *reverse* order of their creation. Destructors are *not* called for `static` objects if the program terminates with a call to function `abort`.

Demonstrating When Constructors and Destructors Are Called

The program of Figs. 9.11–9.13 demonstrates the order in which constructors and destructors are called for objects of class `CreateAndDestroy` (Fig. 9.11 and Fig. 9.12) of various storage classes in several scopes. Each object of class `CreateAndDestroy` contains an integer (`objectID`) and a `string` (`message`) that are used in the program's output to identify the object (Fig. 9.11 lines 16–17). This mechanical example is purely for pedagogic purposes. For this reason, line 21 of the destructor in Fig. 9.12 determines whether the object being destroyed has an `objectID` value `1` or `6` and, if so, outputs a newline character. This line makes the program's output easier to follow.

```
1  // Fig. 9.11: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using namespace std;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Fig. 9.11 | `CreateAndDestroy` class definition.

```
1  // Fig. 9.12: CreateAndDestroy.cpp
2  // CreateAndDestroy class member-function definitions.
3  #include <iostream>
4  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
5  using namespace std;
6
7  // constructor
8  CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
9  {
10    objectID = ID; // set object's ID number
11    message = messageString; // set object's descriptive message
12
```

Fig. 9.12 | `CreateAndDestroy` class member-function definitions. (Part 1 of 2.)

```
   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << "   destructor runs    "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

Fig. 9.12 | `CreateAndDestroy` class member-function definitions. (Part 2 of 2.)

Figure 9.13 defines object `first` (line 10) in global scope. Its constructor is actually called *before* any statements in `main` execute and its destructor is called at program termination *after* the destructors for all other objects have run.

Function `main` (lines 12–23) declares three objects. Objects `second` (line 15) and `fourth` (line 21) are local automatic objects, and object `third` (line 16) is a `static` local object. The constructor for each of these objects is called when execution reaches the point where that object is declared. The destructors for objects `fourth` then `second` are called (i.e., the *reverse* of the order in which their constructors were called) when execution reaches the end of `main`. Because object `third` is `static`, it exists until program termination. The destructor for object `third` is called *before* the destructor for global object `first`, but *after* all other objects are destroyed.

Function `create` (lines 26–33) declares three objects—`fifth` (line 29) and `seventh` (line 31) as local automatic objects, and `sixth` (line 30) as a `static` local object. The destructors for objects `seventh` then `fifth` are called (i.e., the *reverse* of the order in which their constructors were called) when `create` terminates. Because `sixth` is `static`, it exists until program termination. The destructor for `sixth` is called *before* the destructors for `third` and `first`, but *after* all other objects are destroyed.

```
1  // Fig. 9.13: fig09_13.cpp
2  // Demonstrating the order in which constructors and
3  // destructors are called.
4  #include <iostream>
5  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
6  using namespace std;
7
8  void create( void ); // prototype
9
10 CreateAndDestroy first( 1, "(global before main)" ); // global object
11
12 int main()
13 {
14    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
```

Fig. 9.13 | Order in which constructors and destructors are called. (Part 1 of 2.)

```
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

Fig. 9.13 | Order in which constructors and destructors are called. (Part 2 of 2.)

9.9 Time Class Case Study: A Subtle Trap—Returning a Reference to a private Data Member

A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value. One way to use this capability (unfortunately!)

is to have a `public` member function of a class return a reference to a `private` data member of that class. If a function returns a `const` reference, that reference cannot be used as a modifiable *lvalue*.

The program of Figs. 9.14–9.16 uses a simplified `Time` class (Fig. 9.14 and Fig. 9.15) to demonstrate returning a reference to a `private` data member with member function `badSetHour` (declared in Fig. 9.14 in line 15 and defined in Fig. 9.15 in lines 37–45). Such a reference return actually makes a call to member function `badSetHour` an alias for `private` data member `hour`! The function call can be used in any way that the `private` data member can be used, including as an *lvalue* in an assignment statement, thus *enabling clients of the class to clobber the class's* `private` *data at will!* The same problem would occur if a pointer to the `private` data were to be returned by the function.

```
1  // Fig. 9.14: Time.h
2  // Time class declaration.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

Fig. 9.14 | `Time` class declaration.

```
1  // Fig. 9.15: Time.cpp
2  // Time class member-function definitions.
3  #include <stdexcept>
4  #include "Time.h" // include definition of class Time
5  using namespace std;
6
7  // constructor function to initialize private data; calls member function
8  // setTime to set variables; default values are 0 (see class definition)
9  Time::Time( int hr, int min, int sec )
10 {
11    setTime( hr, min, sec );
12 } // end Time constructor
```

Fig. 9.15 | `Time` class member-function definitions. (Part 1 of 2.)

```cpp

// set values of hour, minute and second
void Time::setTime( int h, int m, int s )
{
   // validate hour, minute and second
   if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
      ( s >= 0 && s < 60 ) )
   {
      hour = h;
      minute = m;
      second = s;
   } // end if
   else
      throw invalid_argument(
         "hour, minute and/or second was out of range" );
} // end function setTime

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// POOR PRACTICE: Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   if ( hh >= 0 && hh < 24 )
      hour = hh;
   else
      throw invalid_argument( "hour must be 0-23" );

   return hour; // DANGEROUS reference return
} // end function badSetHour
```

Fig. 9.15 | `Time` class member-function definitions. (Part 2 of 2.)

Figure 9.16 declares `Time` object `t` (line 10) and reference `hourRef` (line 13), which is initialized with the reference returned by the call `t.badSetHour(20)`. Line 15 displays the value of the alias `hourRef`. This shows how `hourRef` *breaks the encapsulation of the class*—statements in `main` should not have access to the `private` data of the class. Next, line 16 uses the alias to set the value of `hour` to 30 (an invalid value) and line 17 displays the value returned by function `getHour` to show that assigning a value to `hourRef` actually modifies the `private` data in the `Time` object `t`. Finally, line 21 uses the `badSetHour` function call itself as an *lvalue* and assigns 74 (another invalid value) to the reference returned by the function. Line 26 again displays the value returned by function `getHour` to show that assigning a value to the result of the function call in line 21 modifies the `private` data in the `Time` object `t`.

Error-Prevention Tip 9.4

Returning a reference or a pointer to a `private` *data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data; this is a dangerous practice that should be avoided.*

```cpp
// Fig. 9.16: fig09_16.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>
#include "Time.h" // include definition of class Time
using namespace std;

int main()
{
   Time t; // create Time object

   // initialize hourRef with the reference returned by badSetHour
   int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in Time object t
   cout << "\nInvalid hour after modification: " << t.getHour();

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour( 12 ) = 74; // assign another invalid value to hour

   cout << "\n\n*************************************************\n"
      << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
      << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
      << t.getHour()
      << "\n*************************************************" << endl;
} // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

Fig. 9.16 | Returning a reference to a `private` data member.

9.10 Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by **memberwise assignment**—each data member of the object on the right of the assignment operator is assigned individually to the *same* data member in the object on the left of the assignment operator. Figures 9.17–9.18 define class `Date` for use in this example. Line 18 of Fig. 9.19 uses **default memberwise assignment** to assign the data members of `Date` object `date1` to the corresponding data members of `Date` object `date2`. In this case, the `month` member of `date1` is assigned to the `month` member of `date2`, the `day` member of `date1` is assigned to the `day` member of `date2` and the `year` member of `date1` is assigned to the `year` member of `date2`. [*Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory; we discuss these problems in Chapter 11 and show how to deal with them.] The `Date` constructor does not contain any error checking; we leave this to the exercises.

Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object. For each class, the compiler provides a default copy constructor that copies each member of the original object

```
// Fig. 9.17: Date.h
// Date class declaration.  Member functions are defined in Date.cpp.

// prevent multiple inclusions of header
#ifndef DATE_H
#define DATE_H

// class Date definition
class Date
{
public:
   Date( int = 1, int = 1, int = 2000 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
}; // end class Date

#endif
```

Fig. 9.17 | Date class declaration.

```
// Fig. 9.18: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

Fig. 9.18 | Date class member-function definitions.

```
// Fig. 9.19: fig09_19.cpp
// Demonstrating that class objects can be assigned
// to each other using default memberwise assignment.
#include <iostream>
#include "Date.h" // include definition of class Date from Date.h
using namespace std;

int main()
{
   Date date1( 7, 4, 2004 );
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1; // default memberwise assignment

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;
} // end main
```

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Fig. 9.19 | Default memberwise assignment.

into the corresponding member of the new object. Like memberwise assignment, copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory. Chapter 11 discusses how to define customized copy constructors that properly copy objects containing pointers to dynamically allocated memory.

Performance Tip 9.3

Passing an object by value is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-`const`*-reference is a safe, good-performing alternative (this can be implemented with a* `const` *reference parameter or with a pointer-to-*`const`*-data parameter).*

9.11 Wrap-Up

This chapter deepened our coverage of classes, using a rich `Time` class case study to introduce several new features. You saw that member functions are usually shorter than global functions because member functions can directly access an object's data members, so the member functions can receive fewer arguments than functions in procedural programming languages. You learned how to use the arrow operator to access an object's members via a pointer of the object's class type.

You learned that member functions have class scope—the member function's name is known only to the class's other members unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the scope resolution operator. We also discussed access functions (commonly used to retrieve the values of data members or to test the truth or falsity of conditions) and utility functions (`private` member functions that support the operation of the class's `public` member functions).

You learned that a constructor can specify default arguments that enable it to be called in a variety of ways. You also learned that any constructor that can be called with no arguments is a default constructor and that there can be at most one default constructor per class. We discussed destructors and their purpose of performing termination housekeeping on an object of a class before that object is destroyed. We also demonstrated the order in which an object's constructors and destructors are called.

We demonstrated the problems that can occur when a member function returns a reference to a `private` data member, which breaks the encapsulation of the class. We also showed that objects of the same type can be assigned to one another using default memberwise assignment. We also discussed the benefits of using class libraries to enhance the speed with which code can be created and to increase the quality of software.

Chapter 10 presents additional class features. We'll demonstrate how `const` can be used to indicate that a member function does not modify an object of a class. You'll build classes with composition, which allows a class to contain objects of other classes as members. We'll show how a class can allow so-called "friend" functions to access the class's non-`public` members. We'll also show how a class's non-`static` member functions can use a special pointer named `this` to access an object's members.

Summary

Section 9.2 `Time` *Class Case Study*

- Preprocessor directives `#ifndef` (which means "if not defined"; p. 382) and `#endif` (p. 382) are used to prevent multiple inclusions of a header. If the code between these directives has not previously been included in an application, `#define` (p. 382) defines a name that can be used to prevent future inclusions, and the code is included in the source code file.
- Data members cannot be initialized where they're declared in the class body (except for a class's `static const` data members of integral or `enum` types). Initialize these data members in the class's constructor (as there is no default initialization for data members of fundamental types).
- A class's functions can throw (p. 384) exceptions (such as `invalid_argument`; p. 384) to indicate invalid data.
- Stream manipulator `setfill` (p. 384) specifies the fill character (p. 384) that's displayed when an integer is output in a field that's wider than the number of digits in the value.
- By default, the fill characters appear before the digits in the number.
- Stream manipulator `setfill` is a "sticky" setting, meaning that once the fill character is set, it applies for all subsequent fields being printed.
- Even though a member function declared in a class definition may be defined outside that class definition (and "tied" to the class via the scope resolution operator), that member function is still within that class's scope.

- If a member function is defined in the body of a class definition, the member function is implicitly declared inline.
- Classes can include objects of other classes as members or they may be derived (p. 387) from other classes that provide attributes and behaviors the new classes can use.

Section 9.3 Class Scope and Accessing Class Members

- A class's data members and member functions belong to that class's scope.
- Nonmember functions are defined at global namespace scope.
- Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.
- Outside a class's scope, class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.
- Member functions of a class can be overloaded, but only by other member functions of that class.
- To overload a member function, provide in the class definition a prototype for each version of the overloaded function, and provide a separate definition for each version of the function.
- Variables declared in a member function have local scope and are known only to that function.
- If a member function defines a variable with the same name as a variable with class scope (p. 385), the class-scope variable is hidden by the block-scope variable in the local scope.
- The dot member selection operator (`.`) is preceded by an object's name or by a reference to an object to access the object's `public` members.
- The arrow member selection operator (`->`; p. 388) is preceded by a pointer to an object to access that object's `public` members.

Section 9.4 Separating Interface from Implementation

- Headers contain some portions of a class's implementation and hints about others. Inline member functions, for example, should be in a header, so that when the compiler compiles a client, the client can include the `inline` function definition in place.
- A class's `private` members that are listed in the class definition in the header are visible to clients, even though the clients may not access the `private` members.

Section 9.5 Access Functions and Utility Functions

- A utility function is a `private` member function that supports the operation of the class's `public` member functions. Utility functions are not intended to be used by clients of a class.

Section 9.6 `Time` Class Case Study: Constructors with Default Arguments

- Like other functions, constructors can specify default arguments.

Section 9.7 Destructors

- A class's destructor (p. 398) is called implicitly when an object of the class is destroyed.
- The name of the destructor for a class is the tilde (~) character followed by the class name.
- A destructor does not release an object's storage—it performs termination housekeeping (p. 399) before the system reclaims an object's memory, so the memory may be reused to hold new objects.
- A destructor receives no parameters and returns no value. A class may have only one destructor.
- If you do not explicitly provide a destructor, the compiler creates an "empty" destructor, so every class has exactly one destructor.

Section 9.8 When Constructors and Destructors Are Called

- The order in which constructors and destructors are called depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but the storage classes of objects can alter the order in which destructors are called.

Section 9.9 `Time` Class Case Study: A Subtle Trap—Returning a Reference to a `private` Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* that can receive a value.
- If the function returns a `const` reference, then the reference cannot be used as a modifiable *lvalue*.

Section 9.10 Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by memberwise assignment (p. 405).
- Objects may be passed by value to or returned by value from functions. C++ creates a new object and uses a copy constructor (p. 406) to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.

Self-Review Exercises

9.1 Fill in the blanks in each of the following:

a) Class members are accessed via the ________ operator in conjunction with the name of an object (or reference to an object) of the class or via the ________ operator in conjunction with a pointer to an object of the class.

b) Class members specified as ________ are accessible only to member functions of the class and friends of the class.

c) Class members specified as ________ are accessible anywhere an object of the class is in scope.

d) ________ can be used to assign an object of a class to another object of the same class.

9.2 Find the error(s) in each of the following and explain how to correct it (them):

a) Assume the following prototype is declared in class `Time`:

```
void ~Time( int );
```

b) The following is a partial definition of class `Time`:

```
class Time
{
public:
   // function prototypes

private:
   int hour = 0;
   int minute = 0;
   int second = 0;
}; // end class Time
```

c) Assume the following prototype is declared in class `Employee`:

```
int Employee( string, string );
```

Answers to Self-Review Exercises

9.1 a) dot (`.`), arrow (`->`). b) `private`. c) `public`. d) Default memberwise assignment (performed by the assignment operator).

9.2 a) *Error:* Destructors are not allowed to return values (or even specify a return type) or take arguments.
Correction: Remove the return type `void` and the parameter `int` from the declaration.
b) *Error:* Members cannot be explicitly initialized in the class definition.
Correction: Remove the explicit initialization from the class definition and initialize the data members in a constructor.
c) *Error:* Constructors are not allowed to return values.
Correction: Remove the return type `int` from the declaration.

Exercises

9.3 *(Scope Resolution Operator)* What's the purpose of the scope resolution operator?

9.4 *(Enhancing Class `Time`)* Provide a constructor that's capable of using the current time from the `time` and `localtime` functions—declared in the C++ Standard Library header `<ctime>`—to initialize an object of the `Time` class.

9.5 *(`Complex` Class)* Create a class called `Complex` for performing arithmetic with complex numbers. Write a program to test your class. Complex numbers have the form

`realPart + imaginaryPart *` i

where i is

$$\sqrt{-1}$$

Use `double` variables to represent the `private` data of the class. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided. Provide `public` member functions that perform the following tasks:

a) Adding two `Complex` numbers: The real parts are added together and the imaginary parts are added together.
b) Subtracting two `Complex` numbers: The real part of the right operand is subtracted from the real part of the left operand, and the imaginary part of the right operand is subtracted from the imaginary part of the left operand.
c) Printing `Complex` numbers in the form `(a, b)`, where `a` is the real part and `b` is the imaginary part.

9.6 *(`Rational` Class)* Create a class called `Rational` for performing arithmetic with fractions. Write a program to test your class.

Use integer variables to represent the `private` data of the class—the `numerator` and the `denominator`. Provide a constructor that enables an object of this class to be initialized when it's declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form. For example, the fraction

$$\frac{2}{4}$$

would be stored in the object as 1 in the `numerator` and 2 in the `denominator`. Provide `public` member functions that perform each of the following tasks:

a) Adding two `Rational` numbers. The result should be stored in reduced form.
b) Subtracting two `Rational` numbers. The result should be stored in reduced form.
c) Multiplying two `Rational` numbers. The result should be stored in reduced form.

d) Dividing two `Rational` numbers. The result should be stored in reduced form.
e) Printing `Rational` numbers in the form `a/b`, where `a` is the numerator and `b` is the denominator.
f) Printing `Rational` numbers in floating-point format.

9.7 *(Enhancing Class `Time`)* Modify the `Time` class of Figs. 9.8–9.9 to include a `tick` member function that increments the time stored in a `Time` object by one second. Write a program that tests the `tick` member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the `tick` member function works correctly. Be sure to test the following cases:
a) Incrementing into the next minute.
b) Incrementing into the next hour.
c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

9.8 *(Enhancing Class `Date`)* Modify the `Date` class of Figs. 9.17–9.18 to perform error checking on the initializer values for data members `month`, `day` and `year`. Also, provide a member function `nextDay` to increment the day by one. Write a program that tests function `nextDay` in a loop that prints the date during each iteration to illustrate that `nextDay` works correctly. Be sure to test the following cases:
a) Incrementing into the next month.
b) Incrementing into the next year.

9.9 *(Combining Class `Time` and Class `Date`)* Combine the modified `Time` class of Exercise 9.7 and the modified `Date` class of Exercise 9.8 into one class called `DateAndTime`. (In Chapter 12, we'll discuss inheritance, which will enable us to accomplish this task quickly without modifying the existing class definitions.) Modify the `tick` function to call the `nextDay` function if the time increments into the next day. Modify functions `printStandard` and `printUniversal` to output the date and time. Write a program to test the new class `DateAndTime`. Specifically, test incrementing the time into the next day.

9.10 *(Returning Error Indicators from Class `Time`'s* set *Functions)* Modify the *set* functions in the `Time` class of Figs. 9.8–9.9 to return appropriate error values if an attempt is made to *set* a data member of an object of class `Time` to an invalid value. Write a program that tests your new version of class `Time`. Display error messages when *set* functions return error values.

9.11 *(`Rectangle` Class)* Create a class `Rectangle` with attributes `length` and `width`, each of which defaults to 1. Provide member functions that calculate the `perimeter` and the `area` of the rectangle. Also, provide *set* and *get* functions for the `length` and `width` attributes. The *set* functions should verify that `length` and `width` are each floating-point numbers larger than 0.0 and less than 20.0.

9.12 *(Enhancing Class `Rectangle`)* Create a more sophisticated `Rectangle` class than the one you created in Exercise 9.11. This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a *set* function that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x- or y-coordinate larger than 20.0. The *set* function also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide member functions that calculate the `length`, `width`, `perimeter` and `area`. The length is the larger of the two dimensions. Include a predicate function `square` that determines whether the rectangle is a square.

9.13 *(Enhancing Class `Rectangle`)* Modify class `Rectangle` from Exercise 9.12 to include a `draw` function that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Include a `setFillCharacter` function to specify the character out of which the body of the rectangle will be drawn. Include a `setPerimeterCharacter` function to specify the character that will be used to draw the border of the rectangle. If you feel ambitious, you might include functions to scale the size of the rectangle, rotate it, and move it around within the designated portion of the first quadrant.

9.14 *(`HugeInteger` Class)* Create a class `HugeInteger` that uses a 40-element array of digits to store integers as large as 40 digits each. Provide member functions `input`, `output`, `add` and `subtract`. For comparing `HugeInteger` objects, provide functions `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`—each of these is a "predicate" function that simply returns `true` if the relationship holds between the two `HugeInteger`s and returns `false` if the relationship does not hold. Also, provide a predicate function `isZero`. If you feel ambitious, provide member functions `multiply`, `divide` and `modulus`.

9.15 *(`TicTacToe` Class)* Create a class `TicTacToe` that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as `private` data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board. [*Caution:* This is an extremely challenging project that could take many weeks of effort!]

10 Classes: A Deeper Look, Part 2

But what, to serve our private ends,
Forbids the cheating of our friends?
—Charles Churchill

Instead of this absurd division into sexes they ought to class people as static and dynamic.
—Evelyn Waugh

Have no friends not equal to yourself.
—Confucius

Objectives

In this chapter you'll learn:

- To specify `const` (constant) objects and `const` member functions.
- To create objects composed of other objects.
- To use `friend` functions and `friend` classes.
- To use the `this` pointer.
- To use `static` data members and member functions.
- The concept of a container class.
- To use proxy classes to hide implementation details from a class's clients.

Outline

10.1 Introduction

In this chapter, we continue our study of classes with several more advanced topics. We use `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege. We discuss *composition*—a form of reuse in which a class can have objects of other classes as members. Next, we introduce *friendship*, which enables a class designer to specify nonmember functions that can access a class's non-`public` members—a technique that is often used in operator overloading (Chapter 11) for performance reasons. We discuss a special pointer (called `this`), which is an *implicit* argument to each of a class's non-`static` member functions. It allows those member functions to access the correct object's data members and non-`static` member functions. We motivate the need for `static` class members and show how to use `static` data members and member functions in your own classes. Finally, we show how to create a proxy class to hide a class's implementation details (including its `private` data) from its clients.

10.2 `const` (Constant) Objects and `const` Member Functions

Let's see how the principle of least privilege applies to objects. Some objects need to be modifiable and some do not. You may use keyword `const` to specify that an object *is not* modifiable and that any attempt to modify the object should result in a compilation error. The statement

```
const Time noon( 12, 0, 0 );
```

declares a `const` object `noon` of class `Time` and initializes it to 12 noon.

Software Engineering Observation 10.1

Attempts to modify a `const` object are caught at compile time rather than causing execution-time errors.

Performance Tip 10.1

Declaring variables and objects `const` when appropriate can improve performance—compilers can perform optimizations on constants that cannot be performed on variables.

C++ disallows member function calls for `const` objects unless the member functions themselves are also declared `const`. This is true even for *get* member functions that do not modify the object.

A member function is specified as `const` *both* in its prototype by inserting the keyword `const` after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

Common Programming Error 10.1

Defining as `const` a member function that modifies a data member of the object is a compilation error.

Common Programming Error 10.2

Defining as `const` a member function that calls a non-`const` member function of the class on the same object is a compilation error.

Common Programming Error 10.3

Invoking a non-`const` member function on a `const` object is a compilation error.

Software Engineering Observation 10.2

A `const` member function can be overloaded with a non-`const` version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is `const`, the compiler uses the `const` version. If the object is not `const`, the compiler uses the non-`const` version.

An interesting problem arises for constructors and destructors, each of which typically modifies objects. A constructor *must* be allowed to modify an object so that the object can be initialized properly. A destructor must be able to perform its termination housekeeping chores before an object's memory is reclaimed by the system.

Common Programming Error 10.4

Attempting to declare a constructor or destructor `const` is a compilation error.

Defining and Using `const` *Member Functions*

The program of Figs. 10.1–10.3 modifies class `Time` of Figs. 9.8–9.9 by making its *get* functions and `printUniversal` function `const`. In the header `Time.h` (Fig. 10.1), lines 19–21 and 24 now include keyword `const` after each function prototype's parameter list. The corresponding definition of each function in Fig. 10.2 (lines 53, 59, 65 and 71, respectively) also specifies keyword `const` after each function's parameter list.

```
1  // Fig. 10.1: Time.h
2  // Time class definition with const member functions.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
```

Fig. 10.1 | `Time` class definition with `const` member functions. (Part 1 of 2.)

```cpp
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions
   void setTime( int, int, int ); // set time
   void setHour( int ); // set hour
   void setMinute( int ); // set minute
   void setSecond( int ); // set second

   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard(); // print standard time (should be const)
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 10.1 | Time class definition with const member functions. (Part 2 of 2.)

```cpp
// Fig. 10.2: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include <stdexcept>
#include "Time.h" // include definition of class Time
using namespace std;

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second );
} // end Time constructor

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
   setHour( hour );
   setMinute( minute );
   setSecond( second );
} // end function setTime

```

Fig. 10.2 | Time class member-function definitions. (Part 1 of 3.)

```
// set hour value
void Time::setHour( int h )
{
   if ( h >= 0 && h < 24 )
      hour = h;
   else
      throw invalid_argument( "hour must be 0-23" );
} // end function setHour

// set minute value
void Time::setMinute( int m )
{
   if ( m >= 0 && m < 60 )
      minute = m;
   else
      throw invalid_argument( "minute must be 0-59" );
} // end function setMinute

// set second value
void Time::setSecond( int s )
{
   if ( s >= 0 && s < 60 )
      second = s;
   else
      throw invalid_argument( "second must be 0-59" );
} // end function setSecond

// return hour value
int Time::getHour() const // get functions should be const
{
   return hour;
} // end function getHour

// return minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

// return second value
int Time::getSecond() const
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

```

Fig. 10.2 | Time class member-function definitions. (Part 2 of 3.)

```
// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() // note lack of const declaration
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 10.2 | Time class member-function definitions. (Part 3 of 3.)

Figure 10.3 instantiates two `Time` objects—non-`const` object `wakeUp` (line 7) and `const` object `noon` (line 8). The program attempts to invoke non-`const` member functions `setHour` (line 13) and `printStandard` (line 20) on the `const` object `noon`. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects—a non-`const` member function on a non-`const` object (line 11), a `const` member function on a non-`const` object (line 15) and a `const` member function on a `const` object (lines 17–18). The error messages generated for non-`const` member functions called on a `const` object are shown in the output window.

```
1  // Fig. 10.3: fig10_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                             // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );  // non-const   non-const
12
13    noon.setHour( 12 );    // const       non-const
14
15    wakeUp.getHour();      // non-const   const
16
17    noon.getMinute();      // const       const
18    noon.printUniversal(); // const       const
19
20    noon.printStandard();  // const       non-const
21 } // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp8_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to 'Time &'
        Conversion loses qualifiers
C:\cpphtp8_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
```

Fig. 10.3 | const objects and const member functions.

A constructor must be a non-`const` *member function* (Fig. 10.2, lines 11–14), but it can still be used to initialize a `const` object (Fig. 10.3, line 8). The `Time` constructor's definition (Fig. 10.2, lines 11–14) shows that it calls another non-`const` member function—`setTime` (lines 17–22)—to perform the initialization of a `Time` object. Invoking a non-`const` member function from the constructor call as part of the initialization of a `const` object is allowed. The "`const`ness" of a `const` object is enforced from the time the constructor *completes* initialization of the object until that object's destructor is called.

Line 20 in Fig. 10.3 generates a compilation error even though member function `printStandard` of class `Time` *does not* modify the object on which it's invoked. The fact that a member function does not modify an object is not sufficient to indicate that the function is a constant function—the function must *explicitly* be declared `const`.

Initializing a `const` *Data Member with a Member Initializer*

The program of Figs. 10.4–10.6 introduces using **member initializer syntax**. All data members *can* be initialized using member initializer syntax, but `const` data members and data members that are references *must* be initialized using member initializers. Later in this chapter, we'll see that member objects must be initialized this way as well.

```
// Fig. 10.4: Increment.h
// Definition of class Increment.
#ifndef INCREMENT_H
#define INCREMENT_H

class Increment
{
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   // function addIncrement definition
   void addIncrement()
   {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment
private:
   int count;
   const int increment; // const data member
}; // end class Increment

#endif
```

Fig. 10.4 | `Increment` class definition containing non-`const` data member `count` and `const` data member `increment`.

```
// Fig. 10.5: Increment.cpp
// Member-function definitions for class Increment demonstrate using a
// member initializer to initialize a constant of a built-in data type.
```

Fig. 10.5 | Member initializer used to initialize a constant of a built-in data type. (Part 1 of 2.)

```cpp
4  #include <iostream>
5  #include "Increment.h" // include definition of class Increment
6  using namespace std;
7
8  // constructor
9  Increment::Increment( int c, int i )
10    : count( c ), // initializer for non-const member
11      increment( i ) // required initializer for const member
12 {
13    // empty body
14 } // end constructor Increment
15
16 // print count and increment values
17 void Increment::print() const
18 {
19    cout << "count = " << count << ", increment = " << increment << endl;
20 } // end function print
```

Fig. 10.5 | Member initializer used to initialize a constant of a built-in data type. (Part 2 of 2.)

```cpp
// Fig. 10.6: fig10_06.cpp
// Program to test class Increment.
#include <iostream>
#include "Increment.h" // include definition of class Increment
using namespace std;

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 1; j <= 3; ++j )
   {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   } // end for
} // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

Fig. 10.6 | Invoking an `Increment` object's `print` and `addIncrement` member functions.

The constructor definition (Fig. 10.5, lines 9–14) uses a **member initializer list** to initialize class `Increment`'s data members—non-`const` integer `count` and `const` integer `increment` (declared in lines 19–20 of Fig. 10.4). *Member initializers* appear between a constructor's parameter list and the left brace that begins the constructor's body. The member initializer list (Fig. 10.5, lines 10–11) is separated from the parameter list with a colon (`:`). Each member initializer consists of the data member name followed by parentheses containing the member's initial value. In this example, `count` is initialized with the value of constructor parameter `c` and `increment` is initialized with the value of constructor parameter `i`. Multiple member initializers are separated by commas. Also, the member initializer list executes *before* the body of the constructor executes.

Software Engineering Observation 10.3

A `const` object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared `const`, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.

Why Is Function `print` Declared `const`?

Function `print` (Fig. 10.5, lines 17–20) is declared `const`. It might seem strange to label this function `const`, because a program probably will never have a `const Increment` object. However, it's possible that a program will have a `const` reference to an `Increment` object or a pointer to `const` that points to an `Increment` object. Typically, this occurs when objects of class `Increment` are passed to functions or returned from functions. In these cases, only class `Increment`'s `const` member functions can be called through the reference or pointer. Thus, it's reasonable to declare function `print` as `const`—doing so prevents errors in these situations where an `Increment` object is treated as a `const` object.

Error-Prevention Tip 10.1

Declare as `const` all of a class's member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you'll have no intention of creating `const` objects of that class or accessing objects of that class through `const` references or pointers to `const`. Declaring such member functions `const` does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

Erroneously Attempting to Initialize a `const` Data Member with an Assignment

Figure 10.7 shows the compilation errors caused by attempting to initialize `const` data member `increment` with an assignment statement in the `Increment` constructor's body rather than with a member initializer.

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp8_examples\ch10\consterror\Increment.cpp(10) : error C2758:
   'Increment::increment' : must be initialized in constructor base/member
   initializer list
       C:\cpphtp8_examples\ch10\consterror\increment.h(20) : see
          declaration of 'Increment::increment'
C:\cpphtp8_examples\ch10\consterror\Increment.cpp(12) : error C2166:
   l-value specifies const object
```

Fig. 10.7 | Compilation errors generated by attempting to initialize a `const` data member in the constructor's body rather than in the member initializer list. (Part 1 of 2.)

GNU C++ compiler error messages:

```
Increment.cpp:9: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:12: error: assignment of read-only data-member
   'Increment::increment'
```

Fig. 10.7 | Compilation errors generated by attempting to initialize a `const` data member in the constructor's body rather than in the member initializer list. (Part 2 of 2.)

Common Programming Error 10.5

Not providing a member initializer for a `const` data member is a compilation error.

10.3 Composition: Objects as Members of Classes

An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class? Such a capability is called **composition** and is sometimes referred to as a ***has-a* relationship**—*a class can have objects of other classes as members.*

Software Engineering Observation 10.4

A common form of software reusability is composition, in which a class has objects of other classes as members.

Previously, we saw how to pass arguments to the constructor of an object we created in `main`. Now we show how *an object's constructor can pass arguments to member-object constructors via member initializers*.

Software Engineering Observation 10.5

*Member objects are constructed in the order in which they're declared in the class definition (not in the order they're listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called **host objects**) are constructed.*

The next program uses classes `Date` (Figs. 10.8–10.9) and `Employee` (Figs. 10.10–10.11) to demonstrate composition. Class `Employee`'s definition (Fig. 10.10) contains `private` data members `firstName`, `lastName`, `birthDate` and `hireDate`. Members `birthDate` and `hireDate` are `const` objects of class `Date`, which contains `private` data members `month`, `day` and `year`. The `Employee` constructor's header (Fig. 10.11, lines 10–11) specifies that the constructor has four parameters (`first`, `last`, `dateOfBirth` and `dateOfHire`). The first two parameters are passed via member initializers to the `string` class constructor. The last two are passed via member initializers to the `Date` class constructor.

```
// Fig. 10.8: Date.h
// Date class definition; Member functions defined in Date.cpp
#ifndef DATE_H
```

Fig. 10.8 | `Date` class definition. (Part 1 of 2.)

```
#define DATE_H

class Date
{
public:
   static const int monthsPerYear = 12; // number of months in a year
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   int month; // 1-12 (January-December)
   int day; // 1-31 based on month
   int year; // any year

   // utility function to check if day is proper for month and year
   int checkDay( int ) const;
}; // end class Date

#endif
```

Fig. 10.8 | Date class definition. (Part 2 of 2.)

```
// Fig. 10.9: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include <stdexcept>
#include "Date.h" // include Date class definition
using namespace std;

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
   if ( mn > 0 && mn <= monthsPerYear ) // validate the month
      month = mn;
   else
      throw invalid_argument( "month must be 1-12" );

   year = yr; // could validate yr
   day = checkDay( dy ); // validate the day

   // output Date object to show when its constructor is called
   cout << "Date object constructor for date ";
   print();
   cout << endl;
} // end Date constructor

// print Date object in form month/day/year
void Date::print() const
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

Fig. 10.9 | Date class member-function definitions. (Part 1 of 2.)

```cpp

// output Date object to show when its destructor is called
Date::~Date()
{
   cout << "Date object destructor for date ";
   print();
   cout << endl;
} // end ~Date destructor

// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
   static const int daysPerMonth[ monthsPerYear + 1 ] =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // determine whether testDay is valid for specified month
   if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
      return testDay;

   // February 29 check for leap year
   if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
      ( year % 4 == 0 && year % 100 != 0 ) ) )
      return testDay;

   throw invalid_argument( "Invalid day for current month and year" );
} // end function checkDay
```

Fig. 10.9 | Date class member-function definitions. (Part 2 of 2.)

```cpp
// Fig. 10.10: Employee.h
// Employee class definition showing composition.
// Member functions defined in Employee.cpp.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition
using namespace std;

class Employee
{
public:
   Employee( const string &, const string &,
      const Date &, const Date & );
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   string firstName; // composition: member object
   string lastName; // composition: member object
   const Date birthDate; // composition: member object
```

Fig. 10.10 | Employee class definition showing composition. (Part 1 of 2.)

```
   const Date hireDate; // composition: member object
}; // end class Employee

#endif
```

Fig. 10.10 | `Employee` class definition showing composition. (Part 2 of 2.)

```
1  // Fig. 10.11: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  #include "Date.h" // Date class definition
6  using namespace std;
7
8  // constructor uses member initializer list to pass initializer
9  // values to constructors of member objects
10 Employee::Employee( const string &first, const string &last,
11    const Date &dateOfBirth, const Date &dateOfHire )
12    : firstName( first ), // initialize firstName
13      lastName( last ), // initialize lastName
14      birthDate( dateOfBirth ), // initialize birthDate
15      hireDate( dateOfHire ) // initialize hireDate
16 {
17    // output Employee object to show when constructor is called
18    cout << "Employee object constructor: "
19       << firstName << ' ' << lastName << endl;
20 } // end Employee constructor
21
22 // print Employee object
23 void Employee::print() const
24 {
25    cout << lastName << ", " << firstName << "  Hired: ";
26    hireDate.print();
27    cout << "  Birthday: ";
28    birthDate.print();
29    cout << endl;
30 } // end function print
31
32 // output Employee object to show when its destructor is called
33 Employee::~Employee()
34 {
35    cout << "Employee object destructor: "
36       << lastName << ", " << firstName << endl;
37 } // end ~Employee destructor
```

Fig. 10.11 | `Employee` class member-function definitions, including constructor with a member initializer list.

`Employee` Constructor's Member Initializer List

The colon (`:`) following the constructor's header (Fig. 10.11, line 12) begins the member initializer list. The member initializers specify the `Employee` constructor parameters being passed to the constructors of the `string` and `Date` data members. Parameters `first`, `last`, `dateOfBirth` and `dateOfHire` are passed to the constructors for objects `firstName`'s (Fig. 10.11, line 12), `lastName` (Fig. 10.11, line 13), `birthDate` (Fig. 10.11, line 14) and `hireDate` (Fig. 10.11, line 15), respectively. Again, member initializers are separated by commas.

`Date` Class's Default Copy Constructor

As you study class `Date` (Fig. 10.8), notice that the class does not provide a constructor that receives a parameter of type `Date`. So, why can the `Employee` constructor's member initializer list initialize the `birthDate` and `hireDate` objects by passing `Date` object's to their `Date` constructors? As we mentioned in Chapter 9, the compiler provides each class with a *default copy constructor* that copies each data member of the constructor's argument object into the corresponding member of the object being initialized. Chapter 11 discusses how you can define customized copy constructors.

Testing Classes `Date` and `Employee`

Figure 10.12 creates two `Date` objects (lines 9–10) and passes them as arguments to the constructor of the `Employee` object created in line 11. Line 14 outputs the `Employee` object's data. When each `Date` object is created in lines 9–10, the `Date` constructor defined in lines 9–26 of Fig. 10.9 displays a line of output to show that the constructor was called (see the first two lines of the sample output). [*Note:* Line 11 of Fig. 10.12 causes two additional `Date` constructor calls that do not appear in the program's output. When each of

```
1  // Fig. 10.12: fig10_12.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
9     Date birth( 7, 24, 1949 );
10    Date hire( 3, 12, 1988 );
11    Employee manager( "Bob", "Blue", birth, hire );
12
13    cout << endl;
14    manager.print();
15 } // end main
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an `Employee` is constructed—two calls to the `string` class's constructor (lines 12–13 of Fig. 10.11), two calls to the `Date` class's default copy constructor (lines 14–15 of Fig. 10.11) and the call to the `Employee` class's constructor.

Fig. 10.12 | Demonstrating composition—an object with member objects.

the `Employee`'s `Date` member object's is initialized in the `Employee` constructor's member initializer list (Fig. 10.11, lines 14–15), the default copy constructor for class `Date` is called. Since this constructor is defined implicitly by the compiler, it does not contain any output statements to demonstrate when it's called.]

Class `Date` and class `Employee` each include a destructor (lines 33–38 of Fig. 10.9 and lines 33–37 of Fig. 10.11, respectively) that prints a message when an object of its class is destructed. This enables us to confirm in the program output that objects are constructed from the *inside out* and destroyed in the *reverse* order, from the *outside in* (i.e., the `Date` member objects are destroyed after the `Employee` object that contains them). Notice the last four lines in the output of Fig. 10.12. The last two lines are the outputs of the `Date` destructor running on `Date` objects `hire` (line 10) and `birth` (line 9), respectively. These outputs confirm that the three objects created in `main` are destructed in the *reverse* of the order in which they were constructed. The `Employee` destructor output is five lines from the bottom. The fourth and third lines from the bottom of the output window show the destructors running for the `Employee`'s member objects `hireDate` (Fig. 10.10, line 22) and `birthDate` (Fig. 10.10, line 21). These outputs confirm that the `Employee` object is destructed from the *outside in*—i.e., the `Employee` destructor runs first (output shown five lines from the bottom of the output window), then the member objects are destructed in the *reverse order* from which they were constructed. Class `string`'s destructor does not contain output statements, so we do not see the `firstName` and `lastName` objects being destructed. Again, Fig. 10.12's output did not show the constructors running for member objects `birthDate` and `hireDate`, because these objects were initialized with the default `Date` class copy constructors provided by the compiler.

What Happens When I Do Not Use the Member Initializer List?

If a member object is *not* initialized through a member initializer, the member object's default constructor will be called implicitly. Values, if any, established by the default constructor can be overridden by *set* functions. However, for complex initialization, this approach may require significant additional work and time.

Common Programming Error 10.6

A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

Performance Tip 10.2

Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects—once when the member object's default constructor is called and again when set *functions are called in the constructor body (or later) to initialize the member object.*

Software Engineering Observation 10.6

If a class member is an object of another class, making that member object `public` *does not violate the encapsulation and hiding of that member object's* `private` *members. But, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be* `private`*, like all other data members.*

10.4 friend Functions and friend Classes

A **friend function** of a class is defined outside that class's scope, yet has the right to access the non-public (and public) members of the class. Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.

Using friend functions can enhance performance. This section presents a mechanical example of how a friend function works. Later in the book, friend functions are used to overload operators for use with class objects (Chapter 11) and to create iterator classes (Chapter 20, Custom Templatized Data Structures). Objects of an iterator class can successively select items or perform an operation on items in a container class object. Objects of container classes can store items. Using friends is often appropriate when a member function cannot be used for certain operations, as we'll see in Chapter 11.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend. To declare all member functions of class ClassTwo as friends of class ClassOne, place a declaration of the form

```
friend class ClassTwo;
```

in the definition of class ClassOne.

Software Engineering Observation 10.7

Even though the prototypes for friend functions appear in the class definition, friends are not *member functions.*

Software Engineering Observation 10.8

Member access notions of private, protected and public are not relevant to friend declarations, so friend declarations can be placed anywhere in a class definition.

Good Programming Practice 10.1

Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Friendship is granted, *not* taken—i.e., for class B to be a friend of class A, class A must explicitly declare that class B is its friend. Also, the friendship relation is neither symmetric nor transitive; i.e., if class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A (again, friendship is not symmetric), that class C is a friend of class B (also because friendship is not symmetric), or that class A is a friend of class C (friendship is not transitive).

Software Engineering Observation 10.9

Some people in the OOP community feel that "friendship" corrupts information hiding and weakens the value of the object-oriented design approach. We will provide several examples of responsible friendship use.

Modifying a Class's private *Data with a Friend Function*

Figure 10.13 is a mechanical example in which we define friend function setX to set the private data member x of class Count. The friend declaration (line 9) appears first (by convention) in the class definition, even before public member functions are declared. Again, this friend declaration can appear *anywhere* in the class.

```
1  // Fig. 10.13: fig10_13.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count
8  {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
23 private:
24    int x; // data member
25 }; // end class Count
26
27 // function setX can modify private data of Count
28 // because setX is declared as a friend of Count (line 9)
29 void setX( Count &c, int val )
30 {
31    c.x = val; // allowed because setX is a friend of Count
32 } // end function setX
33
34 int main()
35 {
36    Count counter; // create Count object
37
38    cout << "counter.x after instantiation: ";
39    counter.print();
40
41    setX( counter, 8 ); // set x using a friend function
42    cout << "counter.x after call to setX friend function: ";
43    counter.print();
44 } // end main
```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 10.13 | Friends can access `private` members of a class.

Function `setX` (lines 29–32) is a C-style, stand-alone function—it isn't a member function of class `Count`. For this reason, when `setX` is invoked for object `counter`, line 41 passes `counter` as an argument to `setX` rather than using a handle (such as the name of the object) to call the function, as in

```
counter.setX( 8 );
```

If you remove the friend declaration in line 9, you'll receive error messages indicating that function `setX` cannot modify class `Count`'s `private` data member `x`.

As we mentioned, Fig. 10.13 is a mechanical example of using the `friend` construct. It would normally be appropriate to define function `setX` as a member function of class `Count`. It would also normally be appropriate to separate the program of Fig. 10.13 into three files:

1. A header (e.g., `Count.h`) containing the `Count` class definition, which in turn contains the prototype of `friend` function `setX`
2. An implementation file (e.g., `Count.cpp`) containing the definitions of class `Count`'s member functions and the definition of `friend` function `setX`
3. A test program (e.g., `fig10_15.cpp`) with `main`.

Overloaded **`friend`** ***Functions***

It's possible to specify overloaded functions as `friends` of a class. Each function intended to be a `friend` must be explicitly declared in the class definition as a `friend` of the class.

10.5 Using the `this` Pointer

We've seen that an object's member functions can manipulate the object's data. How do member functions know *which* object's data members to manipulate? Every object has access to its own address through a pointer called **`this`** (a C++ keyword). The `this` pointer is *not* part of the object itself—i.e., the memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object. Rather, the `this` pointer is passed (by the compiler) as an *implicit* argument to each of the object's non-`static` member functions. Section 10.6 introduces `static` class members and explains why the `this` pointer is *not* implicitly passed to `static` member functions.

Objects use the `this` pointer *implicitly* (as we've done to this point) or *explicitly* to reference their data members and member functions. The type of the `this` pointer depends on the type of the object and whether the member function in which `this` is used is declared `const`. For example, in a nonconstant member function of class `Employee`, the `this` pointer has type `Employee * const` (a constant pointer to a nonconstant `Employee` object). In a constant member function of the class `Employee`, the `this` pointer has the data type `const Employee * const` (a constant pointer to a constant `Employee` object).

The next example shows implicit and explicit use of the `this` pointer; later in this chapter and in Chapter 11, we show some substantial and subtle examples of using `this`.

Implicitly and Explicitly Using the **`this`** ***Pointer to Access an Object's Data Members***

Figure 10.14 demonstrates the implicit and explicit use of the `this` pointer to enable a member function of class `Test` to print the `private` data `x` of a `Test` object.

For illustration purposes, member function `print` (lines 24–36) first prints `x` by using the `this` pointer *implicitly* (line 27)—only the name of the data member is specified. Then `print` uses two different notations to access `x` through the `this` pointer—the arrow operator (`->`) off the `this` pointer (line 31) and the dot operator (`.`) off the dereferenced `this`

```
 1  // Fig. 10.14: fig10_14.cpp
 2  // Using the this pointer to refer to object members.
 3  #include <iostream>
 4  using namespace std;
 5
 6  class Test
 7  {
 8  public:
 9     Test( int = 0 ); // default constructor
10     void print() const;
11  private:
12     int x;
13  }; // end class Test
14
15  // constructor
16  Test::Test( int value )
17     : x( value ) // initialize x to value
18  {
19     // empty body
20  } // end constructor Test
21
22  // print x using implicit and explicit this pointers;
23  // the parentheses around *this are required
24  void Test::print() const
25  {
26     // implicitly use the this pointer to access the member x
27     cout << "        x = " << x;
28
29     // explicitly use the this pointer and the arrow operator
30     // to access the member x
31     cout << "\n  this->x = " << this->x;
32
33     // explicitly use the dereferenced this pointer and
34     // the dot operator to access the member x
35     cout << "\n(*this).x = " << ( *this ).x << endl;
36  } // end function print
37
38  int main()
39  {
40     Test testObject( 12 ); // instantiate and initialize testObject
41
42     testObject.print();
43  } // end main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 10.14 | using the `this` pointer to refer to object members.

pointer (line 35). Note the parentheses around `*this` (line 35) when used with the dot member selection operator (`.`). The parentheses are required because the dot operator has higher precedence than the `*` operator. Without the parentheses, the expression `*this.x` would be evaluated as if it were parenthesized as `*( this.x )`, which is a compilation error, because the dot operator cannot be used with a pointer.

Common Programming Error 10.7

Attempting to use the member selection operator (.) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an lvalue *such as an object's name, a reference to an object or a dereferenced pointer to an object.*

One interesting use of the `this` pointer is to prevent an object from being assigned to itself. As we'll see in Chapter 11, *self-assignment* can cause serious errors when the object contains pointers to dynamically allocated storage.

Using the `this` Pointer to Enable Cascaded Function Calls

Another use of the `this` pointer is to enable **cascaded member-function calls**—that is, invoking multiple functions in the same statement (as in line 12 of Fig. 10.17). The program of Figs. 10.15–10.17 modifies class `Time`'s *set* functions `setTime`, `setHour`, `setMinute` and `setSecond` such that each returns a reference to a `Time` object to enable cascaded member-function calls. Notice in Fig. 10.16 that the last statement in the body of each of these member functions returns `*this` (lines 22, 33, 44 and 55) into a return type of `Time &`.

```
// Fig. 10.15: Time.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions (the Time & return types enable cascading)
   Time &setTime( int, int, int ); // set hour, minute, second
   Time &setHour( int ); // set hour
   Time &setMinute( int ); // set minute
   Time &setSecond( int ); // set second

   // get functions (normally declared const)
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   // print functions (normally declared const)
   void printUniversal() const; // print universal time
   void printStandard() const; // print standard time
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
```

Fig. 10.15 | `Time` class modified to enable cascaded member-function calls. (Part 1 of 2.)

```
   int second; // 0 - 59
}; // end class Time

#endif
```

Fig. 10.15 | `Time` class modified to enable cascaded member-function calls. (Part 2 of 2.)

```
// Fig. 10.16: Time.cpp
// Time class member-function definitions.
#include <iostream>
#include <iomanip>
#include "Time.h" // Time class definition
using namespace std;

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this; // enables cascading
} // end function setTime

// set hour value
Time &Time::setHour( int h ) // note Time & return
{
   if ( h >= 0 && h < 24 )
      hour = h;
   else
      throw invalid_argument( "hour must be 0-23" );

   return *this; // enables cascading
} // end function setHour

// set minute value
Time &Time::setMinute( int m ) // note Time & return
{
   if ( m >= 0 && m < 60 )
      minute = m;
   else
      throw invalid_argument( "minute must be 0-59" );

```

Fig. 10.16 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 1 of 2.)

```
   return *this; // enables cascading
} // end function setMinute

// set second value
Time &Time::setSecond( int s ) // note Time & return
{
   if ( s >= 0 && s < 60 )
      second = s;
   else
      throw invalid_argument( "second must be 0-59" );

   return *this; // enables cascading
} // end function setSecond

// get hour value
int Time::getHour() const
{
   return hour;
} // end function getHour

// get minute value
int Time::getMinute() const
{
   return minute;
} // end function getMinute

// get second value
int Time::getSecond() const
{
   return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
      << ":" << setfill( '0' ) << setw( 2 ) << minute
      << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

Fig. 10.16 | `Time` class member-function definitions modified to enable cascaded member-function calls. (Part 2 of 2.)

The program of Fig. 10.17 creates `Time` object `t` (line 9), then uses it in *cascaded member-function calls* (lines 12 and 24). Why does the technique of returning `*this` as a reference work? The dot operator (`.`) associates from left to right, so line 12 first evaluates

```cpp
1 // Fig. 10.17: fig10_17.cpp
2 // Cascading member-function calls with the this pointer.
3 #include <iostream>
4 #include "Time.h" // Time class definition
5 using namespace std;
6
7 int main()
8 {
9    Time t; // create Time object
10
11    // cascaded function calls
12    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14    // output time in universal and standard formats
15    cout << "Universal time: ";
16    t.printUniversal();
17
18    cout << "\nStandard time: ";
19    t.printStandard();
20
21    cout << "\n\nNew standard time: ";
22
23    // cascaded function calls
24    t.setTime( 20, 20, 20 ).printStandard();
25    cout << endl;
26 } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Fig. 10.17 | Cascading member-function calls with the `this` pointer.

`t.setHour(18)`, then returns a reference to object `t` as the value of this function call. The remaining expression is then interpreted as

```cpp
t.setMinute( 30 ).setSecond( 22 );
```

The `t.setMinute( 30 )` call executes and returns a reference to the object `t`. The remaining expression is interpreted as

```cpp
t.setSecond( 22 );
```

Line 24 also uses cascading. The calls must appear in the order shown in line 24, because `printStandard` as defined in the class does *not* return a reference to `t`. Placing the call to `printStandard` before the call to `setTime` in line 24 results in a compilation error. Chapter 11 presents several practical examples of using cascaded function calls. One such example uses multiple << operators with `cout` to output multiple values in a single statement.

10.6 `static` Class Members

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only *one* copy of a variable should be *shared* by *all* objects of a class. A **`static` data member** is used for these and other reasons. Such a variable represents "class-wide" information (i.e., a property that is shared by all instances and is not specific to any one object of the class). Recall that the versions of class `GradeBook` in Chapter 7 use `static` data members to store constants representing the number of grades that all `GradeBook` objects can hold.

Motivating Class-Wide Data

Let's further motivate the need for `static` class-wide data with an example. Suppose that we have a video game with `Martians` and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that there are at least five `Martians` present. If fewer than five are present, each `Martian` becomes cowardly. So each `Martian` needs to know the `martianCount`. We could endow each instance of class `Martian` with `martianCount` as a data member. If we do, every `Martian` will have a *separate* copy of the data member. Every time we create a new `Martian`, we'll have to update the data member `martianCount` in all `Martian` objects. Doing this would require every `Martian` object to have, or have access to, handles to all other `Martian` objects in memory. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare `martianCount` to be `static`. This makes `martianCount` class-wide data. Every `Martian` can access `martianCount` as if it were a data member of the `Martian`, but only one copy of the `static` variable `martianCount` is maintained by C++. This saves space. We save time by having the `Martian` constructor increment `static` variable `martianCount` and having the `Martian` destructor decrement `martianCount`. Because there's only one copy, we do not have to increment or decrement separate copies of `martianCount` for each `Martian` object.

Performance Tip 10.3

Use `static` data members to save storage when a single copy of the data for all objects of a class will suffice.

Scope and Initialization of `static` Data Members

Although they may seem like global variables, a class's `static` data members have class scope. Also, `static` members can be declared `public`, `private` or `protected`. A fundamental-type `static` data member is initialized by default to `0`. If you want a different initial value, a `static` data member can be initialized *once*. A `static const` data member of `int` or `enum` type can be initialized in its declaration in the class definition. However, all other `static` data members must be defined *at global namespace scope* (i.e., outside the body of the class definition) and can be initialized only in those definitions—again, the next version of the C++ standard will allow initialization where these variables are declared in the class definition. If a `static` data member is an object of a class that provides a default constructor, the `static` data member need not be initialized because its default constructor will be called.

Accessing `static` Data Members

A class's `private` and `protected` `static` members are normally accessed through the class's `public` member functions or `friends`. *A class's `static` members exist even when no objects of that class exist.* To access a `public static` class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (`::`) to the name of the data member. For example, if our preceding variable `martianCount` is `public`, it can be accessed with the expression `Martian::martianCount` when there are no `Martian` objects. (Of course, using `public` data is discouraged.)

To access a `private` or `protected` `static` class member when *no* objects of the class exist, provide a `public` **`static` member function** and call the function by prefixing its name with the class name and scope resolution operator. A `static` member function is a service of the *class*, *not* of a specific object of the class.

Software Engineering Observation 10.10

A class's `static` data members and `static` member functions exist and can be used even if no objects of that class have been instantiated.

Demonstrating `static` Data Members

The program of Figs. 10.18–10.20 demonstrates a `private static` data member called `count` (Fig. 10.18, line 25) and a `public static` member function called `getCount` (Fig. 10.18, line 19). In Fig. 10.19, line 8 defines and initializes the data member `count` to zero *at global namespace scope* and lines 12–15 define `static` member function `getCount`. Notice that neither line 8 nor line 12 includes keyword `static`, yet both lines refer to `static` class members. When `static` is applied to an item at global namespace scope, that item becomes known only in that file. The `static` class members need to be available to any client code that uses the class, so we declare them `static` only in the `.h` file. Data member `count` maintains a count of the number of objects of class `Employee` that have been instantiated. When objects of class `Employee` exist, member `count` can be referenced through any member function of an `Employee` object—in Fig. 10.19, `count` is referenced by both line 22 in the constructor and line 32 in the destructor.

Common Programming Error 10.8

It's a compilation error to include keyword `static` in the definition of a `static` data member at global namespace scope.

```
1  // Fig. 10.18: Employee.h
2  // Employee class definition with a static data member to
3  // track the number of Employee objects in memory
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  using namespace std;
9
10 class Employee
11 {
12 public:
13    Employee( const string &, const string & ); // constructor
14    ~Employee(); // destructor
15    string getFirstName() const; // return first name
16    string getLastName() const; // return last name
```

Fig. 10.18 | `Employee` class definition with a static data member to track the number of `Employee` objects in memory. (Part 1 of 2.)

```
   // static member function
   static int getCount(); // return number of objects instantiated
private:
   string firstName;
   string lastName;

   // static data
   static int count; // number of objects instantiated
}; // end class Employee

#endif
```

Fig. 10.18 | Employee class definition with a static data member to track the number of Employee objects in memory. (Part 2 of 2.)

```
// Fig. 10.19: Employee.cpp
// Employee class member-function definitions.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

// define and initialize static data member at global namespace scope
int Employee::count = 0; // cannot include keyword static

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount

// constructor initializes non-static data members and
// increments static data member count
Employee::Employee( const string &first, const string &last )
   : firstName( first ), lastName( last )
{
   ++count; // increment static count of employees
   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;
   --count; // decrement static count of employees
} // end ~Employee destructor

```

Fig. 10.19 | Employee class member-function definitions. (Part 1 of 2.)

```
// return first name of employee
string Employee::getFirstName() const
{
   return firstName; // return copy of first name
} // end function getFirstName

// return last name of employee
string Employee::getLastName() const
{
   return lastName; // return copy of last name
} // end function getLastName
```

Fig. 10.19 | `Employee` class member-function definitions. (Part 2 of 2.)

Figure 10.20 uses `static` member function `getCount` to determine the number of `Employee` objects in memory at various points in the program. The program calls `Employee::getCount()` before any `Employee` objects have been created (line 12), after two `Employee` objects have been created (line 23) and after those `Employee` objects have been destroyed (line 34). Lines 16–29 in `main` define a *nested scope*. Recall that local variables exist until the scope in which they're defined terminates. In this example, we create two `Employee` objects in lines 17–18 inside the nested scope. As each constructor executes, it increments class `Employee`'s `static` data member `count`. These `Employee` objects are destroyed when the program reaches line 29. At that point, each object's destructor executes and decrements class `Employee`'s `static` data member count.

```
1  // Fig. 10.20: fig10_20.cpp
2  // static data member tracking the number of objects of a class.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
9     // no objects exist; use class name and binary scope resolution
10    // operator to access static member function getCount
11    cout << "Number of employees before instantiation of any objects is "
12       << Employee::getCount() << endl; // use class name
13
14    // the following scope creates and destroys
15    // Employee objects before main terminates
16    {
17       Employee e1( "Susan", "Baker" );
18       Employee e2( "Robert", "Jones" );
19
20       // two objects exist; call static member function getCount again
21       // using the class name and the scope resolution operator
22       cout << "Number of employees after objects are instantiated is "
23          << Employee::getCount();
24
```

Fig. 10.20 | `static` data member tracking the number of objects of a class. (Part 1 of 2.)

```
        cout << "\n\nEmployee 1: "
           << e1.getFirstName() << " " << e1.getLastName()
           << "\nEmployee 2: "
           << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
     } // end nested scope in main

     // no objects exist, so call static member function getCount again
     // using the class name and the scope resolution operator
     cout << "\nNumber of employees after objects are deleted is "
        << Employee::getCount() << endl;
  } // end main
```

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

Fig. 10.20 | `static` data member tracking the number of objects of a class. (Part 2 of 2.)

A member function should be declared `static` if it does *not* access non-`static` data members or non-`static` member functions of the class. Unlike non-`static` member functions, *a `static` member function does not have a `this` pointer*, because *`static` data members and `static` member functions exist independently of any objects of a class*. The `this` pointer *must* refer to a specific object of the class, and when a `static` member function is called, there might *not* be any objects of its class in memory.

Common Programming Error 10.9

Using the `this` pointer in a `static` member function is a compilation error.

Common Programming Error 10.10

Declaring a `static` member function `const` is a compilation error. The `const` qualifier indicates that a function cannot modify the contents of the object in which it operates, but `static` member functions exist and operate independently of any objects of the class.

10.7 Proxy Classes

Two of the fundamental principles of good software engineering are *separating interface from implementation* and *hiding implementation details*. We strive to achieve these goals by defining a class in a header and implementing its member functions in a separate implementation file. As we pointed out in Chapter 9, however, *headers do contain a portion of a class's implementation and hints about others*. For example, a class's `private` members are listed in the class definition in a header, so these members are visible to clients, even though the clients may not access the `private` members. *Revealing a class's* `private` *data in this manner potentially exposes proprietary information to clients of the class.* We now introduce the notion of a **proxy class** that allows you to hide even the `private` data of a class from clients of the class. Providing clients of your class with a *proxy class* that knows only the `public` interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details.

Implementing a proxy class requires several steps, which we demonstrate in Figs. 10.21–10.24. First, we create the class definition for the class that contains the proprietary implementation we would like to hide. Our example class, called `Implementation`, is shown in Fig. 10.21. The proxy class `Interface` is shown in Figs. 10.22–10.23. The test program and sample output are shown in Fig. 10.24.

Class `Implementation` (Fig. 10.21) provides a single `private` data member called `value` (the data we would like to hide from the client), a constructor to initialize `value` and functions `setValue` and `getValue`.

```
// Fig. 10.21: Implementation.h
// Implementation class definition.

class Implementation
{
public:
   // constructor
   Implementation( int v )
      : value( v ) // initialize value with v
   {
      // empty body
   } // end constructor Implementation

   // set value to v
   void setValue( int v )
   {
      value = v; // should validate v
   } // end function setValue

   // return value
   int getValue() const
   {
      return value;
   } // end function getValue
private:
   int value; // data that we would like to hide from the client
}; // end class Implementation
```

Fig. 10.21 | `Implementation` class definition.

We define a proxy class called `Interface` (Fig. 10.22) with an identical `public` interface (except for the constructor and destructor names) to that of class `Implementation`. The proxy class's only `private` member is a pointer to an `Implementation` object. Using a pointer in this manner allows us to hide class `Implementation`'s implementation details from the client. Notice that the only mentions in class `Interface` of the proprietary `Implementation` class are in the pointer declaration (line 17) and in line 6, a **forward class**

declaration. When a class definition uses only a pointer or reference to an object of another class (as in this case), the class header for that other class (which would ordinarily reveal the `private` data of that class) is *not* required to be included with `#include`. This is because the compiler doesn't need to reserve space for an *object* of the class. The compiler *does* need to reserve space for the *pointer* or *reference.* The sizes of pointers and references are characteristics of the hardware platform on which the compiler runs, so the compiler already knows those sizes. You can simply declare that other class as a data type with a *forward class declaration* (line 6) before the type is used in the file.

```
1  // Fig. 10.22: Interface.h
2  // Proxy class Interface definition.
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

Fig. 10.22 | Proxy class `Interface` definition.

The member-function implementation file for proxy class `Interface` (Fig. 10.23) is the only file that includes the header `Implementation.h` (line 5) containing class `Implementation`. The file `Interface.cpp` (Fig. 10.23) is provided to the client as a precompiled object code file along with the header `Interface.h` that includes the function prototypes of the services provided by the proxy class. Because file `Interface.cpp` is made available to the client only as object code, the client is not able to see the interactions between the proxy class and the proprietary class (lines 9, 17, 23 and 29). The proxy class imposes an extra "layer" of function calls as the "price to pay" for hiding the `private` data of class `Implementation`. Given the speed of today's computers and the fact that many compilers can *inline* simple function calls automatically, the effect of these extra function calls on performance is often negligible.

```
1  // Fig. 10.23: Interface.cpp
2  // Implementation of class Interface--client receives this file only
3  // as precompiled object code, keeping the implementation hidden.
4  #include "Interface.h" // Interface class definition
5  #include "Implementation.h" // Implementation class definition
6
```

Fig. 10.23 | `Interface` class member-function definitions. (Part 1 of 2.)

```cpp
// constructor
Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) // initialize ptr to point to
{                                    // a new Implementation object
   // empty body
} // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v )
{
   ptr->setValue( v );
} // end function setValue

// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();
} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;
} // end ~Interface destructor
```

Fig. 10.23 | `Interface` class member-function definitions. (Part 2 of 2.)

Figure 10.24 tests class `Interface`. Notice that only the header for `Interface` is included in the client code (line 4)—there is no mention of the existence of a separate class called `Implementation`. Thus, the client never sees the `private` data of class `Implementation`, nor can the client code become dependent on the `Implementation` code.

Software Engineering Observation 10.11

A proxy class insulates client code from implementation changes.

```cpp
1  // Fig. 10.24: fig10_24.cpp
2  // Hiding a class's private data with a proxy class.
3  #include <iostream>
4  #include "Interface.h" // Interface class definition
5  using namespace std;
6
7  int main()
8  {
9     Interface i( 5 ); // create Interface object
10
11    cout << "Interface contains: " << i.getValue()
12       << " before setValue" << endl;
13
14    i.setValue( 10 );
```

Fig. 10.24 | Hiding a class's `private` data with a proxy class. (Part 1 of 2.)

```

    cout << "Interface contains: " << i.getValue()
       << " after setValue" << endl;
} // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Fig. 10.24 | Hiding a class's `private` data with a proxy class. (Part 2 of 2.)

10.8 Wrap-Up

This chapter introduced several advanced topics related to classes and data abstraction. You learned how to specify `const` objects and `const` member functions to prevent modifications to objects, thus enforcing the principle of least privilege. You also learned that, through composition, a class can have objects of other classes as members. We introduced the topic of friendship and demonstrated how to use `friend` functions.

You learned that the `this` pointer is passed as an implicit argument to each of a class's non-`static` member functions, allowing the functions to access the correct object's data members and other non-`static` member functions. You also saw explicit use of the `this` pointer to access the class's members and to enable cascaded member-function calls. We motivated the need for `static` data members and demonstrated how to declare and use `static` data members and `static` member functions in your own classes. Finally, we showed how to create a proxy class to hide the implementation details of a class from the class's clients.

In Chapter 11, we continue our study of classes and objects by showing how to enable C++'s operators to work with objects—a process called operator overloading. For example, you'll see how to overload the << operator so it can be used to output a complete array without explicitly using a repetition statement.

Summary

Section 10.2 `const` (Constant) Objects and `const` Member Functions

- The keyword `const` can be used to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error.
- C++ compilers disallow non-`const` member function calls on `const` objects.
- An attempt by a `const` member function to modify an object of its class is a compilation error.
- A member function is specified as `const` both in its prototype and in its definition.
- A `const` object must be initialized.
- Constructors and destructors cannot be declared `const`.
- `const` data member and reference data members *must* be initialized using member initializers (p. 420).

Section 10.3 Composition: Objects as Members of Classes

- A class can have objects of other classes as members—this concept is called composition.

- Member objects (p. 423) are constructed in the order in which they're declared in the class definition and before their enclosing class objects are constructed.
- If a member initializer is not provided for a member object, the member object's default constructor (p. 423) will be called implicitly.

Section 10.4 `friend` Functions and `friend` Classes

- A `friend` function (p. 429) of a class is defined outside that class's scope, yet has the right to access all of the class's members. Stand-alone functions or entire classes may be declared to be friends.
- A friend declaration can appear anywhere in the class.
- The friendship relation is neither symmetric nor transitive.

Section 10.5 Using the `this` Pointer

- Every object has access to its own address through the `this` pointer (p. 431).
- An object's `this` pointer is not part of the object itself—i.e., the size of the memory occupied by the `this` pointer is not reflected in the result of a `sizeof` operation on the object.
- The `this` pointer is passed as an implicit argument to each non-`static` member function.
- Objects use the `this` pointer implicitly (as we've done to this point) or explicitly to reference their data members and member functions.
- The `this` pointer enables cascaded member-function calls (p. 433) in which multiple functions are invoked in the same statement.

Section 10.6 `static` Class Members

- A `static` data member (p. 437) represents "class-wide" information (i.e., a property of the class shared by all instances, not a property of a specific object of the class).
- `static` data members have class scope and can be declared `public`, `private` or `protected`.
- A class's `static` members exist even when no objects of that class exist.
- To access a `public static` class member when no objects of the class exist, simply prefix the class name and the scope resolution operator (`::`) to the name of the data member.
- A member function should be declared `static` (p. 438) if it does not access non-`static` data members or non-`static` member functions of the class. Unlike non-`static` member functions, a `static` member function does not have a `this` pointer, because `static` data members and `static` member functions exist independently of any objects of a class.

Section 10.7 Proxy Classes

- Providing clients of your class with a proxy class (p. 442) that knows only the `public` interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details, such as its `private` data.
- When a class definition uses only a pointer or reference to an object of another class, the class header for that other class (which would ordinarily reveal the `private` data of that class) is not required to be included with `#include`. You can simply declare that other class as a data type with a forward class declaration (p. 443) before the type is used in the file.
- The implementation file containing the member functions for a proxy class is the only file that includes the header for the class whose `private` data we would like to hide.
- The implementation file containing the member functions for the proxy class is provided to the client as a precompiled object code file along with the header that includes the function prototypes of the services provided by the proxy class.

Self-Review Exercises

10.1 Fill in the blanks in each of the following:

a) ________ must be used to initialize constant members of a class.
b) A nonmember function must be declared as a(n) ________ of a class to have access to that class's `private` data members.
c) A constant object must be ________; it cannot be modified after it's created.
d) A(n) ________ data member represents class-wide information.
e) An object's non-`static` member functions have access to a "self pointer" to the object called the ________ pointer.
f) Keyword ________ specifies that an object or variable is not modifiable.
g) If a member initializer is not provided for a member object of a class, the object's ________ is called.
h) A member function should be `static` if it does not access ________ class members.
i) Member objects are constructed ________ their enclosing class object.

10.2 Find the errors in the following class and explain how to correct them:

```
class Example
{
public:
   Example( int y = 10 )
      : data( y )
   {
      // empty body
   } // end Example constructor

   int getIncrementedData() const
   {
      return ++data;
   } // end function getIncrementedData

   static int getCount()
   {
      cout << "Data is " << data << endl;
      return count;
   } // end function getCount
private:
   int data;
   static int count;
}; // end class Example
```

Answers to Self-Review Exercises

10.1 a) member initializers. b) `friend`. c) initialized. d) `static`. e) `this`. f) `const`. g) default constructor. h) non-`static`. i) before.

10.2 *Error:* The class definition for `Example` has two errors. The first occurs in function `getIncrementedData`. The function is declared `const`, but it modifies the object.
Correction: To correct the first error, remove the `const` keyword from the definition of `getIncrementedData`.
Error: The second error occurs in function `getCount`. This function is declared `static`, so it isn't allowed to access any non-`static` member (i.e., data) of the class.
Correction: To correct the second error, remove the output line from the `getCount` definition.

Exercises

10.3 *(Friendship)* Explain the notion of friendship. Explain the negative aspects of friendship as described in the text.

10.4 *(Constructor Overloading)* Can a correct `Time` class definition include *both* of the following constructors? If not, explain why not.

```
Time( int h = 0, int m = 0, int s = 0 );
Time();
```

10.5 *(Constructors and Destructors)* What happens when a return type, even `void`, is specified for a constructor or destructor?

10.6 *(`Date` Class Modification)* Modify class `Date` in Fig. 10.8 to have the following capabilities:

a) Output the date in multiple formats such as

```
DDD YYYY
MM/DD/YY
June 14, 1992
```

b) Use overloaded constructors to create `Date` objects initialized with dates of the formats in part (a).

c) Create a `Date` constructor that reads the system date using the standard library functions of the `<ctime>` header and sets the `Date` members. (See your compiler's reference documentation or `www.cplusplus.com/ref/ctime/index.html` for information on the functions in header `<ctime>`.)

In Chapter 11, we'll be able to create operators for testing the equality of two dates and for comparing dates to determine whether one date is prior to, or after, another.

10.7 *(`SavingsAccount` Class)* Create a `SavingsAccount` class. Use a `static` data member `annualInterestRate` to store the annual interest rate for each of the savers. Each member of the class contains a `private` data member `savingsBalance` indicating the amount the saver currently has on deposit. Provide member function `calculateMonthlyInterest` that calculates the monthly interest by multiplying the `balance` by `annualInterestRate` divided by 12; this interest should be added to `savingsBalance`. Provide a `static` member function `modifyInterestRate` that sets the `static` `annualInterestRate` to a new value. Write a driver program to test class `SavingsAccount`. Instantiate two different objects of class `SavingsAccount`, `saver1` and `saver2`, with balances of $2000.00 and $3000.00, respectively. Set the `annualInterestRate` to 3 percent. Then calculate the monthly interest and print the new balances for each of the savers. Then set the `annualInterestRate` to 4 percent, calculate the next month's interest and print the new balances for each of the savers.

10.8 *(`IntegerSet` Class)* Create class `IntegerSet` for which each object can hold integers in the range 0 through 100. Represent the set internally as a `vector` of `bool` values. Element `a[i]` is `true` if integer *i* is in the set. Element `a[j]` is `false` if integer *j* is not in the set. The default constructor initializes a set to the so-called "empty set," i.e., a set for which all elements contain `false`.

Provide member functions for the common set operations. For example, provide a `unionOfSets` member function that creates a third set that is the set-theoretic union of two existing sets (i.e., an element of the result is set to `true` if that element is `true` in either or both of the existing sets, and an element of the result is set to `false` if that element is `false` in each of the existing sets).

Provide an `intersectionOfSets` member function which creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the result is set to `false` if that element is `false` in either or both of the existing sets, and an element of the result is set to `true` if that element is `true` in each of the existing sets).

Provide an `insertElement` member function that places a new integer *k* into a set by setting `a[k]` to `true`. Provide a `deleteElement` member function that deletes integer *m* by setting `a[m]` to `false`.

Provide a `printSet` member function that prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set (i.e., their position in the `vector` has a value of `true`). Print `---` for an empty set.

Provide an `isEqualTo` member function that determines whether two sets are equal.

Provide an additional constructor that receives an array of integers and the size of that array and uses the array to initialize a set object.

Now write a driver program to test your `IntegerSet` class. Instantiate several `IntegerSet` objects. Test that all your member functions work properly.

10.9 *(`Time` Class Modification)* It would be perfectly reasonable for the `Time` class of Figs. 10.15–10.16 to represent the time internally as the number of seconds since midnight rather than the three integer values `hour`, `minute` and `second`. Clients could use the same `public` methods and get the same results. Modify the `Time` class of Fig. 10.15 to implement the time as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class. [*Note:* This exercise nicely demonstrates the virtues of implementation hiding.]

10.10 *(Card Shuffling and Dealing)* Create a program to shuffle and deal a deck of cards. The program should consist of class `Card`, class `DeckOfCards` and a driver program. Class `Card` should provide:

a) Data members `face` and `suit` of type `int`.
b) A constructor that receives two `int`s representing the face and suit and uses them to initialize the data members.
c) Two `static` arrays of `string`s representing the faces and suits.
d) A `toString` function that returns the `Card` as a `string` in the form "*face* `of` *suit*." You can use the `+` operator to concatenate `string`s.

Class `DeckOfCards` should contain:

a) A `vector` of `Card`s named `deck` to store the `Card`s.
b) An integer `currentCard` representing the next card to deal.
c) A default constructor that initializes the `Card`s in the deck. The constructor should use `vector` function `push_back` to add each `Card` to the end of the `vector` after the `Card` is created and initialized. This should be done for each of the 52 `Card`s in the deck.
d) A `shuffle` function that shuffles the `Card`s in the deck. The shuffle algorithm should iterate through the `vector` of `Card`s. For each `Card`, randomly select another `Card` in the deck and swap the two `Card`s.
e) A `dealCard` function that returns the next `Card` object from the deck.
f) A `moreCards` function that returns a `bool` value indicating whether there are more `Card`s to deal.

The driver program should create a `DeckOfCards` object, shuffle the cards, then deal the 52 cards.

10.11 *(Card Shuffling and Dealing)* Modify the program you developed in Exercise 10.10 so that it deals a five-card poker hand. Then write functions to accomplish each of the following:

a) Determine whether the hand contains a pair.
b) Determine whether the hand contains two pairs.
c) Determine whether the hand contains three of a kind (e.g., three jacks).
d) Determine whether the hand contains four of a kind (e.g., four aces).
e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).
f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

Card Shuffling and Dealing Projects

10.12 *(Card Shuffling and Dealing)* Use the functions from Exercise 10.11 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

10.13 *(Card Shuffling and Dealing)* Modify the program you developed in Exercise 10.12 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand.

10.14 *(Card Shuffling and Dealing)* Modify the program you developed in Exercise 10.13 so that it handles the dealer's hand, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. Play 20 more games. Does your modified program play a better game?

Making a Difference

10.15 *(Project: Emergency Response Class)* The North American emergency response service, *9-1-1*, connects callers to a *local* Public Service Answering Point (PSAP). Traditionally, the PSAP would ask the caller for identification information—including the caller's address, phone number and the nature of the emergency, then dispatch the appropriate emergency responders (such as the police, an ambulance or the fire department). *Enhanced 9-1-1 (or E9-1-1)* uses computers and databases to determine the caller's physical address, directs the call to the nearest PSAP, and displays the caller's phone number and address to the call taker. *Wireless Enhanced 9-1-1* provides call takers with identification information for wireless calls. Rolled out in two phases, the first phase required carriers to provide the wireless phone number and the location of the cell site or base station transmitting the call. The second phase required carriers to provide the location of the caller (using technologies such as GPS). To learn more about 9-1-1, visit `www.fcc.gov/pshs/services/911-services/Welcome.html` and `people.howstuffworks.com/9-1-1.htm`.

An important part of creating a class is determining the class's attributes (instance variables). For this class design exercise, research 9-1-1 services on the Internet. Then, design a class called `Emergency` that might be used in an object-oriented 9-1-1 emergency response system. List the attributes that an object of this class might use to represent the emergency. For example, the class might include information on who reported the emergency (including their phone number), the location of the emergency, the time of the report, the nature of the emergency, the type of response and the status of the response. The class attributes should completely describe the nature of the problem and what's happening to resolve that problem.

Operator Overloading; Class `string`

11

There are two men inside the artist, the poet and the craftsman. One is born a poet. One becomes a craftsman.
—Emile Zola

A thing of beauty is a joy forever.
—John Keats

Objectives

In this chapter you'll learn:

- How operator overloading can help you craft valuable classes.
- To overload unary and binary operators.
- To convert objects from one class to another class.
- To use overloaded operators and additional features of C++'s `string` class.
- To create `PhoneNumber`, `Date` and `Array` classes that provide overloaded operators.
- To use keyword `explicit` to indicate that a constructor cannot be used for implicit conversions.
- To experience a "light-bulb moment" when you'll truly appreciate the value of the class concept.

Outline

11.1 Introduction

This chapter shows how to enable C++'s operators to work with class objects—a process called **operator overloading**. One example of an overloaded operator built into C++ is <<, which is used *both* as the stream insertion operator *and* as the bitwise left-shift operator (which is discussed in Chapter 21). Similarly, >> also is overloaded; it's used both as the stream extraction operator and the bitwise right-shift operator. Both of these operators are overloaded in the C++ Standard Library. You've been using overloaded operators since early in the book. The overloads are built into the base C++ language itself. For example, C++ overloads the addition operator (+) and the subtraction operator (-) to perform differently, depending on their context in integer, floating-point and pointer arithmetic with data of fundamental types.

You can overload *most* operators to be used with class objects—the compiler generates the appropriate code based on the *types* of the operands. The jobs performed by overloaded operators also can be performed by explicit function calls, but operator notation is often more natural.

Our examples start by demonstrating the C++ Standard Library's class `string`, which has lots of overloaded operators. This enables you to see overloaded operators in use before implementing your own overloaded operators. Next, we create a `PhoneNumber` class that enables us to use overloaded operators << and >> to conveniently output and input fully formatted, 10-digit phone numbers. We then present a `Date` class that overloads the prefix and postfix increment (++) operators to add one day to the value of a `Date`. The class also overloads the += operator to allow a program to increment a `Date` by the number of days specified on the right side of the operator.

Next, we present a capstone case study—an `Array` class that uses overloaded operators and other capabilities to solve various problems with pointer-based arrays. This is one of the most important case studies in the book. Many of our students have indicated that the `Array` case study is their "light bulb moment" in truly understanding what classes and object technology are all about. As part of this class, we'll overload stream insertion, stream extraction, assignment, equality, relational and subscript operators. Once you master this

Array class, you'll indeed understand the essence of object technology—crafting, using and reusing valuable classes.

The chapter concludes with discussions of how you can convert between types (incuding class types), problems with certain implicit conversions and how to prevent those problems.

11.2 Using the Overloaded Operators of Standard Library Class string

Figure 11.1 demonstrates many of class string's overloaded operators and several other useful member functions, including empty, substr and at. Function empty determines whether a string is empty, function substr returns a string that represents a portion of an existing string and function at returns the character at a specific index in a string (after checking that the index is in range). Chapter 18 presents class string in detail.

```
// Fig. 11.1: fig11_01.cpp
// Standard Library string class test program.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string s1( "happy" );
   string s2( " birthday" );
   string s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
      << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
      << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
      << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
      << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
      << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );

   // test string member-function empty
   cout << "\n\nTesting s3.empty():" << endl;

   if ( s3.empty() )
   {
      cout << "s3 is empty; assigning s1 to s3;" << endl;
      s3 = s1; // assign s1 to s3
      cout << "s3 is \"" << s3 << "\"";
   } // end if

   // test overloaded string concatenation operator
   cout << "\n\ns1 += s2 yields s1 = ";
```

Fig. 11.1 | Standard Library string class test program. (Part 1 of 3.)

```
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test overloaded string concatenation operator with a char * string
   cout << "\n\ns1 += \" to you\" yields" << endl;
   s1 += " to you";
   cout << "s1 = " << s1 << "\n\n";

   // test string member function substr
   cout << "The substring of s1 starting at location 0 for\n"
      << "14 characters, s1.substr(0, 14), is:\n"
      << s1.substr( 0, 14 ) << "\n\n";

   // test substr "to-end-of-string" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1.substr(15), is:\n"
      << s1.substr( 15 ) << endl;

   // test copy constructor
   string s4( s1 );
   cout << "\ns4 = " << s4 << "\n\n";

   // test overloaded assignment (=) operator with self-assignment
   cout << "assigning s4 to s4" << endl;
   s4 = s4;
   cout << "s4 = " << s4 << endl;

   // test using overloaded subscript operator to create lvalue
   s1[ 0 ] = 'H';
   s1[ 6 ] = 'B';
   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
      << s1 << "\n\n";

   // test subscript out of range with string member function "at"
   try
   {
      cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
      s1.at( 30 ) = 'd'; // ERROR: subscript out of range
   } // end try
   catch ( out_of_range &ex )
   {
      cout << "An exception occurred: " << ex.what() << endl;
   } // end catch
} // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
```

Fig. 11.1 | Standard Library `string` class test program. (Part 2 of 3.)

```
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

s4 = happy birthday to you

assigning s4 to s4
s4 = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:
An exception occurred: invalid string position
```

Fig. 11.1 | Standard Library `string` class test program. (Part 3 of 3.)

Lines 9–11 create three `string` objects—`s1` is initialized with the literal `"happy"`, `s2` is initialized with the literal `" birthday"` and `s3` uses the default `string` constructor to create an empty `string`. Lines 14–15 output these three objects, using `cout` and operator `<<`, which the `string` class designers overloaded to handle `string` objects. Then lines 16–22 show the results of comparing `s2` to `s1` by using class `string`'s overloaded equality and relational operators, which perform lexicographical comparisons using the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string`.

Class `string` provides member function **`empty`** to determine whether a `string` is empty, which we demonstrate in line 27. Member function `empty` returns `true` if the `string` is empty; otherwise, it returns `false`.

Line 30 demonstrates class `string`'s overloaded assignment operator by assigning `s1` to `s3`. Line 31 outputs `s3` to demonstrate that the assignment worked correctly.

Line 36 demonstrates class `string`'s overloaded `+=` operator for string concatenation. In this case, the contents of `s2` are appended to `s1`. Then line 37 outputs the resulting string that's stored in `s1`. Line 41 demonstrates that a string literal can be appended to a `string` object by using operator `+=`. Line 42 displays the result.

Class `string` provides member function **`substr`** (lines 47 and 52) to return a portion of a string as a `string` object. The call to `substr` in line 47 obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument).The call to `substr` in line 52 obtains a substring starting from position 15 of `s1`.

When the second argument is not specified, `substr` returns the *remainder* of the `string` on which it's called.

Line 55 creates `string` object `s4` and initializes it with a copy of `s1`. This results in a call to class `string`'s copy constructor. Line 60 uses class `string`'s overloaded `=` operator to demonstrate that it handles *self-assignment* properly—we'll see when we build class `Array` later in the chapter that self-assignment can be dangerous and we'll show how to deal with the issues.

Lines 64–65 used class `string`'s overloaded `[]` operator to create *lvalues* that enable new characters to replace existing characters in `s1`. Line 67 outputs the new value of `s1`. *Class `string`'s overloaded `[]` operator does not perform any bounds checking.* Therefore, *you must ensure that operations using standard class `string`'s overloaded `[]` operator do not accidentally manipulate elements outside the bounds of the `string`.* Class `string` *does* provide bounds checking in its member function **`at`**, which throws an exception if its argument is an invalid subscript. By default, this causes a C++ program to terminate and display a system-specific error message. If the subscript is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue* (i.e., a `const` reference), depending on the context in which the call appears. Line 73 demonstrates a call to function `at` with an invalid subscript; this throws an `out_of_range` exception.

11.3 Fundamentals of Operator Overloading

As you saw in Fig. 11.1, operators provide a concise notation for manipulating `string` objects. You can use operators with your own user-defined types as well. Although C++ does *not* allow *new* operators to be created, it *does* allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

Operator overloading is not automatic—you must write operator-overloading functions to perform the desired operations. An operator is overloaded by writing a non-`static` member function definition or non-member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded. For example, the function name `operator+` would be used to overload the addition operator (`+`) for use with objects of a particular class. When operators are overloaded as member functions, they must be non-`static`, because *they must be called on an object of the class* and operate on that object.

To use an operator on an object of a class, the operator must be overloaded for that class—with three exceptions:

- The assignment operator (`=`) may be used with *every* class to perform *memberwise assignment* of the class's data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left). *Memberwise assignment is dangerous for classes with pointer members*, so we'll explicitly overload the assignment operator for such classes.
- The address (`&`) operator returns a pointer to the object; this operator also can be overloaded.
- The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression. This operator also can be overloaded.

Operators That Cannot Be Overloaded

Most of C++'s operators can be overloaded. Figure 11.2 shows the operators that cannot be overloaded.

Operators that cannot be overloaded			
`.`	`.*`	`::`	`?:`

Fig. 11.2 | Operators that cannot be overloaded.

Rules and Restrictions on Operator Overloading

As you prepare to overload operator with your own classes, there are several rules and restrictions you should keep in mind:

- *The precedence of an operator cannot be changed by overloading.* However, parentheses can be used to *force* the order of evaluation of overloaded operators in an expression.
- *The associativity of an operator cannot be changed by overloading*—if an operator normally associates from left to right, then so do all of its overloaded versions.
- *You cannot change the "arity" of an operator* (that is, the number of operands an operator takes)—overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. Operators `&`, `*`, `+` and `-` all have both unary and binary versions; these unary and binary versions can be separately overloaded.
- *You cannot create new operators; only existing operators can be overloaded.*
- The meaning of how an operator works on values of fundamental types *cannot* be changed by operator overloading. For example, you cannot make the `+` operator subtract two `int`s. Operator overloading works only with *objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.*
- Related operators, like `+` and `+=`, must be overloaded separately.
- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function *must* be declared as a class member. For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

11.4 Overloading Binary Operators

A binary operator can be overloaded as a non-`static` *member function with one parameter or as a non-member function with two parameters (one of those parameters must be either a class object or a reference to a class object).* A non-member operator function is often declared as `friend` of a class for performance reasons.

Binary Overloaded Operators as Member Functions

Consider using `<` to compare two objects of a `String` class that you define. When overloading binary operator `<` as a non-`static` member function of a `String` class, if `y` and `z` are String-class objects, then y < z is treated as if y.operator<(z) had been written, invoking the operator< member function with one argument declared below:

```
class String
{
public:
   bool operator<( const String & ) const;
   ...
}; // end class String
```

Overloaded operator functions for binary operators can be member functions *only* when the *left* operand is an object of the class in which the function is a member.

Binary Overloaded Operators as Non-Member Functions

As a non-member function, binary operator < must take two arguments—one of which must be an object (or a reference to an object) of the class. If y and z are String-class objects or references to String-class objects, then y < z is treated as if the call operator<(y, z) had been written in the program, invoking function operator< which is declared as follows:

```
bool operator<( const String &, const String & );
```

Performance Tip 11.1

*It's possible to overload an operator as a non-member, non-*friend *function, but such a function requiring access to a class's* private *or* protected *data would need to use* set *or* get *functions provided in that class's* public *interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.*

11.5 Overloading the Binary Stream Insertion and Stream Extraction Operators

You can input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<. The C++ class libraries overload these binary operators for each fundamental type, including pointers and char * strings. You can also overload these operators to perform input and output for your own types. The program of Figs. 11.3–11.5 overloads these operators to input and output PhoneNumber objects in the format "(000) 000-0000." The program assumes telephone numbers are input correctly.

```
// Fig. 11.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

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

class PhoneNumber
{
```

Fig. 11.3 | PhoneNumber class with overloaded stream insertion and stream extraction operators as friend functions. (Part 1 of 2.)

```cpp
   friend ostream &operator<<( ostream &, const PhoneNumber & );
   friend istream &operator>>( istream &, PhoneNumber & );
private:
   string areaCode; // 3-digit area code
   string exchange; // 3-digit exchange
   string line; // 4-digit line
}; // end class PhoneNumber

#endif
```

Fig. 11.3 | `PhoneNumber` class with overloaded stream insertion and stream extraction operators as `friend` functions. (Part 2 of 2.)

```cpp
// Fig. 11.4: PhoneNumber.cpp
// Overloaded stream insertion and stream extraction operators
// for class PhoneNumber.
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
   output << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
   input.ignore(); // skip (
   input >> setw( 3 ) >> number.areaCode; // input area code
   input.ignore( 2 ); // skip ) and space
   input >> setw( 3 ) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw( 4 ) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Fig. 11.4 | Overloaded stream insertion and stream extraction operators for class `PhoneNumber`.

```cpp
// Fig. 11.5: fig11_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
```

Fig. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)

```
4  #include <iostream>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  int main()
9  {
10    PhoneNumber phone; // create object phone
11
12    cout << "Enter phone number in the form (123) 456-7890:" << endl;
13
14    // cin >> phone invokes operator>> by implicitly issuing
15    // the non-member function call operator>>( cin, phone )
16    cin >> phone;
17
18    cout << "The phone number entered was: ";
19
20    // cout << phone invokes operator<< by implicitly issuing
21    // the non-member function call operator<<( cout, phone )
22    cout << phone << endl;
23 } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Fig. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 2 of 2.)

Overloading the Stream Extraction (>>) Operator

The stream extraction operator function `operator>>` (Fig. 11.4, lines 21–30) takes the `istream` reference `input` and the `PhoneNumber` reference `number` as arguments and returns an `istream` reference. Operator function `operator>>` inputs phone numbers of the form

```
(800) 555-1212
```

into objects of class `PhoneNumber`. When the compiler sees the expression

```
cin >> phone
```

in line 16 of Fig. 11.5, the compiler generates the *non-member function call*

```
operator>>( cin, phone );
```

When this call executes, reference parameter `input` (Fig. 11.4, line 21) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`. The operator function reads as `string`s the three parts of the telephone number into the `areaCode` (line 24), `exchange` (line 26) and `line` (line 28) members of the `PhoneNumber` object referenced by parameter `number`. Stream manipulator `setw` limits the number of characters read into each `string`. *When used with* `cin` *and* `string`*s,* `setw` *restricts the number of characters read to the number of characters specified by its argument* (i.e., `setw(3)` allows three characters to be read). The parentheses, space and dash characters are skipped by calling `istream` member function `ignore` (Fig. 11.4, lines 23, 25 and 27), which discards the specified number of characters in the input stream (one character by default). Function `operator>>` returns `istream` reference `input` (i.e., `cin`). This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or other data types. For example, a program can input two `PhoneNumber` objects in one statement as follows:

```
cin >> phone1 >> phone2;
```

First, the expression `cin >> phone1` executes by making the non-member function call

```
operator>>( cin, phone1 );
```

This call then returns a reference to `cin` as the value of `cin >> phone1`, so the remaining portion of the expression is interpreted simply as `cin >> phone2`. This executes by making the non-member function call

```
operator>>( cin, phone2 );
```

Good Programming Practice 11.1

Overloaded operators should mimic the functionality of their built-in counterparts—for example, the + operator should be overloaded to perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

Overloading the Stream Insertion (<<) Operator

The stream insertion operator function (Fig. 11.4, lines 11–16) takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference. Function `operator<<` displays objects of type `PhoneNumber`. When the compiler sees the expression

```
cout << phone
```

in line 22 of Fig. 11.5, the compiler generates the non-member function call

```
operator<<( cout, phone );
```

Function `operator<<` displays the parts of the telephone number as `string`s, because they're stored as `string` objects.

Error-Prevention Tip 11.1

Returning a reference from an overloaded << or >> operator function is typically successful because `cout`, `cin` and most stream objects are global, or at least long-lived. Returning a reference to an automatic variable or other temporary object is dangerous—this can create "dangling references" to nonexisting objects.

Overloaded Operators as Non-Member `friend` Functions

The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as non-member, `friend` functions (Fig. 11.3, lines 12–13). They're non-member functions because the object of class `PhoneNumber` must be the operator's *right* operand. If these were to be `PhoneNumber` member functions, the following awkward statements would have to be used to output and input an `Array`:

```
phone << cout;
phone >> cin;
```

Such statements would be confusing to most C++ programmers, who are familiar with `cout` and `cin` appearing as the *left* operands of << and >>, respectively.

Overloaded operator functions for binary operators can be member functions only when the left operand is an object of the class in which the function is a member. *Overloaded input and output operators are declared as `friends` if they need to access non-`public` class members directly for performance reasons or because the class may not offer appropriate* get *functions.* Also, the `PhoneNumber` reference in function `operator<<`'s parameter list (Fig. 11.4, line 11) is `const`, because the `PhoneNumber` will simply be output, and the `PhoneNumber` reference in function `operator>>`'s parameter list (line 21) is non-`const`, because the `PhoneNumber` object must be modified to store the input telephone number in the object.

Software Engineering Observation 11.1

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

Why Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Non-Member Functions

The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`. To use the operator in this manner where the *right* operand is an object of a user-defined class, it must be overloaded as a non-member function. To be a member function, operator << would have to be a member of the `ostream` class. This is not possible for user-defined classes, since we are *not allowed to modify C++ Standard Library classes*. Similarly, the overloaded stream extraction operator (>>) is used in an expression in which the left operand has the type `istream &`, as in the expression `cin >> classObject`, and the *right* operand is an object of a user-defined class, so it, too, must be a non-member function. Also, each of these overloaded operator functions may require access to the `private` data members of the class object being output or input, so these overloaded operator functions can be made `friend` functions of the class for *performance* reasons.

11.6 Overloading Unary Operators

A unary operator for a class can be overloaded as a non-`static` member function with no arguments or as a non-member function with one argument that must be an object (or a reference to an object) of the class. Member functions that implement overloaded operators must be non-`static` so that they can access the non-`static` data in each object of the class.

Unary Overloaded Operators as Member Functions

Consider overloading unary operator `!` to test whether an object of your own `String` class is empty. Such a function would return a `bool` result. When a unary operator such as `!` is overloaded as a member function with no arguments and the compiler sees the expression `!s` (in which `s` is an object of class `String`), the compiler generates the function call `s.operator!()`. The operand `s` is the `String` object for which the `String` class member function `operator!` is being invoked. The function is declared as follows:

```
class String
{
public:
   bool operator!() const;
   ...
}; // end class String
```

Unary Overloaded Operators as Non-Member Functions

A unary operator such as `!` may be overloaded as a non-member function with one parameter in two different ways—either with a parameter that's an object (this requires a copy of the object, so the side effects of the function are *not* applied to the original object), or with a parameter that's a reference to an object (no copy of the original object is made, so all side effects of this function are applied to the original object). If `s` is a `String` class object (or a reference to a `String` class object), then `!s` is treated as if the call `operator!(s)` had been written, invoking the non-member `operator!` function that's declared as follows:

```
bool operator!( const String & );
```

11.7 Overloading the Unary Prefix and Postfix ++ and -- Operators

The prefix and postfix versions of the increment and decrement operators can all be overloaded. We'll see how the compiler distinguishes between the prefix version and the postfix version of an increment or decrement operator.

To overload the prefix and postfix increment operators, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended. The prefix versions are overloaded exactly as any other prefix unary operator would be. Everything stated in this section for overloading prefix and postfix increment operators applies to overloading predecrement and postdecrement operators. In the next section, we examine a `Date` class with overloaded prefix and postfix increment operators.

Overloading the Prefix Increment Operator

Suppose, that we want to add 1 to the day in `Date` object `d1`. When the compiler sees the preincrementing expression `++d1`, the compiler generates the *member-function call*

```
d1.operator++()
```

The prototype for this operator member function would be

```
Date &operator++();
```

If the prefix increment operator is implemented as a *non-member function*, then, when the compiler sees the expression `++d1`, the compiler generates the function call

```
operator++( d1 )
```

The prototype for this non-member operator function would be declared as

```
Date &operator++( Date & );
```

Overloading the Postfix Increment Operator

Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions. The *convention* that has been adopted is that, when the compiler sees the postincrementing expression `d1++`, it generates the *member-function call*

```
d1.operator++( 0 )
```

The prototype for this operator member function is

```
Date operator++( int )
```

The argument 0 is strictly a “dummy value” that enables the compiler to distinguish between the prefix and postfix increment operator functions. The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

If the postfix increment is implemented as a *non-member function*, then, when the compiler sees the expression `d1++`, the compiler generates the function call

```
operator++( d1, 0 )
```

The prototype for this function would be

```
Date operator++( Date &, int );
```

Once again, the 0 argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as non-member functions. Note that the *postfix increment operator* returns `Date` objects *by value*, whereas the prefix increment operator returns `Date` objects *by reference*—the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred. C++ treats such objects as *rvalues*, which *cannot be used on the left side of an assignment*. The prefix increment operator returns the actual incremented object with its new value. Such an object *can* be used as an *lvalue* in a continuing expression.

Performance Tip 11.2

The extra object that’s created by the postfix *increment (or decrement) operator can result in a performance problem—especially when the operator is used in a loop. For this reason, you should prefer the overloaded* prefix *increment and decrement operators.*

11.8 Case Study: A Date Class

The program of Figs. 11.6–11.8 demonstrates a `Date` class, which uses overloaded prefix and postfix increment operators to add 1 to the day in a `Date` object, while causing appropriate increments to the month and year if necessary. The `Date` header (Fig. 11.6) specifies that `Date`’s `public` interface includes an overloaded stream insertion operator (line 11), a default constructor (line 13), a `setDate` function (line 14), an overloaded prefix increment operator (line 15), an overloaded postfix increment operator (line 16), an overloaded `+=` addition assignment operator (line 17), a function to test for leap years (line 18) and a function to determine whether a day is the last day of the month (line 19).

```
1  // Fig. 11.6: Date.h
2  // Date class definition with overloaded increment operators.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
```

Fig. 11.6 | Date class definition with overloaded increment operators. (Part 1 of 2.)

```
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
   void setDate( int, int, int ); // set month, day, year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment operator
   const Date &operator+=( int ); // add days, modify object
   static bool leapYear( int ); // is date in a leap year?
   bool endOfMonth( int ) const; // is date at the end of month?
private:
   int month;
   int day;
   int year;

   static const int days[]; // array of days per month
   void helpIncrement(); // utility function for incrementing date
}; // end class Date

#endif
```

Fig. 11.6 | Date class definition with overloaded increment operators. (Part 2 of 2.)

```
// Fig. 11.7: Date.cpp
// Date class member- and friend-function definitions.
#include <iostream>
#include <string>
#include "Date.h"
using namespace std;

// initialize static member; one classwide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int month, int day, int year )
{
   setDate( month, day, year );
} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   if ( mm >= 1 && mm <= 12 )
      month = mm;
   else
      throw invalid_argument( "Month must be 1-12" );

   if ( yy >= 1900 && yy <= 2100 )
      year = yy;
   else
      throw invalid_argument( "Year must be >= 1900 and <= 2100" );
```

Fig. 11.7 | Date class member- and friend-function definitions. (Part 1 of 3.)

```
   // test for a leap year
   if ( ( month == 2 && leapYear( year ) && dd >= 1 && dd <= 29 ) ||
        ( dd >= 1 && dd <= days[ month ] ) )
      day = dd;
   else
      throw invalid_argument(
         "Day is out of range for current month and year" );
} // end function setDate

// overloaded prefix increment operator
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
} // end function operator++

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; ++i )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=

// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear )
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
```

Fig. 11.7 | Date class member- and friend-function definitions. (Part 2 of 3.)

```
   else
      return testDay == days[ month ];
} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      ++day; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         ++month; // increment month
         day = 1; // first day of new month
      } // end if
      else // last day of year
      {
         ++year; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      } // end else
} // end function helpIncrement

// overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static string monthName[ 13 ] = { "", "January", "February",
      "March", "April", "May", "June", "July", "August",
      "September", "October", "November", "December" };
   output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
   return output; // enables cascading
} // end function operator<<
```

Fig. 11.7 | Date class member- and friend-function definitions. (Part 3 of 3.)

```
// Fig. 11.08: fig11_11.cpp
// Date class test program.
#include <iostream>
#include "Date.h" // Date class definition
using namespace std;

int main()
{
   Date d1( 12, 27, 2010 ); // December 27, 2010
   Date d2; // defaults to January 1, 1900

   cout << "d1 is " << d1 << "\nd2 is " << d2;
   cout << "\n\nd1 += 7 is " << ( d1 += 7 );

   d2.setDate( 2, 28, 2008 );
   cout << "\n\n  d2 is " << d2;
```

Fig. 11.8 | Date class test program. (Part 1 of 2.)

```
17      cout << "\n++d2 is " << ++d2 << " (leap year allows 29th)";
18
19      Date d3( 7, 13, 2010 );
20
21      cout << "\n\nTesting the prefix increment operator:\n"
22         << "  d3 is " << d3 << endl;
23      cout << "++d3 is " << ++d3 << endl;
24      cout << "  d3 is " << d3;
25
26      cout << "\n\nTesting the postfix increment operator:\n"
27         << "  d3 is " << d3 << endl;
28      cout << "d3++ is " << d3++ << endl;
29      cout << "  d3 is " << d3 << endl;
30   } // end main
```

```
d1 is December 27, 2010
d2 is January 1, 1900

d1 += 7 is January 3, 2011

  d2 is February 28, 2008
++d2 is February 29, 2008 (leap year allows 29th)

Testing the prefix increment operator:
  d3 is July 13, 2010
++d3 is July 14, 2010
  d3 is July 14, 2010

Testing the postfix increment operator:
  d3 is July 14, 2010
d3++ is July 14, 2010
  d3 is July 15, 2010
```

Fig. 11.8 | Date class test program. (Part 2 of 2.)

Function main (Fig. 11.8) creates two Date objects (lines 9–10)—d1 is initialized to December 27, 2010 and d2 is initialized by default to January 1, 1900. The Date constructor (defined in Fig. 11.7, lines 13–16) calls setDate (defined in Fig. 11.7, lines 19–38) to validate the month, day and year specified. Invalid values for the month, day or year result in invalid_argument exceptions.

Line 12 of main outputs each of the Date objects, using the overloaded stream insertion operator (defined in Fig. 11.7, lines 107–114). Line 13 of main uses the overloaded operator += (defined in Fig. 11.7, lines 59–65) to add seven days to d1. Line 15 uses function setDate to set d2 to February 28, 2008, which is a leap year. Then, line 17 preincrements d2 to show that the date increments properly to February 29. Next, line 19 creates a Date object, d3, which is initialized with the date July 13, 2010. Then line 23 increments d3 by 1 with the overloaded prefix increment operator. Lines 21–24 output d3 before and after the preincrement operation to confirm that it worked correctly. Finally, line 28 increments d3 with the overloaded postfix increment operator. Lines 26–29 output d3 before and after the postincrement operation to confirm that it worked correctly.

Date Class Prefix Increment Operator

Overloading the prefix increment operator is straightforward. The prefix increment operator (defined in Fig. 11.7, lines 41–45) calls utility function helpIncrement (defined in

Fig. 11.7, lines 87–104) to increment the date. This function deals with "wraparounds" or "carries" that occur when we increment the last day of the month. These carries require incrementing the month. If the month is already 12, then the year must also be incremented and the month must be set to 1. Function `helpIncrement` uses function `endOfMonth` to increment the day correctly.

The overloaded prefix increment operator returns a reference to the current `Date` object (i.e., the one that was just incremented). This occurs because the current object, `*this`, is returned as a `Date &`. This enables a preincremented `Date` object to be used as an *lvalue*, which is how the built-in prefix increment operator works for fundamental types.

`Date` *Class Postfix Increment Operator*

Overloading the postfix increment operator (defined in Fig. 11.7, lines 49–56) is trickier. To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object. For example, if `int` variable `x` has the value `7`, the statement

```
cout << x++ << endl;
```

outputs the original value of variable `x`. So we'd like our postfix increment operator to operate the same way on a `Date` object. On entry to `operator++`, we save the current object (`*this`) in `temp` (line 51). Next, we call `helpIncrement` to increment the current `Date` object. Then, line 55 returns the unincremented copy of the object previously stored in `temp`. This function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it's declared exits. Thus, declaring the return type to this function as `Date &` would return a reference to an object that no longer exists. *Returning a reference (or a pointer) to a local variable is a common error for which most compilers will issue a warning.*

11.9 Dynamic Memory Management

A standard C++ array data structure is fixed in size once it's created. The size is specified with a constant at compile time. Sometimes it's useful to determine the size of an array *dynamically* at execution time and then create the array. C++ enables you to control the *allocation* and *deallocation* of memory in a program for objects and for arrays of any built-in or user-defined type. This is known as **dynamic memory management** and is performed with the operators **`new`** and **`delete`**. We'll use these capabilities to implement our `Array` class in the next section.

You can use the `new` operator to dynamically **allocate** (i.e., reserve) the exact amount of memory required to hold an object or array at execution time. The object or array is created in the **free store** (also called the **heap**)—*a region of memory assigned to each program for storing dynamically allocated objects*. Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns. When you no longer need the memory, you can return it to the free store by using the `delete` operator to **deallocate** (i.e., release) the memory, which can then be *reused* by future `new` operations.

Obtaining Dynamic Memory with `new`

Consider the following statement:

```
Time *timePtr = new Time;
```

The `new` operator allocates storage of the proper size for an object of type `Time`, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the `new` operator (i.e., a `Time *`). If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by throwing an exception.

Releasing Dynamic Memory with `delete`

To destroy a dynamically allocated object and free the space for the object, use the `delete` operator as follows:

```
delete timePtr;
```

This statement first *calls the destructor for the object to which* `timePtr` *points, then deallocates the memory associated with the object, returning the memory to the free store.*

> **Common Programming Error 11.1**
> *Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."*

Initializing Dynamic Memory

You can provide an initializer for a newly created fundamental-type variable, as in

```
double *ptr = new double( 3.14159 );
```

which initializes a newly created `double` to `3.14159` and assigns the resulting pointer to `ptr`. The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object. For example,

```
Time *timePtr = new Time( 12, 45, 0 );
```

initializes a new `Time` object to 12:45 PM and assigns the resulting pointer to `timePtr`.

Dynamically Allocating Arrays with `new []`

You can also use the `new` operator to allocate arrays dynamically. For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:

```
int *gradesArray = new int[ 10 ];
```

which declares `int` pointer `gradesArray` and assigns to it a pointer to the first element of a dynamically allocated 10-element array of `int`s. The size of an array created at compile time must be specified using a constant integral expression; however, a dynamically allocated array's size can be specified using *any* non-negative integral expression that can be evaluated at execution time. Also, when allocating an array of objects dynamically, you *cannot* pass arguments to each object's constructor—each object is initialized by its default constructor. For fundamental types, the elements are initialized to 0 or the equivalent of 0 (e.g., `char`s are initialized to the null character, `'\0'`). Since an array name is a constant pointer to the array's first element, the following is not allowed for dynamically allocated memory:

```
int gradesArray[] = new int[ 10 ];
```

Releasing Dynamically Allocated Arrays with `delete []`

To deallocate the memory to which `gradesArray` points, use the statement

```
delete [] gradesArray;
```

If the pointer points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory. If the preceding statement did not include the square brackets (`[]`) and `gradesArray` pointed to an array of objects, the result is *undefined. Some compilers call the destructor only for the first object in the array. Using* `delete` *on a null pointer (i.e., a pointer with the value 0) has no effect.*

Common Programming Error 11.2

Using `delete` *instead of* `delete []` *for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator* `delete []`*. Similarly, always delete memory allocated as an individual element with operator* `delete`*—the result of deleting a single object with operator* `delete []` *is undefined.*

11.10 Case Study: Array Class

We discussed arrays in Chapter 7. An array is not much more than a pointer to some space in memory. Pointer-based arrays have many problems, including:

- A program can easily "walk off" either end of an array, because C++ *does not check whether subscripts fall outside the range of an array* (though you can still do this explicitly).
- Arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges are *not* allowed.
- An entire array cannot be input or output at once; each array element must be read or written individually (unless the array is a null-terminated `char *` string).
- Two arrays cannot be meaningfully compared with equality or relational operators (because the array names are simply pointers to where the arrays begin in memory and two arrays will always be at different memory locations).
- When an array is passed to a general-purpose function designed to handle arrays of any size, the array's *size* must be passed as an additional argument.
- One array cannot be assigned to another with the assignment operator(s) (because array names are `const` pointers and a *constant* pointer cannot be used on the left side of an assignment operator).

Class development is an interesting, creative and intellectually challenging activity—always with the goal of "crafting valuable classes." With C++, you can implement more robust array capabilities via classes and operator overloading. You can develop an array class that's preferable to "raw" arrays.

In this example, we create a powerful `Array` class that performs range checking to ensure that subscripts remain within the bounds of the `Array`. The class allows one array object to be assigned to another with the assignment operator. `Array` objects know their size, so the size does not need to be passed separately to functions that receive `Array` parameters. Entire `Arrays` can be input or output with the stream extraction and stream insertion operators, respectively. You can compare `Arrays` with the equality operators `==` and `!=`. Recall that C++ Standard Library class template `vector` provides many of these capabilities as well. Chapter 22 explains class template `vector` in detail.

11.10.1 Using the Array Class

The program of Figs. 11.9–11.11 demonstrates class `Array` and its overloaded operators. First we walk through `main` (Fig. 11.9) and the program's output, then we consider the class definition (Fig. 11.10) and each of its member-function definitions (Fig. 11.11).

```cpp
// Fig. 11.9: fig11_09.cpp
// Array class test program.
#include <iostream>
#include "Array.h"
using namespace std;

int main()
{
   Array integers1( 7 ); // seven-element Array
   Array integers2; // 10-element Array by default

   // print integers1 size and contents
   cout << "Size of Array integers1 is "
      << integers1.getSize()
      << "\nArray after initialization:\n" << integers1;

   // print integers2 size and contents
   cout << "\nSize of Array integers2 is "
      << integers2.getSize()
      << "\nArray after initialization:\n" << integers2;

   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   cin >> integers1 >> integers2;

   cout << "\nAfter input, the Arrays contain:\n"
      << "integers1:\n" << integers1
      << "integers2:\n" << integers2;

   // use overloaded inequality (!=) operator
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create Array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 ); // invokes copy constructor

   cout << "\nSize of Array integers3 is "
      << integers3.getSize()
      << "\nArray after initialization:\n" << integers3;

   // use overloaded assignment (=) operator
   cout << "\nAssigning integers2 to integers1:" << endl;
   integers1 = integers2; // note target Array is smaller
```

Fig. 11.9 | Array class test program. (Part 1 of 3.)

```

    cout << "integers1:\n" << integers1
       << "integers2:\n" << integers2;

    // use overloaded equality (==) operator
    cout << "\nEvaluating: integers1 == integers2" << endl;

    if ( integers1 == integers2 )
       cout << "integers1 and integers2 are equal" << endl;

    // use overloaded subscript operator to create rvalue
    cout << "\nintegers1[5] is " << integers1[ 5 ];

    // use overloaded subscript operator to create lvalue
    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
    integers1[ 5 ] = 1000;
    cout << "integers1:\n" << integers1;

    // attempt to use out-of-range subscript
    try
    {
       cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
       integers1[ 15 ] = 1000; // ERROR: subscript out of range
    } // end try
    catch ( out_of_range &ex )
    {
       cout << "An exception occurred: " << ex.what() << endl;
    } // end catch
 } // end main
```

```
Size of Array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17
```

Fig. 11.9 | Array class test program. (Part 2 of 3.)

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]
An exception occurred: Subscript out of range
```

Fig. 11.9 | Array class test program. (Part 3 of 3.)

Creating **Array***s, Outputting Their Size and Displaying Their Contents*

The program begins by instantiating two objects of class `Array`—`integers1` (Fig. 11.9, line 9) with seven elements, and `integers2` (line 10) with the default `Array` size—10 elements (specified by the `Array` default constructor's prototype in Fig. 11.10, line 14). Lines 13–15 use member function `getSize` to determine the size of `integers1` then output `integers1`'s contents, using the `Array` overloaded stream insertion operator. The sample output confirms that the `Array` elements were set correctly to zeros by the constructor. Next, lines 18–20 output the size of `Array` `integers2` then output `integers2`'s contents, using the `Array` overloaded stream insertion operator.

Using the Overloaded Stream Insertion Operator to Fill an **Array**

Line 23 prompts the user to input 17 integers. Line 24 uses the `Array` overloaded stream extraction operator to read the first seven values into `integers1` and the remaining 10 values into `integers2`. Lines 26–28 output the two arrays with the overloaded `Array` stream insertion operator to confirm that the input was performed correctly.

Using the Overloaded Inequality Operator

Line 33 tests the overloaded inequality operator by evaluating the condition

```
integers1 != integers2
```

The program output shows that the `Arrays` are not equal.

Initializing a New **Array** *with a Copy of an Existing* **Array***'s Contents*

Line 38 instantiates a third `Array` called `integers3` and initializes it with a copy of `Array` `integers1`. This invokes class `Array`'s **copy constructor** to copy the elements of `integers1` into `integers3`. We discuss the details of the copy constructor shortly. The copy constructor can also be invoked by writing line 38 as follows:

```
Array integers3 = integers1;
```

The equal sign in the preceding statement is *not* the assignment operator. When an equal sign appears in the declaration of an object, it invokes a constructor for that object. This form can be used to pass only a single argument to a constructor—specifically, the value on the right side of the = symbol.

Lines 40–42 output the size of `integers3` then output `integers3`'s contents, using the `Array` overloaded stream insertion operator to confirm that `integers3`'s elements were set correctly by the copy constructor.

Using the Overloaded Assignment Operator

Line 46 tests the overloaded assignment operator (=) by assigning `integers2` to `integers1`. Lines 48–49 display both `Array` objects' contents to confirm that the assignment was successful. `Array` `integers1` originally held 7 integers, but was resized to hold a copy of the 10 elements in `integers2`. As we'll see, the overloaded assignment operator performs this resizing operation in a manner that's transparent to the client code.

Using the Overloaded Equality Operator

Line 54 uses the overloaded equality operator (==) to confirm that objects `integers1` and `integers2` are indeed identical after the assignment in line 46.

Using the Overloaded Subscript Operator

Line 58 uses the overloaded subscript operator to refer to `integers1[5]`—an in-range element of `integers1`. This subscripted name is used as an *rvalue* to print the value stored in `integers1[5]`. Line 62 uses `integers1[5]` as a modifiable *lvalue* on the left side of an assignment statement to assign a new value, `1000`, to element `5` of `integers1`. We'll see that `operator[]` returns a reference to use as the modifiable *lvalue* after the operator confirms that `5` is a valid subscript for `integers1`.

Line 69 attempts to assign the value `1000` to `integers1[15]`—an *out-of-range* element. In this example, `operator[]` determines that the subscript is out of range and throws an `out_of_range` exception.

Interestingly, *the array subscript operator* `[]` *is not restricted for use only with arrays*; it also can be used, for example, to select elements from other kinds of *container classes*, such as linked lists, strings and dictionaries. Also, when overloaded `operator[]` functions are defined, *subscripts no longer have to be integers*—characters, strings or even objects of user-defined classes also could be used. In Chapter 22, we discuss the STL `map` class that allows `string` subscripts.

11.10.2 Array Class Definition

Now that we've seen how this program operates, let's walk through the class header (Fig. 11.10). As we refer to each member function in the header, we discuss that function's implementation in Fig. 11.11. In Fig. 11.10, lines 34–35 represent the `private` data members of class `Array`. Each `Array` object consists of a `size` member indicating the number of elements in the `Array` and an `int` pointer—`ptr`—that points to the dynamically allocated pointer-based array of integers managed by the `Array` object.

```
// Fig. 11.10: Array.h
// Array class definition with overloaded operators.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

class Array
{
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 ); // default constructor
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size

   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

   // inequality operator; returns opposite of == operator
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==
   } // end function operator!=

   // subscript operator for non-const objects returns modifiable lvalue
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   int operator[]( int ) const;
private:
   int size; // pointer-based array size
   int *ptr; // pointer to first element of pointer-based array
}; // end class Array

#endif
```

Fig. 11.10 | Array class definition with overloaded operators.

```
// Fig 11.11: Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstdlib> // exit function prototype
#include "Array.h" // Array class definition
```

Fig. 11.11 | Array class member- and `friend`-function definitions. (Part 1 of 4.)

```
using namespace std;

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   if ( arraySize > 0 )
      size = arraySize;
   else
      throw invalid_argument( "Array size must be greater than 0" );

   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize

// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) // avoid self-assignment
   {
      // for Arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size )
      {
         delete [] ptr; // release space
         size = right.size; // resize this object
```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 2 of 4.)

```
         ptr = new int[ size ]; // create space for array copy
      } // end inner if

      for ( int i = 0; i < size; ++i )
         ptr[ i ] = right.ptr[ i ]; // copy array into object
   } // end outer if

   return *this; // enables x = y = z, for example
} // end function operator=

// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different number of elements

   for ( int i = 0; i < size; ++i )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // Array contents are not equal

   return true; // Arrays are equal
} // end function operator==

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
      throw out_of_range( "Subscript out of range" );

   return ptr[ subscript ]; // reference return
} // end function operator[]

// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
      throw out_of_range( "Subscript out of range" );

   return ptr[ subscript ]; // returns copy of this element
} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; ++i )
      input >> a.ptr[ i ];

```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 3 of 4.)

```
112    return input; // enables cin >> x >> y;
113 } // end function
114
115 // overloaded output operator for class Array
116 ostream &operator<<( ostream &output, const Array &a )
117 {
118    int i;
119
120    // output private ptr-based array
121    for ( i = 0; i < a.size; ++i )
122    {
123       output << setw( 12 ) << a.ptr[ i ];
124
125       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
126          output << endl;
127    } // end for
128
129    if ( i % 4 != 0 ) // end last line of output
130       output << endl;
131
132    return output; // enables cout << x << y;
133 } // end function operator<<
```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 4 of 4.)

Overloading the Stream Insertion and Stream Extraction Operators as `friend`*s*

Lines 11–12 of Fig. 11.10 declare the overloaded stream insertion operator and the overloaded stream extraction operator as `friend`s of class `Array`. When the compiler sees an expression like `cout << arrayObject`, it invokes non-member function `operator<<` with the call

```
operator<<( cout, arrayObject )
```

When the compiler sees an expression like `cin >> arrayObject`, it invokes non-member function `operator>>` with the call

```
operator>>( cin, arrayObject )
```

Again, these stream insertion and stream extraction operator functions cannot be members of class `Array`, because the `Array` object is always mentioned on the *right* side of the stream insertion or stream extraction operator.

Function `operator<<` (defined in Fig. 11.11, lines 116–133) prints the number of elements indicated by `size` from the integer array to which `ptr` points. Function `operator>>` (defined in Fig. 11.11, lines 107–113) inputs directly into the array to which `ptr` points. Each of these operator functions returns an appropriate reference to enable *cascaded* output or input statements, respectively. These functions have access to an `Array`'s `private` data because they're declared as `friend`s of class `Array`. We could have used class `Array`'s `getSize` and `operator[]` functions in the bodies of `operator<<` and `operator>>`, in which case these operator functions would not need to be `friend`s of class `Array`. However, the additional function calls might degrade performance.

`Array` *Default Constructor*

Line 14 of Fig. 11.10 declares the *default constructor* for the class and specifies a default size of 10 elements. When the compiler sees a declaration like line 10 in Fig. 11.9, it invokes class `Array`'s default constructor to set the size of the `Array` to 10 elements. The default constructor (defined in Fig. 11.11, lines 10–22) validates and assigns the argument to data member `size`, uses `new` to obtain the memory for the internal pointer-based representation of this array and assigns the pointer returned by `new` to data member `ptr`. Then the constructor uses a `for` statement to set all the elements of the array to zero. It's possible to have an `Array` class that does not initialize its members if, for example, these members are to be read at some later time; but this is considered to be a poor programming practice. `Array`s, and *objects in general, should be properly initialized as they're created*.

`Array` Copy Constructor

Line 15 of Fig. 11.10 declares a *copy constructor* (defined in Fig. 11.11, lines 26–33) that initializes an `Array` by making a copy of an existing `Array` object. *Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory.* This is exactly the problem that would occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class. Copy constructors are invoked whenever a copy of an object is needed, such as in

- passing an object by value to a function,
- returning an object by value from a function or
- initializing an object with a copy of another object of the same class.

The copy constructor is called in a declaration when an object of class `Array` is instantiated and initialized with another object of class `Array`, as in the declaration in line 38 of Fig. 11.9.

The copy constructor for `Array` uses a member initializer (Fig. 11.11, line 27) to copy the `size` of the initializer `Array` into data member `size`, uses `new` (line 29) to obtain the memory for the internal pointer-based representation of this `Array` and assigns the pointer returned by `new` to data member `ptr`.[1] Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object. An object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

Software Engineering Observation 11.2

The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.

Common Programming Error 11.3

A copy constructor must *receive its argument by reference, not by value. Otherwise, the copy constructor call results in* infinite recursion *(a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!*

---

1. Operator `new` could fail to obtain the needed memory, in which case a `bad_alloc` exception will occur. We deal with `new` failures in Chapter 16.

Common Programming Error 11.4

If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both would point to the same *dynamically allocated memory. The first destructor to execute would delete the dynamically allocated memory, and the other object's* `ptr` *would be* undefined, *a situation called a **dangling pointer**—this would likely result in a serious runtime error (such as early program termination) when the pointer was used.*

`Array` Destructor

Line 16 of Fig. 11.10 declares the class's destructor (defined in Fig. 11.11, lines 36–39). The destructor is invoked when an object of class `Array` goes out of scope. The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

Error-Prevention Tip 11.2

If after deleting dynamically allocated memory, the pointer will continue to exist in memory, set the pointer's value to `0` *to indicate that the pointer no longer points to memory in the free store. By setting the pointer to* `0`*, the program loses access to that free-store space, which could be reallocated for a different purpose. If you do not set the pointer to* `0`*, your code could* inadvertently access the reallocated memory, causing subtle, nonrepeatable logic errors.

`getSize` Member Function

Line 17 of Fig. 11.10 declares function `getSize` (defined in Fig. 11.11, lines 42–45) that returns the number of elements in the `Array`.

Overloaded Assignment Operator

Line 19 of Fig. 11.10 declares the overloaded assignment operator function for the class. When the compiler sees the expression `integers1 = integers2` in line 46 of Fig. 11.9, the compiler invokes member function `operator=` with the call

```
integers1.operator=( integers2 )
```

Member function `operator=`'s implementation (Fig. 11.11, lines 49–67) tests for **self-assignment** (line 51) in which an `Array` object is being assigned to itself. When `this` is equal to the `right` operand's address, a self-assignment is being attempted, so the assignment is skipped (i.e., the object already is itself; in a moment we'll see why self-assignment is dangerous). If it isn't a self-assignment, then the function determines whether the sizes of the two arrays are identical (line 55); in that case, the original array of integers in the left-side `Array` object is not reallocated. Otherwise, `operator=` uses `delete` (line 57) to release the memory originally allocated to the target array, copies the `size` of the source array to the `size` of the target array (line 58), uses `new` to allocate the memory for the target array and places the pointer returned by `new` into the array's `ptr` member. Then the `for` statement in lines 62–63 copies the array elements from the source array to the target array. Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this` in line 66) as a constant reference; this enables cascaded `Array` assignments such as `x = y = z`, but prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array` reference that's returned by `(x = y)`. If self-assignment occurs, and function `operator=` did not test for this case, `operator=` would unnecessarily copy the elements of the `Array` into itself.

Software Engineering Observation 11.3

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

Common Programming Error 11.5

Not providing an overloaded assignment operator and a copy constructor for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

Software Engineering Observation 11.4

It's possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a `private` *member of the class.*

Software Engineering Observation 11.5

It's possible to prevent class objects from being copied*; to do this, simply make both the overloaded assignment operator and the copy constructor of that class* `private`*.*

Overloaded Equality and Inequality Operators

Line 20 of Fig. 11.10 declares the overloaded equality operator (==) for the class. When the compiler sees the expression `integers1 == integers2` in line 54 of Fig. 11.9, the compiler invokes member function `operator==` with the call

```
integers1.operator==( integers2 )
```

Member function `operator==` (defined in Fig. 11.11, lines 71–81) immediately returns `false` if the `size` members of the arrays are not equal. Otherwise, `operator==` compares each pair of elements. If they're all equal, the function returns `true`. The first pair of elements to differ causes the function to return `false` immediately.

Lines 23–26 of the header define the overloaded inequality operator (`!=`) for the class. Member function `operator!=` uses the overloaded `operator==` function to determine whether one `Array` is equal to another, then returns the *opposite* of that result. Writing `operator!=` in this manner enables you to *reuse* `operator==`, which *reduces the amount of code that must be written in the class*. Also, the full function definition for `operator!=` is in the `Array` header. This allows the compiler to *inline* the definition of `operator!=` to eliminate the overhead of the extra function call.

Overloaded Subscript Operators

Lines 29 and 32 of Fig. 11.10 declare two overloaded subscript operators (defined in Fig. 11.11 in lines 85–92 and 95–103, respectively). When the compiler sees the expression `integers1[5]` (Fig. 11.9, line 58), it invokes the appropriate overloaded `operator[]` member function by generating the call

```
integers1.operator[]( 5 )
```

The compiler creates a call to the `const` version of `operator[]` (Fig. 11.11, lines 95–103) when the subscript operator is used on a `const Array` object. For example, if `const` object `z` is instantiated with the statement

```
const Array z( 5 );
```

then the `const` version of `operator[]` is required to execute a statement such as

```
cout << z[ 3 ] << endl;
```

Remember, a program can invoke only the `const` member functions of a `const` object.

Each definition of `operator[]` determines whether the subscript it receives as an argument is *in range* and—if not, each throws an `out_of_range` exception. If the subscript is in range, the non-`const` version of `operator[]` returns the appropriate array element as a reference so that it may be used as a modifiable *lvalue* (e.g., on the left side of an assignment statement). If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the array. The returned character is an *rvalue*.

11.11 Operators as Member Functions vs. Non-Member Functions

Whether an operator function is implemented as a member function or as a non-member function, the operator is still used the same way in expressions. So which is best?

When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class. If the left operand *must* be an object of a different class or a fundamental type, this operator function *must* be implemented as a non-member function (as we did in Section 11.5 when overloading `<<` and `>>` as the stream insertion and stream extraction operators, respectively). A non-member operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.

Operator member functions of a specific class are called (implicitly by the compiler) only when the *left* operand of a binary operator is specifically an object of that class, or when the *single operand of a unary operator* is an object of that class.

Commutative Operators

Another reason why you might choose a non-member function to overload an operator is to enable the operator to be *commutative*. For example, suppose we have a *fundamental type variable*, `number`, of type `long int`, and an *object* `bigInteger1`, of class `HugeInteger` (a class in which integers may be arbitrarily large rather than being limited by the machine word size of the underlying hardware; class `HugeInteger` is developed in the chapter exercises). The addition operator (+) produces a *temporary* `HugeInteger` object as the sum of a `HugeInteger` and a `long int` (as in the expression `bigInteger1 + number`), *or* as the sum of a `long int` and a `HugeInteger` (as in the expression `number + bigInteger1`). Thus, we require the addition operator to be *commutative* (exactly as it is with two fundamental-type operands). The problem is that the class object *must* appear on the *left* of the addition operator if that operator is to be overloaded as a member function. So, we *also* overload the operator as a non-member function to allow the `HugeInteger` to appear on the *right* of the addition. The `operator+` function that deals with the `HugeInteger` on the left can still be a member function. The non-member function can simply swap its arguments and call the member function.

11.12 Converting between Types

Most programs process information of many types. Sometimes all the operations "stay within a type." For example, adding an `int` to an `int` produces an `int`. It's often necessary, however, to convert data of one type to data of another type. This can happen in assignments, in calculations, in passing values to functions and in returning values from functions. The compiler knows how to perform certain conversions among fundamental types. You can use *cast operators* to *force* conversions among fundamental types.

But what about user-defined types? The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this. Such conversions can be performed with **conversion constructors**—**single-argument constructors** that turn objects of other types (including fundamental types) into objects of a particular class.

A **conversion operator** (also called a *cast operator*) can be used to convert an object of one class into an object of another class or into an object of a fundamental type. Such a conversion operator must be a non-`static` member function. The function prototype

```
A::operator char *() const;
```

declares an overloaded cast operator function for converting an object of user-defined type `A` into a temporary `char *` object. The operator function is declared `const` because it does not modify the original object. An overloaded **cast operator function** does not specify a return type—the return type is the type to which the object is being converted. If `s` is a class object, when the compiler sees the expression `static_cast< char * >( s )`, the compiler generates the call

```
s.operator char *()
```

The operand `s` is the class object `s` for which the member function `operator char *` is being invoked.

Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types. The prototypes

```
A::operator int() const;
A::operator OtherClass() const;
```

declare *overloaded cast operator functions* that can convert an object of user-defined type `A` into an integer or into an object of user-defined type `OtherClass`, respectively.

One of the nice features of cast operators and conversion constructors is that, when necessary, the compiler can call these functions implicitly to create temporary objects. For example, if an object `s` of a user-defined `String` class appears in a program at a location where an ordinary `char *` is expected, such as

```
cout << s;
```

the compiler can call the overloaded cast-operator function `operator char *` to convert the object into a `char *` and use the resulting `char *` in the expression. With this cast operator provided for a `String` class, the stream insertion operator does not have to be overloaded to output a `String` using `cout`.

Software Engineering Observation 11.6

When a conversion constructor is used to perform an implicit conversion*, C++ can apply* only one *implicit constructor call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator.* The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions.

11.13 explicit Constructors

Any single-argument constructor—except a copy constructor—can be used by the compiler to perform an implicit conversion. The constructor's argument is converted to an object of the class in which the constructor is defined. The conversion is automatic and you need not use a cast operator. *In some situations, implicit conversions are undesirable or error-prone.* For example, our `Array` class in Fig. 11.10 defines a constructor that takes a single `int` argument. The intent of this constructor is to create an `Array` object containing the number of elements specified by the `int` argument. However, this constructor can be misused by the compiler to perform an *implicit conversion*.

> **Common Programming Error 11.6**
>
> *Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.*

Accidentally Using a Single-Argument Constructor as a Conversion Constructor

The program (Fig. 11.12) uses the `Array` class of Figs. 11.10–11.11 to demonstrate an improper implicit conversion.

```cpp
// Fig. 11.12: Fig11_12.cpp
// Driver for simple class Array.
#include <iostream>
#include "Array.h"
using namespace std;

void outputArray( const Array & ); // prototype

int main()
{
   Array integers1( 7 ); // 7-element array
   outputArray( integers1 ); // output Array integers1
   outputArray( 3 ); // convert 3 to an Array and output Array's contents
}  // end main

// print Array contents
void outputArray( const Array &arrayToOutput )
{
   cout << "The Array received has " << arrayToOutput.getSize()
      << " elements. The contents are:\n" << arrayToOutput << endl;
} // end outputArray
```

```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

Fig. 11.12 | Single-argument constructors and implicit conversions.

Line 11 in `main` instantiates `Array` object `integers1` and calls the single argument constructor with the `int` value `7` to specify the number of elements in the `Array`. Recall from Fig. 11.11 that the `Array` constructor that receives an `int` argument initializes all the array elements to 0. Line 12 calls function `outputArray` (defined in lines 17–21), which receives as its argument a `const Array &` to an `Array`. The function outputs the number of elements in its `Array` argument and the contents of the `Array`. In this case, the size of the `Array` is `7`, so seven `0`s are output.

Line 13 calls function `outputArray` with the `int` value `3` as an argument. However, this program does *not* contain a function called `outputArray` that takes an `int` argument. So, the compiler determines whether class `Array` provides a *conversion constructor* that can convert an `int` into an `Array`. Since the `Array` constructor receives one `int` argument, the compiler assumes that the constructor is a conversion constructor that can be used to convert the argument `3` into a temporary `Array` object containing three elements. Then, the compiler passes the temporary `Array` object to function `outputArray` to output the `Array`'s contents. Thus, even though we do not explicitly provide an `outputArray` function that receives an `int` argument, the compiler is able to compile line 13. The output shows the contents of the three-element `Array` containing `0`s.

Preventing Implicit Conversions with Single-Argument Constructors

C++ provides the keyword **`explicit`** to *suppress implicit conversions via conversion constructors when such conversions should not be allowed*. A constructor that's declared `explicit` *cannot* be used in an implicit conversion. In the example of Figure 11.13, the only modification to `Array.h` from Fig. 11.10 was the addition of the keyword `explicit` to the declaration of the single-argument constructor in line 14, as in

```
explicit Array( int = 10 ); // default constructor
```

No modifications are required to the source-code file containing class `Array`'s member-function definitions.

Figure 11.13 presents a slightly modified version of the program in Fig. 11.12. When this program is compiled, the compiler produces an error message indicating that the integer value passed to `outputArray` in line 13 cannot be converted to a `const Array &`. The compiler error message (from Visual C++) is shown in the output window. Line 14 demonstrates how the explicit constructor can be used to create a temporary `Array` of `3` elements and pass it to function `outputArray`.

Error-Prevention Tip 11.3

Use the `explicit` keyword on single-argument constructors that should not be used by the compiler to perform implicit conversions.

```
1  // Fig. 11.13: Fig11_13.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
```

Fig. 11.13 | Demonstrating an `explicit` constructor. (Part 1 of 2.)

```
void outputArray( const Array & ); // prototype

int main()
{
   Array integers1( 7 ); // 7-element array
   outputArray( integers1 ); // output Array integers1
   outputArray( 3 ); // convert 3 to an Array and output Array's contents
   outputArray( Array( 3 ) ); // explicit single-argument constructor call
}  // end main

// print array contents
void outputArray( const Array &arrayToOutput )
{
   cout << "The Array received has " << arrayToOutput.getSize()
      << " elements. The contents are:\n" << arrayToOutput << endl;
} // end outputArray
```

```
c:\cpphtp8_examples\ch11\fig11_13\fig11_13.cpp(13) : error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
        Reason: cannot convert from 'int' to 'const Array'
        Constructor for class 'Array' is declared 'explicit'
```

Fig. 11.13 | Demonstrating an `explicit` constructor. (Part 2 of 2.)

11.14 Building a String Class

In Section 8.10, we introduced C-style, pointer-based string processing with character arrays. Our discussion of `char *` strings continues in Section 21.10. As part of our coverage of crafting valuable classes, we implement our own `String` class that encapsulates a dynamically allocated `char *` string and provides many capabilities that are similar to those we introduced in the `Array` class. To implement this class, we use several of the capabilities introduced in Sections 8.10 and 21.10. Because classes `Array` and `String` are so similar, we placed our `String` class's code and discussion online at `www.deitel.com/books/cpphtp8/` under **Downloads and Resources for Registered Users**. The C++ standard library includes the similar, more robust class `string`, which we demonstrated in Section 11.2 and study in detail in Chapter 18. In this section, we discuss one key feature of our `String` class.

Overloaded Function Call Operator

Overloading the **function call operator `()`** is powerful, because functions can take an *arbitrary* number of parameters. In our `String` class, we overload this operator to select a substring from a `String`. The operator's two integer parameters specify the start location and the length of the substring to be selected. If the start location is out of range or the substring length is negative, the operator simply returns an empty `String`. If the substring length is 0, then the substring is selected to the end of the `String` object. Suppose `string1` is a `String` object containing the string `"AEIOU"`. When the compiler encounters the expression `string1(2, 2)`, it generates the member-function call

```
string1.operator()( 2, 2 )
```

which returns a `String` containing `"IO"`.

11.15 Wrap-Up

In this chapter, you learned how to overload operators to work with class objects. We demonstrated standard C++ class `string`, which makes extensive use of overloaded operators to create a robust, reusable class that can replace C-style, pointer-based strings. Next, we discussed several restrictions that the C++ standard places on overloaded operators. We then presented a `PhoneNumber` class that overloaded operators << and >> to conveniently output and input phone numbers. You also saw a `Date` class that overloaded the prefix and postfix increment (++) operators and we showed a special syntax that's required to differentiate between the prefix and postfix versions of the increment (++) operator.

Next, we introduced the concept of dynamic memory management. You learned that you can create and destroy objects dynamically with the `new` and `delete` operators, respectively. Then, we presented a capstone `Array` class case study that used overloaded operators and other capabilities to solve various problems with pointer-based arrays. This case study helped you truly understand what classes and object technology are all about—crafting, using and reusing valuable classes. As part of this class, you saw overloaded stream insertion, stream extraction, assignment, equality, relational and subscript operators.

You learned reasons for implementing overloaded operators as member functions or as non-member functions. The chapter concluded with discussions of converting between types (incuding class types), problems with certain implicit conversions defined by single-argument constructors and how to prevent those problems by using `explicit` constructors.

In the next chapter, we continue our discussion of classes by introducing a form of software reuse called inheritance. We'll see that when classes share common attributes and behaviors, it's possible to define those attributes and behaviors in a common "base" class and "inherit" those capabilities into new class definitions, enabling you to create the new classes with a minimal amount of code.

Summary

Section 11.1 Introduction

- C++ enables you to overload most operators to be sensitive to the context in which they're used—the compiler generates the appropriate code based on the types of the operands.
- One example of an overloaded operator built into C++ is operator <<, which is used both as the stream insertion operator and as the bitwise left-shift operator. Similarly, >> is also overloaded; it's used both as the stream extraction operator and as the bitwise right-shift operator. Both of these operators are overloaded in the C++ Standard Library.
- C++ overloads + and - to perform differently, depending on their context in integer arithmetic, floating-point arithmetic and pointer arithmetic.
- The jobs performed by overloaded operators can also be performed by explicit function calls, but operator notation is often more natural.

Section 11.2 Using the Overloaded Operators of Standard Library Class `string`

- Standard class `string` is defined in header <string> and belongs to namespace `std`.
- Class `string` provides many overloaded operators, including equality, relational, assignment, addition assignment (for concatenation) and subscript operators.
- Class `string` provides member function `empty` (p. 455), which returns `true` if the `string` is empty; otherwise, it returns `false`.

- Standard class `string` member function `substr` (p. 455) obtains a substring of a length specified by the second argument, starting at the position specified by the first argument. When the second argument is not specified, `substr` returns the remainder of the `string` on which it's called.
- Class `string's` overloaded `[]` operator does not perform any bounds checking. Therefore, you must ensure that operations using standard class `string`'s overloaded `[]` operator do not accidentally manipulate elements outside the bounds of the `string`.
- Standard class `string` provides bounds checking with member function `at` (p. 456), which "throws an exception" if its argument is an invalid subscript. By default, this causes the program to terminate. If the subscript is valid, function `at` returns a reference or a `const` reference to the character at the specified location depending on the context.

Section 11.3 Fundamentals of Operator Overloading

- An operator is overloaded by writing a non-`static` member-function definition or non-member function definition in which the function name is the keyword `operator` followed by the symbol for the operator being overloaded.
- When operators are overloaded as member functions, they must be non-`static`, because they must be called on an object of the class and operate on that object.
- To use an operator on class objects, that operator *must* be overloaded, with three exceptions—the assignment operator (`=`), the address operator (`&`) and the comma operator (`,`).
- You cannot change the precedence and associativity of an operator by overloading.
- You cannot change the "arity" of an operator (i.e., the number of operands an operator takes).
- You cannot create new operators—only existing operators can be overloaded.
- You cannot change the meaning of how an operator works on objects of fundamental types.
- Overloading an assignment operator and an addition operator for a class does not imply that the `+=` operator is also overloaded. Such behavior can be achieved only by explicitly overloading operator `+=` for that class.
- Overloaded `()`, `[]`, `->` and assignment operators must be declared as class members. For the other operators, the operator overloading functions can be class members or non-member functions.

Section 11.4 Overloading Binary Operators

- A binary operator can be overloaded as a non-`static` member function with one argument or as a non-member function with two arguments (one of those arguments must be either a class object or a reference to a class object).

Section 11.5 Overloading the Binary Stream Insertion and Stream Extraction Operators

- The overloaded stream insertion operator (`<<`) is used in an expression in which the left operand has type `ostream &`. For this reason, it must be overloaded as a non-member function. To be a member function, operator `<<` would have to be a member of the `ostream` class, but this is not possible, since we are not allowed to modify C++ Standard Library classes. Similarly, the overloaded stream extraction operator (`>>`) must be a non-member function.
- Another reason to choose a non-member function to overload an operator is to enable the operator to be commutative.
- When used with `cin`, `setw` restricts the number of characters read to the number of characters specified by its argument.
- `istream` member function `ignore` discards the specified number of characters in the input stream (one character by default).

- Overloaded input and output operators are declared as friends if they need to access non-public class members directly for performance reasons.

Section 11.6 Overloading Unary Operators

- A unary operator for a class can be overloaded as a non-static member function with no arguments or as a non-member function with one argument; that argument must be either an object of the class or a reference to an object of the class.
- Member functions that implement overloaded operators must be non-static so that they can access the non-static data in each object of the class.

Section 11.7 Overloading the Unary Prefix and Postfix ++ and -- Operators

- The prefix and postfix increment and decrement operators can all be overloaded.
- To overload the pre- and post-increment operators, each overloaded operator function must have a distinct signature. The prefix versions are overloaded like any other unary operator. The postfix increment operator's unique signature is accomplished by providing a second argument, which must be of type int. This argument is not supplied in the client code. It's used implicitly by the compiler to distinguish between the prefix and postfix versions of the increment operator. The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

Section 11.9 Dynamic Memory Management

- Dynamic memory management (p. 469) enables you to control the allocation and deallocation of memory in a program for any built-in or user-defined type.
- The free store (sometimes called the heap; p. 469) is a region of memory assigned to each program for storing objects dynamically allocated at execution time.
- The new operator (p. 469) allocates storage of the proper size for an object, runs the object's constructor and returns a pointer of the correct type. The new operator can be used to dynamically allocate (p. 469) any fundamental type (such as int or double) or class type. If new is unable to find space in memory for the object, it indicates that an error occurred by "throwing" an "exception." This usually causes the program to terminate immediately, unless the exception is handled.
- To destroy a dynamically allocated object and free its space, use the delete operator (p. 469).
- An array of objects can be allocated dynamically with new as in

  ```
  int *ptr = new int[ 100 ];
  ```

  which allocates an array of 100 integers and assigns the starting location of the array to ptr. The preceding array of integers is deleted (p. 469) with the statement

  ```
  delete [] ptr;
  ```

Section 11.10 Case Study: Array Class

- A copy constructor initializes a new object of a class by copying the members of an existing one. Classes that contain dynamically allocated memory typically provide a copy constructor, a destructor and an overloaded assignment operator.
- The implementation of member function operator= should test for self-assignment (p. 481), in which an object is being assigned to itself.
- The compiler calls the const version of operator[] when the subscript operator is used on a const object and calls the non-const version of the operator when it's used on a non-const object.
- The array subscript operator ([]) can be used to select elements from other types of containers. Also, with overloading, the index values no longer need to be integers.

Section 11.11 Operators as Member Functions vs. Non-Member Functions

- Operator functions can be member functions or non-member functions—non-member functions are often made `friend`s for performance reasons. Member functions use the `this` pointer implicitly to obtain one of their class object arguments (the left operand for binary operators). Arguments for both operands of a binary operator must be explicitly listed in a non-member function call.
- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a non-member function.
- A non-member operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.

Section 11.12 Converting between Types

- The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this. Such conversions can be performed with conversion constructors (p. 484)—single-argument constructors (p. 484) that turn objects of other types (including fundamental types) into objects of a particular class.
- Any single-argument constructor can be thought of as a conversion constructor.
- A conversion operator (p. 484) must be a non-`static` member function. Overloaded cast-operator functions (p. 484) can be defined for converting objects of user-defined types into fundamental types or into objects of other user-defined types.
- An overloaded cast operator function does not specify a return type—the return type is the type to which the object is being converted.
- When necessary, the compiler can call cast operators and conversion constructors implicitly to create temporary objects.

Section 11.13 `explicit` Constructors

- C++ provides the keyword `explicit` (p. 486) to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that's declared `explicit` cannot be used in an implicit conversion.

Section 11.14 Building a `String` Class

- Overloading the function call operator `()` (p. 487)is powerful, because functions can take an arbitrary number of parameters.

Self-Review Exercises

11.1 Fill in the blanks in each of the following:

a) Suppose `a` and `b` are integer variables and we form the sum `a + b`. Now suppose `c` and `d` are floating-point variables and we form the sum `c + d`. The two `+` operators here are clearly being used for different purposes. This is an example of ________.

b) Keyword ________ introduces an overloaded-operator function definition.

c) To use operators on class objects, they must be overloaded, with the exception of operators ________, ________ and ________.

d) The ________, ________ and ________ of an operator cannot be changed by overloading the operator.

e) The operators that cannot be overloaded are ________, ________, ________ and ________.

f) The ________ operator reclaims memory previously allocated by `new`.

g) The ________ operator dynamically allocates memory for an object of a specified type and returns a(n) ________ to that type.

11.2 Explain the multiple meanings of the operators `<<` and `>>`.

11.3 In what context might the name `operator/` be used?

11.4 (True/False) Only existing operators can be overloaded.

11.5 How does the precedence of an overloaded operator compare with the precedence of the original operator?

Answers to Self-Review Exercises

11.1 a) operator overloading. b) `operator`. c) assignment (`=`), address (`&`), comma (`,`). d) precedence, associativity, "arity." e) `.`, `?:`, `.*`, and `::`. f) `delete`. g) `new`, pointer.

11.2 Operator `>>` is both the right-shift operator and the stream extraction operator, depending on its context. Operator `<<` is both the left-shift operator and the stream insertion operator, depending on its context.

11.3 For operator overloading: It would be the name of a function that would provide an overloaded version of the `/` operator for a specific class.

11.4 True.

11.5 The precedence is identical.

Exercises

11.6 *(Memory Allocation and Deallocation Operators)* Compare and contrast dynamic memory allocation and deallocation operators `new`, `new []`, `delete` and `delete []`.

11.7 *(Overloading the Parentheses Operator)* One nice example of overloading the function call operator `()` is to allow another form of double-array subscripting popular in some programming languages. Instead of saying

```
chessBoard[ row ][ column ]
```

for an array of objects, overload the function call operator to allow the alternate form

```
chessBoard( row, column )
```

Create a class `DoubleSubscriptedArray` that has similar features to class `Array` in Figs. 11.10–11.11. At construction time, the class should be able to create an array of any number of rows and any number of columns. The class should supply `operator()` to perform double-subscripting operations. For example, in a 3-by-5 `DoubleSubscriptedArray` called `a`, the user could write `a(1, 3)` to access the element at row `1` and column `3`. Remember that `operator()` can receive any number of arguments. The underlying representation of the double-subscripted array should be a single-subscripted array of integers with *rows* * *columns* number of elements. Function `operator()` should perform the proper pointer arithmetic to access each element of the array. There should be two versions of `operator()`—one that returns `int &` (so that an element of a `DoubleSubscriptedArray` can be used as an *lvalue*) and one that returns `const int &`. The class should also provide the following operators: `==`, `!=`, `=`, `<<` (for outputting the array in row and column format) and `>>` (for inputting the entire array contents).

11.8 *(`Complex` Class)* Consider class `Complex` shown in Figs. 11.14–11.16. The class enables operations on so-called *complex numbers*. These are numbers of the form `realPart + imaginaryPart * i`, where *i* has the value

$$\sqrt{-1}$$

a) Modify the class to enable input and output of complex numbers via overloaded `>>` and `<<` operators, respectively (you should remove the `print` function from the class).
b) Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.
c) Overload the `==` and `!=` operators to allow comparisons of complex numbers.

```cpp
// Fig. 11.14: Complex.h
// Complex class definition.
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex
{
public:
   Complex( double = 0.0, double = 0.0 ); // constructor
   Complex operator+( const Complex & ) const; // addition
   Complex operator-( const Complex & ) const; // subtraction
   void print() const; // output
private:
   double real; // real part
   double imaginary; // imaginary part
}; // end class Complex

#endif
```

Fig. 11.14 | `Complex` class definition.

```cpp
// Fig. 11.15: Complex.cpp
// Complex class member-function definitions.
#include <iostream>
#include "Complex.h" // Complex class definition
using namespace std;

// Constructor
Complex::Complex( double realPart, double imaginaryPart )
   : real( realPart ),
   imaginary( imaginaryPart )
{
   // empty body
} // end Complex constructor

// addition operator
Complex Complex::operator+( const Complex &operand2 ) const
{
   return Complex( real + operand2.real,
      imaginary + operand2.imaginary );
} // end function operator+

// subtraction operator
Complex Complex::operator-( const Complex &operand2 ) const
{
   return Complex( real - operand2.real,
      imaginary - operand2.imaginary );
} // end function operator-

```

Fig. 11.15 | `Complex` class member-function definitions. (Part 1 of 2.)

```
// display a Complex object in the form: (a, b)
void Complex::print() const
{
   cout << '(' << real << ", " << imaginary << ')';
} // end function print
```

Fig. 11.15 | Complex class member-function definitions. (Part 2 of 2.)

```
// Fig. 11.16: fig11_16.cpp
// Complex class test program.
#include <iostream>
#include "Complex.h"
using namespace std;

int main()
{
   Complex x;
   Complex y( 4.3, 8.2 );
   Complex z( 3.3, 1.1 );

   cout << "x: ";
   x.print();
   cout << "\ny: ";
   y.print();
   cout << "\nz: ";
   z.print();

   x = y + z;
   cout << "\n\nx = y + z:" << endl;
   x.print();
   cout << " = ";
   y.print();
   cout << " + ";
   z.print();

   x = y - z;
   cout << "\n\nx = y - z:" << endl;
   x.print();
   cout << " = ";
   y.print();
   cout << " - ";
   z.print();
   cout << endl;
} // end main
```

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 11.16 | Complex class test program.

11.9 *(HugeInt Class)* A machine with 32-bit integers can represent integers in the range of approximately –2 billion to +2 billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what

C++ was built to do, namely, create powerful new data types. Consider class `HugeInt` of Figs. 11.17–11.19. Study the class carefully, then answer the following:

a) Describe precisely how it operates.
b) What restrictions does the class have?
c) Overload the * multiplication operator.
d) Overload the / division operator.
e) Overload all the relational and equality operators.

[*Note:* We do not show an assignment operator or copy constructor for class `HugeInteger`, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

```
// Fig. 11.17: Hugeint.h
// HugeInt class definition.
#ifndef HUGEINT_H
#define HUGEINT_H

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

class HugeInt
{
   friend ostream &operator<<( ostream &, const HugeInt & );
public:
   static const int digits = 30; // maximum digits in a HugeInt

   HugeInt( long = 0 ); // conversion/default constructor
   HugeInt( const string & ); // conversion constructor

   // addition operator; HugeInt + HugeInt
   HugeInt operator+( const HugeInt & ) const;

   // addition operator; HugeInt + int
   HugeInt operator+( int ) const;

   // addition operator;
   // HugeInt + string that represents large integer value
   HugeInt operator+( const string & ) const;
private:
   short integer[ digits ];
}; // end class HugetInt

#endif
```

Fig. 11.17 | `HugeInt` class definition.

```
// Fig. 11.18: Hugeint.cpp
// HugeInt member-function and friend-function definitions.
#include <cctype> // isdigit function prototype
#include "Hugeint.h" // HugeInt class definition
using namespace std;

// default constructor; conversion constructor that converts
// a long integer into a HugeInt object
HugeInt::HugeInt( long value )
{
```

Fig. 11.18 | `HugeInt` member-function and `friend`-function definitions. (Part 1 of 3.)

```
   // initialize array to zero
   for ( int i = 0; i < digits; ++i )
      integer[ i ] = 0;

   // place digits of argument into array
   for ( int j = digits - 1; value != 0 && j >= 0; j-- )
   {
      integer[ j ] = value % 10;
      value /= 10;
   } // end for
} // end HugeInt default/conversion constructor

// conversion constructor that converts a character string
// representing a large integer into a HugeInt object
HugeInt::HugeInt( const string &number )
{
   // initialize array to zero
   for ( int i = 0; i < digits; ++i )
      integer[ i ] = 0;

   // place digits of argument into array
   int length = number.size();

   for ( int j = digits - length, k = 0; j < digits; ++j, ++k )
      if ( isdigit( number[ k ] ) ) // ensure that character is a digit
         integer[ j ] = number[ k ] - '0';
} // end HugeInt conversion constructor

// addition operator; HugeInt + HugeInt
HugeInt HugeInt::operator+( const HugeInt &op2 ) const
{
   HugeInt temp; // temporary result
   int carry = 0;

   for ( int i = digits - 1; i >= 0; i-- )
   {
      temp.integer[ i ] = integer[ i ] + op2.integer[ i ] + carry;

      // determine whether to carry a 1
      if ( temp.integer[ i ] > 9 )
      {
         temp.integer[ i ] %= 10;  // reduce to 0-9
         carry = 1;
      } // end if
      else // no carry
         carry = 0;
   } // end for

   return temp; // return copy of temporary object
} // end function operator+

// addition operator; HugeInt + int
HugeInt HugeInt::operator+( int op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end function operator+

```

Fig. 11.18 | HugeInt member-function and friend-function definitions. (Part 2 of 3.)

```
// addition operator;
// HugeInt + string that represents large integer value
HugeInt HugeInt::operator+( const string &op2 ) const
{
   // convert op2 to a HugeInt, then invoke
   // operator+ for two HugeInt objects
   return *this + HugeInt( op2 );
} // end operator+

// overloaded output operator
ostream& operator<<( ostream &output, const HugeInt &num )
{
   int i;

   for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= HugeInt::digits ); ++i )
      ; // skip leading zeros

   if ( i == HugeInt::digits )
      output << 0;
   else
      for ( ; i < HugeInt::digits; ++i )
         output << num.integer[ i ];

   return output;
} // end function operator<<
```

Fig. 11.18 | HugeInt member-function and friend-function definitions. (Part 3 of 3.)

```
// Fig. 11.19: fig11_19.cpp
// HugeInt test program.
#include <iostream>
#include "Hugeint.h"
using namespace std;

int main()
{
   HugeInt n1( 7654321 );
   HugeInt n2( 7891234 );
   HugeInt n3( "99999999999999999999999999999" );
   HugeInt n4( "1" );
   HugeInt n5;

   cout << "n1 is " << n1 << "\nn2 is " << n2
      << "\nn3 is " << n3 << "\nn4 is " << n4
      << "\nn5 is " << n5 << "\n\n";

   n5 = n1 + n2;
   cout << n1 << " + " << n2 << " = " << n5 << "\n\n";

   cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";

   n5 = n1 + 9;
   cout << n1 << " + " << 9 << " = " << n5 << "\n\n";

   n5 = n2 + "10000";
   cout << n2 << " + " << "10000" << " = " << n5 << endl;
} // end main
```

Fig. 11.19 | HugeInt test program. (Part 1 of 2.)

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 11.19 | HugeInt test program. (Part 2 of 2.)

11.10 *(`RationalNumber` Class)* Create a class `RationalNumber` (fractions) with the following capabilities:

a) Create a constructor that prevents a 0 denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.
b) Overload the addition, subtraction, multiplication and division operators for this class.
c) Overload the relational and equality operators for this class.

11.11 *(`Polynomial` Class)* Develop class `Polynomial`. The internal representation of a `Polynomial` is an array of terms. Each term contains a coefficient and an exponent, e.g., the term

$$2x^4$$

has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as *set* and *get* functions. The class should also provide the following overloaded operator capabilities:

a) Overload the addition operator (+) to add two `Polynomial`s.
b) Overload the subtraction operator (-) to subtract two `Polynomial`s.
c) Overload the assignment operator to assign one `Polynomial` to another.
d) Overload the multiplication operator (*) to multiply two `Polynomial`s.
e) Overload the addition assignment operator (+=), subtraction assignment operator (-=), and multiplication assignment operator (*=).

Object-Oriented Programming: Inheritance

12

Say not you know another entirely, till you have divided an inheritance with him.
—Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
—Bertrand Russell

Good as it is to inherit a library, it is better to collect one.
—Augustine Birrell

Save base authority from others' books.
—William Shakespeare

Objectives

In this chapter you'll learn:

- What inheritance is and how it promotes software reuse.
- The notions of base classes and derived classes and the relationships between them.
- The `protected` member access specifier.
- The use of constructors and destructors in inheritance hierarchies.
- The order in which constructors and destructors are called in inheritance hierarchies.
- The differences between `public`, `protected` and `private` inheritance.
- To use inheritance to customize existing software.

Outline

12.1 Introduction

This chapter continues our discussion of object-oriented programming (OOP) by introducing **inheritance**—a form of software reuse in which you create a class that absorbs an existing class's capabilities, then *customizes* or enhances them. Software reuse saves time during program development by taking advantage of proven, high-quality software.

When creating a class, instead of writing completely new data members and member functions, you can specify that the new class should **inherit** the members of an existing class. This existing class is called the **base class**, and the new class is called the **derived class**. Other programming languages, such as Java and C#, refer to the base class as the **superclass** and the derived class as the **subclass**. A derived class represents a *more specialized* group of objects.

C++ offers `public`, `protected` and `private` inheritance. In this chapter, we concentrate on `public` inheritance and briefly explain the other two. *With `public` inheritance, every object of a derived class is also an object of that derived class's base class.* However, base-class objects are *not* objects of their derived classes. For example, if we have `Vehicle` as a base class and `Car` as a derived class, then all `Car`s are `Vehicle`s, but not all `Vehicle`s are `Car`s—for example, a `Vehicle` could also be a `Truck` or a `Boat`.

We distinguish between the ***is-a*** **relationship** and the *has-a* relationship. The *is-a* relationship represents inheritance. In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class—for example, a `Car` *is a* `Vehicle`, so any attributes and behaviors of a `Vehicle` are also attributes and behaviors of a `Car`. By contrast, the *has-a* relationship represents *composition*, which was discussed in Chapter 10. In a *has-a* relationship, an object *contains* one or more objects of other classes as members. For example, a `Car` has many components—it *has a* steering wheel, *has a* brake pedal, *has a* transmission, etc.

12.2 Base Classes and Derived Classes

Figure 12.1 lists several simple examples of base classes and derived classes. Base classes tend to be *more general* and derived classes tend to be *more specific*.

Base class	Derived classes
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`, `Sphere`, `Cube`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`Account`	`CheckingAccount`, `SavingsAccount`

Fig. 12.1 | Inheritance examples.

Because every derived-class object *is an* object of its base class, and one base class can have *many* derived classes, the set of objects represented by a base class typically is *larger* than the set of objects represented by any of its derived classes. For example, the base class `Vehicle` represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived class `Car` represents a *smaller*, *more specific* subset of all vehicles.

Inheritance relationships form **class hierarchies**. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they're employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class—supplying members to other classes, a derived class—inheriting its members from other classes, or *both*.

`CommunityMember` Class Hierarchy

Let's develop a simple inheritance hierarchy with five levels (represented by the UML class diagram in Fig. 12.2). A university community has thousands of `CommunityMembers`.

These `CommunityMembers` consist of `Employees`, `Students` and alumni (each of class `Alumnus`). `Employees` are either `Faculty` or `Staff`. `Faculty` are either `Administrators` or

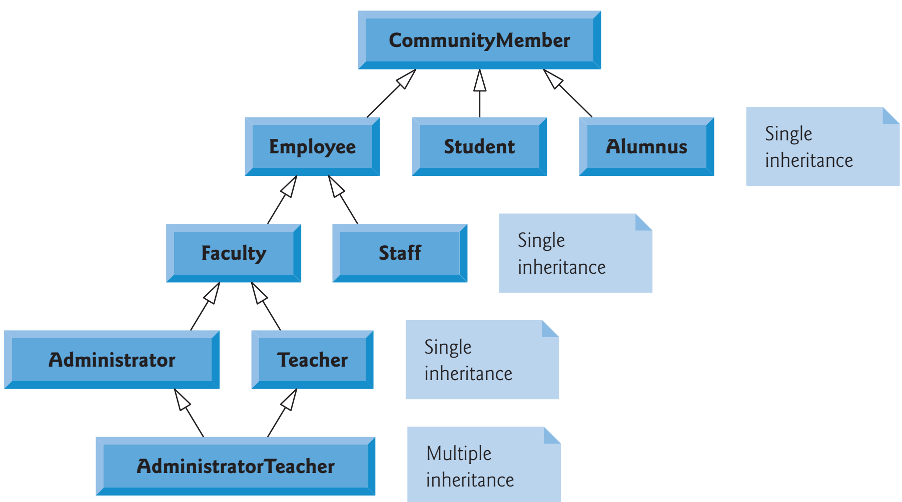


Fig. 12.2 | Inheritance hierarchy for university `CommunityMembers`.

Teachers. Some Administrators, however, are also Teachers. We've used *multiple inheritance* to form class AdministratorTeacher. With **single inheritance**, a class is derived from *one* base class. With **multiple inheritance**, a derived class inherits from *two or more* (possibly unrelated) base classes. We discuss multiple inheritance in Chapter 24, Other Topics.

Each arrow in the hierarchy (Fig. 12.2) represents an *is-a* relationship. For example, as we follow the arrows in this class hierarchy, we can state "an Employee *is a* CommunityMember" and "a Teacher *is a* Faculty member." CommunityMember is the **direct base class** of Employee, Student and Alumnus. In addition, CommunityMember is an **indirect base class** of all the other classes in the diagram. An indirect base class is inherited from two or more levels up the class hierarchy.

Starting from the bottom of the diagram, you can follow the arrows upwards and apply the *is-a* relationship to the topmost base class. For example, an AdministratorTeacher *is an* Administrator, *is a* Faculty member, *is an* Employee and *is a* CommunityMember.

Shape Class Hierarchy

Now consider the Shape inheritance hierarchy in Fig. 12.3. This hierarchy begins with base class Shape. Classes TwoDimensionalShape and ThreeDimensionalShape derive from base class Shape—a Shape *is a* TwoDimensionalShape or *is a* ThreeDimensionalShape. The third level of this hierarchy contains *more specific* types of TwoDimensionalShapes and ThreeDimensionalShapes. As in Fig. 12.2, we can follow the arrows from the bottom of the diagram upwards to the topmost base class in this hierarchy to identify several *is-a* relationships. For instance, a Triangle *is a* TwoDimensionalShape and *is a* Shape, while a Sphere *is a* ThreeDimensionalShape and *is a* Shape.

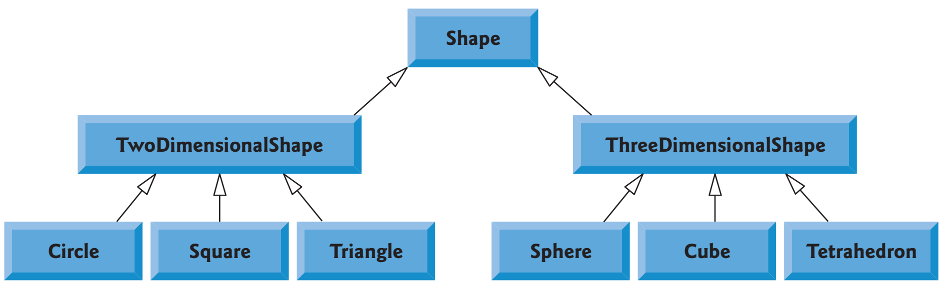


Fig. 12.3 | Inheritance hierarchy for Shapes.

To specify that class TwoDimensionalShape (Fig. 12.3) is derived from (or inherits from) class Shape, class TwoDimensionalShape's definition could begin as follows:

```
class TwoDimensionalShape : public Shape
```

This is an example of **public inheritance**, the most commonly used form. We'll also discuss **private inheritance** and **protected inheritance** (Section 12.6). With all forms of inheritance, private members of a base class are *not* accessible directly from that class's derived classes, but these private base-class members are still inherited (i.e., they're still considered parts of the derived classes). With public inheritance, all other base-class members retain their original member access when they become members of the derived class (e.g., `public` members of the base class become `public` members of the derived class, and, as we'll soon see, `protected` members of the base class become `protected` members of the derived class). Through these inherited base-class members, the derived class can manipulate `private` members of the base class (if these inherited members provide such functionality in the base class). Note that `friend` functions are *not* inherited.

Inheritance is not appropriate for every class relationship. In Chapter 10, we discussed the *has-a* relationship, in which classes have members that are objects of other classes. Such relationships create classes by *composition* of existing classes. For example, given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`. However, it is appropriate to say that an `Employee` *has a* `BirthDate` and that an `Employee` *has a* `TelephoneNumber`.

It's possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the members of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class (i.e., such objects have an *is-a* relationship with the base class). In Chapter 13, we consider many examples that take advantage of this relationship.

12.3 protected Members

Chapter 3 introduced access specifiers `public` and `private`. A base class's `public` members are accessible within its body and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that class or one of its derived classes. A base class's `private` members are accessible only within its body and to the `friend`s of that base class. In this section, we introduce the access specifier **`protected`**.

Using `protected` access offers an intermediate level of protection between `public` and `private` access. A base class's `protected` members can be accessed within the body of that base class, by members and `friend`s of that base class, and by members and `friend`s of any classes derived from that base class.

Derived-class member functions can refer to `public` and `protected` members of the base class simply by using the member names. When a derived-class member function *redefines* a base-class member function, the base-class member can still be accessed from the derived class by preceding the base-class member name with the base-class name and the scope resolution operator (`::`). We discuss accessing redefined members of the base class in Section 12.4.5 and using `protected` data in Section 12.4.4.

12.4 Relationship between Base Classes and Derived Classes

In this section, we use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a base class and a derived class. Commission employees (who will be represented as objects of a base class) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a derived class) receive a base salary plus a percentage of their sales. We divide our discussion of the relationship between commission employees and base-salaried commission employees into a carefully paced series of five examples.

12.4.1 Creating and Using a CommissionEmployee Class

Let's examine CommissionEmployee's class definition (Figs. 12.4–12.5). The CommissionEmployee header (Fig. 12.4) specifies class CommissionEmployee's public services, which include a constructor (lines 12–13) and member functions earnings (line 30) and print (line 31). Lines 15–28 declare public *get* and *set* functions that manipulate the class's data members (declared in lines 33–37) firstName, lastName, socialSecurityNumber, grossSales and commissionRate. Member functions setGrossSales (defined in lines 56–62 of Fig. 12.5) and setCommissionRate (defined in lines 71–77 of Fig. 12.5), for example, validate their arguments before assigning the values to data members grossSales and commissionRate, respectively.

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate (percentage)
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

Fig. 12.4 | CommissionEmployee class header.

```
// Fig. 12.5: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate )
{
   firstName = first; // should validate
   lastName = last; // should validate
   socialSecurityNumber = ssn; // should validate
   setGrossSales( sales ); // validate and store gross sales
   setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
   firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
   return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
   lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
```

Fig. 12.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 1 of 2.)

```
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   if ( sales >= 0.0 )
      grossSales = sales;
   else
      throw invalid_argument( "Gross sales must be >= 0.0" );
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   if ( rate > 0.0 && rate < 1.0 )
      commissionRate = rate;
   else
      throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
   return commissionRate * grossSales;
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
   cout << "commission employee: " << firstName << ' ' << lastName
      << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate;
} // end function print
```

Fig. 12.5 | Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales. (Part 2 of 2.)

`CommissionEmployee` Constructor

The `CommissionEmployee` constructor definition purposely does not use member-initializer syntax in the first several examples of this section, so that we can demonstrate how private and protected specifiers affect member access in derived classes. As shown in Fig. 12.5, lines 12–14, we assign values to data members firstName, lastName and socialSecurityNumber in the constructor body. Later in this section, we'll return to using member-initializer lists in the constructors.

We do not validate the values of the constructor's arguments first, last and ssn before assigning them to the corresponding data members. We certainly could validate the first and last names—perhaps by ensuring that they're of a reasonable length. Similarly, a social security number could be validated to ensure that it contains nine digits, with or without dashes (e.g., 123-45-6789 or 123456789).

CommissionEmployee Member Functions **earnings** *and* **print**

Member function earnings (lines 86–89) calculates a CommissionEmployee's earnings. Line 88 multiplies the commissionRate by the grossSales and returns the result. Member function print (lines 92–98) displays the values of a CommissionEmployee object's data members.

Testing Class **CommissionEmployee**

Figure 12.6 tests class CommissionEmployee. Lines 11–12 instantiate object employee of class CommissionEmployee and invoke CommissionEmployee's constructor to initialize the object with "Sue" as the first name, "Jones" as the last name, "222-22-2222" as the social security number, 10000 as the gross sales amount and .06 as the commission rate. Lines 19–24 use employee's *get* functions to display the values of its data members. Lines 26–27 invoke the object's member functions setGrossSales and setCommissionRate to change the values of data members grossSales and commissionRate, respectively. Line 31 then calls employee's print member function to output the updated CommissionEmployee information. Finally, line 34 displays the CommissionEmployee's earnings, calculated by the object's earnings member function using the updated values of data members grossSales and commissionRate.

```
1  // Fig. 12.6: fig12_06.cpp
2  // Testing class CommissionEmployee.
3  #include <iostream>
4  #include <iomanip>
5  #include "CommissionEmployee.h" // CommissionEmployee class definition
6  using namespace std;
7
8  int main()
9  {
10    // instantiate a CommissionEmployee object
11    CommissionEmployee employee(
12       "Sue", "Jones", "222-22-2222", 10000, .06 );
13
14    // set floating-point output formatting
15    cout << fixed << setprecision( 2 );
16
17    // get commission employee data
18    cout << "Employee information obtained by get functions: \n"
19       << "\nFirst name is " << employee.getFirstName()
```

Fig. 12.6 | CommissionEmployee class test program. (Part 1 of 2.)

```
         << "\nLast name is " << employee.getLastName()
         << "\nSocial security number is "
         << employee.getSocialSecurityNumber()
         << "\nGross sales is " << employee.getGrossSales()
         << "\nCommission rate is " << employee.getCommissionRate() << endl;

      employee.setGrossSales( 8000 ); // set gross sales
      employee.setCommissionRate( .1 ); // set commission rate

      cout << "\nUpdated employee information output by print function: \n"
         << endl;
      employee.print(); // display the new employee information

      // display the employee's earnings
      cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
   } // end main
```

```
Employee information obtained by get functions:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information output by print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10

Employee's earnings: $800.00
```

Fig. 12.6 | `CommissionEmployee` class test program. (Part 2 of 2.)

12.4.2 Creating a `BasePlusCommissionEmployee` Class Without Using Inheritance

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new and independent) class `BasePlusCommissionEmployee` (Figs. 12.7–12.8), which contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary.

```
// Fig. 12.7: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class definition represents an employee
// that receives a base salary in addition to commission.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using namespace std;

```

Fig. 12.7 | `BasePlusCommissionEmployee` class header. (Part 1 of 2.)

```
class BasePlusCommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   double earnings() const; // calculate earnings
   void print() const; // print BasePlusCommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Fig. 12.7 | BasePlusCommissionEmployee class header. (Part 2 of 2.)

```
// Fig. 12.8: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
{
```

Fig. 12.8 | BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission. (Part 1 of 4.)

```
   firstName = first; // should validate
   lastName = last; // should validate
   socialSecurityNumber = ssn; // should validate
   setGrossSales( sales ); // validate and store gross sales
   setCommissionRate( rate ); // validate and store commission rate
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set first name
void BasePlusCommissionEmployee::setFirstName( const string &first )
{
   firstName = first; // should validate
} // end function setFirstName

// return first name
string BasePlusCommissionEmployee::getFirstName() const
{
   return firstName;
} // end function getFirstName

// set last name
void BasePlusCommissionEmployee::setLastName( const string &last )
{
   lastName = last; // should validate
} // end function setLastName

// return last name
string BasePlusCommissionEmployee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void BasePlusCommissionEmployee::setSocialSecurityNumber(
   const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string BasePlusCommissionEmployee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void BasePlusCommissionEmployee::setGrossSales( double sales )
{
   if ( sales >= 0.0 )
      grossSales = sales;
```

Fig. 12.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 4.)

```
   else
      throw invalid_argument( "Gross sales must be >= 0.0" );
} // end function setGrossSales

// return gross sales amount
double BasePlusCommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

// set commission rate
void BasePlusCommissionEmployee::setCommissionRate( double rate )
{
   if ( rate > 0.0 && rate < 1.0 )
      commissionRate = rate;
   else
      throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
} // end function setCommissionRate

// return commission rate
double BasePlusCommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   if ( salary >= 0.0 )
      baseSalary = salary;
   else
      throw invalid_argument( "Salary must be >= 0.0" );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
   return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
   return baseSalary + ( commissionRate * grossSales );
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
```

Fig. 12.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 4.)

```
113        << "\ngross sales: " << grossSales
114        << "\ncommission rate: " << commissionRate
115        << "\nbase salary: " << baseSalary;
116 } // end function print
```

Fig. 12.8 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 4.)

Defining Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployee` header (Fig. 12.7) specifies class `BasePlusCommissionEmployee`'s `public` services, which include the `BasePlusCommissionEmployee` constructor (lines 13–14) and member functions `earnings` (line 34) and `print` (line 35). Lines 16–32 declare `public` *get* and *set* functions for the class's `private` data members (declared in lines 37–42) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` and `baseSalary`. These variables and member functions encapsulate all the necessary features of a base-salaried commission employee. Note the similarity between this class and class `CommissionEmployee` (Figs. 12.4–12.5)—in this example, we do not yet exploit that similarity.

Class `BasePlusCommissionEmployee`'s `earnings` member function (defined in lines 100–103 of Fig. 12.8) computes the earnings of a base-salaried commission employee. Line 102 returns the result of adding the employee's base salary to the product of the commission rate and the employee's gross sales.

Testing Class `BasePlusCommissionEmployee`

Figure 12.9 tests class `BasePlusCommissionEmployee`. Lines 11–12 instantiate object `employee` of class `BasePlusCommissionEmployee`, passing `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` and `300` to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 19–25 use `BasePlusCommissionEmployee`'s *get* functions to retrieve the values of the object's data members for output. Line 27 invokes the object's `setBaseSalary` member function to change the base salary. Member function `setBaseSalary` (Fig. 12.8, lines 88–94) ensures that data member `baseSalary` is not assigned a negative value, because an employee's base salary cannot be negative. Line 31 of Fig. 12.9 invokes the object's `print` member function to output the updated `BasePlusCommissionEmployee`'s information, and line 34 calls member function `earnings` to display the `BasePlusCommissionEmployee`'s earnings.

```
1  // Fig. 12.9: fig12_09.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  #include <iomanip>
5  #include "BasePlusCommissionEmployee.h"
6  using namespace std;
7
8  int main()
9  {
```

Fig. 12.9 | `BasePlusCommissionEmployee` class test program. (Part 1 of 2.)

```
    // instantiate BasePlusCommissionEmployee object
    BasePlusCommissionEmployee
       employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

    // set floating-point output formatting
    cout << fixed << setprecision( 2 );

    // get commission employee data
    cout << "Employee information obtained by get functions: \n"
       << "\nFirst name is " << employee.getFirstName()
       << "\nLast name is " << employee.getLastName()
       << "\nSocial security number is "
       << employee.getSocialSecurityNumber()
       << "\nGross sales is " << employee.getGrossSales()
       << "\nCommission rate is " << employee.getCommissionRate()
       << "\nBase salary is " << employee.getBaseSalary() << endl;

    employee.setBaseSalary( 1000 ); // set base salary

    cout << "\nUpdated employee information output by print function: \n"
       << endl;
    employee.print(); // display the new employee information

    // display the employee's earnings
    cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
 } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Fig. 12.9 | `BasePlusCommissionEmployee` class test program. (Part 2 of 2.)

Exploring the Similarities Between Class `BasePlusCommissionEmployee` *and Class* `CommissionEmployee`

Most of the code for class `BasePlusCommissionEmployee` (Figs. 12.7–12.8) is similar, if not identical, to the code for class `CommissionEmployee` (Figs. 12.4–12.5). For example, in class `BasePlusCommissionEmployee`, `private` data members `firstName` and `lastName` and member functions `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. Classes `CommissionEmployee` and `BasePlusCommissionEmployee` also both contain `private` data members `socialSecurityNumber`, `commissionRate` and `grossSales`, as well as *get* and *set* functions to manipulate these members. In addition, the `BasePlusCommissionEmployee` constructor is *almost* identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are `private` data member `baseSalary` and member functions `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s `print` member function is *nearly identical* to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s `print` also outputs the value of data member `baseSalary`.

We literally *copied* code from class `CommissionEmployee` and *pasted* it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and member functions that manipulate the base salary. This *copy-and-paste approach* is error prone and time consuming.

Software Engineering Observation 12.1

Copying and pasting code *from one class to another can spread many physical* copies of the same code *and can spread* errors *throughout a system, creating a code-maintenance nightmare. To avoid duplicating code (and possibly errors), use* inheritance, *rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.*

Software Engineering Observation 12.2

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, you need to make the changes only *in the base class—derived classes then inherit the changes. Without inheritance, changes would need to be made to* all *the source code files that contain a copy of the code in question.*

12.4.3 Creating a `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy

Now we create and test a new `BasePlusCommissionEmployee` class (Figs. 12.10–12.11) that *derives from* class `CommissionEmployee` (Figs. 12.4–12.5). In this example, a `BasePlusCommissionEmployee` object *is a* `CommissionEmployee` (because inheritance passes on the capabilities of class `CommissionEmployee`), but class `BasePlusCommissionEmployee` *also* has data member `baseSalary` (Fig. 12.10, line 23). The *colon (:)* in line 11 of the class definition indicates inheritance. Keyword `public` indicates the *type of inheritance*. As a derived class (formed with `public` inheritance), `BasePlusCommissionEmployee` inherits *all* the members of class `CommissionEmployee`, *except* for the constructor—each class provides its *own* constructors that are specific to the class. (Destructors, too, are not inherited.) Thus, the `public` services of `BasePlusCommissionEmployee` include its constructor (lines 14–15) and the `public` member functions inherited from class `CommissionEmployee`—*although we cannot see these inherited member functions* in `BasePlusCommissionEmployee`'s source code, they're nevertheless a part of derived class `BasePlusCommissionEmployee`. The derived class's `public` services also include member functions `setBaseSalary`, `getBaseSalary`, `earnings` and `print` (lines 17–21).

```
// Fig. 12.10: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   double earnings() const; // calculate earnings
   void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Fig. 12.10 | BasePlusCommissionEmployee class definition indicating inheritance relationship with class CommissionEmployee.

```
// Fig. 12.11: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   // explicitly call base-class constructor
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
```

Fig. 12.11 | BasePlusCommissionEmployee implementation file: private base-class data cannot be accessed from derived class. (Part 1 of 3.)

```
   if ( salary >= 0.0 )
      baseSalary = salary;
   else
      throw invalid_argument( "Salary must be >= 0.0" );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
   return baseSalary;
} // end function getBaseSalary

// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
   // derived class cannot access the base class's private data
   return baseSalary + ( commissionRate * grossSales );
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
   // derived class cannot access the base class's private data
   cout << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate
      << "\nbase salary: " << baseSalary;
} // end function print
```

```
C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(36) :
   error C2248: 'CommissionEmployee::commissionRate' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(36) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(43) :
   error C2248: 'CommissionEmployee::firstName' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(44) :
   error C2248: 'CommissionEmployee::lastName' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(44) :
   error C2248: 'CommissionEmployee::socialSecurityNumber' :
   cannot access private member declared in class 'CommissionEmployee'

C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(45) :
   error C2248: 'CommissionEmployee::grossSales' :
   cannot access private member declared in class 'CommissionEmployee'
```

Fig. 12.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 2 of 3.)

```
C:\chhhtp8_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(46) :
   error C2248: 'CommissionEmployee::commissionRate' :
   cannot access private member declared in class 'CommissionEmployee'
```

Fig. 12.11 | `BasePlusCommissionEmployee` implementation file: `private` base-class data cannot be accessed from derived class. (Part 3 of 3.)

Figure 12.11 shows `BasePlusCommissionEmployee`'s member-function implementations. The constructor (lines 8–15) introduces **base-class initializer syntax** (line 12), which uses a member initializer to pass arguments to the base-class (`CommissionEmployee`) constructor. C++ requires that a derived-class constructor call its base-class constructor to initialize the base-class data members that are inherited into the derived class. Line 12 does this by *explicitly* invoking the `CommissionEmployee` constructor by name, passing the constructor's parameters `first`, `last`, `ssn`, `sales` and `rate` as arguments to initialize the base-class data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. If `BasePlusCommissionEmployee`'s constructor did *not* invoke class `CommissionEmployee`'s constructor *explicitly*, C++ would attempt to invoke class `CommissionEmployee`'s default constructor implicitly—but the class does *not* have such a constructor, so the compiler would issue an error. Recall from Chapter 3 that the compiler provides a default constructor with no parameters in any class that does *not* explicitly include a constructor. However, `CommissionEmployee` *does* explicitly include a constructor, so a default constructor is *not* provided

Common Programming Error 12.1

When a derived-class constructor calls a base-class constructor, the arguments passed to the base-class constructor must be consistent with the number and types of parameters specified in one of the base-class constructors; otherwise, a compilation error occurs.

Performance Tip 12.1

In a derived-class constructor, invoking base-class constructors and initializing member object explicitly in the member initializer list prevents duplicate initialization in which a default constructor is called, then data members are modified again in the derived-class constructor's body.

Compilation Errors from Accessing Base-Class `private` Members

The compiler generates errors for line 36 of Fig. 12.11 because base class `CommissionEmployee`'s data members `commissionRate` and `grossSales` are `private`—derived class `BasePlusCommissionEmployee`'s member functions are *not* allowed to access base class `CommissionEmployee`'s `private` data. The compiler issues additional errors in lines 43–46 of `BasePlusCommissionEmployee`'s `print` member function for the same reason. As you can see, C++ rigidly enforces restrictions on accessing `private` data members, so that *even a derived class (which is intimately related to its base class) cannot access the base class's `private` data.*

Preventing the Errors in `BasePlusCommissionEmployee`

We purposely included the erroneous code in Fig. 12.11 to emphasize that a derived class's member functions cannot access its base class's `private` data. The errors in `BasePlusCom-`

missionEmployee could have been prevented by using the *get* member functions inherited from class CommissionEmployee. For example, line 36 could have invoked getCommissionRate and getGrossSales to access CommissionEmployee's private data members commissionRate and grossSales, respectively. Similarly, lines 43–46 could have used appropriate *get* member functions to retrieve the values of the base class's data members. In the next example, we show how using protected data *also* allows us to avoid the errors encountered in this example.

Including the Base-Class Header in the Derived-Class Header with #include

Notice that we #include the base class's header in the derived class's header (line 8 of Fig. 12.10). This is necessary for three reasons. First, for the derived class to use the base class's name in line 11, we must tell the compiler that the base class exists—the class definition in CommissionEmployee.h does exactly that.

The second reason is that the compiler uses a class definition to determine the *size* of an object of that class (as we discussed in Section 3.6). A client program that creates an object of a class must #include the class definition to enable the compiler to reserve the proper amount of memory for the object. When using inheritance, a derived-class object's size depends on the data members declared explicitly in its class definition *and* the data members inherited from its direct and indirect base classes. Including the base class's definition in line 8 allows the compiler to determine the memory requirements for the base class's data members that become part of a derived-class object and thus contribute to the total size of the derived-class object.

The last reason for line 8 is to allow the compiler to determine whether the derived class uses the base class's inherited members properly. For example, in the program of Figs. 12.10–12.11, the compiler uses the base-class header to determine that the data members being accessed by the derived class are private in the base class. Since these are *inaccessible* to the derived class, the compiler generates errors. The compiler also uses the base class's *function prototypes* to *validate* function calls made by the derived class to the inherited base-class functions.

Linking Process in an Inheritance Hierarchy

In Section 3.7, we discussed the linking process for creating an executable GradeBook application. In that example, you saw that the client's object code was linked with the object code for class GradeBook, as well as the object code for any C++ Standard Library classes used in either the client code or in class GradeBook.

The linking process is similar for a program that uses classes in an inheritance hierarchy. The process requires the object code for all classes used in the program and the object code for the direct and indirect base classes of any derived classes used by the program. Suppose a client wants to create an application that uses class BasePlusCommissionEmployee, which is a derived class of CommissionEmployee (we'll see an example of this in Section 12.4.4). When compiling the client application, the client's object code must be linked with the object code for classes BasePlusCommissionEmployee and CommissionEmployee, because BasePlusCommissionEmployee inherits member functions from its base class CommissionEmployee. The code is also linked with the object code for any C++ Standard Library classes used in class CommissionEmployee, class BasePlusCommissionEmployee or the client code. This provides the program with access to the implementations of all of the functionality that the program may use.

12.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

To enable class `BasePlusCommissionEmployee` to directly access `CommissionEmployee` data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, we can declare those members as `protected` in the base class. As we discussed in Section 12.3, a base class's `protected` members can be accessed by members and `friend`s of the base class and by members and `friend`s of any classes derived from that base class.

*Defining Base Class **CommissionEmployee** with **protected** Data*

Class `CommissionEmployee` (Fig. 12.12) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `protected` (lines 32–37) rather than `private`. The member-function implementations are identical to those in Fig. 12.5, so `CommissionEmployee.cpp` is not shown here.

```
1  // Fig. 12.12: CommissionEmployee.h
2  // CommissionEmployee class definition with protected data.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 protected:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
```

Fig. 12.12 | `CommissionEmployee` class definition that declares `protected` data to allow access by derived classes. (Part 1 of 2.)

```
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Fig. 12.12 | `CommissionEmployee` class definition that declares `protected` data to allow access by derived classes. (Part 2 of 2.)

Class `BasePlusCommissionEmployee`

The definition of class `BasePlusCommissionEmployee` from Figs. 12.10–12.11 remains unchanged, so we do not show it again here. Now that `BasePlusCommissionEmployee` inherits from the updated class `CommissionEmployee` Fig. 12.12, `BasePlusCommissionEmployee` objects can access inherited data members that are declared `protected` in class `CommissionEmployee` (i.e., data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). As a result, the compiler does *not* generate errors when compiling the `BasePlusCommissionEmployee` `earnings` and `print` member-function definitions in Fig. 12.11 (lines 33–37 and 40–48, respectively). This shows the special privileges that a derived class is granted to access `protected` base-class data members. Objects of a derived class also can access `protected` members in *any* of that derived class's *indirect* base classes.

Class `BasePlusCommissionEmployee` does *not* inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s constructor (Fig. 12.11, lines 8–15) calls class `CommissionEmployee`'s constructor explicitly with member initializer syntax (line 12). Recall that `BasePlusCommissionEmployee`'s constructor must explicitly call the constructor of class `CommissionEmployee`, because `CommissionEmployee` does not contain a default constructor that could be invoked implicitly.

Testing the Modified `BasePlusCommissionEmployee` *Class*

To test the updated class hierarchy, we reused the test program from Fig. 12.9. As shown in Fig. 12.13, the output is identical to that of Fig. 12.9. We created the first class `BasePlusCommissionEmployee` *without using inheritance* and created this version of `BasePlusCommissionEmployee` *using inheritance*; however, both classes provide the *same* functionality. The code for class `BasePlusCommissionEmployee` (i.e., the header and implementation files), which is 74 lines, is considerably *shorter* than the code for the noninherited version of the class, which is 161 lines, because the inherited version absorbs part of its functionality from `CommissionEmployee`, whereas the noninherited version does not absorb any functionality. Also, there is now only *one* copy of the `CommissionEmployee` functionality declared and defined in class `CommissionEmployee`. This makes the source code easier to maintain, modify and debug, because the source code related to a `CommissionEmployee` exists only in the files `CommissionEmployee.h` and `CommissionEmployee.cpp`.

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
```

Fig. 12.13 | `protected` base-class data can be accessed from derived class. (Part 1 of 2.)

```
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Fig. 12.13 | `protected` base-class data can be accessed from derived class. (Part 2 of 2.)

Notes on Using `protected` *Data*

In this example, we declared base-class data members as `protected`, so derived classes can modify the data directly. Inheriting `protected` data members slightly improves performance, because we can directly access the members without incurring the overhead of calls to *set* or *get* member functions.

> **Software Engineering Observation 12.3**
> *In most cases, it's better to use* `private` *data members to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.*

Using `protected` data members creates two serious problems. First, the derived-class object does *not* have to use a member function to set the value of the base class's `protected` data member. An invalid value can easily be assigned to the `protected` data member, thus leaving the object in an *inconsistent* state—e.g., with `CommissionEmployee`'s data member `grossSales` declared as `protected`, a derived-class object can assign a negative value to `grossSales`. The second problem with using `protected` data members is that derived-class member functions are more likely to be written so that they *depend on the base-class implementation*. Derived classes should depend only on the base-class services (i.e., non-`private` member functions) and *not* on the base-class implementation. With `protected` data members in the base class, if the base-class implementation changes, we may need to modify *all* derived classes of that base class. For example, if for some reason we were to change the names of data members `firstName` and `lastName` to `first` and `last`, then we'd have to do so for all occurrences in which a derived class references these base-class data members directly. Such software is said to be **fragile** or **brittle**, because a small change in the base class can "break" derived-class implementation. You should be able to change the base-class implementation while still providing the same services to derived classes. Of course, if the base-class services change, we must reimplement our derived classes—good object-oriented design attempts to prevent this.

> **Software Engineering Observation 12.4**
> *It's appropriate to use the* `protected` *access specifier when a base class should provide a service (i.e., a member function) only to its derived classes and* `friend`*s.*

Software Engineering Observation 12.5

Declaring base-class data members `private` *(as opposed to declaring them* `protected`*) enables you to change the base-class implementation without having to change derived-class implementations.*

12.4.5 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `private` Data

We now reexamine our hierarchy once more, this time using the best software engineering practices. Class `CommissionEmployee` now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `private` as shown previously in lines 32–37 of Fig. 12.4.

Changes to Class `CommissionEmployee`'s Member Function Definitions

In the `CommissionEmployee` constructor implementation (Fig. 12.14, lines 8–15), we use member initializers (line 11) to set the values of the members `firstName`, `lastName` and `socialSecurityNumber`. We show how the derived-class `BasePlusCommissionEmployee` (Fig. 12.15) can invoke non-`private` base-class member functions (`setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber` and `getSocialSecurityNumber`) to manipulate these data members.

In the body of the constructor and in the bodies of member function's earnings (lines 84–87) and print (lines 90–97), we call the class's *set* and *get* member functions to access the class's private data members. If we decide to change the data member names, the `earnings` and `print` definitions will *not* require modification—only the definitions of the *get* and *set* member functions that directly manipulate the data members will need to change. These changes occur solely within the base class—no changes to the derived class are needed. Localizing the effects of changes like this is a good software engineering practice.

Performance Tip 12.2

Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, today's optimizing compilers are carefully designed to perform many optimizations implicitly (such as inlining set *and* get *member-function calls). You should write code that adheres to proper software engineering principles, and leave optimization to the compiler. A good rule is, "Do not second-guess the compiler."*

```
1  // Fig. 12.14: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  #include "CommissionEmployee.h" // CommissionEmployee class definition
5  using namespace std;
6
7  // constructor
8  CommissionEmployee::CommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate )
```

Fig. 12.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 1 of 3.)

```
   : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
   setGrossSales( sales ); // validate and store gross sales
   setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
   firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
   return firstName;
} // end function getFirstName

// set last name
void CommissionEmployee::setLastName( const string &last )
{
   lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   if ( sales >= 0.0 )
      grossSales = sales;
   else
      throw invalid_argument( "Gross sales must be >= 0.0" );
} // end function setGrossSales

```

Fig. 12.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 2 of 3.)

```
// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
   return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   if ( rate > 0.0 && rate < 1.0 )
      commissionRate = rate;
   else
      throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee object
void CommissionEmployee::print() const
{
   cout << "commission employee: "
      << getFirstName() << ' ' << getLastName()
      << "\nsocial security number: " << getSocialSecurityNumber()
      << "\ngross sales: " << getGrossSales()
      << "\ncommission rate: " << getCommissionRate();
} // end function print
```

Fig. 12.14 | `CommissionEmployee` class implementation file: `CommissionEmployee` class uses member functions to manipulate its `private` data. (Part 3 of 3.)

Changes to Class `BasePlusCommissionEmployee`'s Member Function Definitions

Class `BasePlusCommissionEmployee` inherits `CommissionEmployee`'s `public` member functions and can access the `private` base-class members via the inherited member functions. Class `BasePlusCommissionEmployee`'s header remains unchanged from Fig. 12.10. The class has several changes to its member-function implementations (Fig. 12.15) that distinguish it from the previous version of the class (Figs. 12.10–12.11). Member functions `earnings` (Fig. 12.15, lines 33–36) and `print` (lines 39–47) each invoke member function `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. This insulates `earnings` and `print` from potential changes to the implementation of data member `baseSalary`. For example, if we decide to rename data member `baseSalary` or change its type, only member functions `setBaseSalary` and `getBaseSalary` will need to change.

```
1  // Fig. 12.15: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  #include "BasePlusCommissionEmployee.h"
5  using namespace std;
6
7  // constructor
8  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
9     const string &first, const string &last, const string &ssn,
10    double sales, double rate, double salary )
11    // explicitly call base-class constructor
12    : CommissionEmployee( first, last, ssn, sales, rate )
13 {
14    setBaseSalary( salary ); // validate and store base salary
15 } // end BasePlusCommissionEmployee constructor
16
17 // set base salary
18 void BasePlusCommissionEmployee::setBaseSalary( double salary )
19 {
20    if ( salary >= 0.0 )
21       baseSalary = salary;
22    else
23       throw invalid_argument( "Salary must be >= 0.0" );
24 } // end function setBaseSalary
25
26 // return base salary
27 double BasePlusCommissionEmployee::getBaseSalary() const
28 {
29    return baseSalary;
30 } // end function getBaseSalary
31
32 // calculate earnings
33 double BasePlusCommissionEmployee::earnings() const
34 {
                                 CommissionEmployee::earnings()
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41    cout << "base-salaried ";
42
43    // invoke CommissionEmployee's print function
44    CommissionEmployee::print();
45
46    cout << "\nbase salary: " << getBaseSalary();
47 } // end function print
```

Fig. 12.15 | BasePlusCommissionEmployee class that inherits from class CommissionEmployee but cannot directly access the class's private data.

BasePlusCommissionEmployee* *Member Function* *earnings

Class BasePlusCommissionEmployee's earnings function (Fig. 12.15, lines 33–36) redefines class CommissionEmployee's earnings member function (Fig. 12.14, lines 84–87) to calculate the earnings of a base-salaried commission employee. Class `BasePlusCommissionEmployee`'s version of `earnings` obtains the portion of the employee's earnings based on commission alone by calling base-class `CommissionEmployee`'s `earnings` function with the expression `CommissionEmployee::earnings()` (Fig. 12.15, line 35). `BasePlusCommissionEmployee`'s `earnings` function then adds the base salary to this value to calculate the total earnings of the employee. Note the syntax used to invoke a redefined base-class member function from a derived class—place the base-class name and the scope resolution operator (`::`) before the base-class member-function name. This member-function invocation is a good software engineering practice: Recall from Chapter 9 that, if an object's member function performs the actions needed by another object, we should call that member function rather than duplicating its code body. By having `BasePlusCommissionEmployee`'s `earnings` function invoke `CommissionEmployee`'s `earnings` function to calculate part of a `BasePlusCommissionEmployee` object's earnings, we avoid duplicating the code and reduce code-maintenance problems.

Common Programming Error 12.2

When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the `::` *operator prefixed with the name of the base class when referencing the base class's member function causes* infinite recursion*, because the derived-class member function would then call itself.*

`BasePlusCommissionEmployee` *Member Function* `print`

Similarly, `BasePlusCommissionEmployee`'s `print` function (Fig. 12.15, lines 39–47) redefines class `CommissionEmployee`'s `print` function (Fig. 12.14, lines 90–97) to output the appropriate base-salaried commission employee information. The new version displays part of a `BasePlusCommissionEmployee` object's information (i.e., the string `"commission employee"` and the values of class `CommissionEmployee`'s `private` data members) by calling `CommissionEmployee`'s `print` member function with the qualified name `CommissionEmployee::print()` (Fig. 12.15, line 44). `BasePlusCommissionEmployee`'s `print` function then outputs the remainder of a `BasePlusCommissionEmployee` object's information (i.e., the value of class `BasePlusCommissionEmployee`'s base salary).

Testing the Modified Class Hierarchy

Once again, this example uses the `BasePlusCommissionEmployee` test program from Fig. 12.9 and produces the same output. Although each "base-salaried commission employee" class behaves identically, the version in this example is the best engineered. *By using inheritance and by calling member functions that hide the data and ensure consistency, we've efficiently and effectively constructed a well-engineered class.*

Summary of the `CommissionEmployee`–`BasePlusCommissionEmployee` *Examples*

In this section, you saw an evolutionary set of examples that was carefully designed to teach key capabilities for good software engineering with inheritance. You learned how to create a derived class using inheritance, how to use `protected` base-class members to enable a derived class to access inherited base-class data members and how to redefine base-class functions to provide versions that are more appropriate for derived-class objects. In addition, you learned how to apply software engineering techniques from Chapters 9–10 and this chapter to create classes that are easy to maintain, modify and debug.

12.5 Constructors and Destructors in Derived Classes

As we explained in the preceding section, instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor). Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the one of the class at the base of the hierarchy, whose body actually finishes executing *first*. The original derived-class constructor's body finishes executing *last*. Each base-class constructor initializes the base-class data members that the derived-class object inherits. In the `CommissionEmployee`/`BasePlusCommissionEmployee` hierarchy that we've been studying, when a program creates an object of class `BasePlusCommissionEmployee`, the `CommissionEmployee` constructor is called. Since class `CommissionEmployee` is at the base of the hierarchy, its constructor executes, initializing the `private` data members of `CommissionEmployee` that are part of the `BasePlusCommissionEmployee` object. When `CommissionEmployee`'s constructor completes execution, it returns control to `BasePlusCommissionEmployee`'s constructor, which initializes the `BasePlusCommissionEmployee` object's `baseSalary`.

Software Engineering Observation 12.6

When a program creates a derived-class object, the derived-class constructor immediately calls the base-class constructor, the base-class constructor's body executes, then the derived class's member initializers *execute and finally the derived-class constructor's body executes. This process cascades up the hierarchy if it contains more than two levels.*

When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in *reverse* of the order in which the constructors executed. When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

Software Engineering Observation 12.7

Suppose that we create an object of a derived class where both the base class and the derived class contain (via composition) objects of other classes. When an object of that derived class is created, first the constructors for the base class's member objects execute, then the base-class constructor body executes, then the constructors for the derived class's member objects execute, then the derived class's constructor body executes. Destructors for derived-class objects are called in the reverse of the order in which their corresponding constructors are called.

Base-class constructors, destructors and overloaded assignment operators (Chapter 11) are *not* inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

12.6 `public`, `protected` and `private` Inheritance

When deriving a class from a base class, the base class may be inherited through `public`, `protected` or `private` inheritance. We normally use `public` inheritance in this book. Use of `protected` inheritance is rare. Chapter 20 demonstrates `private` inheritance as an alternative to composition. Figure 12.16 summarizes for each type of inheritance the accessibility of base-class members in a derived class. The first column contains the base-class access specifiers.

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Fig. 12.16 | Summary of base-class member accessibility in a derived class.

When deriving a class from a `public` base class, `public` members of the base class become `public` members of the derived class, and `protected` members of the base class become `protected` members of the derived class. A base class's `private` members are *never* accessible directly from a derived class, but can be accessed through calls to the `public` and `protected` members of the base class.

When deriving from a `protected` base class, `public` and `protected` members of the base class become `protected` members of the derived class. When deriving from a `private` base class, `public` and `protected` members of the base class become `private` members (e.g., the functions become utility functions) of the derived class. `Private` and `protected` inheritance are not *is-a* relationships.

12.7 Software Engineering with Inheritance

Sometimes it's difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software development process. Object-oriented programming facilitates software reuse, thus shortening development times and enhancing software quality.

When we use inheritance to create a new class from an existing one, the new class inherits the data members and member functions of the existing class, as described in Fig. 12.16. We can customize the new class to meet our needs by redefining base-class members and by including additional members. The derived-class programmer does this in C++ *without* accessing the base class's source code (the derived class must be able to *link* to the base class's object code). This powerful capability is attractive to software developers. They can develop proprietary classes for sale or license and make these classes available to users in object-code format. Users then can derive new classes from these library classes rapidly and without accessing the proprietary source code. The software developers need to supply the headers along with the object code

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. Interest in creating and selling class libraries is growing exponentially. The standard C++ libraries tend to be general purpose and limited in scope. There is a worldwide commitment to the development of class libraries for a huge variety of application arenas.

Software Engineering Observation 12.8

At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should "factor out" common attributes and behaviors and place these in a base class, then use inheritance to form derived classes.

Software Engineering Observation 12.9

Creating a derived class does not affect its base class's source code. Inheritance preserves the integrity of a base class.

12.8 Wrap-Up

This chapter introduced inheritance—the ability to create a class by absorbing an existing class's data members and member functions and embellishing them with new capabilities. Through a series of examples using an employee inheritance hierarchy, you learned the notions of base classes and derived classes and used `public` inheritance to create a derived class that inherits members from a base class. The chapter introduced the access specifier `protected`—derived-class member functions can access `protected` base-class members. You learned how to access redefined base-class members by qualifying their names with the base-class name and scope resolution operator (`::`). You also saw the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Finally, we explained the three types of inheritance—`public`, `protected` and `private`—and the accessibility of base-class members in a derived class when using each type.

In Chapter 13, Object-Oriented Programming: Polymorphism, we build on our discussion of inheritance by introducing polymorphism—an object-oriented concept that enables us to write programs that handle, in a more general manner, objects of a wide variety of classes related by inheritance. After studying Chapter 13, you'll be familiar with classes, objects, encapsulation, inheritance and polymorphism—the essential concepts of object-oriented programming.

Summary

Section 12.1 Introduction

- Software reuse reduces program development time and cost.
- Inheritance (p. 500) is a form of software reuse in which you create a class that absorbs an existing class's capabilities, then customizes or enhances them. The existing class is called the base class (p. 500), and the new class is referred to as the derived class (p. 500).
- Every object of a derived class is also an object of that class's base class. However, a base-class object is not an object of that class's derived classes.
- The *is-a* relationship (p. 500) represents inheritance. In an *is-a* relationship, an object of a derived class also can be treated as an object of its base class.
- The *has-a* relationship (p. 500) represents composition—an object contains one or more objects of other classes as members, but does not disclose their behavior directly in its interface.

Section 12.2 Base Classes and Derived Classes

- A direct base class (p. 502) is the one from which a derived class explicitly inherits. An indirect base class (p. 502) is inherited from two or more levels up the class hierarchy (p. 501).
- With single inheritance (p. 502), a class is derived from one base class. With multiple inheritance (p. 502), a class inherits from multiple (possibly unrelated) base classes.
- A derived class represents a more specialized group of objects.
- Inheritance relationships form class hierarchies.
- It's possible to treat base-class objects and derived-class objects similarly; the commonality shared between the object types is expressed in the base class's data members and member functions.

Section 12.3 `protected` Members

- A base class's `public` members are accessible anywhere that the program has a handle to an object of that base class or to an object of one of that base class's derived classes—or, when using the scope resolution operator, whenever the class's name is in scope.
- A base class's `private` members are accessible only within the base class or from its friends.
- A base class's `protected` members can be accessed by members and `friend`s of that base class and by members and `friend`s of any classes derived from that base class.
- When a derived-class member function redefines a base-class member function, the base-class member function can still be accessed from the derived class by qualifying the base-class member function name with the base-class name and the scope resolution operator (`::`).

Section 12.5 Constructors and Destructors in Derived Classes

- When an object of a derived class is instantiated, the base class's constructor is called immediately to initialize the base-class data members in the derived-class object, then the derived-class constructor initializes the additional derived-class data members.
- When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors—first the derived-class destructor is called, then the base-class destructor is called.

Section 12.6 `public`, `protected` and `private` Inheritance

- Declaring data members `private`, while providing non-`private` member functions to manipulate and perform validity checking on this data, enforces good software engineering.
- When deriving a class, the base class may be declared as either `public`, `protected` or `private`.

- When deriving a class from a `public` base class (p. 502), `public` members of the base class become `public` members of the derived class, and `protected` members of the base class become `protected` members of the derived class.
- When deriving a class from a `protected` base class (p. 502), `public` and `protected` members of the base class become `protected` members of the derived class.
- When deriving a class from a `private` base class (p. 502), `public` and `protected` members of the base class become `private` members of the derived class.

Self-Review Exercises

12.1 Fill in the blanks in each of the following statements:

a) ________ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

b) A base class's ________ members can be accessed in the base-class definition, in derived-class definitions and in `friend`s of the base class its derived classes.

c) In a(n) ________ relationship, an object of a derived class also can be treated as an object of its base class.

d) In a(n) ________ relationship, a class object has one or more objects of other classes as members.

e) In single inheritance, a class exists in a(n) ________ relationship with its derived classes.

f) A base class's ________ members are accessible within that base class and anywhere that the program has a handle to an object of that class or one of its derived classes.

g) A base class's `protected` access members have a level of protection between those of `public` and ________ access.

h) C++ provides for ________, which allows a derived class to inherit from many base classes, even if the base classes are unrelated.

i) When an object of a derived class is instantiated, the base class's ________ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

j) When deriving a class from a base class with `public` inheritance, `public` members of the base class become ________ members of the derived class, and `protected` members of the base class become ________ members of the derived class.

k) When deriving a class from a base class with `protected` inheritance, `public` members of the base class become ________ members of the derived class, and `protected` members of the base class become ________ members of the derived class.

12.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Base-class constructors are not inherited by derived classes.

b) A *has-a* relationship is implemented via inheritance.

c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.

d) Inheritance encourages the reuse of proven high-quality software.

e) When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

Answers to Self-Review Exercises

12.1 a) Inheritance. b) `protected`. c) *is-a* or inheritance (for `public` inheritance). d) *has-a* or composition or aggregation. e) hierarchical. f) `public`. g) `private`. h) multiple inheritance. i) constructor. j) `public`, `protected`. k) `protected`, `protected`.

12.2 a) True. b) False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance. c) False. This is an example of a *has-a* relationship. Class `Car` has an *is-a* relationship with class `Vehicle`. d) True. e) True.

Exercises

12.3 *(Composition as an Alternative to Inheritance)* Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class `BasePlusCommissionEmployee` of the `CommissionEmployee`–`BasePlusCommissionEmployee` hierarchy to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for designing classes `CommissionEmployee` and `BasePlusCommissionEmployee`, as well as for object-oriented programs in general. Which approach is more natural? Why?

12.4 *(Inheritance Advantage)* Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

12.5 *(Protected vs. Private Base Classes)* Some programmers prefer not to use `protected` access because they believe it breaks the encapsulation of the base class. Discuss the relative merits of using `protected` access vs. using `private` access in base classes.

12.6 *(Student Inheritance Hierarchy)* Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 12.2. Use `Student` as the base class of the hierarchy, then include classes `UndergraduateStudent` and `GraduateStudent` that derive from `Student`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might derive from `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might derive from `GraduateStudent`. After drawing the hierarchy, discuss the relationships that exist between the classes. [*Note:* You do not need to write any code for this exercise.]

12.7 *(Richer Shape Hierarchy)* The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 12.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have the base class `Shape` from which class `TwoDimensionalShape` and class `ThreeDimensionalShape` are derived. [*Note:* You do not need to write any code for this exercise.] We'll use this hierarchy in the exercises of Chapter 13 to process a set of distinct shapes as objects of base-class `Shape`. (This technique, called polymorphism, is the subject of Chapter 13.)

12.8 *(Quadrilateral Inheritance Hierarchy)* Draw an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the base class of the hierarchy. Make the hierarchy as deep as possible.

12.9 (***Package** Inheritance Hierarchy*) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use class `Package` as the base class of the hierarchy, then include classes `TwoDayPackage` and `OvernightPackage` that derive from `Package`. Base class `Package` should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. `Package`'s constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. `Package` should provide a `public` member function `calculateCost` that returns a `double` indicating the cost associated with shipping the package. `Package`'s `calculateCost` function should determine the cost by multiplying the weight by the cost per ounce. Derived class `TwoDayPackage` should inherit the functionality of base class `Package`, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. `TwoDayPackage`'s constructor should receive a value to initialize this data member. `TwoDayPackage` should redefine member function `calculateCost` so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class `Package`'s `calculateCost` function. Class `OvernightPackage` should inherit directly from class `Package` and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. `OvernightPackage` should redefine member function `calcu-`

`lateCost` so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a test program that creates objects of each type of `Package` and tests member function `calculateCost`.

12.10 ***(Account Inheritance Hierarchy)*** Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class `Account` and derived classes `SavingsAccount` and `CheckingAccount` that inherit from class `Account`. Base class `Account` should include one data member of type `double` to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it's greater than or equal to `0.0`. If not, the balance should be set to `0.0` and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print the message `"Debit amount exceeded account balance."` Member function `getBalance` should return the current balance.

Derived class `SavingsAccount` should inherit the functionality of an `Account`, but also include a data member of type `double` indicating the interest rate (percentage) assigned to the `Account`. `SavingsAccount`'s constructor should receive the initial balance, as well as an initial value for the `SavingsAccount`'s interest rate. `SavingsAccount` should provide a `public` member function `calculateInterest` that returns a `double` indicating the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance. [*Note:* `SavingsAccount` should inherit member functions `credit` and `debit` as is without redefining them.]

Derived class `CheckingAccount` should inherit from base class `Account` and include an additional data member of type `double` that represents the fee charged per transaction. `CheckingAccount`'s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class `CheckingAccount` should redefine member functions `credit` and `debit` so that they subtract the fee from the account balance whenever either transaction is performed successfully. `CheckingAccount`'s versions of these functions should invoke the base-class `Account` version to perform the updates to an account balance. `CheckingAccount`'s `debit` function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [*Hint:* Define `Account`'s `debit` function so that it returns a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the `SavingsAccount` object by first invoking its `calculateInterest` function, then passing the returned interest amount to the object's `credit` function.

13 Object-Oriented Programming: Polymorphism

One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind them.
—John Ronald Reuel Tolkien

The silence often of pure innocence
Persuades when speaking fails.
—William Shakespeare

General propositions do not decide concrete cases.
—Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
—Alfred North Whitehead

Objectives

In this chapter you'll learn:

- How polymorphism makes programming more convenient and systems more extensible.
- The distinction between abstract and concrete classes and how to create abstract classes.
- To use runtime type information (RTTI).
- How C++ implements `virtual` functions and dynamic binding.
- How `virtual` destructors ensure that all appropriate destructors run on an object.

Outline

13.1 Introduction

We now continue our study of OOP by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables you to "program in the general" rather than "program in the specific." In particular, polymorphism enables you to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class. As we'll soon see, polymorphism works off base-class pointer handles and base-class reference handles, but not off name handles.

Implementing for Extensibility

With polymorphism, you can design and implement systems that are easily *extensible*—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy. For example, if we create class `Tortoise` that inherits from class `Animal` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that process each `Animal` generically can remain the same.

Optional Discussion of Polymorphism "Under the Hood"

A key feature of this chapter is its (optional) detailed discussion of polymorphism, `virtual` functions and dynamic binding "under the hood," which uses a detailed diagram to explain how polymorphism can be implemented in C++.

13.2 Introduction to Polymorphism: Polymorphic Video Game

Suppose that we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venutian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each of these classes inherits from the common base class `SpaceObject`, which contains member function `draw`. Each derived class implements this function in a manner appropriate for that class. A screen-manager program maintains a container (e.g., a `vector`) that holds `SpaceObject` pointers to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message—namely, `draw`. Each type of object responds in a unique way. For example, a `Martian` object might draw itself in red with the appropriate number of antennae, a `SpaceShip` object might draw itself as a silver flying saucer, and a `LaserBeam` object might draw itself as a bright red beam across the screen. The *same* message (in this case, `draw`) sent to a variety of objects has *many forms* of results—hence the term polymorphism.

A polymorphic screen manager facilitates adding new classes to a system with minimal modifications to its code. Suppose that we want to add objects of class `Mercurian` to our video game. To do so, we must build a class `Mercurian` that inherits from `SpaceObject`, but provides its own definition of member function `draw`. Then, when pointers to objects of class `Mercurian` appear in the container, you do not need to modify the code for the screen manager. The screen manager invokes member function `draw` on every object in the container, regardless of the object's type, so the new `Mercurian` objects simply "plug right in." Thus, without modifying the system (other than to build and include the classes themselves), you can use polymorphism to accommodate additional classes, including ones that were *not even envisioned* when the system was created.

Software Engineering Observation 13.1

Polymorphism enables you to deal in generalities *and let the execution-time environment concern itself with the* specifics. *You can direct a variety of objects to behave in manners appropriate to those objects* without even knowing their types*—as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer or a common base-class reference.*

Software Engineering Observation 13.2

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new *types of objects that can respond to* existing *messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.*

13.3 Relationships Among Objects in an Inheritance Hierarchy

Section 12.4 created an employee class hierarchy, in which class `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. The Chapter 12 examples manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using the objects' names to invoke their member functions. We now examine the relationships among classes in a hierarchy more closely. The next several sections present a series of examples that demonstrate how base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects.

- In Section 13.3.1, we assign the address of a derived-class object to a base-class pointer, then show that invoking a function via the base-class pointer invokes the base-class functionality in the derived-class object—i.e., the *type of the handle determines which function is called*.
- In Section 13.3.2, we assign the address of a base-class object to a derived-class pointer, which results in a compilation error. We discuss the error message and investigate why the compiler does not allow such an assignment.
- In Section 13.3.3, we assign the address of a derived-class object to a base-class pointer, then examine how the base-class pointer can be used to invoke only the base-class functionality—*when we attempt to invoke derived-class member functions through the base-class pointer, compilation errors occur*.
- Finally, in Section 13.3.4, we demonstrate how to get polymorphic behavior from base-class pointers aimed at derived-class objects. We introduce `virtual` functions and polymorphism by declaring a base-class function as `virtual`. We then assign the address of a derived-class object to the base-class pointer and use that pointer to invoke derived-class functionality—*precisely the capability we need to achieve polymorphic behavior*.

A key concept in these examples is to demonstrate that with `public` inheritance *an object of a derived class can be treated as an object of its base class*. This enables various interesting manipulations. For example, a program can create an array of base-class pointers that point to objects of many derived-class types. Despite the fact that the derived-class objects are of *different types*, the compiler allows this because each derived-class object *is an* object of its base class. However, *we cannot treat a base-class object as an object of any of its derived classes*. For example, a `CommissionEmployee` is not a `BasePlusCommissionEmployee` in the hierarchy defined in Chapter 12—a `CommissionEmployee` does *not* have a `baseSalary` data member and does *not* have member functions `setBaseSalary` and `getBaseSalary`. The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

13.3.1 Invoking Base-Class Functions from Derived-Class Objects

The example in Fig. 13.1 reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Section 12.4.5. The example demonstrates three ways to aim base and derived-class pointers at base and derived-class objects. The first two are straightforward—we aim a base-class pointer at a base-class object and invoke base-class functionality, and we aim a derived-class pointer at a derived-class object and invoke derived-class functionality. Then, we demonstrate the relationship between derived classes and base classes (i.e., the *is-a* relationship of inheritance) by aiming a base-class pointer at a derived-class object and showing that the base-class functionality is indeed available in the derived-class object.

```cpp
// Fig. 13.1: fig13_01.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects commissionEmployee and basePlusCommissionEmployee
   cout << "Print base-class and derived-class objects:\n\n";
   commissionEmployee.print(); // invokes base-class print
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // invokes derived-class print

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee; // perfectly natural
   cout << "\n\n\nCalling print with base-class pointer to "
      << "\nbase-class object invokes base-class print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
   cout << "\n\n\nCalling print with derived-class pointer to "
      << "\nderived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\n\nCalling print with base-class pointer to "
      << "derived-class object\ninvokes base-class print "
```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 1 of 2.)

```
         << "function on that derived-class object:\n\n";
      commissionEmployeePtr->print(); // invokes base-class print
      cout << endl;
   } // end main
```

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06


Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

Fig. 13.1 | Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers. (Part 2 of 2.)

Recall that each `BasePlusCommissionEmployee` object *is a* `CommissionEmployee` that also has a base salary. Class `BasePlusCommissionEmployee`'s `earnings` member function (lines 33–36 of Fig. 12.15) redefines class `CommissionEmployee`'s `earnings` member function (lines 84–87 of Fig. 12.14) to include the object's base salary. Class `BasePlusCommissionEmployee`'s `print` member function (lines 39–47 of Fig. 12.15) redefines class `CommissionEmployee`'s `version` (lines 90–97 of Fig. 12.14) to display the same information plus the employee's base salary.

Creating Objects and Displaying Their Contents

In Fig. 13.1, lines 13–14 create a `CommissionEmployee` object and line 17 creates a pointer to a `CommissionEmployee` object; lines 20–21 create a `BasePlusCommissionEmployee` object and line 24 creates a pointer to a `BasePlusCommissionEmployee` object. Lines 31 and 33 use each object's name to invoke its `print` member function.

Aiming a Base-Class Pointer at a Base-Class Object

Line 36 assigns the address of base-class object `commissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 39 uses to invoke member function `print` on that `CommissionEmployee` object. This invokes the version of `print` defined in base class `CommissionEmployee`.

Aiming a Derived-Class Pointer at a Derived-Class Object

Similarly, line 42 assigns the address of derived-class object `basePlusCommissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, which line 46 uses to invoke member function `print` on that `BasePlusCommissionEmployee` object. This invokes the version of `print` defined in derived class `BasePlusCommissionEmployee`.

Aiming a Base-Class Pointer at a Derived-Class Object

Line 49 then assigns the address of derived-class object `basePlusCommissionEmployee` to base-class pointer `commissionEmployeePtr`, which line 53 uses to invoke member function `print`. This "crossover" is allowed because an object of a derived class *is an* object of its base class. Despite the fact that the base class `CommissionEmployee` pointer points to a derived class `BasePlusCommissionEmployee` object, the base class `CommissionEmployee`'s `print` member function is invoked (rather than `BasePlusCommissionEmployee`'s `print` function). The output of each `print` member-function invocation in this program reveals that *the invoked functionality depends on the type of the pointer (or reference) used to invoke the function, not the type of the object for which the member function is called*. In Section 13.3.4, when we introduce `virtual` functions, we demonstrate that it's possible to invoke the object type's functionality, rather than invoke the handle type's functionality. We'll see that this is crucial to implementing polymorphic behavior—the key topic of this chapter.

13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

In Section 13.3.1, we assigned the address of a derived-class object to a base-class pointer and explained that the C++ compiler allows this assignment, because a derived-class object *is a* base-class object. We take the opposite approach in Fig. 13.2, as we aim a derived-class pointer at a base-class object. [*Note:* This program reuses the final versions of classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Section 12.4.5.] Lines 8–9 of Fig. 13.2 create a `CommissionEmployee` object, and line 10 creates a `BasePlusCommissionEmployee` pointer. Line 14 attempts to assign the address of base-class object `commissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, but the compiler generates an error. The compiler prevents this assignment, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`. Consider the consequences if the compiler were to allow this assignment. Through a `BasePlusCommissionEmployee` pointer, we can invoke *every* `BasePlusCommissionEmployee` member function, including `setBaseSalary`, for the object to which the pointer points (i.e., the base-class object `commissionEmployee`). However, the `CommissionEmployee` object does *not* provide a `set-`

BaseSalary member function, *nor* does it provide a baseSalary data member to set. This could lead to problems, because member function setBaseSalary would assume that there is a baseSalary data member to set at its "usual location" in a BasePlusCommissionEmployee object. This memory does not belong to the CommissionEmployee object, so member function setBaseSalary might overwrite other important data in memory, possibly data that belongs to a different object.

```
// Fig. 13.2: fig13_02.cpp
// Aiming a derived-class pointer at a base-class object.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // aim derived-class pointer at base-class object
   // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
   basePlusCommissionEmployeePtr = &commissionEmployee;
} // end main
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp8_examples\ch13\Fig13_02\fig13_02.cpp(14) : error C2440: '=' :
   cannot convert from 'CommissionEmployee *' to 'BasePlusCommissionEmployee *'
         Cast from base to derived requires dynamic_cast or static_cast
```

Fig. 13.2 | Aiming a derived-class pointer at a base-class object.

13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

Off a base-class pointer, the compiler allows us to invoke *only* base-class member functions. Thus, if a base-class pointer is aimed at a derived-class object, and an attempt is made to access a *derived-class-only member function*, a compilation error will occur.

Figure 13.3 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer. [*Note:* We're again reusing the versions of classes CommissionEmployee and BasePlusCommissionEmployee from Section 12.4.5.] Line 9 creates commissionEmployeePtr—a pointer to a CommissionEmployee object—and lines 10–11 create a BasePlusCommissionEmployee object. Line 14 aims commissionEmployeePtr at derived-class object basePlusCommissionEmployee. Recall from Section 13.3.1 that this is allowed, because a BasePlusCommissionEmployee *is a* CommissionEmployee (in the sense that a BasePlusCommissionEmployee object contains all the functionality of a CommissionEmployee object). Lines 18–22 invoke base-class member functions getFirstName, getLastName, getSocialSecurityNumber, getGrossSales and getCommissionRate off the base-class pointer. All of these calls are legitimate, because BasePlusCommissionEmployee inherits these member functions from CommissionEmployee. We know that commissionEmployeePtr is aimed at a BasePlusCommissionEmployee object, so in lines 26–27 we attempt to invoke BasePlusCommissionEmployee member functions getBaseSalary and setBaseSalary. The compiler generates errors on both of these calls, because they're *not* made to member functions of base-class CommissionEmployee. The handle can be used to invoke *only* those functions that are members of that handle's associated class type. (In this case, off a CommissionEmployee *, we can invoke only CommissionEmployee member functions setFirstName, getFirstName, setLastName, getLastName, setSocialSecurityNumber, getSocialSecurityNumber, setGrossSales, getGrossSales, setCommissionRate, getCommissionRate, earnings and print.)

```
// Fig. 13.3: fig13_03.cpp
// Attempting to invoke derived-class-only member functions
// through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
   CommissionEmployee *commissionEmployeePtr = 0; // base class
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class

   // aim base-class pointer at derived-class object
   commissionEmployeePtr = &basePlusCommissionEmployee;

   // invoke base-class member functions on derived-class
   // object through base-class pointer (allowed)
   string firstName = commissionEmployeePtr->getFirstName();
   string lastName = commissionEmployeePtr->getLastName();
   string ssn = commissionEmployeePtr->getSocialSecurityNumber();
   double grossSales = commissionEmployeePtr->getGrossSales();
   double commissionRate = commissionEmployeePtr->getCommissionRate();

   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer (disallowed)
   double baseSalary = commissionEmployeePtr->getBaseSalary();
   commissionEmployeePtr->setBaseSalary( 500 );
} // end main
```

Microsoft Visual C++ compiler error messages:

```
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(26) : error C2039:
   'getBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
C:\cpphtp8_examples\ch13\Fig13_03\fig13_03.cpp(27) : error C2039:
   'setBaseSalary' : is not a member of 'CommissionEmployee'
      C:\cpphtp8_examples\ch13\Fig13_03\CommissionEmployee.h(10) :
         see declaration of 'CommissionEmployee'
```

GNU C++ compiler error messages:

```
fig13_03.cpp:26: error: 'getBaseSalary' undeclared (first use this function)
fig13_03.cpp:27: error: 'setBaseSalary' undeclared (first use this function)
```

Fig. 13.3 | Attempting to invoke derived-class-only functions via a base-class pointer.

The compiler will allow access to derived-class-only members from a base-class pointer that's aimed at a derived-class object *if* we explicitly cast the base-class pointer to a derived-class pointer—this is known as **downcasting**. As you know, it's possible to aim a base-class pointer at a derived-class object. However, as we demonstrated in Fig. 13.3, a base-class pointer can be used to invoke *only* the functions declared in the base class. Downcasting allows a derived-class-specific operation on a derived-class object pointed to by a base-class pointer. After a downcast, the program *can* invoke derived-class functions that are not in the base class. Downcasting is a potentially dangerous operation. Section 13.8 demonstrates how to *safely* use downcasting.

Software Engineering Observation 13.3

If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it's acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must *be done to send that derived-class object messages that do not appear in the base class.*

13.3.4 Virtual Functions

In Section 13.3.1, we aimed a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, then invoked member function `print` through that pointer. Recall that the *type of the handle* determined which class's functionality to invoke. In that case, the `CommissionEmployee` pointer invoked the `CommissionEmployee` member function `print` on the `BasePlusCommissionEmployee` object, even though the pointer was aimed at a `BasePlusCommissionEmployee` object that has its own custom `print` function.

Software Engineering Observation 13.4

With `virtual` functions, the type of the object, not the type of the handle used to invoke the member function, determines which version of a `virtual` function to invoke.

First, we consider why `virtual` functions are useful. Suppose that shape classes such as `Circle`, `Triangle`, `Rectangle` and `Square` are all derived from base class `Shape`. Each of these classes might be endowed with the ability to draw itself via a member function `draw`. Although each class has its own `draw` function, the function for each shape is quite different. In a program that draws a set of shapes, it would be useful to be able to treat all the shapes generically as objects of the base class `Shape`. Then, to draw any shape, we could simply use a base-class `Shape` pointer to invoke function `draw` and let the program determine *dynamically* (i.e., at runtime) which derived-class `draw` function to use, based on the type of the object to which the base-class `Shape` pointer points at any given time.

To enable this behavior, we declare `draw` in the base class as a **`virtual` function**, and we **override** `draw` in each of the derived classes to draw the appropriate shape. From an implementation perspective, *overriding* a function is no different than *redefining* one (which is the approach we've been using until now). An overridden function in a derived class has the *same signature and return type* (i.e., *prototype*) as the function it overrides in its base class. If we do not declare the base-class function as `virtual`, we can redefine that function. By contrast, if we declare the base-class function as `virtual`, we can override that function to enable polymorphic behavior. We declare a `virtual` function by preceding the function's prototype with the keyword `virtual` in the base class. For example,

```
virtual void draw() const;
```

would appear in base class `Shape`. The preceding prototype declares that function `draw` is a `virtual` function that takes no arguments and returns nothing. This function is declared `const` because a `draw` function typically would not make changes to the `Shape` object on which it's invoked—virtual functions do *not* have to be `const` functions.

Software Engineering Observation 13.5

Once a function is declared `virtual`*, it remains* `virtual` all the way down the inheritance hierarchy from that point, *even if that function is not explicitly declared* `virtual` *when a derived class overrides it.*

Good Programming Practice 13.1

Even though certain functions are implicitly `virtual` *because of a declaration made higher in the class hierarchy, explicitly declare these functions* `virtual` *at every level of the class hierarchy to promote program clarity.*

Error-Prevention Tip 13.1

When you browse a class hierarchy to locate a class to reuse, it's possible that a function in that class will exhibit `virtual` *function behavior even though it isn't explicitly declared* `virtual`*. This happens when the class inherits a* `virtual` *function from its base class, and it can lead to subtle logic errors. Such errors can be avoided by explicitly declaring all* `virtual` *functions* `virtual` *throughout the inheritance hierarchy.*

Software Engineering Observation 13.6

When a derived class chooses not *to override a* `virtual` *function from its base class, the derived class simply inherits its base class's* `virtual` *function implementation.*

If a program invokes a `virtual` function through a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`) or a base-class reference to a derived-class object (e.g., `shapeRef.draw()`), the program will choose the correct derived-class `draw` function dynamically (i.e., at execution time) *based on the object type—not the pointer or reference type*. Choosing the appropriate function to call at execution time (rather than at compile time) is known as **dynamic binding** or **late binding**.

When a `virtual` function is called by referencing a specific object by *name* and using the dot member-selection operator (e.g., `squareObject.draw()`), the function invocation is resolved at compile time (this is called **static binding**) and the `virtual` function that's called is the one defined for (or inherited by) the class of that particular object—this is not polymorphic behavior. Thus, dynamic binding with `virtual` functions occurs only off pointer (and, as we'll soon see, reference) handles.

Now let's see how `virtual` functions can enable polymorphic behavior in our employee hierarchy. Figures 13.4–13.5 are the headers for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively. The only new features in these files is that we specify each class's `earnings` and `print` member functions as `virtual` (lines 30–31 of Fig. 13.4 and lines 20–21 of Fig. 13.5). Because functions `earnings` and `print` are `virtual` in class `CommissionEmployee`, class `BasePlusCommissionEmployee`'s `earnings` and `print` functions override class `CommissionEmployee`'s. Now, if we aim a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, and the program uses that pointer to call either function `earnings` or `print`, the `BasePlusCommissionEmployee` object's corresponding function will be invoked. There were no changes to the member-function implementations of classes `CommissionEmployee` and `BasePlusCommissionEmployee`, so we reuse the versions of Figs. 12.14 and 12.15.

```
// Fig. 13.4: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using namespace std;

class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
      double = 0.0, double = 0.0 );

   void setFirstName( const string & ); // set first name
   string getFirstName() const; // return first name

   void setLastName( const string & ); // set last name
   string getLastName() const; // return last name

   void setSocialSecurityNumber( const string & ); // set SSN
   string getSocialSecurityNumber() const; // return SSN

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   string firstName;
   string lastName;
   string socialSecurityNumber;
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif
```

Fig. 13.4 | `CommissionEmployee` class header declares `earnings` and `print` as `virtual`.

```
// Fig. 13.5: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H
```

Fig. 13.5 | `BasePlusCommissionEmployee` class header declares `earnings` and `print` functions as `virtual`. (Part 1 of 2.)

```

#include <string> // C++ standard string class
#include "CommissionEmployee.h" // CommissionEmployee class declaration
using namespace std;

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Fig. 13.5 | `BasePlusCommissionEmployee` class header declares `earnings` and `print` functions as `virtual`. (Part 2 of 2.)

We modified Fig. 13.1 to create the program of Fig. 13.6. Lines 40–51 demonstrate again that a `CommissionEmployee` pointer aimed at a `CommissionEmployee` object can be used to invoke `CommissionEmployee` functionality, and a `BasePlusCommissionEmployee` pointer aimed at a `BasePlusCommissionEmployee` object can be used to invoke `BasePlusCommissionEmployee` functionality. Line 54 aims the base-class pointer `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`. Note that when line 61 invokes member function `print` off the base-class pointer, the derived-class `BasePlusCommissionEmployee`'s `print` member function is invoked, so line 61 outputs different text than line 53 does in Fig. 13.1 (when member function `print` was *not* declared `virtual`). We see that declaring a member function `virtual` causes the program to dynamically determine which function to invoke *based on the type of object to which the handle points, rather than on the type of the handle*. Note again that when `commissionEmployeePtr` points to a `CommissionEmployee` object (line 40), class `CommissionEmployee`'s `print` function is invoked, and when `CommissionEmployeePtr` points to a `BasePlusCommissionEmployee` object, class `BasePlusCommissionEmployee`'s `print` function is invoked. Thus, the same message—`print`, in this case—sent (off a base-class pointer) to a variety of objects related by inheritance to that base class, takes on many forms—this is polymorphic behavior.

```
1 // Fig. 13.6: fig13_06.cpp
2 // Introducing polymorphism, virtual functions and dynamic binding.
3 #include <iostream>
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 1 of 3.)

```
#include <iomanip>
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

int main()
{
   // create base-class object
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "222-22-2222", 10000, .06 );

   // create base-class pointer
   CommissionEmployee *commissionEmployeePtr = 0;

   // create derived-class object
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

   // create derived-class pointer
   BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // output objects using static binding
   cout << "Invoking print function on base-class and derived-class "
      << "\nobjects with static binding\n\n";
   commissionEmployee.print(); // static binding
   cout << "\n\n";
   basePlusCommissionEmployee.print(); // static binding

   // output objects using dynamic binding
   cout << "\n\n\nInvoking print function on base-class and "
      << "derived-class \nobjects with dynamic binding";

   // aim base-class pointer at base-class object and print
   commissionEmployeePtr = &commissionEmployee;
   cout << "\n\nCalling virtual function print with base-class pointer"
      << "\nto base-class object invokes base-class "
      << "print function:\n\n";
   commissionEmployeePtr->print(); // invokes base-class print

   // aim derived-class pointer at derived-class object and print
   basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
   cout << "\n\nCalling virtual function print with derived-class "
      << "pointer\nto derived-class object invokes derived-class "
      << "print function:\n\n";
   basePlusCommissionEmployeePtr->print(); // invokes derived-class print

   // aim base-class pointer at derived-class object and print
   commissionEmployeePtr = &basePlusCommissionEmployee;
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 2 of 3.)

```
    cout << "\n\nCalling virtual function print with base-class pointer"
       << "\nto derived-class object invokes derived-class "
       << "print function:\n\n";

    // polymorphism; invokes BasePlusCommissionEmployee's print;
    // base-class pointer to derived-class object
    commissionEmployeePtr->print();
    cout << endl;
} // end main
```

```
Invoking print function on base-class and derived-class
objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Invoking print function on base-class and derived-class
objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 13.6 | Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object. (Part 3 of 3.)

13.4 Type Fields and `switch` Statements

One way to determine the type of an object is to use a `switch` statement to check the value of a field in the object. This allows us to distinguish among object types, then invoke an appropriate action for a particular object. For example, in a hierarchy of shapes in which each shape object has a `shapeType` attribute, a `switch` statement could check the object's `shapeType` to determine which `print` function to call.

Using `switch` logic exposes programs to a variety of potential problems. For example, you might forget to include a type test when one is warranted, or might forget to test all possible cases in a `switch` statement. When modifying a `switch`-based system by adding new types, you might forget to insert the new cases in *all* relevant `switch` statements. Every addition or deletion of a class requires the modification of every `switch` statement in the system; tracking these statements down can be time consuming and error prone.

Software Engineering Observation 13.7

Polymorphic programming can eliminate the need for `switch` *logic. By using the polymorphism mechanism to perform the equivalent logic, you can avoid the kinds of errors typically associated with* `switch` *logic.*

Software Engineering Observation 13.8

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and simpler sequential code. This simplification facilitates testing, debugging and program maintenance.

13.5 Abstract Classes and Pure `virtual` Functions

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it's useful to define *classes from which you never intend to instantiate any objects*. Such classes are called **abstract classes**. Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as **abstract base classes**. These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*—derived classes must define the "missing pieces" before objects of these classes can be instantiated. We build programs with abstract classes in Section 13.6.

An abstract class provides a base class from which other classes can inherit. Classes that can be used to instantiate objects are called **concrete classes**. Such classes define or inherit implementations for every member function they declare. We could have an *abstract* base class `TwoDimensionalShape` and derive such *concrete* classes as `Square`, `Circle` and `Triangle`. We could also have an *abstract* base class `ThreeDimensionalShape` and derive such *concrete* classes as `Cube`, `Sphere` and `Cylinder`. Abstract base classes are *too generic* to define real objects; we need to be *more specific* before we can think of instantiating objects. For example, if someone tells you to "draw the two-dimensional shape," what shape would you draw? Concrete classes provide the *specifics* that make it reasonable to instantiate objects.

An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes. In some cases, abstract classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 12.3, which begins with abstract base class `Shape`. On the next level of the hierarchy we have two more abstract base classes—`TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy defines *concrete* classes for two-dimensional shapes (namely, `Circle`, `Square` and `Triangle`) and for three-dimensional shapes (namely, `Sphere`, `Cube` and `Tetrahedron`).

Pure Virtual Functions

A class is made abstract by declaring one or more of its `virtual` functions to be "pure." A **pure `virtual` function** is specified by placing "`= 0`" in its declaration, as in

```
virtual void draw() const = 0; // pure virtual function
```

The "`= 0`" is a **pure specifier**. Pure `virtual` functions do *not* provide implementations. Every concrete derived class *must override all* base-class pure `virtual` functions with concrete implementations of those functions. The difference between a `virtual` function and a pure `virtual` function is that a `virtual` function has an implementation and gives the derived class the *option* of overriding the function; by contrast, a pure `virtual` function does *not* provide an implementation and *requires* the derived class to override the function for that derived class to be concrete; otherwise the derived class remains *abstract*.

Pure `virtual` functions are used when it does *not* make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function. Returning to our earlier example of space objects, it does not make sense for the base class `SpaceObject` to have an implementation for function `draw` (as there is no way to draw a generic space object without having more information about what type of space object is being drawn). An example of a function that would be defined as `virtual` (and not pure `virtual`) would be one that returns a name for the object. We can name a generic `SpaceObject` (for instance, as `"space object"`), so a default implementation for this function can be provided, and the function does not need to be pure `virtual`. The function is still declared `virtual`, however, because it's expected that derived classes will override this function to provide *more specific* names for the derived-class objects.

Software Engineering Observation 13.9

An abstract class defines a common public interface for the various classes in a class hierarchy. An abstract class contains one or more pure `virtual` functions that concrete derived classes must *override.*

Common Programming Error 13.1

Failure to override a pure `virtual` function in a derived classmakes that class abstract. Attempting to instantiate an object of an abstract class causes a compilation error.

Software Engineering Observation 13.10

An abstract class has at least one pure `virtual` function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

Although we *cannot* instantiate objects of an abstract base class, we *can* use the abstract base class to declare pointers and references that can refer to objects of any concrete classes derived from the abstract class. Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

Device Drivers and Polymorphism

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to *read* or *write* data from and to devices may have a certain uniformity. The *write* message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the *write* call itself really is no different from the *write* to any other device in the system—place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the `public` functions) offered by the device drivers are provided as pure `virtual` functions in the abstract base class. The implementations of these pure `virtual` functions are provided in the derived classes that correspond to the specific types of device drivers. This architecture also allows new devices to be added to a system easily, even after the operating system has been defined. The user can just plug in the device and install its new device driver. The operating system "talks" to this new device through its device driver, which has the same `public` member functions as all other device drivers—those defined in the device driver abstract base class.

Iterators and Polymorphism

It's common in object-oriented programming to define an **iterator class** that can traverse all the objects in a container (such as an array). For example, a program can print a list of objects in a `vector` by creating an iterator object, then using the iterator to obtain the next element of the list each time the iterator is called. Iterators often are used in polymorphic programming to traverse an array or a linked list of pointers to objects from various levels of a hierarchy. The pointers in such a list are all base-class pointers. (Chapter 22, Standard Template Library (STL), presents a thorough treatment of iterators.) A list of pointers to objects of thebase class `TwoDimensionalShape` could contain pointers to objects of the classes `Square`, `Circle`, `Triangle` and so on. Using polymorphism to send a `draw` message, off a `TwoDimensionalShape *` pointer, to each object in the list would draw them correctly on the screen.

13.6 Case Study: Payroll System Using Polymorphism

This section reexamines the `CommissionEmployee`-`BasePlusCommissionEmployee` hierarchy that we explored throughout Section 12.4. In this example, we use an abstract class and polymorphism to perform payroll calculations based on the type of employee. We create an enhanced employee hierarchy to solve the following problem:

> *A company pays its employees weekly. The employees are of three types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, commission employees are paid a percentage of their sales and base-salary-plus-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically.*

We use abstract class `Employee` to represent the general concept of an employee. The classes that derive directly from `Employee` are `SalariedEmployee` and `CommissionEmployee`. Class `BasePlusCommissionEmployee`—derived from `CommissionEmployee`—represents the last employee type. The UML class diagram in Fig. 13.7 shows the inheritance hierarchy for our polymorphic employee payroll application. The abstract class name `Employee` is italicized, as per the convention of the UML.

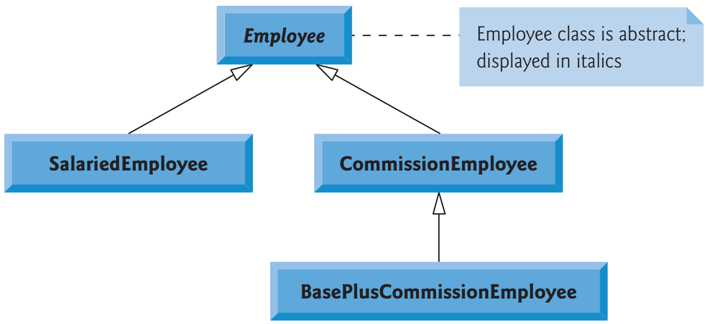


Fig. 13.7 | `Employee` hierarchy UML class diagram.

Abstract base class `Employee` declares the "interface" to the hierarchy—that is, the set of member functions that a program can invoke on all `Employee` objects. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` data members `firstName`, `lastName` and `socialSecurityNumber` appear in abstract base class `Employee`.

Software Engineering Observation 13.11

A derived class can inherit interface and/or implementation from a base class. Hierarchies designed for ***implementation inheritance*** *tend to have their functionality high in the hierarchy—each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions. Hierarchies designed for* ***interface inheritance*** *tend to have their functionality lower in the hierarchy—a base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).*

The following sections implement the `Employee` class hierarchy. The first five each implement one of the abstract or concrete classes. The last section implements a test program that builds objects of all these classes and processes the objects polymorphically.

13.6.1 Creating Abstract Base Class `Employee`

Class `Employee` (Figs. 13.9–13.10, discussed in further detail shortly) provides functions `earnings` and `print`, in addition to various *get* and *set* functions that manipulate `Employee`'s data members. An `earnings` function certainly applies generically to all employees, but each earnings calculation depends on the employee's class. So we declare `earnings` as pure `virtual` in base class `Employee` because *a default implementation does not make sense* for that function—there is not enough information to determine what amount `earnings` should return. Each derived class *overrides* `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns the address of an employee's object to a base class `Employee` pointer, then invokes the `earnings` function on that object. We maintain a `vector` of `Employee` pointers, each of which points to an `Employee` object. *Of course, there cannot be `Employee` objects, because `Employee` is an abstract class—because of inheritance, however, all objects of all derived classes of `Employee` may nevertheless be thought of as `Employee` objects*. The program iterates through the `vector` and calls function `earnings` for each `Employee` object. C++ processes these function calls *polymorphically*. Including `earnings` as a pure `virtual` function in `Employee` forces every direct derived class of `Employee` that wishes to be a *concrete* class to *override* `earnings`.

Function `print` in class `Employee` displays the first name, last name and social security number of the employee. As we'll see, each derived class of `Employee` overrides function `print` to output the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information. Function `print` in the derived classes could also call `earnings`, even though `earnings` is a pure-`virtual` function in base class `Employee`.

The diagram in Fig. 13.8 shows each of the four classes in the hierarchy down the left side and functions `earnings` and `print` across the top. For each class, the diagram shows the desired results of each function. Class `Employee` specifies "`= 0`" for function `earnings` to indicate that this is a pure `virtual` function and hence has *no* implementation. Each derived class overrides this function to provide an appropriate implementation. We do not list base class `Employee`'s *get* and *set* functions because they're not overridden in any of the derived classes—each of these functions is inherited and used "as is" by each of the derived classes.

	`earnings`	`print`
`Employee`	`= 0`	*firstName lastName* `social security number:` *SSN*
`Salaried-Employee`	*weeklySalary*	`salaried employee:` *firstName lastName* `social security number:` *SSN* `weekly salary:` *weeklySalary*
`Commission-Employee`	*commissionRate* `*` *grossSales*	`commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*`;` `commission rate:` *commissionRate*

`BasePlus-Commission-Employee`	`(`*commissionRate* `*` *grossSales*`) +` *baseSalary*	`base-salaried commission employee:` *firstName lastName*
		`social security number:` *SSN*
		`gross sales:` *grossSales*`;`
		`commission rate:` *commissionRate*`;`
		`base salary:` *baseSalary*

Fig. 13.8 | Polymorphic interface for the `Employee` hierarchy classes.

`Employee` Class Header

Let's consider class `Employee`'s header (Fig. 13.9). The `public` member functions include a constructor that takes the first name, last name and social security number as arguments (line 12); *set* functions that set the first name, last name and social security number (lines 14, 17 and 20, respectively); *get* functions that return the first name, last name and social security number (lines 15, 18 and 21, respectively); pure `virtual` function `earnings` (line 24) and `virtual` function `print` (line 25).

```
1  // Fig. 13.9: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using namespace std;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
22
23    // pure virtual function makes Employee an abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
29    string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H
```

Fig. 13.9 | `Employee` class header.

Recall that we declared `earnings` as a pure `virtual` function because first we must know the *specific* `Employee` type to determine the appropriate `earnings` calculations. Declaring this function as pure `virtual` indicates that each concrete derived class *must* provide an `earnings` implementation and that a program can use base-class `Employee` pointers to invoke function `earnings` *polymorphically* for any type of `Employee`.

`Employee` Class Member-Function Definitions

Figure 13.10 contains the member-function implementations for class `Employee`. No implementation is provided for `virtual` function `earnings`. The `Employee` constructor (lines 9–14) does not validate the social security number. Normally, such validation should be provided.

```
// Fig. 13.10: Employee.cpp
// Abstract-base-class Employee member-function definitions.
// Note: No definitions are given for pure virtual functions.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

// constructor
Employee::Employee( const string &first, const string &last,
   const string &ssn )
   : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
   // empty body
} // end Employee constructor

// set first name
void Employee::setFirstName( const string &first )
{
   firstName = first;
} // end function setFirstName

// return first name
string Employee::getFirstName() const
{
   return firstName;
} // end function getFirstName

// set last name
void Employee::setLastName( const string &last )
{
   lastName = last;
} // end function setLastName

// return last name
string Employee::getLastName() const
{
   return lastName;
} // end function getLastName

// set social security number
void Employee::setSocialSecurityNumber( const string &ssn )
{
   socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string Employee::getSocialSecurityNumber() const
{
   return socialSecurityNumber;
} // end function getSocialSecurityNumber

```

Fig. 13.10 | Employee class implementation file. (Part 1 of 2.)

```
52 // print Employee's information (virtual, but not pure virtual)
53 void Employee::print() const
54 {
55    cout << getFirstName() << ' ' << getLastName()
56       << "\nsocial security number: " << getSocialSecurityNumber();
57 } // end function print
```

Fig. 13.10 | `Employee` class implementation file. (Part 2 of 2.)

The `virtual` function `print` (Fig. 13.10, lines 53–57) provides an *implementation* that will be *overridden* in each of the derived classes. Each of these functions will, however, use the abstract class's version of `print` to print information *common to all classes* in the `Employee` hierarchy.

13.6.2 Creating Concrete Derived Class `SalariedEmployee`

Class `SalariedEmployee` (Figs. 13.11–13.12) derives from class `Employee` (line 8 of Fig. 13.11). The `public` member functions include a constructor that takes a first name, a last name, a social security number and a weekly salary as arguments (lines 11–12); a *set* function to assign a new nonnegative value to data member `weeklySalary` (line 14); a *get* function to return `weeklySalary`'s value (line 15); a `virtual` function `earnings` that calculates a `SalariedEmployee`'s earnings (line 18) and a `virtual` function `print` (line 19) that outputs the employee's type, namely, `"salaried employee: "` followed by employee-specific information produced by base class `Employee`'s `print` function and `SalariedEmployee`'s `getWeeklySalary` function.

```
1  // Fig. 13.11: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

Fig. 13.11 | `SalariedEmployee` class header.

`SalariedEmployee` *Class Member-Function Definitions*

Figure 13.12 contains the member-function implementations for `SalariedEmployee`. The class's constructor passes the first name, last name and social security number to the `Employee` constructor (line 10) to initialize the `private` data members that are inherited from the base class, but not directly accessible in the derived class. Function `earnings` (lines 32–35) overrides pure `virtual` function `earnings` in `Employee` to provide a *concrete* implementation that returns the `SalariedEmployee`'s weekly salary. If we did not implement `earnings`, class `SalariedEmployee` would be an *abstract* class, and any attempt to instantiate an object of the class would result in a compilation error (and, of course, we want `SalariedEmployee` here to be a concrete class). In class `SalariedEmployee`'s header, we declared member functions `earnings` and `print` as `virtual` (lines 18–19 of Fig. 13.11)—actually, placing the `virtual` keyword before these member functions is *redundant*. We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy. Explicitly declaring such functions `virtual` at every level of the hierarchy can promote program clarity. Not declaring `earnings` as pure `virtual` signals our intent to provide an implementation in this concrete class.

```
1  // Fig. 13.12: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  #include "SalariedEmployee.h" // SalariedEmployee class definition
5  using namespace std;
6
7  // constructor
8  SalariedEmployee::SalariedEmployee( const string &first,
9     const string &last, const string &ssn, double salary )
10    : Employee( first, last, ssn )
11 {
12    setWeeklySalary( salary );
13 } // end SalariedEmployee constructor
14
15 // set salary
16 void SalariedEmployee::setWeeklySalary( double salary )
17 {
18    if ( salary >= 0.0 )
19       weeklySalary = salary;
20    else
21       throw invalid_argument( "Weekly salary must be >= 0.0" );
22 } // end function setWeeklySalary
23
24 // return salary
25 double SalariedEmployee::getWeeklySalary() const
26 {
27    return weeklySalary;
28 } // end function getWeeklySalary
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double SalariedEmployee::earnings() const
33 {
```

Fig. 13.12 | `SalariedEmployee` class implementation file. (Part 1 of 2.)

```
34       return getWeeklySalary();
35    } // end function earnings
36
37    // print SalariedEmployee's information
38    void SalariedEmployee::print() const
39    {
40       cout << "salaried employee: ";
41       Employee::print(); // reuse abstract base-class print function
42       cout << "\nweekly salary: " << getWeeklySalary();
43    } // end function print
```

Fig. 13.12 | `SalariedEmployee` class implementation file. (Part 2 of 2.)

Function `print` of class `SalariedEmployee` (lines 38–43 of Fig. 13.12) overrides `Employee` function `print`. If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`. In that case, `SalariedEmployee`'s `print` function would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To print a `SalariedEmployee`'s complete information, the derived class's `print` function outputs `"salaried employee: "` followed by the base-class `Employee`-specific information (i.e., first name, last name and social security number) printed by invoking the base class's `print` function using the scope resolution operator (line 41)—this is a nice example of code reuse. Without the scope resolution operator, the `print` call would cause *infinite recursion*. The output produced by `SalariedEmployee`'s `print` function contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` function.

13.6.3 Creating Concrete Derived Class `CommissionEmployee`

Class `CommissionEmployee` (Figs. 13.13–13.14) derives from `Employee` (Fig. 13.13, line 8). The member-function implementations (Fig. 13.14) include a constructor (lines 8–14) that takes a first name, last name, social security number, sales amount and commission rate; *set* functions (lines 17–23 and 32–38) to assign new values to data members `commissionRate` and `grossSales`, respectively; *get* functions (lines 26–29 and 41–44) that retrieve their values; function `earnings` (lines 47–50) to calculate a `CommissionEmployee`'s earnings; and function `print` (lines 53–59) to output the employee's type, namely, `"commission employee: "` and employee-specific information. The constructor passes the first name, last name and social security number to the `Employee` constructor (line 10) to initialize `Employee`'s `private` data members. Function `print` calls base-class function `print` (line 56) to display the `Employee`-specific information.

```
1    // Fig. 13.13: CommissionEmployee.h
2    // CommissionEmployee class derived from Employee.
3    #ifndef COMMISSION_H
4    #define COMMISSION_H
5
6    #include "Employee.h" // Employee class definition
7
```

Fig. 13.13 | `CommissionEmployee` class header. (Part 1 of 2.)

```
class CommissionEmployee : public Employee
{
public:
   CommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0 );

   void setCommissionRate( double ); // set commission rate
   double getCommissionRate() const; // return commission rate

   void setGrossSales( double ); // set gross sales amount
   double getGrossSales() const; // return gross sales amount

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print CommissionEmployee object
private:
   double grossSales; // gross weekly sales
   double commissionRate; // commission percentage
}; // end class CommissionEmployee

#endif // COMMISSION_H
```

Fig. 13.13 | CommissionEmployee class header. (Part 2 of 2.)

```
// Fig. 13.14: CommissionEmployee.cpp
// CommissionEmployee class member-function definitions.
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition
using namespace std;

// constructor
CommissionEmployee::CommissionEmployee( const string &first,
   const string &last, const string &ssn, double sales, double rate )
   : Employee( first, last, ssn )
{
   setGrossSales( sales );
   setCommissionRate( rate );
} // end CommissionEmployee constructor

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
   if ( sales >= 0.0 )
      grossSales = sales;
   else
      throw invalid_argument( "Gross sales must be >= 0.0" );
} // end function setGrossSales

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
```

Fig. 13.14 | CommissionEmployee class implementation file. (Part 1 of 2.)

```
   return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
   if ( rate > 0.0 && rate < 1.0 )
      commissionRate = rate;
   else
      throw invalid_argument( "Commission rate must be > 0.0 and < 1.0" );
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
   return commissionRate;
} // end function getCommissionRate

// calculate earnings; override pure virtual function earnings in Employee
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // end function earnings

// print CommissionEmployee's information
void CommissionEmployee::print() const
{
   cout << "commission employee: ";
   Employee::print(); // code reuse
   cout << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
} // end function print
```

Fig. 13.14 | `CommissionEmployee` class implementation file. (Part 2 of 2.)

13.6.4 Creating Indirect Concrete Derived Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployee` (Figs. 13.15–13.16) directly inherits from class `CommissionEmployee` (line 8 of Fig. 13.15) and therefore is an *indirect* derived class of class `Employee`. Class `BasePlusCommissionEmployee`'s member-function implementations include a constructor (lines 8–14 of Fig. 13.16) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes the first name, last name, social security number, sales amount and commission rate to the `CommissionEmployee` constructor (line 11) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a *set* function (lines 17–23) to assign a new value to data member `baseSalary` and a *get* function (lines 26–29) to return `baseSalary`'s value. Function `earnings` (lines 33–36) calculates a `BasePlusCommissionEmployee`'s earnings. Line 35 in function `earnings` calls base-class `CommissionEmployee`'s `earnings` function to calculate the commission-based portion of the employee's earnings. This is another nice example of code reuse. `BasePlusCommissionEmployee`'s `print` function (lines 39–44) outputs `"base-salaried"`, followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary. The resulting output begins with `"base-salaried commission employee: "` followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `print` displays the employee's first name, last name and social security number by invoking the `print` function of its base class (i.e., `Employee`)—yet another example of code reuse. `BasePlusCommissionEmployee`'s `print` initiates a chain of functions calls that spans *all three levels* of the `Employee` hierarchy.

```cpp
// Fig. 13.15: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include "CommissionEmployee.h" // CommissionEmployee class definition

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
   BasePlusCommissionEmployee( const string &, const string &,
      const string &, double = 0.0, double = 0.0, double = 0.0 );

   void setBaseSalary( double ); // set base salary
   double getBaseSalary() const; // return base salary

   // keyword virtual signals intent to override
   virtual double earnings() const; // calculate earnings
   virtual void print() const; // print BasePlusCommissionEmployee object
private:
   double baseSalary; // base salary per week
}; // end class BasePlusCommissionEmployee

#endif // BASEPLUS_H
```

Fig. 13.15 | `BasePlusCommissionEmployee` class header.

```cpp
// Fig. 13.16: BasePlusCommissionEmployee.cpp
// BasePlusCommissionEmployee member-function definitions.
#include <iostream>
#include "BasePlusCommissionEmployee.h"
using namespace std;

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first, const string &last, const string &ssn,
   double sales, double rate, double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

```

Fig. 13.16 | `BasePlusCommissionEmployee` class implementation file. (Part 1 of 2.)

```cpp
// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
   if ( salary >= 0.0 )
      baseSalary = salary;
   else
      throw invalid_argument( "Salary must be >= 0.0" );
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
    return baseSalary;
} // end function getBaseSalary

// calculate earnings;
// override virtual function earnings in CommissionEmployee
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee's information
void BasePlusCommissionEmployee::print() const
{
   cout << "base-salaried ";
   CommissionEmployee::print(); // code reuse
   cout << "; base salary: " << getBaseSalary();
} // end function print
```

Fig. 13.16 | BasePlusCommissionEmployee class implementation file. (Part 2 of 2.)

13.6.5 Demonstrating Polymorphic Processing

To test our Employee hierarchy, the program in Fig. 13.17 creates an object of each of the three concrete classes SalariedEmployee, CommissionEmployee and BasePlusCommissionEmployee. The program manipulates these objects, first with *static binding*, then *polymorphically*, using a vector of Employee pointers. Lines 22–27 create objects of each of the three concrete Employee derived classes. Lines 32–38 output each Employee's information and earnings. Each member-function invocation in lines 32–37 is an example of static binding—at *compile time*, because we are using *name handles* (not *pointers* or *references* that could be set at *execution time*), the *compiler* can identify each object's type to determine which print and earnings functions are called.

```cpp
1  // Fig. 13.17: fig13_17.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>
```

Fig. 13.17 | Employee class hierarchy driver program. (Part 1 of 4.)

```
#include "Employee.h"
#include "SalariedEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"
using namespace std;

void virtualViaPointer( const Employee * const ); // prototype
void virtualViaReference( const Employee & ); // prototype

int main()
{
   // set floating-point output formatting
   cout << fixed << setprecision( 2 );

   // create derived-class objects
   SalariedEmployee salariedEmployee(
      "John", "Smith", "111-11-1111", 800 );
   CommissionEmployee commissionEmployee(
      "Sue", "Jones", "333-33-3333", 10000, .06 );
   BasePlusCommissionEmployee basePlusCommissionEmployee(
      "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

   cout << "Employees processed individually using static binding:\n\n";

   // output each Employee’s information and earnings using static binding
   salariedEmployee.print();
   cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
   commissionEmployee.print();
   cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
   basePlusCommissionEmployee.print();
   cout << "\nearned $" << basePlusCommissionEmployee.earnings()
      << "\n\n";

   // create vector of three base-class pointers
   vector < Employee * > employees( 3 );

   // initialize vector with Employees
   employees[ 0 ] = &salariedEmployee;
   employees[ 1 ] = &commissionEmployee;
   employees[ 2 ] = &basePlusCommissionEmployee;

   cout << "Employees processed polymorphically via dynamic binding:\n\n";

   // call virtualViaPointer to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class pointers:\n\n";

   for ( size_t i = 0; i < employees.size(); ++i )
      virtualViaPointer( employees[ i ] );

   // call virtualViaReference to print each Employee's information
   // and earnings using dynamic binding
   cout << "Virtual function calls made off base-class references:\n\n";
```

Fig. 13.17 | Employee class hierarchy driver program. (Part 2 of 4.)

```cpp

   for ( size_t i = 0; i < employees.size(); ++i )
      virtualViaReference( *employees[ i ] ); // note dereferencing
} // end main

// call Employee virtual functions print and earnings off a
// base-class pointer using dynamic binding
void virtualViaPointer( const Employee * const baseClassPtr )
{
   baseClassPtr->print();
   cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
} // end function virtualViaPointer

// call Employee virtual functions print and earnings off a
// base-class reference using dynamic binding
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();
   cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
} // end function virtualViaReference
```

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 13.17 | `Employee` class hierarchy driver program. (Part 3 of 4.)

```
Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Fig. 13.17 | `Employee` class hierarchy driver program. (Part 4 of 4.)

Line 41 allocates `vector employees`, which contains three `Employee` pointers. Line 44 aims `employees[0]` at object `salariedEmployee`. Line 45 aims `employees[1]` at object `commissionEmployee`. Line 46 aims `employee[2]` at object `basePlusCommissionEmployee`. The compiler allows these assignments, because a `SalariedEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`. Therefore, we can assign the addresses of `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to base-class `Employee` pointers, even though `Employee` is an *abstract* class.

The loop in lines 54–55 traverses `vector employees` and invokes function `virtualViaPointer` (lines 67–71) for each element in `employees`. Function `virtualViaPointer` receives in parameter `baseClassPtr` the address stored in an `employees` element. Each call to `virtualViaPointer` uses `baseClassPtr` to invoke `virtual` functions `print` (line 69) and `earnings` (line 70). Function `virtualViaPointer` does not contain any `SalariedEmployee`, `CommissionEmployee` or `BasePlusCommissionEmployee` type information. The function knows only about base-class type `Employee`. Therefore, the compiler *cannot know* which concrete class's functions to call through `baseClassPtr`. Yet at execution time, each virtual-function invocation calls the function on the object to which `baseClassPtr` points at that moment. The output illustrates that *the appropriate functions for each class are indeed invoked* and that each object's proper information is displayed. For instance, the weekly salary is displayed for the `SalariedEmployee`, and the gross sales are displayed for the `CommissionEmployee` and `BasePlusCommissionEmployee`. Also, obtaining the earnings of each `Employee` polymorphically in line 70 produces the same results as obtaining these employees' earnings via static binding in lines 33, 35 and 37. All `virtual` function calls to `print` and `earnings` are resolved at runtime with dynamic binding.

Finally, another `for` statement (lines 61–62) traverses `employees` and invokes function `virtualViaReference` (lines 75–79) for each element in the `vector`. Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a *reference* to the object obtained by *dereferencing the pointer* stored in each `employees` element (line 62). Each call to `virtualViaReference` invokes `virtual` functions `print` (line 77) and `earnings` (line 78) via reference `baseClassRef` to demonstrate that *polymorphic* *processing occurs with base-class references as well*. Each virtual-function invocation calls the function on the object to which baseClassRef refers at runtime. This is another example of *dynamic binding*. The output produced using base-class references is identical to the output produced using base-class pointers.

13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

C++ makes polymorphism easy to program. It's certainly possible to program for polymorphism in non-object-oriented languages such as C, but doing so requires complex and potentially dangerous pointer manipulations. This section discusses how C++ can implement polymorphism, virtual functions and dynamic binding internally. This will give you a solid understanding of how these capabilities really work. More importantly, it will help you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time. This will help you determine when to use polymorphism and when to avoid it. As you'll see in Chapter 22, the STL components were implemented *without* polymorphism and virtual functions—this was done to avoid the associated execution-time overhead and achieve optimal performance to meet the unique requirements of the STL.

First, we'll explain the data structures that the compiler builds at *compile time* to support polymorphism at execution time. You'll see that polymorphism is accomplished through three levels of pointers, i.e., *triple indirection*. Then we'll show how an executing program uses these data structures to execute virtual functions and achieve the dynamic binding associated with polymorphism. Our discussion explains one possible implementation; this is not a language requirement.

When C++ compiles a class that has one or more virtual functions, it builds a **virtual function table (*vtable*)** for that class. An executing program uses the *vtable* to select the proper function implementation each time a virtual function of that class is called. The leftmost column of Fig. 13.18 illustrates the *vtables* for the classes Employee, SalariedEmployee, CommissionEmployee and BasePlusCommissionEmployee.

Employee Class vtable

In the *vtable* for class Employee, the first function pointer is set to 0 (i.e., the null pointer). This is done because function earnings is a pure virtual function and therefore *lacks an implementation*. The second function pointer points to function print, which displays the employee's full name and social security number. [*Note:* We've abbreviated the output of each print function in this figure to conserve space.] Any class that has one or more null pointers in its *vtable* is an *abstract* class. Classes without any null *vtable* pointers (such as SalariedEmployee, CommissionEmployee and BasePlusCommissionEmployee) are *concrete* classes.

SalariedEmployee Class vtable

Class SalariedEmployee overrides function earnings to return the employee's weekly salary, so the function pointer points to the earnings function of class SalariedEmployee. SalariedEmployee also overrides print, so the corresponding function pointer points to the SalariedEmployee member function that prints "salaried employee: " followed by the employee's name, social security number and weekly salary.

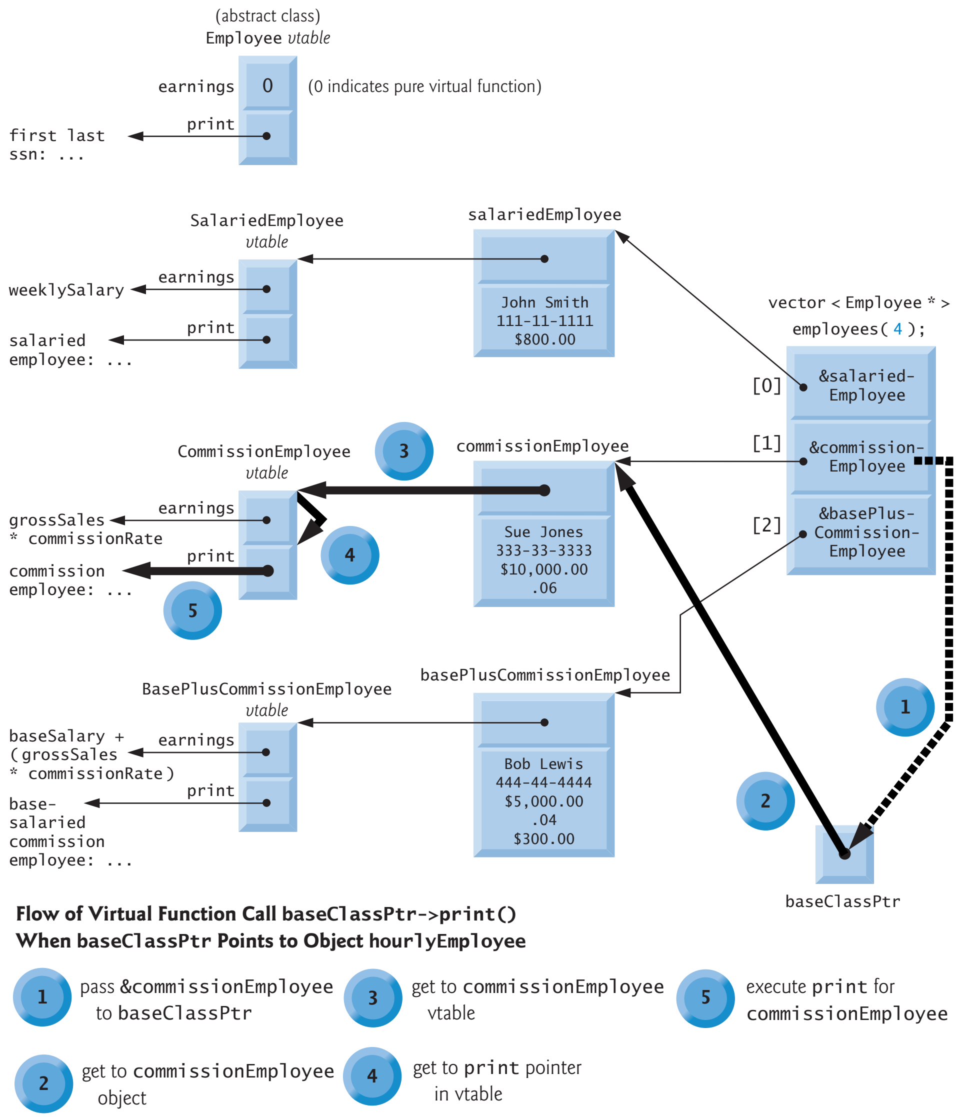


Fig. 13.18 | How `virtual` function calls work.

`CommissionEmployee` *Class* vtable

The `earnings` function pointer in the *vtable* for class `CommissionEmployee` points to `CommissionEmployee`’s `earnings` function that returns the employee’s gross sales multiplied by the commission rate. The `print` function pointer points to the `CommissionEmployee` version of the function, which prints the employee’s type, name, social security number, commission rate and gross sales. As in class `HourlyEmployee`, both functions override the functions in class `Employee`.

`BasePlusCommissionEmployee` Class vtable

The `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee` points to the `BasePlusCommissionEmployee`'s `earnings` function, which returns the employee's base salary plus gross sales multiplied by commission rate. The `print` function pointer points to the `BasePlusCommissionEmployee` version of the function, which prints the employee's base salary plus the type, name, social security number, commission rate and gross sales. Both functions override the functions in class `CommissionEmployee`.

Inheriting Concrete `virtual` Functions

In our `Employee` case study, each concrete class provides its own implementation for `virtual` functions `earnings` and `print`. You've learned that each class which inherits directly from abstract base class `Employee` *must implement `earnings`* in order to be a concrete class, because `earnings` is a pure `virtual` function. These classes do *not* need to implement function `print`, however, to be considered concrete—`print` is not a pure `virtual` function and derived classes can inherit class `Employee`'s implementation of `print`. Furthermore, class `BasePlusCommissionEmployee` does *not* have to implement either function `print` or `earnings`—both function implementations can be inherited from class `CommissionEmployee`. If a class in our hierarchy were to inherit function implementations in this manner, the *vtable* pointers for these functions would simply point to the function implementation that was being inherited. For example, if `BasePlusCommissionEmployee` did not override `earnings`, the `earnings` function pointer in the *vtable* for class `BasePlusCommissionEmployee` would point to the same `earnings` function as the *vtable* for class `CommissionEmployee` points to.

Three Levels of Pointers to Implement Polymorphism

Polymorphism is accomplished through an elegant data structure involving *three levels of pointers*. We've discussed one level—the function pointers in the *vtable*. These point to the actual functions that execute when a `virtual` function is invoked.

Now we consider the second level of pointers. *Whenever an object of a class with one or more `virtual` functions is instantiated, the compiler attaches to the object a pointer to the* vtable *for that class.* This pointer is normally at the front of the object, but it isn't required to be implemented that way. In Fig. 13.18, these pointers are associated with the objects created in Fig. 13.17 (one object for each of the types `SalariedEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`). The diagram displays each of the object's data member values. For example, the `salariedEmployee` object contains a pointer to the `SalariedEmployee` *vtable*; the object also contains the values `John Smith`, `111-11-1111` and `$800.00`.

The third level of pointers simply contains the handles to the objects that receive the `virtual` function calls. The handles in this level may also be references. Fig. 13.18 depicts the `vector employees` that contains `Employee` pointers.

Now let's see how a typical `virtual` function call executes. Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 69 of Fig. 13.17). Assume that `baseClassPtr` contains `employees[ 1 ]` (i.e., the address of object `commissionEmployee` in `employees`). When the compiler compiles this statement, it determines that the call is indeed being made via a base-class pointer and that `print` is a `virtual` function.

The compiler determines that `print` is the *second* entry in each of the *vtables*. To locate this entry, the compiler notes that it will need to skip the first entry. Thus, the compiler compiles an **offset** or **displacement** into the table of machine-language object-code pointers to find the code that will execute the `virtual` function call. The size in bytes of the offset depends on the number of bytes used to represent a pointer on an individual platform. For example, on a 32-bit platform, a pointer is typically stored in 4 bytes, whereas on a 64-bit platform, a pointer could be stored in 8 bytes.

The compiler generates code that performs the following operations [*Note:* The numbers in the list correspond to the circled numbers in Fig. 13.18]:

1. Select the i^{th} entry of `employees` (in this case, the address of object `commissionEmployee`), and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `commissionEmployee`.
2. *Dereference* that pointer to get to the `commissionEmployee` object—which, as you recall, begins with a pointer to the `CommissionEmployee` *vtable*.
3. *Dereference* `commissionEmployee`'s *vtable* pointer to get to the `CommissionEmployee` *vtable*.
4. Skip the offset of four bytes to select the `print` function pointer.
5. *Dereference* the `print` function pointer to form the "name" of the actual function to execute, and use the function call operator `()` to execute the appropriate `print` function, which in this case prints the employee's type, name, social security number, gross sales and commission rate.

Fig. 13.18's data structures may appear to be complex, but this complexity is managed by the compiler and hidden from you, making polymorphic programming straightforward. The pointer dereferencing operations and memory accesses that occur on every `virtual` function call require some additional execution time. The *vtables* and the *vtable* pointers added to the objects require some additional memory. You now have enough information to determine whether `virtual` functions are appropriate for your programs.

Performance Tip 13.1

Polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient. You can use these capabilities with nominal impact on performance.

Performance Tip 13.2

Virtual functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that runs as efficiently as hand-coded `switch`-based logic. Polymorphism's overhead is acceptable for most applications. But in some situations—such as real-time applications with stringent performance requirements—polymorphism's overhead may be too high.

13.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

Recall from the problem statement at the beginning of Section 13.6 that, for the current pay period, our fictitious company has decided to reward `BasePlusCommissionEmployees` by adding 10 percent to their base salaries. When processing `Employee` objects polymorphically in Section 13.6.5, we did not need to worry about the "specifics." Now, however, to adjust the base salaries of `BasePlusCommissionEmployees`, we have to determine the specific type of each `Employee` object at execution time, then act appropriately. This section demonstrates the powerful capabilities of **runtime type information (RTTI)** and **dynamic casting**, which enable a program to determine the type of an object at execution time and act on that object accordingly.

[*Note:* Some compilers require that RTTI be enabled before it can be used in a program. In Visual C++ 2010, this option is enabled by default.]

Figure 13.19 uses the `Employee` hierarchy developed in Section 13.6 and increases by 10 percent the base salary of each `BasePlusCommissionEmployee`. Line 21 declares three-element `vector employees` that stores pointers to `Employee` objects. Lines 24–29 populate the `vector` with the addresses of dynamically allocated objects of classes `SalariedEmployee` (Figs. 13.11–13.12), `CommissionEmployee` (Figs. 13.13–13.14) and `BasePlusCommissionEmployee` (Figs. 13.15–13.16).

```
1  // Fig. 13.19: fig13_19.cpp
2  // Demonstrating downcasting and runtime type information.
3  // NOTE: You may need to enable RTTI on your compiler
4  // before you can execute this application.
5  #include <iostream>
6  #include <iomanip>
7  #include <vector>
8  #include <typeinfo>
9  #include "Employee.h"
10 #include "SalariedEmployee.h"
11 #include "CommissionEmployee.h"
12 #include "BasePlusCommissionEmployee.h"
13 using namespace std;
14
15 int main()
16 {
17    // set floating-point output formatting
18    cout << fixed << setprecision( 2 );
19
20    // create vector of three base-class pointers
21    vector < Employee * > employees( 3 );
22
23    // initialize vector with various kinds of Employees
24    employees[ 0 ] = new SalariedEmployee(
25       "John", "Smith", "111-11-1111", 800 );
26    employees[ 1 ] = new CommissionEmployee(
27       "Sue", "Jones", "333-33-3333", 10000, .06 );
28    employees[ 2 ] = new BasePlusCommissionEmployee(
29       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
30
31    // polymorphically process each element in vector employees
32    for ( size_t i = 0; i < employees.size(); ++i )
33    {
34       employees[ i ]->print(); // output employee information
35       cout << endl;
36
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 1 of 2.)

```
37         // downcast pointer
38         BasePlusCommissionEmployee *derivedPtr =
39            dynamic_cast < BasePlusCommissionEmployee * >
40               ( employees[ i ] );
41
42         // determine whether element points to base-salaried
43         // commission employee
44         if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
45         {
46            double oldBaseSalary = derivedPtr->getBaseSalary();
47            cout << "old base salary: $" << oldBaseSalary << endl;
48            derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
49            cout << "new base salary with 10% increase is: $"
50               << derivedPtr->getBaseSalary() << endl;
51         } // end if
52
53         cout << "earned $" << employees[ i ]->earnings() << "\n\n";
54      } // end for
55
56      // release objects pointed to by vector's elements
57      for ( size_t j = 0; j < employees.size(); ++j )
58      {
59         // output class name
60         cout << "deleting object of "
61            << typeid( *employees[ j ] ).name() << endl;
62
63         delete employees[ j ];
64      } // end for
65   } // end main
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

Fig. 13.19 | Demonstrating downcasting and runtime type information. (Part 2 of 2.)

The `for` statement in lines 32–54 iterates through the `employees vector` and displays each `Employee`'s information by invoking member function `print` (line 34). Recall that because print is declared virtual in base class Employee, the system invokes the appropriate derived-class object's print function.

In this example, as we encounter BasePlusCommissionEmployee objects, we wish to increase their base salary by 10 percent. Since we process the employees generically (i.e., polymorphically), we cannot (with the techniques we've learned) be certain as to which type of Employee is being manipulated at any given time. This creates a problem, because BasePlusCommissionEmployee employees *must* be identified when we encounter them so they can receive the 10 percent salary increase. To accomplish this, we use operator **dynamic_cast** (line 39) to determine whether the type of each object is BasePlusCommissionEmployee. This is the *downcast* operation we referred to in Section 13.3.3. Lines 38–40 dynamically downcast employees[i] from type Employee * to type BasePlusCommissionEmployee *. If the vector element points to an object that *is a* BasePlusCommissionEmployee object, then that object's address is assigned to commissionPtr; otherwise, 0 is assigned to derived-class pointer derivedPtr.

If the value returned by the dynamic_cast operator in lines 38–40 *is not* 0, the object *is* the correct type, and the if statement (lines 44–51) performs the special processing required for the BasePlusCommissionEmployee object. Lines 46, 48 and 50 invoke BasePlusCommissionEmployee functions getBaseSalary and setBaseSalary to retrieve and update the employee's salary.

Line 53 invokes member function earnings on the object to which employees[i] points. Recall that earnings is declared virtual in the base class, so the program invokes the derived-class object's earnings function—another example of dynamic binding.

Lines 57–64 display each employee's object type and uses the delete operator to deallocate the dynamic memory to which each vector element points. Operator **typeid** (line 61) returns a reference to an object of class **type_info** that contains the information about the type of its operand, including the name of that type. When invoked, type_info member function **name** (line 51) returns a pointer-based string that contains the type name (e.g., "class BasePlusCommissionEmployee") of the argument passed to typeid. To use typeid, the program must include header **<typeinfo>** (line 8).

Portability Tip 13.1

The string returned by type_info *member function* name *may vary by compiler.*

We avoid several compilation errors in this example by *downcasting* an Employee pointer to a BasePlusCommissionEmployee pointer (lines 38–40). If we remove the dynamic_cast from line 39 and attempt to assign the current Employee pointer directly to BasePlusCommissionEmployee pointer derivedPtr, we'll receive a compilation error. C++ does not allow a program to assign a base-class pointer to a derived-class pointer because the *is-a* relationship does not apply—a CommissionEmployee is *not* a BasePlusCommissionEmployee. The *is-a* relationship applies only between the derived class and its base classes, not vice versa.

Similarly, if lines 46, 48 and 50 used the current base-class pointer from employees, rather than derived-class pointer derivedPtr, to invoke derived-class-only functions getBaseSalary and setBaseSalary, we would receive a compilation error at each of these lines. As you learned in Section 13.3.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed. Although lines 46, 48 and 50 execute only if `commissionPtr` is not `0` (i.e., if the cast *can* be performed), we *cannot* attempt to invoke derived-class `BasePlusCommissionEmployee` functions `getBaseSalary` and `setBaseSalary` on the base-class `Employee` pointer. Recall that, using a base class `Employee` pointer, we can invoke only functions found in base class `Employee`—`earnings`, `print` and `Employee`'s *get* and *set* functions.

13.9 Virtual Destructors

A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy. So far you've seen **nonvirtual destructors**—destructors that are not declared with keyword `virtual`. If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the `delete` operator to a base-class pointer to the object, the C++ standard specifies that the behavior is *undefined*.

The simple solution to this problem is to create a **`virtual` destructor** (i.e., a destructor that is declared with keyword `virtual`) in the base class. This makes all derived-class destructors `virtual` *even though they do not have the same name as the base-class destructor*. Now, if an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a base-class pointer, the destructor for the appropriate class is called based on the object to which the base-class pointer points. Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it's important for the destructors of both the derived class and base class to execute. The base-class destructor automatically executes after the derived-class destructor.

Error-Prevention Tip 13.2

If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base class pointer.

Common Programming Error 13.2

Constructors cannot be `virtual`. Declaring a constructor `virtual` is a compilation error.

13.10 Wrap-Up

In this chapter we discussed polymorphism, which enables us to "program in the general" rather than "program in the specific," and we showed how this makes programs more extensible. We began with an example of how polymorphism would allow a screen manager to display several "space" objects. We then demonstrated how base-class and derived-class pointers can be aimed at base-class and derived-class objects. We said that aiming base-class pointers at base-class objects is natural, as is aiming derived-class pointers at derived-class objects. Aiming base-class pointers at derived-class objects is also natural because a derived-class object *is an* object of its base class. You learned why aiming derived-class pointers at base-class objects is dangerous and why the compiler disallows such assignments. We introduced `virtual` functions, which enable the proper functions to be called when objects at various levels of an inheritance hierarchy are referenced (at execution time) via base-class pointers or references. This is known as dynamic or late binding. We then discussed pure `virtual` functions and abstract classes (classes with one or more pure vir-

`tual` functions). You learned that abstract classes cannot be used to instantiate objects, while concrete classes can. We then demonstrated using abstract classes in an inheritance hierarchy. You learned how polymorphism works "under the hood" with *vtables* that are created by the compiler. We used runtime type information (RTTI) and dynamic casting to determine the type of an object at execution time and act on that object accordingly. The chapter concluded with a discussion of `virtual` destructors, and how they ensure that all appropriate destructors in an inheritance hierarchy run on a derived-class object when that object is deleted via a base-class pointer or reference.

In the next chapter, we discuss templates, a sophisticated feature of C++ that enables you to define a family of related classes or functions with a single code segment.

Summary

Section 13.1 Introduction

- Polymorphism (p. 535) enables us to "program in the general" rather than "program in the specific."
- Polymorphism enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.
- With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.

Section 13.2 Introduction to Polymorphism: Polymorphic Video Game

- With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked.
- This makes it possible to design and implement more extensible systems. Programs can be written to process objects of types that may not exist when the program is under development.

Section 13.3 Relationships Among Objects in an Inheritance Hierarchy

- C++ enables polymorphism—the ability for objects of different classes related by inheritance to respond differently to the same member-function call.
- Polymorphism is implemented via `virtual` functions (p. 543) and dynamic binding (p. 544).
- When a base-class pointer or reference is used to call a `virtual` function, C++ chooses the correct overridden function in the appropriate derived class associated with the object.
- If a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator, the reference is resolved at compile time (this is called static binding; p. 544); the `virtual` function that is called is the one defined for the class of that particular object.
- Derived classes can provide their own implementations of a base-class `virtual` function if necessary, but if they do not, the base class's implementation is used.

Section 13.4 Type Fields and `switch` Statements

- Polymorphic programming with `virtual` functions can eliminate the need for `switch` logic. You can use the `virtual` function mechanism to perform the equivalent logic automatically, thus avoiding the kinds of errors typically associated with `switch` logic.

Section 13.5 Abstract Classes and Pure `virtual` Functions

- Abstract classes (p. 550) are typically used as base classes, so we refer to them as abstract base classes (p. 550). No objects of an abstract class may be instantiated.
- Classes from which objects can be instantiated are concrete classes (p. 550).
- You create an abstract class by declaring one or more pure `virtual` functions (p. 551) with pure specifiers (`= 0`) in their declarations.
- If a class is derived from a class with a pure `virtual` function and that derived class does not supply a definition for that pure `virtual` function, then that `virtual` function remains pure in the derived class. Consequently, the derived class is also an abstract class.
- Although we cannot instantiate objects of abstract base classes, we can declare pointers and references to objects of abstract base classes. Such pointers and references can be used to enable polymorphic manipulations of derived-class objects instantiated from concrete derived classes.

Section 13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Dynamic binding requires that at runtime, the call to a virtual member function be routed to the `virtual` function version appropriate for the class. A `virtual` function table called the *vtable* (p. 567) is implemented as an array containing function pointers. Each class with `virtual` functions has a *vtable*. For each `virtual` function in the class, the *vtable* has an entry containing a function pointer to the version of the `virtual` function to use for an object of that class. The `virtual` function to use for a particular class could be the function defined in that class, or it could be a function inherited either directly or indirectly from a base class higher in the hierarchy.
- When a base class provides a `virtual` member function, derived classes can override the `virtual` function, but they do not have to override it.
- Each object of a class with `virtual` functions contains a pointer to the *vtable* for that class. When a function call is made from a base-class pointer to a derived-class object, the appropriate function pointer in the *vtable* is obtained and dereferenced to complete the call at execution time.
- Any class that has one or more `0` pointers in its *vtable* is an abstract class. Classes without any `0` *vtable* pointers are concrete classes.
- New kinds of classes are regularly added to systems and accommodated by dynamic binding.

Section 13.8 Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, `dynamic_cast`, `typeid` and `type_info`

- Operator `dynamic_cast` (p. 571) checks the type of the object to which a pointer points, then determines whether the type has an *is-a* relationship with the type to which the pointer is being converted. If so, `dynamic_cast` returns the object's address. If not, `dynamic_cast` returns `0`.
- Operator `typeid` (p. 573) returns a reference to a `type_info` object (p. 573) that contains information about the operand's type, including the type name. To use `typeid`, the program must include header `<typeinfo>` (p. 573).
- When invoked, `type_info` member function `name` (p. 573) returns a pointer-based string that contains the name of the type that the `type_info` object represents.
- Operators `dynamic_cast` and `typeid` are part of C++'s runtime type information (RTTI; p. 571) feature, which allows a program to determine an object's type at runtime.

Section 13.9 Virtual Destructors

- Declare the base-class destructor `virtual` (p. 574) if the class contains `virtual` functions. This makes all derived-class destructors virtual, even though they do not have the same name as the base-class destructor. If an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called. After a derived-class destructor runs, the destructors for all of that class's base classes run all the way up the hierarchy.

Self-Review Exercises

13.1 Fill in the blanks in each of the following statements:

a) Treating a base-class object as a(n) ________ can cause errors.
b) Polymorphism helps eliminate ________ logic.
c) If a class contains at least one pure `virtual` function, it's a(n) ________ class.
d) Classes from which objects can be instantiated are called ________ classes.
e) Operator ________ can be used to downcast base-class pointers safely.
f) Operator `typeid` returns a reference to a(n) ________ object.
g) ________ involves using a base-class pointer or reference to invoke `virtual` functions on base-class and derived-class objects.
h) Overridable functions are declared using keyword ________.
i) Casting a base-class pointer to a derived-class pointer is called ________.

13.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) All `virtual` functions in an abstract base class must be declared as pure `virtual` functions.
b) Referring to a derived-class object with a base-class handle is dangerous.
c) A class is made abstract by declaring that class `virtual`.
d) If a base class declares a pure `virtual` function, a derived class must implement that function to become a concrete class.
e) Polymorphic programming can eliminate the need for `switch` logic.

Answers to Self-Review Exercises

13.1 a) derived-class object. b) `switch`. c) abstract. d) concrete. e) `dynamic_cast`. f) `type_info`. g) Polymorphism. h) `virtual`. i) downcasting.

13.2 a) False. An abstract base class can include virtual functions with implementations. b) False. Referring to a base-class object with a derived-class handle is dangerous. c) False. Classes are never declared `virtual`. Rather, a class is made abstract by including at least one pure virtual function in the class. d) True. e) True.

Exercises

13.3 *(Programming in the General)* How is it that polymorphism enables you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

13.4 *(Polymorphism vs. `switch` logic)* Discuss the problems of programming with `switch` logic. Explain why polymorphism can be an effective alternative to using `switch` logic.

13.5 *(Inheriting Interface vs. Implementation)* Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?

13.6 *(Virtual Functions)* What are `virtual` functions? Describe a circumstance in which `virtual` functions would be appropriate.

13.7 *(Dynamic Binding vs. Static Binding)* Distinguish between static binding and dynamic binding. Explain the use of `virtual` functions and the *vtable* in dynamic binding.

13.8 *(Virtual Functions)* Distinguish between `virtual` functions and pure `virtual` functions.

13.9 *(Abstract Base Classes)* Suggest one or more levels of abstract base classes for the `Shape` hierarchy discussed in this chapter and shown in Fig. 12.3. (The first level is `Shape`, and the second level consists of the classes `TwoDimensionalShape` and `ThreeDimensionalShape`.)

13.10 *(Polymorphism and Extensibility)* How does polymorphism promote extensibility?

13.11 *(Polymorphic Application)* You've been asked to develop a flight simulator that will have elaborate graphical outputs. Explain why polymorphic programming could be especially effective for a problem of this nature.

13.12 *(Payroll System Modification)* Modify the payroll system of Figs. 13.9–13.17 to include `private` data member `birthDate` in class `Employee`. Use class `Date` from Figs. 11.6–11.7 to represent an employee's birthday. Assume that payroll is processed once per month. Create a `vector` of `Employee` references to store the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the month in which the `Employee`'s birthday occurs.

13.13 *(`Shape` Hierarchy)* Implement the `Shape` hierarchy designed in Exercise 12.7 (which is based on the hierarchy in Fig. 12.3). Each `TwoDimensionalShape` should contain function `getArea` to calculate the area of the two-dimensional shape. Each `ThreeDimensionalShape` should have member functions `getArea` and `getVolume` to calculate the surface area and volume, respectively, of the three-dimensional shape. Create a program that uses a `vector` of `Shape` pointers to objects of each concrete class in the hierarchy. The program should print the object to which each `vector` element points. Also, in the loop that processes all the shapes in the `vector`, determine whether each shape is a `TwoDimensionalShape` or a `ThreeDimensionalShape`. If a shape is a `TwoDimensionalShape`, display its area. If a shape is a `ThreeDimensionalShape`, display its area and volume.

13.14 *(Project: Polymorphic Screen Manager Using `Shape` Hierarchy)* Develop a basic graphics package. Use the `Shape` hierarchy implemented in Exercise 13.13. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill characters to be used in drawing each shape. The user can specify more than one of the same shape. As you create each shape, place a `Shape *` pointer to each new `Shape` object into an array. Each `Shape` class should now have its own `draw` member function. Write a polymorphic screen manager that walks through the array, sending `draw` messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape.

13.15 *(`Package` Inheritance Hierarchy)* Use the `Package` inheritance hierarchy created in Exercise 12.9 to create a program that displays the address information and calculates the shipping costs for several `Packages`. The program should contain a `vector` of `Package` pointers to objects of classes `TwoDayPackage` and `OvernightPackage`. Loop through the `vector` to process the `Packages` polymorphically. For each `Package`, invoke *get* functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each `Package`'s `calculateCost` member function and print the result. Keep track of the total shipping cost for all `Packages` in the `vector`, and display this total when the loop terminates.

13.16 *(Polymorphic Banking Program Using `Account` Hierarchy)* Develop a polymorphic banking program using the `Account` hierarchy created in Exercise 12.10. Create a `vector` of `Account` pointers to `SavingsAccount` and `CheckingAccount` objects. For each `Account` in the `vector`, allow the user to specify an amount of money to withdraw from the `Account` using member function `debit` and an amount of money to deposit into the `Account` using member function `credit`. As you process each `Account`, determine its type. If an `Account` is a `SavingsAccount`, calculate the amount of interest owed to the `Account` using member function `calculateInterest`, then add the interest to the account balance using member function `credit`. After processing an `Account`, print the updated account balance obtained by invoking base-class member function `getBalance`.

13.17 *(Payroll System Modification)* Modify the payroll system of Figs. 13.9–13.17 to include an additional `Employee` subclasses `PieceWorker` and `HourlyWorker`. A `PieceWorker` represents an employee whose pay is based on the number of pieces of merchandise produced. An `HourlyWorker` represents an employee whose pay is based on an hourly wage and the number of hours worked. Hourly workers receive overtime pay (1.5 times the hourly wage) for all hours worked in excess of 40 hours.

Class `PieceWorker` should contain `private` instance variables `wage` (to store the employee's wage per piece) and `pieces` (to store the number of pieces produced). Class `HourlyWorker` should contain `private` instance variables `wage` (to store the employee's wage per hour) and `hours` (to store the hours worked). In class `PieceWorker`, provide a concrete implementation of method `earnings` that calculates the employee's earnings by multiplying the number of pieces produced by the wage per piece. In class `HourlyWorker`, provide a concrete implementation of method `earnings` that calculates the employee's earnings by multiplying the number of hours worked by the wage per hour. If the number of hours worked is over 40, be sure to pay the `HourlyWorker` for the overtime hours. Add a pointer to an object of each new class into the vector of `Employee` pointers in `main`. For each `Employee`, display its `string` representation and earnings.

Making a Difference

13.18 *(`CarbonFootprint` Abstract Class: Polymorphism)* Using an abstract class with only pure virtual functions, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three small classes unrelated by inheritance—classes `Building`, `Car` and `Bicycle`. Give each class some unique appropriate attributes and behaviors that it does not have in common with other classes. Write an abstract class `CarbonFootprint` with only a pure virtual `getCarbonFootprint` method. Have each of your classes inherit from that abstract class and implement the `getCarbonFootprint` method to calculate an appropriate carbon footprint for that class (check out a few websites that explain how to calculate carbon footprints). Write an application that creates objects of each of the three classes, places pointers to those objects in a `vector` of `CarbonFootprint` pointers, then iterates through the `vector`, polymorphically invoking each object's `getCarbonFootprint` method. For each object, print some identifying information and the object's carbon footprint.

Templates

14

Behind that outside pattern the dim shapes get clearer every day.
It is always the same shape, only very numerous.
—Charlotte Perkins Gilman

Every man of genius sees the world at a different angle from his fellows.
—Havelock Ellis

…our special individuality, as distinguished from our generic humanity.
—Oliver Wendell Holmes, Sr.

Objectives

In this chapter you'll learn:

- To use function templates to conveniently create a group of related (overloaded) functions.
- To distinguish between function templates and function-template specializations.
- To use class templates to create groups of related types.
- To distinguish between class templates and class-template specializations.
- To overload function templates.

Outline

14.1 Introduction

In this chapter, we discuss one of C++'s more powerful software reuse features, namely **templates**. **Function templates** and **class templates** enable you to specify, with a single code segment, an entire range of related (overloaded) functions—called **function-template specializations**—or an entire range of related classes—called **class-template specializations**. This technique is called **generic programming**.

We might write a single function template for an array-sort function, then have C++ generate separate function-template specializations that will sort `int` arrays, `float` arrays, `string` arrays and so on. We introduced function templates in Chapter 6. We present an additional discussion and example in this chapter.

We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a stack-of-`int` class, a stack-of-`float` class, a stack-of-`string` class and so on.

Note the distinction between templates and template specializations: Function templates and class templates are like stencils out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors.

In this chapter, we present a function template and a class template. This chapter is only an introduction to templates. Chapter 22, Standard Template Library (STL), presents a rich treatment of the template container classes, iterators and algorithms of the STL. Chapter 22 contains dozens of complete template-based examples illustrating more sophisticated template-programming techniques than those used here.

Software Engineering Observation 14.1

Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in headers, which are then `#include`d in the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header.

14.2 Function Templates

Overloaded functions normally perform *similar* or *identical* operations on *different* types of data. If the operations are *identical* for each type, they can be expressed more compactly and conveniently using function templates. Initially, you write a single function-template definition. Based on the argument types provided explicitly or inferred from calls to this function, the compiler generates separate source-code functions (i.e., *function-template specializations*) to handle each function call appropriately. In the C programming language, this task can be performed using **macros** created with the preprocessor directive `#define` (see Appendix E, Preprocessor). However, macros can have serious *side effects* and do *not* enable the compiler to perform *type checking*.

Error-Prevention Tip 14.1

Function templates, like macros, enable software reuse. Unlike macros, function templates help eliminate many types of errors through the scrutiny of full C++ type checking.

All **function-template definitions** begin with keyword **`template`** followed by a list of **template parameters** to the function template enclosed in **angle brackets** (**<** and **>**); each template parameter that represents a type *must* be preceded by either of the interchangeable keywords `class` or **`typename`**, as in

```
template< typename T >
```

or

```
template< class ElementType >
```

or

```
template< typename BorderType, typename FillType >
```

The type template parameters of a function-template definition are used to specify the types of the function's parameters, to specify the return type of the function and to declare variables within the function. The function definition follows and appears like any other function definition. Keywords `typename` and `class` used to specify function-template parameters actually mean "any fundamental type or user-defined type."

Common Programming Error 14.1

Not placing keyword `class` or keyword `typename` before each type template parameter of a function template is a syntax error.

Example: Function Template `printArray`

Let's examine function template `printArray` in Fig. 14.1, lines 7–14. Function template `printArray` declares (line 7) a single template parameter `T` (`T` can be any valid identifier) for the type of the array to be printed by function `printArray`; `T` is referred to as a **type template parameter**, or type parameter. You'll see nontype template parameters in Section 14.5.

```
1  // Fig. 14.1: fig14_01.cpp
2  // Using function-template specializations.
3  #include <iostream>
4  using namespace std;
5
6  // function template printArray definition
7  template< typename T >
8  void printArray( const T * const array, int count )
9  {
10    for ( int i = 0; i < count; ++i )
11       cout << array[ i ] << " ";
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 1 of 2.)

```
12
13    cout << endl;
14 } // end function template printArray
15
16 int main()
17 {
18    const int aCount = 5; // size of array a
19    const int bCount = 7; // size of array b
20    const int cCount = 6; // size of array c
21
22    int a[ aCount ] = { 1, 2, 3, 4, 5 };
23    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
24    char c[ cCount ] = "HELLO"; // 6th position for null
25
26    cout << "Array a contains:" << endl;
27
28    // call integer function-template specialization
29    printArray( a, aCount );
30
31    cout << "Array b contains:" << endl;
32
33    // call double function-template specialization
34    printArray( b, bCount );
35
36    cout << "Array c contains:" << endl;
37
38    // call character function-template specialization
39    printArray( c, cCount );
40 } // end main
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

Fig. 14.1 | Function-template specializations of function template `printArray`. (Part 2 of 2.)

When the compiler detects a `printArray` function invocation in the client program (e.g., lines 29 and 34), the compiler uses its overload resolution capabilities to find a definition of function `printArray` that best matches the function call. In this case, the only `printArray` function with the appropriate number of parameters is the `printArray` function template (lines 7–14). Consider the function call at line 29. The compiler compares the type of `printArray`'s first argument (`int *` at line 29) to the `printArray` function template's first parameter (`const T * const` at line 8) and deduces that replacing the type parameter `T` with `int` would make the argument consistent with the parameter. Then, the compiler substitutes `int` for `T` throughout the template definition and compiles a `printArray` specialization that can display an array of `int` values. In Fig. 14.1, the compiler creates two `printArray` specializations—one that expects an `int` array and one that expects a `double` array. For example, the function-template specialization for type `int` is

```
void printArray( const int * const array, int count )
{
   for ( int i = 0; i < count; ++i )
      cout << array[ i ] << " ";

   cout << endl;
} // end function printArray
```

As with function parameters, the names of template parameters must be *unique* inside a template definition. Template parameter names need not be unique across *different* function templates.

Figure 14.1 demonstrates function template `printArray` (lines 7–14). The program begins by declaring five-element `int` array `a` and seven-element `double` array `b` (lines 22–23). Then, the program outputs each array by calling `printArray`—once with a first argument `a` of type `int *` (line 29) and once with a first argument `b` of type `double *` (line 34). The call in line 29, for example, causes the compiler to infer that `T` is `int` and to instantiate a `printArray` function-template specialization, for which type parameter `T` is `int`. The call in line 34 causes the compiler to infer that `T` is `double` and to instantiate a second `printArray` function-template specialization, for which type parameter `T` is `double`. It's important to note that if `T` (line 7) represents a user-defined type (which it does not in Fig. 14.1), there must be an overloaded stream insertion operator for that type; otherwise, the first stream insertion operator in line 11 will not compile.

Common Programming Error 14.2

If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., ==, +, <=) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

In this example, the template mechanism saves you from having to write two separate overloaded functions with prototypes

```
void printArray( const int * const, int );
void printArray( const double * const, int );
void printArray( const char * const, int );
```

that all use the same code, except for type `T` (as used in line 8).

Performance Tip 14.1

Although templates offer software-reusability benefits, remember that multiple function-template specializations and class-template specializations are instantiated in a program (at compile time), despite the fact that the templates are written only once. These copies can consume considerable memory. This is not normally an issue, though, because the code generated by the template is the same size as the code you'd have written to produce the separate overloaded functions.

14.3 Overloading Function Templates

Function templates and overloading are intimately related. The function-template specializations generated from a function template all have the same name, so the compiler uses overload resolution to invoke the proper function.

A function template may be overloaded in several ways. We can provide other function templates that specify the same function name but different function parameters. For example, function template `printArray` of Fig. 14.1 could be overloaded with another `printArray` function template with additional parameters `lowSubscript` and `highSubscript` to specify the portion of the array to output (see Exercise 14.4).

A function template also can be overloaded by providing nontemplate functions with the same function name but different function arguments. For example, function template `printArray` of Fig. 14.1 could be overloaded with a nontemplate version that specifically prints an array of character strings in neat, tabular format (see Exercise 14.5).

The compiler performs a matching process to determine what function to call when a function is invoked. First, the compiler tries to find and use a precise match in which the function names and argument types are consistent with those of the function call. If this fails, the compiler determines whether a function template is available that can be used to generate a function-template specialization with a precise match of function name and argument types that are consistent with those of the function call. If such a template is found, the compiler generates and uses the appropriate function-template specialization. If not, the compiler generates an error message. Also, if there are multiple matches for the function call, the compiler attempts to determine the best match. If there is more than one best match, the call is ambiguous and the compiler generates an error message.

Common Programming Error 14.3

A compilation error occurs if no matching function definition can be found for a particular function call or if there are multiple matches that the compiler considers ambiguous.

14.4 Class Templates

It's possible to understand the concept of a "stack" (a data structure into which we insert items at the top and retrieve those items in last-in, first-out order) *independent of the type of the items* being placed in the stack. However, to instantiate a stack, a data type must be specified. This creates a wonderful opportunity for software reusability. We need the means for describing the notion of a stack *generically* and instantiating classes that are *type-specific* versions of this generic stack class. C++ provides this capability through **class templates**.

Software Engineering Observation 14.2

Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.

Class templates are called **parameterized types**, because they require one or more *type parameters* to specify how to customize a "generic class" template to form a class-template specialization. To produce many specializations you write only one class-template definition. When an additional specialization is needed, you use a concise, simple notation, and the compiler writes the source code for that specialization. One `Stack` class template, for example, could thus become the basis for creating many `Stack` classes (such as "`Stack` of `double`," "`Stack` of `int`," "`Stack` of `char`," "`Stack` of `Employee`," etc.) used in a program.

Creating Class Template `Stack< T >`

Note the `Stack` class-template definition in Fig. 14.2. It looks like a conventional class definition, except that it's preceded by the header (line 6)

```
template< typename T >
```

to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created. You need not specifically use identifier `T`—any valid identifier can be used. The type of element to be stored on this `Stack` is mentioned generically as `T` throughout the `Stack` class header and member-function definitions. We'll show how `T` becomes associated with a specific type, such as `double` or `int`. Due to the way this class template is designed, there are two constraints for class types used with this `Stack`—they must have a *default constructor* (for use in line 44 to create the array that stores the stack elements), and their *assignment operators* must properly copy objects into the `Stack` (lines 56 and 70).

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
7  class Stack
8  {
9  public:
10    explicit Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T & ); // push an element onto the Stack
19    bool pop( T & ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the Stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
```

Fig. 14.2 | Stack class template. (Part 1 of 2.)

```
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif
```

Fig. 14.2 | Stack class template. (Part 2 of 2.)

The member-function definitions of a class template are function templates. The member-function definitions that appear outside the class template definition each begin with the header

```
template< typename T >
```

(lines 40, 51 and 65). Thus, each definition resembles a conventional function definition, except that the `Stack` element type always is listed generically as type parameter `T`. The scope resolution operator is used with the class-template name `Stack<T>` (lines 41, 52 and 66) to tie each member-function definition to the class template's scope. In this case, the generic class name is `Stack<T>`. When `doubleStack` is instantiated as type `Stack<double>`, the Stack constructor function-template specialization uses new to create an array of elements of type double to represent the stack (line 44). The statement

```
      stackPtr( new T[ size ] );
```

in the Stack class-template definition is generated by the compiler in the class-template specialization Stack<double> as

```
      stackPtr( new double[ size ] );
```

Driver to Test Class Template Stack< T >

Now, let's consider the driver (Fig. 14.3) that exercises the Stack class template. The driver begins by instantiating object doubleStack of size 5 (line 9). This object is declared to be of class Stack< double > (pronounced "Stack of double"). The compiler associates type double with type parameter T in the class template to produce the source code for a Stack class of type double. Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

```
1  // Fig. 14.3: fig14_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< double > doubleStack( 5 ); // size 5
10    double doubleValue = 1.1;
11
12    cout << "Pushing elements onto doubleStack\n";
13
14    // push 5 doubles onto doubleStack
15    while ( doubleStack.push( doubleValue ) )
16    {
17       cout << doubleValue << ' ';
18       doubleValue += 1.1;
19    } // end while
20
21    cout << "\nStack is full. Cannot push " << doubleValue
22       << "\n\nPopping elements from doubleStack\n";
23
24    // pop elements from doubleStack
25    while ( doubleStack.pop( doubleValue ) )
26       cout << doubleValue << ' ';
27
28    cout << "\nStack is empty. Cannot pop\n";
29
30    Stack< int > intStack; // default size 10
31    int intValue = 1;
32    cout << "\nPushing elements onto intStack\n";
```

Fig. 14.3 | Stack class template test program. (Part 1 of 2.)

```
33
34      // push 10 integers onto intStack
35      while ( intStack.push( intValue ) )
36      {
37         cout << intValue++ << ' ';
38      } // end while
39
40      cout << "\nStack is full. Cannot push " << intValue
41         << "\n\nPopping elements from intStack\n";
42
43      // pop elements from intStack
44      while ( intStack.pop( intValue ) )
45         cout << intValue << ' ';
46
47      cout << "\nStack is empty. Cannot pop" << endl;
48   } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 14.3 | Stack class template test program. (Part 2 of 2.)

Lines 15–19 invoke `push` to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 onto `doubleStack`. The `while` loop terminates when the driver attempts to `push` a sixth value onto `doubleStack` (which is full, because it holds a maximum of five elements). Function `push` returns `false` when it's unable to push a value onto the stack.[1]

Lines 25–26 invoke `pop` in a `while` loop to remove the five values from the stack (note, in the output of Fig. 14.3, that the values do `pop` off in *last-in, first-out order*). When the driver attempts to `pop` a sixth value, the `doubleStack` is empty, so the `pop` loop terminates.

Line 30 instantiates integer stack `intStack` with the declaration

```
Stack< int > intStack;
```

1. Class `Stack` (Fig. 14.2) provides the function `isFull`, which you can use to determine whether the stack is full before attempting a push operation. This avoids the potential error of pushing onto a full stack. We could also have function `push` throw an exception. You could catch that exception, then decide how to handle it appropriately for the application. The same technique can be used with function `pop` when an attempt is made to pop an element from an empty stack.

(pronounced "`intStack` is a `Stack` of `int`"). Because no size is specified, the size defaults to `10` as specified in the default constructor (Fig. 14.2, line 10). Lines 35–38 loop and invoke `push` to place values onto `intStack` until it's full, then lines 44–45 loop and invoke `pop` to remove values from `intStack` until it's empty. Once again, notice in the output that the values `pop` off in last-in, first-out order.

Creating Function Templates to Test Class Template `Stack< T >`

Notice that the code in function `main` of Fig. 14.3 is *almost identical* for both the `doubleStack` manipulations in lines 9–28 and the `intStack` manipulations in lines 30–47. This presents another opportunity to use a function template. Figure 14.4 defines function template `testStack` (lines 10–34) to perform the same tasks as `main` in Fig. 14.3—`push` a series of values onto a `Stack< T >` and `pop` the values off a `Stack< T >`. Function template `testStack` uses template parameter `T` (specified at line 10) to represent the data type stored in the `Stack< T >`. The function template takes four arguments (lines 12–15)—a reference to an object of type `Stack< T >`, a value of type `T` that will be the first value `push`ed onto the `Stack< T >`, a value of type `T` used to increment the values `push`ed onto the `Stack< T >` and a `string` that represents the name of the `Stack< T >` object for output purposes. Function `main` (lines 36–43) instantiates an object of type `Stack< double >` called `doubleStack` (line 38) and an object of type `Stack< int >` called `intStack` (line 39) and uses these objects in lines 41 and 42. The compiler infers the type of `T` for `testStack` from the type used to instantiate the function's first argument (i.e., the type used to instantiate `doubleStack` or `intStack`). The output of Fig. 14.4 precisely matches the output of Fig. 14.3.

```
1  // Fig. 14.4: fig14_04.cpp
2  // Stack class template test program. Function main uses a
3  // function template to manipulate objects of type Stack< T >.
4  #include <iostream>
5  #include <string>
6  #include "Stack.h" // Stack class template definition
7  using namespace std;
8
9  // function template to manipulate Stack< T >
10 template< typename T >
11 void testStack(
12    Stack< T > &theStack, // reference to Stack< T >
13    T value, // initial value to push
14    T increment, // increment for subsequent values
15    const string stackName ) // name of the Stack< T > object
16 {
17    cout << "\nPushing elements onto " << stackName << '\n';
18
19    // push element onto Stack
20    while ( theStack.push( value ) )
21    {
22       cout << value << ' ';
23       value += increment;
24    } // end while
25
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 1 of 2.)

```
   cout << "\nStack is full. Cannot push " << value
      << "\n\nPopping elements from " << stackName << '\n';

   // pop elements from Stack
   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop" << endl;
} // end function template testStack

int main()
{
   Stack< double > doubleStack( 5 ); // size 5
   Stack< int > intStack; // default size 10

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );
} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Fig. 14.4 | Passing a `Stack` template object to a function template. (Part 2 of 2.)

14.5 Nontype Parameters and Default Types for Class Templates

Class template `Stack` of Section 14.4 used only a type parameter in the template header (Fig. 14.2, line 6). It's also possible to use **non-type template parameters**, which can have default arguments and are treated as `const`s. For example, the template header could be modified to take an `int elements` parameter as follows:

```
template< typename T, int elements > // nontype parameter elements
```

Then, a declaration such as

```
Stack< double, 100 > mostRecentSalesFigures;
```

could be used to instantiate (at compile time) a 100-element `Stack` class-template specialization of `double` values named `mostRecentSalesFigures`; this class-template specialization would be of type `Stack<double, 100>`. The class definition then might contain a `private` data member with an array declaration such as

```
T stackHolder[ elements ]; // array to hold Stack contents
```

In addition, a type parameter can specify a **default type**. For example,

```
template< typename T = string > // defaults to type string
```

might specify that a `Stack` contains `string` objects by default. Then, a declaration such as

```
Stack<> jobDescriptions;
```

could be used to instantiate a `Stack` class-template specialization of `string`s named `jobDescriptions`; this class-template specialization would be of type `Stack<string>`. Default type parameters must be the *rightmost* (trailing) parameters in a template's type-parameter list. When one is instantiating a class with two or more default types, if an omitted type is not the rightmost type parameter in the type-parameter list, then all type parameters to the right of that type also must be omitted.

Performance Tip 14.2

When appropriate, specify the size of a container class (such as an array class or a stack class) at compile time (possibly through a nontype template parameter). This eliminates the execution-time overhead of using `new` *to create the space dynamically.*

Software Engineering Observation 14.3

Specifying the size of a container at compile time avoids the potentially fatal execution-time error if `new` *is unable to obtain the needed memory.*

In the exercises, you'll be asked to use a nontype parameter to create a template for our class `Array` from Chapter 11. This template will enable `Array` objects to be instantiated with a specified number of elements of a specified type at compile time, rather than creating space for the `Array` objects at execution time.

In some cases, it may not be possible to use a particular type with a class template. For example, our `Stack` template (Fig. 14.2) requires that class types that will be stored in a `Stack` must provide a default constructor and an assignment operator that properly copies objects. If a particular user-defined type will not work with our `Stack` template or requires customized processing, you can define an **explicit specialization** of the class template for a particular type. Let's assume we want to create an explicit specialization `Stack` for `Employee` objects. To do this, form a new class with the name `Stack<Employee>` as follows:

```
template<>
class Stack< Employee >
{
   // body of class definition
};
```

The `Stack<Employee>` explicit specialization is a complete replacement for the `Stack` class template that is specific to type `Employee`—it does *not* use anything from the original class template and can even have different members.

14.6 Wrap-Up

This chapter presented one of C++'s most powerful features—templates. You learned how to use function templates to enable the compiler to produce a set of function-template specializations that represent a group of related overloaded functions. We also discussed how to overload a function template to create a specialized version of a function that handles a particular data type's processing in a manner that differs from the other function-template specializations. Next, you learned about class templates and class-template specializations. You saw examples of how to use a class template to create a group of related types that each perform identical processing on different data types. In the next chapter, we discuss many of C++'s I/O capabilities and demonstrate several stream manipulators that perform various formatting tasks.

Summary

Section 14.1 Introduction

- Templates (p. 580) enable us to specify a range of related (overloaded) functions—called function-template specializations—or a range of related classes—called class-template specializations.

Section 14.2 Function Templates

- To use function-template specializations (p. 580), you write a single function-template definition (p. 581). Based on the argument types provided in calls to this function, C++ generates separate specializations to handle each type of call appropriately.
- All function-template definitions begin with the keyword `template` (p. 581) followed by template parameters (p. 581) enclosed in angle brackets (`<` and `>`); each template parameter that represents a type must be preceded by keyword `class` or `typename` (p. 581). Keywords `typename` and `class` used to specify function-template parameters mean "any fundamental type or user-defined type."
- Template-definition template parameters are used to specify the kinds of arguments to the function, the return type of the function and to declare variables in the function.
- As with function parameters, the names of template parameters must be unique inside a template definition. Template parameter names need not be unique across different function templates.

Section 14.3 Overloading Function Templates

- A function template may be overloaded (p. 583) in several ways. We can provide other function templates that specify the same function name but different function parameters. A function template can also be overloaded by providing other nontemplate functions with the same function name, but different function parameters. If both the template and non-template versions match a call, the non-template version will be used.

Section 14.4 Class Templates

- Class templates provide the means for describing a class generically and for instantiating classes that are type-specific versions of this generic class.
- Class templates are called parameterized types (p. 584); they require type parameters to specify how to customize a generic class template to form a specific class-template specialization.
- To use class-template specializations you write one class template. When you need a new type-specific class, the compiler writes the source code for the class-template specialization.
- A class-template definition (p. 584) looks like a conventional class definition, but it's preceded by `template<typename T>` (or `template<class T>`) to indicate this is a class-template definition. Type parameter `T` acts as a placeholder for the type of the class to create. The type `T` is mentioned throughout the class definition and member-function definitions as a generic type name.

- Member-function definitions outside a class template each begin with `template<typename T>` (or `template<class T>`). Then, each function definition resembles a conventional function definition, except that the generic data in the class always is listed generically as type parameter `T`. The binary scope-resolution operator is used with the class-template name to tie each member-function definition to the class template's scope.

Section 14.5 Nontype Parameters and Default Types for Class Templates

- It's possible to use nontype parameters (p. 590) in the header of a class or function template.
- You can specify a default type (p. 591) for a type parameter in the type-parameter list.
- An explicit specialization (p. 591) of a class template overrides a class template for a specific type.

Self-Review Exercises

14.1 State which of the following are *true* and which are *false*. If *false*, explain why.

a) The template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.
b) Keywords `typename` and `class` as used with a template type parameter specifically mean "any user-defined class type."
c) A function template can be overloaded by another function template with the same function name.
d) Template parameter names among template definitions must be unique.
e) Each member-function definition outside a class template must begin with a template header.

14.2 Fill in the blanks in each of the following:

a) Templates enable us to specify, with a single code segment, an entire range of related functions called ________, or an entire range of related classes called ________.
b) All function-template definitions begin with the keyword ________, followed by a list of template parameters to the function template enclosed in ________.
c) The related functions generated from a function template all have the same name, so the compiler uses ________ resolution to invoke the proper function.
d) Class templates also are called ________ types.
e) The ________ operator is used with a class-template name to tie each member-function definition to the class template's scope.

Answers to Self-Review Exercises

14.1 a) True. b) False. Keywords `typename` and `class` in this context also allow for a type parameter of a fundamental type. c) True. d) False. Template parameter names among function templates need not be unique. e) True.

14.2 a) function-template specializations, class-template specializations. b) `template`, angle brackets (< and >). c) overload. d) parameterized. e) scope resolution.

Exercises

14.3 *(Selection Sort Function Template)* Write a function template `selectionSort` based on Fig. 8.13. Write a driver program that inputs, sorts and outputs an `int` array and a `float` array.

14.4 *(Print Array Range)* Overload function template `printArray` of Fig. 14.1 so that it takes two additional integer arguments, namely `int lowSubscript` and `int highSubscript`. A call to this function will print only the designated portion of the array. Validate `lowSubscript` and `highSub-`

script; if either is out of range or if highSubscript is less than or equal to lowSubscript, the overloaded printArray function should return 0; otherwise, printArray should return the number of elements printed. Then modify main to exercise both versions of printArray on arrays a, b and c (lines 22–24 of Fig. 14.1). Be sure to test all capabilities of both versions of printArray.

14.5 *(Function Template Overloading)* Overload function template printArray of Fig. 14.1 with a nontemplate version that prints an array of character strings in neat, tabular, column format.

14.6 *(Operator Overloads in Templates)* Write a simple function template for predicate function isEqualTo that compares its two arguments of the same type with the equality operator (==) and returns true if they are equal and false otherwise. Use this function template in a program that calls isEqualTo only with a variety of fundamental types. Now write a separate version of the program that calls isEqualTo with a user-defined class type, but does not overload the equality operator. What happens when you attempt to run this program? Now overload the equality operator (with the operator function) operator==. Now what happens when you attempt to run this program?

14.7 *(Array Class Template)* Reimplement class Array from Figs. 11.10–11.11 as a class template. Demonstrate the new Array class template in a program.

14.8 Distinguish between the terms "function template" and "function-template specialization."

14.9 Explain which is more like a stencil—a class template or a class-template specialization?

14.10 What's the relationship between function templates and overloading?

14.11 Why might you choose to use a function template instead of a macro?

14.12 What performance problem can result from using function templates and class templates?

14.13 The compiler performs a matching process to determine which function-template specialization to call when a function is invoked. Under what circumstances does an attempt to make a match result in a compile error?

14.14 Why is it appropriate to refer to a class template as a parameterized type?

14.15 Explain why a C++ program would use the statement

```
Array< Employee > workerList( 100 );
```

14.16 Review your answer to Exercise 14.15. Explain why a C++ program might use the statement

```
Array< Employee > workerList;
```

14.17 Explain the use of the following notation in a C++ program:

```
template< typename T > Array< T >::Array( int s )
```

14.18 Why might you use a nontype parameter with a class template for a container such as an array or stack?

15

Stream Input/Output

Consciousness … does not appear to itself chopped up in bits … A "river" or a "stream" are the metaphors by which it is most naturally described.

—William James

Objectives

In this chapter you'll learn:

- To use C++ object-oriented stream input/output.
- To format input and output.
- The stream-I/O class hierarchy.
- To use stream manipulators.
- To control justification and padding.
- To determine the success or failure of input/output operations.
- To tie output streams to input streams.

Outline

15.1 Introduction

The C++ standard libraries provide an extensive set of input/output capabilities. This chapter discusses a range of capabilities sufficient for performing most common I/O operations and overviews the remaining capabilities. We discussed some of these features earlier in the text; now we provide a more complete treatment. Many of the I/O features that we'll discuss are object oriented. This style of I/O makes use of other C++ features, such as references, function overloading and operator overloading.

C++ uses **type-safe I/O**. Each I/O operation is executed in a manner sensitive to the data type. If an I/O function has been defined to handle a particular data type, then that member function is called to handle that data type. If there is no match between the type of the actual data and a function for handling that data type, the compiler generates an error. Thus, improper data cannot "sneak" through the system (as can occur in C, allowing for some subtle and bizarre errors).

Users can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>). This **extensibility** is one of C++'s most valuable features.

Software Engineering Observation 15.1

Use the C++-style I/O exclusively in C++ programs, even though C-style I/O is available to C++ programmers.

Error-Prevention Tip 15.1

C++ I/O is type safe.

Software Engineering Observation 15.2

C++ enables a common treatment of I/O for predefined types and user-defined types. This commonality facilitates software development and reuse.

15.2 Streams

C++ I/O occurs in **streams**, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).

An application associates meaning with bytes. The bytes could represent characters, raw data, graphics images, digital speech, digital video or any other information an application may require. The system I/O mechanisms should transfer bytes from devices to memory (and vice versa) consistently and reliably. Such transfers often involve some mechanical motion, such as the rotation of a disk or a tape, or the typing of keystrokes at a keyboard. The time these transfers take typically is much greater than the time the processor requires to manipulate data internally. Thus, I/O operations require careful planning and tuning to ensure optimal performance.

C++ provides both “low-level” and “high-level” I/O capabilities. Low-level I/O capabilities (i.e., **unformatted I/O**) specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient.

Programmers generally prefer a higher-level view of I/O (i.e., **formatted I/O**), in which bytes are grouped into meaningful units, such as integers, floating-point numbers, characters, strings and user-defined types. These type-oriented capabilities are satisfactory for most I/O other than high-volume file processing.

Performance Tip 15.1

Use unformatted I/O for the best performance in high-volume file processing.

Portability Tip 15.1

Using unformatted I/O can lead to portability problems, because unformatted data is not portable across all platforms.

15.2.1 Classic Streams vs. Standard Streams

In the past, the C++ **classic stream libraries** enabled input and output of `char`s. Because a `char` normally occupies one byte, it can represent only a limited set of characters (such as those in the ASCII character set used by most readers of this book, or other popular character sets). However, many languages use alphabets that contain more characters than a single-byte `char` can represent. The ASCII character set does not provide these characters; the **Unicode® character set** does. Unicode is an extensive international character set that represents the majority of the world's "commercially viable" languages, mathematical symbols and much more. For more information on Unicode, visit www.unicode.org.

C++ includes the **standard stream libraries**, which enable developers to build systems capable of performing I/O operations with Unicode characters. For this purpose, C++ includes an additional character type called **wchar_t**, which among other uses can store Unicode characters. The C++ standard also redesigned the classic C++ stream classes, which processed only chars, as class templates with separate specializations for processing characters of types char and wchar_t, respectively. We use the char type of class templates throughout this book.

15.2.2 iostream Library Headers

The C++ iostream library provides hundreds of I/O capabilities. Several headers contain portions of the library interface.

Most C++ programs include the <iostream> header, which declares basic services required for all stream-I/O operations. The <iostream> header defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. (cerr and clog are discussed in Section 15.2.3.) Both unformatted- and formatted-I/O services are provided.

The <iomanip> header declares services useful for performing formatted I/O with so-called **parameterized stream manipulators**, such as setw and setprecision.

The <fstream> header declares services for file processing. We use this header in the file-processing programs of Chapter 17.

C++ implementations generally contain other I/O-related libraries that provide system-specific capabilities, such as the controlling of special-purpose devices for audio and video I/O.

15.2.3 Stream Input/Output Classes and Objects

The iostream library provides many templates for handling common I/O operations. For example, class template **basic_istream** supports stream-input operations, class template **basic_ostream** supports stream-output operations, and class template **basic_iostream** supports both stream-input and stream-output operations. Each template has a predefined template specialization that enables char I/O. In addition, the iostream library provides a set of typedefs that provide aliases for these template specializations. The **typedef** specifier declares synonyms (aliases) for data types. You'll sometimes use typedef to create shorter or more readable type names. For example, the statement

```
typedef Card *CardPtr;
```

defines an additional type name, CardPtr, as a synonym for type Card *. Creating a name using typedef does not create a data type; typedef creates only a type name. Section 21.5 discusses typedef in detail. The typedef **istream** represents a specialization of basic_istream that enables char input. Similarly, the typedef **ostream** represents a specialization of basic_ostream that enables char output. Also, the typedef **iostream** represents a specialization of basic_iostream that enables both char input and output. We use these typedefs throughout this chapter.

Stream-I/O Template Hierarchy and Operator Overloading

Templates `basic_istream` and `basic_ostream` both derive through single inheritance from base template `basic_ios`.[1] Template `basic_iostream` derives through multiple inheritance[2] from templates `basic_istream` and `basic_ostream`. The UML class diagram of Fig. 15.1 summarizes these inheritance relationships.

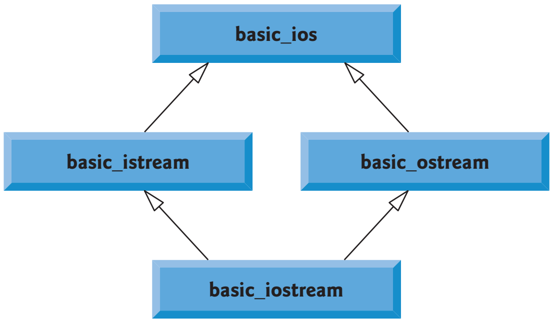


Fig. 15.1 | Stream-I/O template hierarchy portion.

Operator overloading provides a convenient notation for performing input/output. The left-shift operator (<<) is overloaded to designate stream output and is referred to as the stream insertion operator. The right-shift operator (>>) is overloaded to designate stream input and is referred to as the stream extraction operator. These operators are used with the standard stream objects `cin`, `cout`, `cerr` and `clog` and, commonly, with user-defined stream objects.

Standard Stream Objects `cin`*,* `cout`*,* `cerr` *and* `clog`

Predefined object `cin` is an `istream` instance and is said to be "connected to" (or attached to) the standard input device, which usually is the keyboard. The stream extraction operator (>>) as used in the following statement causes a value for integer variable `grade` (assuming that `grade` has been declared as an `int` variable) to be input from `cin` to memory:

```
cin >> grade; // data "flows" in the direction of the arrows
```

The compiler determines the data type of `grade` and selects the appropriate overloaded stream extraction operator. Assuming that `grade` has been declared properly, the stream extraction operator does not require additional type information (as is the case, for example, in C-style I/O). The >> operator is overloaded to input data items of fundamental types, strings and pointer values.

The predefined object `cout` is an `ostream` instance and is said to be "connected to" the standard output device, which usually is the display screen. The stream insertion operator (<<), as used in the following statement, causes the value of variable `grade` to be output from memory to the standard output device:

```
cout << grade; // data "flows" in the direction of the arrows
```

1. This chapter discusses templates only in the context of the template specializations for `char` I/O.
2. Multiple inheritance is discussed in Chapter 24, Other Topics.

The compiler determines the data type of grade (assuming grade has been declared properly) and selects the appropriate stream insertion operator. The << operator is overloaded to output data items of fundamental types, strings and pointer values.

The predefined object cerr is an ostream instance and is said to be "connected to" the standard error device, normally the screen. Outputs to object cerr are **unbuffered**, implying that each stream insertion to cerr causes its output to appear immediately—this is appropriate for notifying a user promptly about errors.

The predefined object clog is an instance of the ostream class and is said to be "connected to" the standard error device. Outputs to clog are **buffered**. This means that each insertion to clog could cause its output to be held in a buffer (that is, an area in memory) until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique discussed in operating-systems courses.

File-Processing Templates

C++ file processing uses class templates **basic_ifstream** (for file input), **basic_ofstream** (for file output) and **basic_fstream** (for file input and output). Each class template has a predefined template specialization that enables char I/O. C++ provides a set of typedefs that provide aliases for these template specializations. For example, the typedef **ifstream** represents a specialization of basic_ifstream that enables char input from a file. Similarly, typedef **ofstream** represents a specialization of basic_ofstream that enables char output to a file. Also, typedef **fstream** represents a specialization of basic_fstream that enables char input from, and output to, a file. Template basic_ifstream inherits from basic_istream, basic_ofstream inherits from basic_ostream and basic_fstream inherits from basic_iostream. The UML class diagram of Fig. 15.2 summarizes the various inheritance relationships of the I/O-related classes. The full stream-I/O class hierarchy provides most of the capabilities that you need. Consult the class-library reference for your C++ system for additional file-processing information.

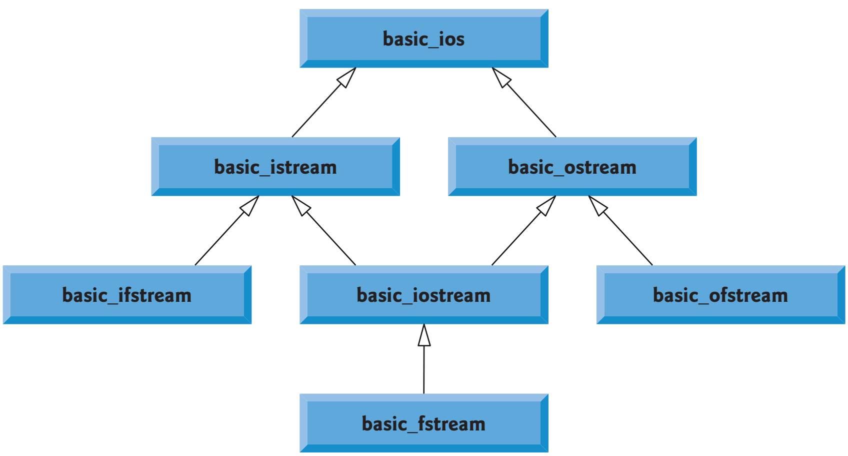


Fig. 15.2 | Stream-I/O template hierarchy portion showing the main file-processing templates.

15.3 Stream Output

Formatted and unformatted output capabilities are provided by `ostream`. Capabilities include output of standard data types with the stream insertion operator (<<); output of characters via the `put` member function; unformatted output via the `write` member function (Section 15.5); output of integers in decimal, octal and hexadecimal formats (Section 15.6.1); output of floating-point values with various precision (Section 15.6.2), with forced decimal points (Section 15.7.1), in scientific notation and in fixed notation (Section 15.7.5); output of data justified in fields of designated widths (Section 15.7.2); output of data in fields padded with specified characters (Section 15.7.3); and output of uppercase letters in scientific notation and hexadecimal notation (Section 15.7.6).

15.3.1 Output of char * Variables

C++ determines data types automatically—an improvement over C. This feature sometimes "gets in the way." For example, suppose we want to print the address stored in a `char *` pointer. The << operator has been overloaded to output a `char *` as a *null-terminated string*. To output the *address*, you can cast the `char *` to a `void *` (this can be done to any pointer variable). Figure 15.3 demonstrates printing a `char *` variable in both string and address formats. The address prints here as a hexadecimal (base-16) number—in general, the way addresses print is implementation dependent. To learn more about hexadecimal numbers, read Appendix D. We say more about controlling the bases of numbers in Section 15.6.1 and Section 15.7.4.

```cpp
// Fig. 15.3: Fig15_03.cpp
// Printing the address stored in a char * variable.
#include <iostream>
using namespace std;

int main()
{
   const char *const word = "again";

   // display value of char *, then display value of char *
   // static_cast to void *
   cout << "Value of word is: " << word << endl
      << "Value of static_cast< void * >( word ) is: "
      << static_cast< void * >( word ) << endl;
} // end main
```

```
Value of word is: again
Value of static_cast< void * >( word ) is: 00428300
```

Fig. 15.3 | Printing the address stored in a `char *` variable.

15.3.2 Character Output Using Member Function put

We can use the `put` member function to output characters. For example, the statement

```cpp
cout.put( 'A' );
```

displays a single character `A`. Calls to `put` may be cascaded, as in the statement

```
cout.put( 'A' ).put( '\n' );
```

which outputs the letter `A` followed by a newline character. As with `<<`, the preceding statement executes in this manner, because the dot operator (`.`) associates from left to right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call. The `put` function also may be called with a numeric expression that represents an ASCII value, as in the following statement, which also outputs `A`:

```
cout.put( 65 );
```

15.4 Stream Input

Now let's consider stream input. Formatted and unformatted input capabilities are provided by `istream`. The stream extraction operator (>>) normally skips **white-space characters** (such as blanks, tabs and newlines) in the input stream; later we'll see how to change this behavior. After each input, the stream extraction operator returns a *reference* to the stream object that received the extraction message (e.g., `cin` in the expression `cin >> grade`). If that reference is used as a condition (e.g., in a `while` statement's loop-continuation condition), the stream's overloaded `void *` cast operator function is implicitly invoked to convert the reference into a non-null pointer value or the null pointer based on the success or failure of the last input operation. A non-null pointer converts to the `bool` value `true` to indicate success and the null pointer converts to the `bool` value `false` to indicate failure. When an attempt is made to read past the end of a stream, the stream's overloaded `void *` cast operator returns the null pointer to indicate end-of-file.

Each stream object contains a set of **state bits** used to control the stream's state (i.e., formatting, setting error states, etc.). These bits are used by the stream's overloaded `void *` cast operator to determine whether to return a non-null pointer or the null pointer. Stream extraction causes the stream's **`failbit`** to be set if data of the wrong type is input and causes the stream's **`badbit`** to be set if the operation fails. Section 15.7 and Section 15.8 discuss stream state bits in detail, then show how to test these bits after an I/O operation.

15.4.1 `get` and `getline` Member Functions

The **`get`** member function with no arguments inputs one character from the designated stream (including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file) and returns it as the value of the function call. This version of `get` returns `EOF` when end-of-file is encountered on the stream.

Using Member Functions `eof`*,* `get` *and* `put`

Figure 15.4 demonstrates the use of member functions `eof` and `get` on input stream `cin` and member function `put` on output stream `cout`. The program first prints the value of `cin.eof()`—i.e., `false` (`0` on the output)—to show that end-of-file has not occurred on `cin`. The user enters a line of text and presses *Enter* followed by end-of-file (*<Ctrl>-z* on Microsoft Windows systems, *<Ctrl>-d* on UNIX and Macintosh systems). Line 15 reads each character, which line 16 outputs to `cout` using member function `put`. When end-of-file is encountered, the `while` statement ends, and line 20 displays the value of `cin.eof()`, which is now `true` (`1` on the output), to show that end-of-file has been set on `cin`. This program uses the version of `istream` member function `get` that takes no arguments and returns the character being input (line 15). Function `eof` returns `true` only after the program attempts to read past the last character in the stream.

The `get` member function with a character-reference argument inputs the next character from the input stream (even if this is a white-space character) and stores it in the character argument. This version of `get` returns a reference to the `istream` object for which the `get` member function is being invoked.

A third version of `get` takes three arguments—a character array, a size limit and a delimiter (with default value `'\n'`). This version reads characters from the input stream. It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array used as a buffer by the program. The delimiter is not placed in the character array but *does remain in the input stream* (the delimiter will be the next character read). Thus, the result of a second consecutive `get` is an empty line, unless the delimiter character is removed from the input stream (possibly with `cin.ignore()`).

```
1  // Fig. 15.4: Fig15_04.cpp
2  // Using member functions get, put and eof.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int character; // use int, because char cannot represent EOF
9
10    // prompt user to enter line of text
11    cout << "Before input, cin.eof() is " << cin.eof() << endl
12       << "Enter a sentence followed by end-of-file:" << endl;
13
14    // use get to read each character; use put to display it
15    while ( ( character = cin.get() ) != EOF )
16       cout.put( character );
17
18    // display end-of-file character
19    cout << "\nEOF in this system is: " << character << endl;
20    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
21 } // end main
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF in this system is: -1
After input of EOF, cin.eof() is 1
```

Fig. 15.4 | `get`, `put` and `eof` member functions.

Comparing `cin` *and* `cin.get`

Figure 15.5 compares input using stream extraction with `cin` (which reads characters until a white-space character is encountered) and input using `cin.get`. The call to `cin.get` (line 22) does not specify a delimiter, so the default `'\n'` character is used.

```
// Fig. 15.5: Fig15_05.cpp
// Contrasting input of a string via cin and cin.get.
#include <iostream>
using namespace std;

int main()
{
   // create two char arrays, each with 80 elements
   const int SIZE = 80;
   char buffer1[ SIZE ];
   char buffer2[ SIZE ];

   // use cin to input characters into buffer1
   cout << "Enter a sentence:" << endl;
   cin >> buffer1;

   // display buffer1 contents
   cout << "\nThe string read with cin was:" << endl
      << buffer1 << endl << endl;

   // use cin.get to input characters into buffer2
   cin.get( buffer2, SIZE );

   // display buffer2 contents
   cout << "The string read with cin.get was:" << endl
      << buffer2 << endl;
} // end main
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

Fig. 15.5 | Contrasting input of a string via `cin` and `cin.get`.

Using Member Function `getline`

Member function **`getline`** operates similarly to the third version of the `get` member function and inserts a null character after the line in the character array. The `getline` function removes the delimiter from the stream (i.e., reads the character and discards it), but does not store it in the character array. The program of Fig. 15.6 demonstrates the use of the `getline` member function to input a line of text (line 13).

```
1  // Fig. 15.6: Fig15_06.cpp
2  // Inputting characters using cin member function getline.
3  #include <iostream>
4  using namespace std;
5
```

Fig. 15.6 | Inputting characters with `cin` member function `getline`. (Part 1 of 2.)

```
int main()
{
   const int SIZE = 80;
   char buffer[ SIZE ]; // create array of 80 characters

   // input characters in buffer via cin function getline
   cout << "Enter a sentence:" << endl;
   cin.getline( buffer, SIZE );

   // display buffer contents
   cout << "\nThe sentence entered is:" << endl << buffer << endl;
} // end main
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Fig. 15.6 | Inputting characters with `cin` member function `getline`. (Part 2 of 2.)

15.4.2 istream Member Functions peek, putback and ignore

The **`ignore`** member function of `istream` reads and discards a designated number of characters (the default is one) or terminates upon encountering a designated delimiter (the default is `EOF`, which causes `ignore` to skip to the end of the file when reading from a file).

The **`putback`** member function places the previous character obtained by a `get` from an input stream back into that stream. This function is useful for applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream, so the character can be included in the input data.

The **`peek`** member function returns the next character from an input stream but does not remove the character from the stream.

15.4.3 Type-Safe I/O

C++ offers *type-safe I/O*. The << and >> operators are overloaded to accept data items of specific types. If unexpected data is processed, various error bits are set, which the user may test to determine whether an I/O operation succeeded or failed. If operators << and >> have not been overloaded for a user-defined type and you attempt to input into or output the contents of an object of that user-defined type, the compiler reports an error. This enables the program to "stay in control." We discuss these error states in Section 15.8.

15.5 Unformatted I/O Using read, write and gcount

Unformatted input/output is performed using the **`read`** and **`write`** member functions of `istream` and `ostream`, respectively. Member function `read` inputs bytes to a character array in memory; member function `write` outputs bytes from a character array. These bytes are not formatted in any way. They're input or output as raw bytes. For example, the call

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

outputs the first 10 bytes of `buffer` (including null characters, if any, that would cause output with `cout` and `<<` to terminate). The call

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

displays the first 10 characters of the alphabet.

The `read` member function inputs a designated number of characters into a character array. If fewer than the designated number of characters are read, `failbit` is set. Section 15.8 shows how to determine whether `failbit` has been set. Member function **`gcount`** reports the number of characters read by the last input operation.

Figure 15.7 demonstrates `istream` member functions `read` and `gcount`, and `ostream` member function `write`. The program inputs 20 characters (from a longer input sequence) into the array `buffer` with `read` (line 13), determines the number of characters input with `gcount` (line 17) and outputs the characters in `buffer` with `write` (line 17).

```
1  // Fig. 15.7: Fig15_07.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 80;
9     char buffer[ SIZE ]; // create array of 80 characters
10
11    // use function read to input characters into buffer
12    cout << "Enter a sentence:" << endl;
13    cin.read( buffer, 20 );
14
15    // use functions write and gcount to display buffer characters
16    cout << endl << "The sentence entered was:" << endl;
17    cout.write( buffer, cin.gcount() );
18    cout << endl;
19 } // end main
```

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence entered was:
Using the read, writ
```

Fig. 15.7 | Unformatted I/O using the `read`, `gcount` and `write` member functions.

15.6 Introduction to Stream Manipulators

C++ provides various **stream manipulators** that perform formatting tasks. The stream manipulators provide capabilities such as setting field widths, setting precision, setting and unsetting format state, setting the fill character in fields, flushing streams, inserting a newline into the output stream (and flushing the stream), inserting a null character into the output stream and skipping white space in the input stream. These features are described in the following sections.

15.6.1 Integral Stream Base: dec, oct, hex and setbase

Integers are interpreted normally as decimal (base-10) values. To change the base in which integers are interpreted on a stream, insert the **hex** manipulator to set the base to hexadecimal (base 16) or insert the **oct** manipulator to set the base to octal (base 8). Insert the **dec** manipulator to reset the stream base to decimal. These are all sticky manipulators.

A stream's base also may be changed by the **setbase** stream manipulator, which takes an `int` argument of `10`, `8`, or `16` to set the base to decimal, octal or hexadecimal, respectively. Because `setbase` takes an argument, it's called a parameterized stream manipulator. Using `setbase` (or any other parameterized manipulator) requires the inclusion of the `<iomanip>` header. The stream base value remains the same until changed explicitly; `setbase` settings are "sticky." Figure 15.8 demonstrates stream manipulators `hex`, `oct`, `dec` and `setbase`.

```cpp
// Fig. 15.8: Fig15_08.cpp
// Using stream manipulators hex, oct, dec and setbase.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int number;

   cout << "Enter a decimal number: ";
   cin >> number; // input number

   // use hex stream manipulator to show hexadecimal number
   cout << number << " in hexadecimal is: " << hex
      << number << endl;

   // use oct stream manipulator to show octal number
   cout << dec << number << " in octal is: "
      << oct << number << endl;

   // use setbase stream manipulator to show decimal number
   cout << setbase( 10 ) << number << " in decimal is: "
      << number << endl;
} // end main
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 15.8 | Stream manipulators `hex`, `oct`, `dec` and `setbase`.

15.6.2 Floating-Point Precision (precision, setprecision)

We can control the **precision** of floating-point numbers (i.e., the number of digits to the right of the decimal point) by using either the `setprecision` stream manipulator or the **precision** member function of `ios_base`. A call to either of these sets the precision for all subsequent output operations until the next precision-setting call. A call to member function `precision` with no argument returns the current precision setting (this is what you need to use so that you can restore the original precision eventually after a "sticky" setting is no longer needed). The program of Fig. 15.9 uses both member function `precision` (line 22) and the `setprecision` manipulator (line 31) to print a table that shows the square root of 2, with precision varying from 0 to 9.

```
1  // Fig. 15.9: Fig15_09.cpp
2  // Controlling precision of floating-point values.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>
6  using namespace std;
7
8  int main()
9  {
10    double root2 = sqrt( 2.0 ); // calculate square root of 2
11    int places; // precision, vary from 0-9
12
13    cout << "Square root of 2 with precisions 0-9." << endl
14       << "Precision set by ios_base member function "
15       << "precision:" << endl;
16
17    cout << fixed; // use fixed-point notation
18
19    // display square root using ios_base function precision
20    for ( places = 0; places <= 9; ++places )
21    {
22       cout.precision( places );
23       cout << root2 << endl;
24    } // end for
25
26    cout << "\nPrecision set by stream manipulator "
27       << "setprecision:" << endl;
28
29    // set precision for each digit, then display square root
30    for ( places = 0; places <= 9; ++places )
31       cout << setprecision( places ) << root2 << endl;
32 } // end main
```

```
Square root of 2 with precisions 0-9.
Precision set by ios_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 15.9 | Precision of floating-point values. (Part 1 of 2.)

```
Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 15.9 | Precision of floating-point values. (Part 2 of 2.)

15.6.3 Field Width (`width`, `setw`)

The **`width`** member function (of base class `ios_base`) sets the field width (i.e., the number of character positions in which a value should be output or the maximum number of characters that should be input) and returns the previous width. If values output are narrower than the field width, **fill characters** are inserted as **padding**. A value wider than the designated width will not be truncated—the full number will be printed. The `width` function with no argument returns the current setting.

Common Programming Error 15.1

The width setting applies only for the next insertion or extraction (i.e., the width setting is not "sticky"); afterward, the width is set implicitly to `0` (that is, input and output will be performed with default settings*). Assuming that the width setting applies to all subsequent outputs is a logic error.*

Common Programming Error 15.2

When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs.

Figure 15.10 demonstrates the use of the `width` member function on both input and output. On input into a `char` array, *a maximum of one fewer characters than the width will be read*, because provision is made for the null character to be placed in the input string. Remember that stream extraction terminates when nonleading white space is encountered. The `setw` stream manipulator also may be used to set the field width. [*Note:* When prompted for input in Fig. 15.10, the user should enter a line of text and press *Enter* followed by end-of-file (*<Ctrl>-z* on Microsoft Windows systems and *<Ctrl>-d* on UNIX and Macintosh systems).]

```
// Fig. 15.10: Fig15_10.cpp
// Demonstrating member function width.
#include <iostream>
using namespace std;
```

Fig. 15.10 | `width` member function of class `ios_base`. (Part 1 of 2.)

```

int main()
{
   int widthValue = 4;
   char sentence[ 10 ];

   cout << "Enter a sentence:" << endl;
   cin.width( 5 ); // input only 5 characters from sentence

   // set field width, then display characters based on that width
   while ( cin >> sentence )
   {
      cout.width( widthValue++ );
      cout << sentence << endl;
      cin.width( 5 ); // input 5 more characters from sentence
   } // end while
} // end main
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

Fig. 15.10 | `width` member function of class `ios_base`. (Part 2 of 2.)

15.6.4 User-Defined Output Stream Manipulators

You can create your own stream manipulators.[3] Figure 15.11 shows the creation and use of new nonparameterized stream manipulators `bell` (lines 8–11), `carriageReturn` (lines 14–17), `tab` (lines 20–23) and `endLine` (lines 27–30). For output stream manipulators, the return type and parameter must be of type `ostream &`. When line 35 inserts the `endLine` manipulator in the output stream, function `endLine` is called and line 29 outputs the escape sequence `\n` and the `flush` manipulator to the standard output stream `cout`. Similarly, when lines 35–44 insert the manipulators `tab`, `bell` and `carriageReturn` in the output stream, their corresponding functions—`tab` (line 20), `bell` (line 8) and `carriage-Return` (line 14) are called, which in turn output various escape sequences.

3. You can also create your own parameterized stream manipulators. This capability is beyond the scope of this book.

```cpp
// Fig. 15.11: Fig15_11.cpp
// Creating and testing user-defined, nonparameterized
// stream manipulators.
#include <iostream>
using namespace std;

// bell manipulator (using escape sequence \a)
ostream& bell( ostream& output )
{
   return output << '\a'; // issue system beep
} // end bell manipulator

// carriageReturn manipulator (using escape sequence \r)
ostream& carriageReturn( ostream& output )
{
   return output << '\r'; // issue carriage return
} // end carriageReturn manipulator

// tab manipulator (using escape sequence \t)
ostream& tab( ostream& output )
{
   return output << '\t'; // issue tab
} // end tab manipulator

// endLine manipulator (using escape sequence \n and member
// function flush)
ostream& endLine( ostream& output )
{
   return output << '\n' << flush; // issue endl-like end of line
} // end endLine manipulator

int main()
{
   // use tab and endLine manipulators
   cout << "Testing the tab manipulator:" << endLine
      << 'a' << tab << 'b' << tab << 'c' << endLine;

   cout << "Testing the carriageReturn and bell manipulators:"
      << endLine << "..........";

   cout << bell; // use bell manipulator

   // use carriageReturn and endLine manipulators
   cout << carriageReturn << "-----" << endLine;
} // end main
```

```
Testing the tab manipulator:
a       b       c
Testing the carriageReturn and bell manipulators:
-----.....
```

Fig. 15.11 | User-defined, nonparameterized stream manipulators.

15.7 Stream Format States and Stream Manipulators

Various stream manipulators can be used to specify the kinds of formatting to be performed during stream-I/O operations. Stream manipulators control the output's format settings. Figure 15.12 lists each stream manipulator that controls a given stream's format state. All these manipulators belong to class `ios_base`. We show examples of most of these stream manipulators in the next several sections.

Manipulator	Description
`skipws`	Skip white-space characters on an input stream. This setting is reset with stream manipulator `noskipws`.
`left`	Left justify output in a field. Padding characters appear to the right if necessary.
`right`	Right justify output in a field. Padding characters appear to the left if necessary.
`internal`	Indicate that a number's sign should be left justified in a field and a number's magnitude should be right justified in that same field (i.e., padding characters appear between the sign and the number).
`boolalpha`	Specify that bool values should be displayed as the word `true` or `false`. The manipulator noboolalpha sets the stream back to displaying bool values as 1 (true) and 0 (false).
`dec`	Specify that integers should be treated as decimal (base 10) values.
`oct`	Specify that integers should be treated as octal (base 8) values.
`hex`	Specify that integers should be treated as hexadecimal (base 16) values.
`showbase`	Specify that the base of a number is to be output ahead of the number (a leading `0` for octals; a leading `0x` or `0X` for hexadecimals). This setting is reset with stream manipulator `noshowbase`.
`showpoint`	Specify that floating-point numbers should be output with a decimal point. This is used normally with `fixed` to guarantee a certain number of digits to the right of the decimal point, even if they're zeros. This setting is reset with stream manipulator `noshowpoint`.
`uppercase`	Specify that uppercase letters (i.e., `X` and `A` through `F`) should be used in a hexadecimal integer and that uppercase `E` should be used when representing a floating-point value in scientific notation. This setting is reset with stream manipulator `nouppercase`.
`showpos`	Specify that positive numbers should be preceded by a plus sign (`+`). This setting is reset with stream manipulator `noshowpos`.
`scientific`	Specify output of a floating-point value in scientific notation.
`fixed`	Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.

Fig. 15.12 | Format state stream manipulators from `<iostream>`.

15.7.1 Trailing Zeros and Decimal Points (showpoint)

Stream manipulator **`showpoint`** forces a floating-point number to be output with its decimal point and trailing zeros. For example, the floating-point value `79.0` prints as `79` without using `showpoint` and prints as `79.000000` (or as many trailing zeros as are specified by the current precision) using `showpoint`. To reset the `showpoint` setting, output the stream manipulator **`noshowpoint`**. The program in Fig. 15.13 shows how to use stream manipulator `showpoint` to control the printing of trailing zeros and decimal points for floating-point values. Recall that the default precision of a floating-point number is 6. When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after decimal point.

```cpp
// Fig. 15.13: Fig15_13.cpp
// Controlling the printing of trailing zeros and
// decimal points in floating-point values.
#include <iostream>
using namespace std;

int main()
{
   // display double values with default stream format
   cout << "Before using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl << endl;

   // display double value after showpoint
   cout << showpoint
      << "After using showpoint" << endl
      << "9.9900 prints as: " << 9.9900 << endl
      << "9.9000 prints as: " << 9.9000 << endl
      << "9.0000 prints as: " << 9.0000 << endl;
} // end main
```

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 15.13 | Controlling the printing of trailing zeros and decimal points in floating-point values.

15.7.2 Justification (`left`, `right` and `internal`)

Stream manipulators **`left`** and **`right`** enable fields to be left justified with padding characters to the right or right justified with padding characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 15.7.3). Figure 15.14 uses the `setw`, `left` and `right` manipulators to left justify and right justify integer data in a field.

```
// Fig. 15.14: Fig15_14.cpp
// Left and right justification with stream manipulators left and right.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int x = 12345;

   // display x right justified (default)
   cout << "Default is right justified:" << endl
      << setw( 10 ) << x;

   // use left manipulator to display x left justified
   cout << "\n\nUse std::left to left justify x:\n"
      << left << setw( 10 ) << x;

   // use right manipulator to display x right justified
   cout << "\n\nUse std::right to right justify x:\n"
      << right << setw( 10 ) << x << endl;
} // end main
```

```
Default is right justified:
     12345

Use std::left to left justify x:
12345

Use std::right to right justify x:
     12345
```

Fig. 15.14 | Left justification and right justification with stream manipulators `left` and `right`.

Stream manipulator **`internal`** indicates that a number's sign (or base when using stream manipulator `showbase`) should be left justified within a field, that the number's magnitude should be right justified and that intervening spaces should be padded with the fill character. Figure 15.15 shows the `internal` stream manipulator specifying internal spacing (line 10). Note that **`showpos`** forces the plus sign to print (line 10). To reset the `showpos` setting, output the stream manipulator **`noshowpos`**.

```
1  // Fig. 15.15: Fig15_15.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // display value with internal spacing and plus sign
10    cout << internal << showpos << setw( 10 ) << 123 << endl;
11 } // end main
```

Fig. 15.15 | Printing an integer with internal spacing and plus sign. (Part 1 of 2.)

```
+      123
```

Fig. 15.15 | Printing an integer with internal spacing and plus sign. (Part 2 of 2.)

15.7.3 Padding (fill, setfill)

The **fill member function** specifies the fill character to be used with justified fields; spaces are used for padding by default. The function returns the prior padding character. The **setfill manipulator** also sets the padding character. Figure 15.16 demonstrates function fill (line 30) and stream manipulator setfill (lines 34 and 37) to set the fill character.

```
1  // Fig. 15.16: Fig15_16.cpp
2  // Using member function fill and stream manipulator setfill to change
3  // the padding character for fields larger than the printed value.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    int x = 10000;
11
12    // display x
13    cout << x << " printed as int right and left justified\n"
14       << "and as hex with internal justification.\n"
15       << "Using the default pad character (space):" << endl;
16
17    // display x with base
18    cout << showbase << setw( 10 ) << x << endl;
19
20    // display x with left justification
21    cout << left << setw( 10 ) << x << endl;
22
23    // display x as hex with internal justification
24    cout << internal << setw( 10 ) << hex << x << endl << endl;
25
26    cout << "Using various padding characters:" << endl;
27
28    // display x using padded characters (right justification)
29    cout << right;
30    cout.fill( '*' );
31    cout << setw( 10 ) << dec << x << endl;
32
33    // display x using padded characters (left justification)
34    cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
35
36    // display x using padded characters (internal justification)
37    cout << internal << setw( 10 ) << setfill( '^' ) << hex
38       << x << endl;
39 } // end main
```

Fig. 15.16 | Using member function fill and stream manipulator setfill to change the padding character for fields larger than the values being printed. (Part 1 of 2.)

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Fig. 15.16 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed. (Part 2 of 2.)

15.7.4 Integral Stream Base (dec, oct, hex, showbase)

C++ provides stream manipulators **`dec`**, **`hex`** and **`oct`** to specify that integers are to be displayed as decimal, hexadecimal and octal values, respectively. Stream insertions default to decimal if none of these manipulators is used. With stream extraction, integers prefixed with `0` (zero) are treated as octal values, integers prefixed with `0x` or `0X` are treated as hexadecimal values, and all other integers are treated as decimal values. Once a particular base is specified for a stream, all integers on that stream are processed using that base until a different base is specified or until the program terminates.

Stream manipulator **`showbase`** forces the base of an integral value to be output. Decimal numbers are output by default, octal numbers are output with a leading `0`, and hexadecimal numbers are output with either a leading `0x` or a leading `0X` (as we discuss in Section 15.7.6, stream manipulator `uppercase` determines which option is chosen). Figure 15.17 demonstrates the use of stream manipulator `showbase` to force an integer to print in decimal, octal and hexadecimal formats. To reset the `showbase` setting, output the stream manipulator **`noshowbase`**.

```
// Fig. 15.17: Fig15_17.cpp
// Using stream manipulator showbase.
#include <iostream>
using namespace std;

int main()
{
   int x = 100;

   // use showbase to show number base
   cout << "Printing integers preceded by their base:" << endl
      << showbase;

   cout << x << endl; // print decimal value
   cout << oct << x << endl; // print octal value
   cout << hex << x << endl; // print hexadecimal value
} // end main
```

Fig. 15.17 | Stream manipulator `showbase`. (Part 1 of 2.)

```
Printing integers preceded by their base:
100
0144
0x64
```

Fig. 15.17 | Stream manipulator `showbase`. (Part 2 of 2.)

15.7.5 Floating-Point Numbers; Scientific and Fixed Notation (scientific, fixed)

Stream manipulators `scientific` and `fixed` control the output format of floating-point numbers. Stream manipulator **`scientific`** forces the output of a floating-point number to display in scientific format. Stream manipulator **`fixed`** forces a floating-point number to display a specific number of digits (as specified by member function `precision` or stream manipulator `setprecision`) to the right of the decimal point. Without using another manipulator, the floating-point-number value determines the output format.

Figure 15.18 demonstrates displaying floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 18) and `fixed` (line 22). The exponent format in scientific notation might differ across different compilers.

```
1  // Fig. 15.18: Fig15_18.cpp
2  // Displaying floating-point values in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     double x = 0.001234567;
10    double y = 1.946e9;
11
12    // display x and y in default format
13    cout << "Displayed in default format:" << endl
14       << x << '\t' << y << endl;
15
16    // display x and y in scientific format
17    cout << "\nDisplayed in scientific format:" << endl
18       << scientific << x << '\t' << y << endl;
19
20    // display x and y in fixed format
21    cout << "\nDisplayed in fixed format:" << endl
22       << fixed << x << '\t' << y << endl;
23 } // end main
```

```
Displayed in default format:
0.00123457      1.946e+009

Displayed in scientific format:
1.234567e-003   1.946000e+009
```

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 1 of 2.)

```
Displayed in fixed format:
0.001235        1946000000.000000
```

Fig. 15.18 | Floating-point values displayed in default, scientific and fixed formats. (Part 2 of 2.)

15.7.6 Uppercase/Lowercase Control (uppercase)

Stream manipulator **uppercase** outputs an uppercase `X` or `E` with hexadecimal-integer values or with scientific notation floating-point values, respectively (Fig. 15.19). Using stream manipulator `uppercase` also causes all letters in a hexadecimal value to be uppercase. By default, the letters for hexadecimal values and the exponents in scientific notation floating-point values appear in lowercase. To reset the `uppercase` setting, output the stream manipulator **nouppercase**.

```
// Fig. 15.19: Fig15_19.cpp
// Stream manipulator uppercase.
#include <iostream>
using namespace std;

int main()
{
   cout << "Printing uppercase letters in scientific" << endl
      << "notation exponents and hexadecimal values:" << endl;

   // use std:uppercase to display uppercase letters; use std::hex and
   // std::showbase to display hexadecimal value and its base
   cout << uppercase << 4.345e10 << endl
      << hex << showbase << 123456789 << endl;
} // end main
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
0X75BCD15
```

Fig. 15.19 | Stream manipulator `uppercase`.

15.7.7 Specifying Boolean Format (boolalpha)

C++ provides data type `bool`, whose values may be `false` or `true`, as a preferred alternative to the old style of using `0` to indicate `false` and nonzero to indicate `true`. A `bool` variable outputs as `0` or `1` by default. However, we can use stream manipulator **boolalpha** to set the output stream to display `bool` values as the strings `"true"` and `"false"`. Use stream manipulator **noboolalpha** to set the output stream to display `bool` values as integers (i.e., the default setting). The program of Fig. 15.20 demonstrates these stream manipulators. Line 11 displays the `bool` value, which line 8 sets to `true`, as an integer. Line 15 uses manipulator `boolalpha` to display the `bool` value as a string. Lines 18–19 then change the `bool`'s value and use manipulator `noboolalpha`, so line 22 can display the `bool` value as an integer. Line 26 uses manipulator `boolalpha` to display the `bool` value as a string. Both `boolalpha` and `noboolalpha` are "sticky" settings.

Good Programming Practice 15.1

Displaying `bool` *values as* `true` *or* `false`*, rather than nonzero or* 0*, respectively, makes program outputs clearer.*

```
// Fig. 15.20: Fig15_20.cpp
// Demonstrating stream manipulators boolalpha and noboolalpha.
#include <iostream>
using namespace std;

int main()
{
   bool booleanValue = true;

   // display default true booleanValue
   cout << "booleanValue is " << booleanValue << endl;

   // display booleanValue after using boolalpha
   cout << "booleanValue (after using boolalpha) is "
      << boolalpha << booleanValue << endl << endl;

   cout << "switch booleanValue and use noboolalpha" << endl;
   booleanValue = false; // change booleanValue
   cout << noboolalpha << endl; // use noboolalpha

   // display default false booleanValue after using noboolalpha
   cout << "booleanValue is " << booleanValue << endl;

   // display booleanValue after using boolalpha again
   cout << "booleanValue (after using boolalpha) is "
      << boolalpha << booleanValue << endl;
} // end main
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

Fig. 15.20 | Stream manipulators `boolalpha` and `noboolalpha`.

15.7.8 Setting and Resetting the Format State via Member Function `flags`

Throughout Section 15.7, we've been using stream manipulators to change output format characteristics. We now discuss how to return an output stream's format to its default state after having applied several manipulations. Member function **`flags`** without an argument returns the current format settings as a **`fmtflags`** data type (of class `ios_base`), which represents the **format state**. Member function `flags` with a `fmtflags` argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that `flags` returns might differ across several systems. The program of Fig. 15.21 uses member function `flags` to save the stream's original format state (line 17), then restore the original format settings (line 25).

```
1  // Fig. 15.21: Fig15_21.cpp
2  // Demonstrating the flags member function.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int integerValue = 1000;
9     double doubleValue = 0.0947628;
10
11    // display flags value, int and double values (original format)
12    cout << "The value of the flags variable is: " << cout.flags()
13       << "\nPrint int and double in original format:\n"
14       << integerValue << '\t' << doubleValue << endl << endl;
15
16    // use cout flags function to save original format
17    ios_base::fmtflags originalFormat = cout.flags();
18    cout << showbase << oct << scientific; // change format
19
20    // display flags value, int and double values (new format)
21    cout << "The value of the flags variable is: " << cout.flags()
22       << "\nPrint int and double in a new format:\n"
23       << integerValue << '\t' << doubleValue << endl << endl;
24
25    cout.flags( originalFormat ); // restore format
26
27    // display flags value, int and double values (original format)
28    cout << "The restored value of the flags variable is: "
29       << cout.flags()
30       << "\nPrint values in original format again:\n"
31       << integerValue << '\t' << doubleValue << endl;
32 } // end main
```

```
The value of the flags variable is: 513
Print int and double in original format:
1000    0.0947628

The value of the flags variable is: 012011
Print int and double in a new format:
01750   9.476280e-002

The restored value of the flags variable is: 513
Print values in original format again:
1000    0.0947628
```

Fig. 15.21 | `flags` member function.

15.8 Stream Error States

The state of a stream may be tested through bits in class `ios_base`. In a moment, we show how to test these bits, in the example of Fig. 15.22.

```cpp
// Fig. 15.22: Fig15_22.cpp
// Testing error states.
#include <iostream>
using namespace std;

int main()
{
   int integerValue;

   // display results of cin functions
   cout << "Before a bad input operation:"
      << "\ncin.rdstate(): " << cin.rdstate()
      << "\n    cin.eof(): " << cin.eof()
      << "\n   cin.fail(): " << cin.fail()
      << "\n    cin.bad(): " << cin.bad()
      << "\n   cin.good(): " << cin.good()
      << "\n\nExpects an integer, but enter a character: ";

   cin >> integerValue; // enter character value
   cout << endl;

   // display results of cin functions after bad input
   cout << "After a bad input operation:"
      << "\ncin.rdstate(): " << cin.rdstate()
      << "\n    cin.eof(): " << cin.eof()
      << "\n   cin.fail(): " << cin.fail()
      << "\n    cin.bad(): " << cin.bad()
      << "\n   cin.good(): " << cin.good() << endl << endl;

   cin.clear(); // clear stream

   // display results of cin functions after clearing cin
   cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
      << "\ncin.good(): " << cin.good() << endl;
} // end main
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
    cin.bad(): 0
   cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1
```

Fig. 15.22 | Testing error states.

The **eofbit** is set for an input stream after end-of-file is encountered. A program can use member function `eof` to determine whether end-of-file has been encountered on a stream after an attempt to extract data beyond the end of the stream. The call

```
cin.eof()
```

returns `true` if end-of-file has been encountered on `cin` and `false` otherwise.

The `failbit` is set for a stream when a format error occurs on the stream and no characters are input (e.g., when you attempt to read a number and the user enters a string). When such an error occurs, the characters are not lost. The **fail** member function reports whether a stream operation has failed. Usually, recovering from such errors is possible.

The `badbit` is set for a stream when an error occurs that results in the loss of data. The **bad** member function reports whether a stream operation failed. Generally, such serious failures are nonrecoverable.

The **goodbit** is set for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set for the stream.

The **good** member function returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. I/O operations should be performed only on "good" streams.

The **rdstate** member function returns the stream's error state. Calling `cout.rdstate`, for example, would return the stream's state, which then could be tested by a `switch` statement that examines `eofbit`, `badbit`, `failbit` and `goodbit`. The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`—using these functions does not require you to be familiar with particular status bits.

The **clear** member function is used to restore a stream's state to "good," so that I/O may proceed on that stream. The default argument for `clear` is `goodbit`, so the statement

```
cin.clear();
```

clears `cin` and sets `goodbit` for the stream. The statement

```
cin.clear( ios::failbit )
```

sets the `failbit`. You might want to do this when performing input on `cin` with a user-defined type and encountering a problem. The name `clear` might seem inappropriate in this context, but it's correct.

The program of Fig. 15.22 demonstrates member functions `rdstate`, `eof`, `fail`, `bad`, `good` and `clear`. The actual values output may differ across different compilers.

The `operator!` member function of `basic_ios` returns `true` if the `badbit` is set, the `failbit` is set or both are set. The `operator void *` member function returns `false` (0) if the `badbit` is set, the `failbit` is set or both are set. These functions are useful in file processing when a `true`/`false` condition is being tested under the control of a selection statement or repetition statement.

15.9 Tying an Output Stream to an Input Stream

Interactive applications generally involve an `istream` for input and an `ostream` for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only when the buffer fills, when outputs are flushed explicitly by the program or automatically at the end of the program. C++ provides member function **tie** to synchronize (i.e., "tie together") the operation of an `istream` and an `ostream` to ensure that outputs appear before their subsequent inputs. The call

```
cin.tie( &cout );
```

ties `cout` (an `ostream`) to `cin` (an `istream`). Actually, this particular call is redundant, because C++ performs this operation automatically to create a user's standard input/output environment. However, the user would tie other `istream`/`ostream` pairs explicitly. To untie an input stream, `inputStream`, from an output stream, use the call

```
inputStream.tie( 0 );
```

15.10 Wrap-Up

This chapter summarized how C++ performs input/output using streams. You learned about the stream-I/O classes and objects, as well as the stream I/O template class hierarchy. We discussed `ostream`'s formatted and unformatted output capabilities performed by the `put` and `write` functions. You saw examples using `istream`'s formatted and unformatted input capabilities performed by the `eof`, `get`, `getline`, `peek`, `putback`, `ignore` and `read` functions. We discussed stream manipulators and member functions that perform formatting tasks—`dec`, `oct`, `hex` and `setbase` for displaying integers; `precision` and `setprecision` for controlling floating-point precision; and `width` and `setw` for setting field width. You also learned additional formatting `iostream` manipulators and member functions—`showpoint` for displaying decimal point and trailing zeros; `left`, `right` and `internal` for justification; `fill` and `setfill` for padding; `scientific` and `fixed` for displaying floating-point numbers in scientific and fixed notation; `uppercase` for uppercase/lowercase control; `boolalpha` for specifying boolean format; and `flags` and `fmtflags` for resetting the format state.

We introduced exception handling earlier in the book in our discussion of arrays. In the next chapter, we take a deeper look at C++'s rich set of exception handling capabilities.

Summary

Section 15.1 Introduction

- I/O operations are performed in a manner sensitive to the type of the data.

Section 15.2 Streams

- C++ I/O occurs in streams (p. 597). A stream is a sequence of bytes.
- Low-level I/O-capabilities specify that bytes should be transferred device-to-memory or memory-to-device. High-level I/O is performed with bytes grouped into meaningful units such as integers, strings and user-defined types.
- C++ provides both unformatted-I/O and formatted-I/O operations. Unformatted-I/O (p. 597) transfers are fast, but process raw data that is difficult for people to use. Formatted I/O processes data in meaningful units, but requires extra processing time that can degrade the performance.
- The `<iostream>` header declares all stream-I/O operations (p. 598).
- The `<iomanip>` header declares the parameterized stream manipulators (p. 598).
- The `<fstream>` header declares file-processing operations (p. 600).
- The `basic_istream` template (p. 598) supports stream-input operations.

- The `basic_ostream` template (p. 598) supports stream-output operations.
- The `basic_iostream` template supports both stream-input and stream-output operations.
- Templates `basic_istream` and the `basic_ostream` each derive from the `basic_ios` (p. 599) template.
- Template `basic_iostream` derives from both the `basic_istream` and `basic_ostream` templates.
- The `istream` object `cin` is tied to the standard input device, normally the keyboard.
- The `ostream` object `cout` is tied to the standard output device, normally the screen.
- The `ostream` object `cerr` is tied to the standard error device, normally the screen. Outputs to `cerr` are unbuffered (p. 600)—each insertion to `cerr` appears immediately.
- The `ostream` object `clog` is tied to the standard error device, normally the screen. Outputs to `clog` are buffered (p. 600).
- The C++ compiler determines data types automatically for input and output.

Section 15.3 Stream Output

- Addresses are displayed in hexadecimal format by default.
- To print the address in a pointer variable, cast the pointer to `void *`.
- Member function `put` outputs one character. Calls to `put` may be cascaded.

Section 15.4 Stream Input

- Stream input is performed with the stream extraction operator >>, which automatically skips white-space characters (p. 602) in the input stream and returns `false` after end-of-file is encountered.
- Stream extraction causes `failbit` (p. 602) to be set for improper input and `badbit` (p. 602) to be set if the operation fails.
- A series of values can be input using the stream extraction operation in a `while` loop header. The extraction returns `0` when end-of-file is encountered or an error occurs.
- The `get` member function (p. 602) with no arguments inputs one character and returns the character; `EOF` is returned if end-of-file is encountered on the stream.
- Member function `get` with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object (p. 598) for which the `get` member function is being invoked.
- Member function `get` with three arguments—a character array, a size limit and a delimiter (with default value newline)—reads characters from the input stream up to a maximum of limit – 1 characters, or until the delimiter is read. The input string is terminated with a null character. The delimiter is not placed in the character array but remains in the input stream.
- Member function `getline` (p. 604) operates like the three-argument `get` member function. The `getline` function removes the delimiter from the input stream but does not store it in the string.
- Member function `ignore` (p. 605) skips the specified number of characters (the default is 1) in the input stream; it terminates if the specified delimiter is encountered (the default delimiter is `EOF`).
- The `putback` member function (p. 605) places the previous character obtained by a `get` on a stream back into that stream.
- The `peek` member function (p. 605) returns the next character from an input stream but does not extract (remove) the character from the stream.
- C++ offers type-safe I/O (p. 596). If unexpected data is processed by the << and >> operators, various error bits are set, which can be tested to determine whether an I/O operation succeeded or failed. If operator << has not been overloaded for a user-defined type, a compiler error is reported.

Section 15.5 Unformatted I/O Using `read`*,* `write` *and* `gcount`

- Unformatted I/O is performed with member functions `read` and `write` (p. 605). These input or output bytes to or from memory, beginning at a designated memory address.
- The `gcount` member function (p. 606) returns the number of characters input by the previous `read` operation on that stream.
- Member function `read` inputs a specified number of characters into a character array. `failbit` is set if fewer than the specified number of characters are read.

Section 15.6 Introduction to Stream Manipulators

- To change the base in which integers output, use the manipulator `hex` (p. 607) to set the base to hexadecimal (base 16) or `oct` (p. 607) to set the base to octal (base 8). Use manipulator `dec` (p. 607) to reset the base to decimal. The base remains the same until changed explicitly.
- The parameterized stream manipulator `setbase` (p. 607) also sets the base for integer output. `setbase` takes one integer argument of `10`, `8` or `16` to set the base.
- Floating-point precision can be controlled with the `setprecision` stream manipulator or the `precision` member function (p. 607). Both set the precision for all subsequent output operations until the next precision-setting call. The `precision` member function with no argument returns the current precision value.
- Parameterized manipulators require the inclusion of the `<iomanip>` header.
- Member function `width` sets the field width and returns the previous width. Values narrower than the field are padded with fill characters (p. 609). The field-width setting applies only for the next insertion or extraction; the field width is set to `0` implicitly (subsequent values will be output as large as necessary). Values wider than a field are printed in their entirety. Function `width` with no argument returns the current width setting. Manipulator `setw` also sets the width.
- For input, the `setw` stream manipulator establishes a maximum string size; if a larger string is entered, the larger line is broken into pieces no larger than the designated size.
- You can create your own stream manipulators.

Section 15.7 Stream Format States and Stream Manipulators

- Stream manipulator `showpoint` (p. 612) forces a floating-point number to be output with a decimal point and with the number of significant digits specified by the precision.
- Stream manipulators `left` and `right` (p. 613) cause fields to be left justified with padding characters to the right or right justified with padding characters to the left.
- Stream manipulator `internal` (p. 614) indicates that a number's sign (or base when using stream manipulator `showbase;` p. 616) should be left justified within a field, its magnitude should be right justified and intervening spaces should be padded with the fill character.
- Member function `fill` (p. 615) specifies the fill character to be used with stream manipulators `left`, `right` and `internal` (space is the default); the prior padding character is returned. Stream manipulator `setfill` (p. 615) also sets the fill character.
- Stream manipulators `oct`, `hex` and `dec` specify that integers are to be treated as octal, hexadecimal or decimal values, respectively. Integer output defaults to decimal if none of these bits is set; stream extractions process the data in the form the data is supplied.
- Stream manipulator `showbase` forces the base of an integral value to be output.
- Stream manipulator `scientific` (p. 617) is used to output a floating-point number in scientific format. Stream manipulator `fixed` (p. 617) is used to output a floating-point number with the precision specified by the `precision` member function.

- Stream manipulator `uppercase` (p. 612) outputs an uppercase `X` or `E` for hexadecimal integers and scientific notation floating-point values, respectively. Hexadecimal values appear in all uppercase.
- Member function `flags` (p. 619) with no argument returns the current format state (p. 619) as a `long` value. Function `flags` with a `long` argument sets the format state specified by the argument.

Section 15.8 Stream Error States

- The state of a stream may be tested through bits in class `ios_base`.
- The `eofbit` (p. 622) is set for an input stream after end-of-file is encountered during an input operation. The `eof` member function (p. 622) reports whether the `eofbit` has been set.
- A stream's `failbit` is set when a format error occurs. The `fail` member function (p. 622) reports whether a stream operation has failed; it's normally possible to recover from such errors.
- A stream's `badbit` is set when an error occurs that results in data loss. Member function `bad` reports whether a stream operation failed. Such serious failures are normally nonrecoverable.
- The `good` member function (p. 622) returns true if the `bad`, `fail` and `eof` functions would all return `false`. I/O operations should be performed only on "good" streams.
- The `rdstate` member function (p. 622) returns the error state of the stream.
- Member function `clear` (p. 622) restores a stream's state to "good," so that I/O may proceed.

Section 15.9 Tying an Output Stream to an Input Stream

- C++ provides the `tie` member function (p. 623) to synchronize `istream` and `ostream` operations to ensure that outputs appear before subsequent inputs.

Self-Review Exercises

15.1 *(Fill in the Blanks)* Answer each of the following:

a) Input/output in C++ occurs as ________ of bytes.
b) The stream manipulators for justification are ________, ________ and ________.
c) Member function ________ can be used to set and reset format state.
d) Most C++ programs that do I/O should include the ________ header that contains the declarations required for all stream-I/O operations.
e) When using parameterized manipulators, the header ________ must be included.
f) Header ________ contains the declarations required for file processing.
g) The `ostream` member function ________ is used to perform unformatted output.
h) Input operations are supported by class ________.
i) Standard error stream outputs are directed to the stream objects ________ or ________.
j) Output operations are supported by class ________.
k) The symbol for the stream insertion operator is ________.
l) The four objects that correspond to the standard devices on the system include ________, ________, ________ and ________.
m) The symbol for the stream extraction operator is ________.
n) The stream manipulators ________, ________ and ________ specify that integers should be displayed in octal, hexadecimal and decimal formats, respectively.
o) The ________ stream manipulator causes positive numbers to display with a plus sign.

15.2 *(True or False)* State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The stream member function `flags` with a `long` argument sets the `flags` state variable to its argument and returns its previous value.
b) The stream insertion operator << and the stream extraction operator >> are overloaded to handle all standard data types—including strings and memory addresses (stream insertion only)—and all user-defined data types.

c) The stream member function `flags` with no arguments resets the stream's format state.
d) The stream extraction operator `>>` can be overloaded with an operator function that takes an `istream` reference and a reference to a user-defined type as arguments and returns an `istream` reference.
e) The stream insertion operator `<<` can be overloaded with an operator function that takes an `istream` reference and a reference to a user-defined type as arguments and returns an `istream` reference.
f) Input with the stream extraction operator `>>` always skips leading white-space characters in the input stream, by default.
g) The stream member function `rdstate` returns the current state of the stream.
h) The `cout` stream normally is connected to the display screen.
i) The stream member function `good` returns `true` if the `bad`, `fail` and `eof` member functions all return `false`.
j) The `cin` stream normally is connected to the display screen.
k) If a nonrecoverable error occurs during a stream operation, the `bad` member function will return `true`.
l) Output to `cerr` is unbuffered and output to `clog` is buffered.
m) Stream manipulator `showpoint` forces floating-point values to print with the default six digits of precision unless the precision value has been changed, in which case floating-point values print with the specified precision.
n) The `ostream` member function `put` outputs the specified number of characters.
o) The stream manipulators `dec`, `oct` and `hex` affect only the next integer output operation.

15.3 *(Write a C++ Statement)* For each of the following, write a single statement that performs the indicated task.

a) Output the string `"Enter your name: "`.
b) Use a stream manipulator that causes the exponent in scientific notation and the letters in hexadecimal values to print in capital letters.
c) Output the address of the variable `myString` of type `char *`.
d) Use a stream manipulator to ensure that floating-point values print in scientific notation.
e) Output the address in variable `integerPtr` of type `int *`.
f) Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.
g) Output the value pointed to by `floatPtr` of type `float *`.
h) Use a stream member function to set the fill character to `'*'` for printing in field widths larger than the values being output. Repeat this statement with a stream manipulator.
i) Output the characters `'O'` and `'K'` in one statement with `ostream` function `put`.
j) Get the value of the next character to input without extracting it from the stream.
k) Input a single character into variable `charValue` of type `char`, using the `istream` member function `get` in two different ways.
l) Input and discard the next six characters in the input stream.
m) Use `istream` member function `read` to input 50 characters into `char` array `line`.
n) Read 10 characters into character array `name`. Stop reading characters if the `'.'` delimiter is encountered. Do not remove the delimiter from the input stream. Write another statement that performs this task and removes the delimiter from the input.
o) Use the `istream` member function `gcount` to determine the number of characters input into character array `line` by the last call to `istream` member function `read`, and output that number of characters, using `ostream` member function `write`.
p) Output `124`, `18.376`, `'Z'`, `1000000` and `"String"`, separated by spaces.
q) Print the current precision setting, using a member function of object `cout`.

r) Input an integer value into `int` variable `months` and a floating-point value into `float` variable `percentageRate`.
s) Print `1.92`, `1.925` and `1.9258` separated by tabs and with `3` digits of precision, using a stream manipulator.
t) Print integer `100` in octal, hexadecimal and decimal, using stream manipulators and separated by tabs.
u) Print integer `100` in decimal, octal and hexadecimal separated by tabs, using a stream manipulator to change the base.
v) Print `1234` right justified in a `10`-digit field.
w) Read characters into character array `line` until the character `'z'` is encountered, up to a limit of `20` characters (including a terminating null character). Do not extract the delimiter character from the stream.
x) Use integer variables `x` and `y` to specify the field width and precision used to display the `double` value `87.4573`, and display the value.

15.4 *(Find and Correct Code Errors)* Identify the error in each of the following statements and explain how to correct it.

a) `cout << "Value of x <= y is: " << x <= y;`
b) The following statement should print the integer value of `'c'`.
```
cout << 'c';
```
c) `cout << ""A string in quotes"";`

15.5 *(Show Outputs)* For each of the following, show the output.

a)
```
cout << "12345" << endl;
cout.width( 5 );
cout.fill( '*' );
cout << 123 << endl << 123;
```
b) `cout << setw( 10 ) << setfill( '$' ) << 10000;`
c) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`
d) `cout << showbase << oct << 99 << endl << hex << 99;`
e) `cout << 100000 << endl << showpos << 100000;`
f) `cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;`

Answers to Self-Review Exercises

15.1 a) streams. b) `left`, `right` and `internal`. c) `flags`. d) `<iostream>`. e) `<iomanip>`. f) `<fstream>`. g) `write`. h) `istream`. i) `cerr` or `clog`. j) `ostream`. k) `<<`. l) `cin`, `cout`, `cerr` and `clog`. m) `>>`. n) `oct`, `hex` and `dec`. o) `showpos`.

15.2 a) False. The stream member function `flags` with a `fmtflags` argument sets the `flags` state variable to its argument and returns the prior state settings. b) False. The stream insertion and stream extraction operators are not overloaded for all user-defined types. You must specifically provide the overloaded operator functions to overload the stream operators for use with each user-defined type you create. c) False. The stream member function `flags` with no arguments returns the current format settings as a `fmtflags` data type, which represents the format state. d) True. e) False. To overload the stream insertion operator `<<`, the overloaded operator function must take an `ostream` reference and a reference to a user-defined type as arguments and return an `ostream` reference. f) True. g) True. h) True. i) True. j) False. The `cin` stream is connected to the standard input of the computer, which normally is the keyboard. k) True. l) True. m) True. n) False. The `ostream` member function `put` outputs its single-character argument. o) False. The stream manipulators `dec`, `oct` and `hex` set the output format state for integers to the specified base until the base is changed again or the program terminates. p) False. Memory addresses are displayed in hexadecimal format by default. To display addresses as `long` integers, the address must be cast to a `long` value.

15.3 a) `cout << "Enter your name: ";`
b) `cout << uppercase;`
c) `cout << static_cast< void * >( myString );`
d) `cout << scientific;`
e) `cout << integerPtr;`
f) `cout << showbase;`
g) `cout << *floatPtr;`
h) `cout.fill( '*' );`
`cout << setfill( '*' );`
i) `cout.put( 'O' ).put( 'K' );`
j) `cin.peek();`
k) `charValue = cin.get();`
`cin.get( charValue );`
l) `cin.ignore( 6 );`
m) `cin.read( line, 50 );`
n) `cin.get( name, 10, '.' );`
`cin.getline( name, 10, '.' );`
o) `cout.write( line, cin.gcount() );`
p) `cout << 124 << ' ' << 18.376 << ' ' << "Z " << 1000000 << " String";`
q) `cout << cout.precision();`
r) `cin >> months >> percentageRate;`
s) `cout << setprecision( 3 ) << 1.92 << '\t' << 1.925 << '\t' << 1.9258;`
t) `cout << oct << 100 << '\t' << hex << 100 << '\t' << dec << 100;`
u) `cout << 100 << '\t' << setbase( 8 ) << 100 << '\t' << setbase( 16 ) << 100;`
v) `cout << setw( 10 ) << 1234;`
w) `cin.get( line, 20, 'z' );`
x) `cout << setw( x ) << setprecision( y ) << 87.4573;`

15.4 a) *Error:* The precedence of the << operator is higher than that of <=, which causes the statement to be evaluated improperly and also causes a compiler error.
Correction: Place parentheses around the expression `x <= y`.
b) *Error:* In C++, characters are not treated as small integers, as they are in C.
Correction: To print the numerical value for a character in the computer's character set, the character must be cast to an integer value, as in the following:
```
cout << static_cast< int >( 'c' );
```
c) *Error:* Quote characters cannot be printed in a string unless an escape sequence is used.
Correction: Print the string in one of the following ways:
```
cout << "\"A string in quotes\"";
```

15.5 a)
```
12345
**123
123
```
b) `$$$$$10000`
c) `1024.988`
d)
```
0143
0x63
```
e)
```
100000
+100000
```
f) ` 4.45e+002`

Exercises

15.6 *(Write C++ Statements)* Write a statement for each of the following:

a) Print integer `40000` left justified in a 15-digit field.
b) Read a string into character array variable `state`.
c) Print `200` with and without a sign.
d) Print the decimal value `100` in hexadecimal form preceded by `0x`.
e) Read characters into array `charArray` until the character `'p'` is encountered, up to a limit of 10 characters (including the terminating null character). Extract the delimiter from the input stream, and discard it.
f) Print `1.234` in a 9-digit field with preceding zeros.

15.7 *(Inputting Decimal, Octal and Hexadecimal Values)* Write a program to test the inputting of integer values in decimal, octal and hexadecimal formats. Output each integer read by the program in all three formats. Test the program with the following input data: `10`, `010`, `0x10`.

15.8 *(Printing Pointer Values as Integers)* Write a program that prints pointer values, using casts to all the integer data types. Which ones print strange values? Which ones cause errors?

15.9 *(Printing with Field Widths)* Write a program to test the results of printing the integer value `12345` and the floating-point value `1.2345` in various-sized fields. What happens when the values are printed in fields containing fewer digits than the values?

15.10 *(Rounding)* Write a program that prints the value `100.453627` rounded to the nearest digit, tenth, hundredth, thousandth and ten-thousandth.

15.11 *(Length of a String)* Write a program that inputs a string from the keyboard and determines the length of the string. Print the string in a field width that is twice the length of the string.

15.12 *(Converting Fahrenheit to Celsius)* Write a program that converts integer Fahrenheit temperatures from `0` to `212` degrees to floating-point Celsius temperatures with `3` digits of precision. Use the formula

```
celsius = 5.0 / 9.0 * ( fahrenheit - 32 );
```

to perform the calculation. The output should be printed in two right-justified columns and the Celsius temperatures should be preceded by a sign for both positive and negative values.

15.13 In some programming languages, strings are entered surrounded by either single or double quotation marks. Write a program that reads the three strings `suzy`, `"suzy"` and `'suzy'`. Are the single and double quotes ignored or read as part of the string?

15.14 *(Reading Phone Numbers with and Overloaded Stream Extraction Operator)* In Fig. 11.5, the stream extraction and stream insertion operators were overloaded for input and output of objects of the `PhoneNumber` class. Rewrite the stream extraction operator to perform the following error checking on input. The `operator>>` function will need to be reimplemented.

a) Input the entire phone number into an array. Test that the proper number of characters has been entered. There should be a total of 14 characters read for a phone number of the form `(800) 555-1212`. Use `ios_base`-member-function `clear` to set `failbit` for improper input.
b) The area code and exchange do not begin with `0` or `1`. Test the first digit of the area-code and exchange portions of the phone number to be sure that neither begins with `0` or `1`. Use `ios_base`-member-function `clear` to set `failbit` for improper input.
c) The middle digit of an area code used to be limited to `0` or `1` (though this has changed). Test the middle digit for a value of `0` or `1`. Use the `ios_base`-member-function `clear` to set `failbit` for improper input. If none of the above operations results in `failbit` being set for improper input, copy the parts of the telephone number into the `PhoneNumber`

object's `areaCode`, `exchange` and `line` members. If `failbit` has been set on the input, have the program print an error message and end, rather than print the phone number.

15.15 *(`Point` Class)* Write a program that accomplishes each of the following:

a) Create a user-defined class `Point` that contains the private integer data members `xCoordinate` and `yCoordinate` and declares stream insertion and stream extraction overloaded operator functions as `friends` of the class.

b) Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set the `failbit` to indicate improper input. The stream insertion operator should not be able to display the point after an input error occurred.

c) Write a `main` function that tests input and output of user-defined class `Point`, using the overloaded stream extraction and stream insertion operators.

15.16 *(`Complex` Class)* Write a program that accomplishes each of the following:

a) Create a user-defined class `Complex` that contains the private integer data members `real` and `imaginary` and declares stream insertion and stream extraction overloaded operator functions as `friends` of the class.

b) Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set `failbit` to indicate improper input. The input should be of the form

```
3 + 8i
```

c) The values can be negative or positive, and it's possible that one of the two values is not provided, in which case the appropriate data member should be set to 0. The stream insertion operator should not be able to display the point if an input error occurred. For negative imaginary values, a minus sign should be printed rather than a plus sign.

d) Write a `main` function that tests input and output of user-defined class `Complex`, using the overloaded stream extraction and stream insertion operators.

15.17 *(Printing a Table of ASCII Values)* Write a program that uses a `for` statement to print a table of ASCII values for the characters in the ASCII character set from `33` to `126`. The program should print the decimal value, octal value, hexadecimal value and character value for each character. Use the stream manipulators `dec`, `oct` and `hex` to print the integer values.

15.18 *(String-Terminating Null Character)* Write a program to show that the `getline` and three-argument `get` `istream` member functions both end the input string with a string-terminating null character. Also, show that `get` leaves the delimiter character on the input stream, whereas `getline` extracts the delimiter character and discards it. What happens to the unread characters in the stream?

16 Exception Handling: A Deeper Look

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.
—Franklin Delano Roosevelt

If they're running and they don't look where they're going I have to come out from somewhere and catch them.
—Jerome David Salinger

I never forget a face, but in your case I'll make an exception.
—Groucho Marx

Objectives

In this chapter you'll learn:

- To use `try`, `catch` and `throw` to detect, handle and indicate exceptions, respectively.
- To process uncaught and unexpected exceptions.
- To declare new exception classes.
- How stack unwinding enables exceptions not caught in one scope to be caught in another.
- To handle `new` failures.
- To use `unique_ptr` to prevent memory leaks.
- To understand the standard exception hierarchy.

Outline

16.1 Introduction

As you know, an **exception** is an indication of a problem that occurs during a program's execution. **Exception handling** enables you to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. The features presented in this chapter enable you to write **robust** and **fault-tolerant programs** that can deal with problems continue executing or terminate gracefully.

We begin with a review of exception-handling concepts via an example that demonstrates handling an exception that occurs when a function attempts to divide by zero. We show how to handle exceptions that occur in a constructor or destructor and exceptions that occur if operator `new` fails to allocate memory for an object. We introduce several C++ Standard Library exception handling classes.

Software Engineering Observation 16.1

Exception handling provides a standard mechanism for processing errors. This is especially important when working on a project with a large team of programmers.

Software Engineering Observation 16.2

Incorporate your exception-handling strategy into your system from inception. Including effective exception handling after a system has been implemented can be difficult.

16.2 Example: Handling an Attempt to Divide by Zero

Let's consider a simple example of exception handling (Figs. 16.1–16.2). We show how to deal with a common arithmetic problem—division by zero. In C++, *division by zero* using integer arithmetic typically causes a program to terminate prematurely. In floating-point arithmetic, some C++ implementations allow division by zero, in which case a result of positive or negative infinity is displayed as `INF` or `-INF`, respectively.

In this example, we define a function named `quotient` that receives two integers input by the user and divides its first `int` parameter by its second `int` parameter. Before performing the division, the function casts the first `int` parameter's value to type `double`. Then, the second `int` parameter's value is (implicitly) promoted to type `double` for the calculation. So function `quotient` actually performs the division using two `double` values and returns a `double` result.

Although division by zero is often allowed in floating-point arithmetic, for the purpose of this example we treat any attempt to divide by zero as an error. Thus, function `quotient` tests its second parameter to ensure that it isn't zero before allowing the division to proceed. If the second parameter is zero, the function throws an exception to indicate to the caller that a problem occurred. The caller (`main` in this example) can then process the exception and allow the user to type two new values before calling function `quotient` again. In this way, the program can continue executing even after an improper value is entered, thus making the program more robust.

The example consists of two files. `DivideByZeroException.h` (Fig. 16.1) defines an *exception class* that represents the type of the problem that might occur in the example, and `fig16_02.cpp` (Fig. 16.2) defines the `quotient` function and the `main` function that calls it. Function `main` contains the code that demonstrates exception handling.

Defining an Exception Class to Represent the Type of Problem That Might Occur

Figure 16.1 defines class `DivideByZeroException` as a derived class of Standard Library class **`runtime_error`** (defined in header **`<stdexcept>`**). Class `runtime_error`—a derived class of Standard Library class **`exception`** (defined in header **`<exception>`**)—is the C++ standard base class for representing runtime errors. Class `exception` is the standard C++ base class for all exceptions. (Section 16.12 discusses class `exception` and its derived classes in detail.) A typical exception class that derives from the `runtime_error` class defines only a constructor (e.g., lines 12–13) that passes an error-message string to the base-class `runtime_error` constructor. Every exception class that derives directly or indirectly from `exception` contains the `virtual` function **`what`**, which returns an exception object's error message. You're not required to derive a custom exception class, such as `DivideByZeroException`, from the standard exception classes provided by C++. However, doing so allows you to use the `virtual` function `what` to obtain an appropriate error message. We use an object of this `DivideByZeroException` class in Fig. 16.2 to indicate when an attempt is made to divide by zero.

```
1  // Fig. 16.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header contains runtime_error
4  using namespace std;
5
6  // DivideByZeroException objects should be thrown by functions
7  // upon detecting division-by-zero exceptions
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // constructor specifies default error message
12    DivideByZeroException()
13          runtime_error( "attempted to divide by zero" )
14 }; // end class DivideByZeroException
```

Fig. 16.1 | Class `DivideByZeroException` definition.

Demonstrating Exception Handling

Figure 16.2 uses exception handling to wrap code that might throw a "divide-by-zero" exception and to handle that exception, should one occur. The user enters two integers, which are passed as arguments to function `quotient` (lines 10–18). This function divides its first parameter (`numerator`) by its second parameter (`denominator`). Assuming that the user does not specify `0` as the denominator for the division, function `quotient` returns the division result. If the user inputs `0` for the denominator, `quotient` throws an exception. In the sample output, the first two lines show a successful calculation, and the next two show a failure due to an attempt to divide by zero. When the exception occurs, the program informs the user of the mistake and prompts the user to input two new integers. After we discuss the code, we'll consider the user inputs and flow of program control that yield these outputs.

```cpp
// Fig. 16.2: Fig16_02.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>
#include "DivideByZeroException.h" // DivideByZeroException class
using namespace std;

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
   // throw DivideByZeroException if trying to divide by zero
   if ( denominator == 0 )
      throw DivideByZeroException(); // terminate function

   // return division result
   return static_cast< double >( numerator ) / denominator;
} // end function quotient

int main()
{
   int number1; // user-specified numerator
   int number2; // user-specified denominator
   double result; // result of division

   cout << "Enter two integers (end-of-file to end): ";

   // enable user to enter two integers to divide
   while ( cin >> number1 >> number2 )
   {
      // try block contains code that might throw exception
      // and code that will not execute if an exception occurs
      try
      {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      } // end try
      catch ( DivideByZeroException &divideByZeroException )
      {
         cout << "Exception occurred: "
            << divideByZeroException.what() << endl;
      } // end catch
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 1 of 2.)

```
43
44         cout << "\nEnter two integers (end-of-file to end): ";
45      } // end while
46
47      cout << endl;
48   } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 16.2 | Exception-handling example that throws exceptions on attempts to divide by zero. (Part 2 of 2.)

Enclosing Code in a `try` Block

The program begins by prompting the user to enter two integers. The integers are input in the condition of the `while` loop (line 29). Line 35 passes the values to function `quotient` (lines 10–18), which either divides the integers and returns a result, or **throws an exception** (i.e., indicates that an error occurred) on an attempt to divide by zero. Exception handling is geared to situations in which the function that detects an error is unable to handle it.

As you learned in Section 7.11, `try` blocks enable exception handling. A `try` block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs. The `try` block in lines 33–37 encloses the invocation of function `quotient` and the statement that displays the division result. In this example, because the invocation of function `quotient` (line 35) can *throw* an exception, we enclose this function invocation in a `try` block. Enclosing the output statement (line 36) in the `try` block ensures that the output will occur only if function `quotient` returns a result.

Software Engineering Observation 16.3

Exceptions may surface through explicitly mentioned code in a `try` block, through calls to other functions and through deeply nested function calls initiated by code in a `try` block.

Defining a `catch` Handler to Process a `DivideByZeroException`

You saw in Section 7.11 that exceptions are processed by `catch` handlers. At least one `catch` handler (lines 38–42) must immediately follow each `try` block. The exception parameter is declared as a *reference* to the type of exception the `catch` handler can process (`DivideByZeroException` in this case). When an exception occurs in a `try` block, the `catch` handler that executes is the one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a base class of it). If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with the caught exception in the body of the `catch` handler, which is delimited by braces (`{` and `}`). A `catch` handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task. In this example, the `catch` handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

Common Programming Error 16.1

It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.

Common Programming Error 16.2

Each `catch` handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

Common Programming Error 16.3

It's a logic error to catch the same type in two different `catch` handlers following a single `try` block.

Termination Model of Exception Handling

If an exception occurs as the result of a statement in a `try` block, the `try` block expires (i.e., terminates immediately). Next, the program searches for the first `catch` handler that can process the type of exception that occurred. The program locates the matching `catch` by comparing the thrown exception's type to each `catch`'s exception-parameter type until the program finds a match. A match occurs if the types are *identical* or if the thrown exception's type is a *derived class* of the exception-parameter type. When a match occurs, the code contained in the matching `catch` handler executes. When a `catch` handler finishes processing by reaching its closing right brace (`}`), the exception is considered handled and the local variables defined within the `catch` handler (including the `catch` parameter) go out of scope. Program control does *not* return to the point at which the exception occurred (known as the **throw point**), because the `try` block has *expired*. Rather, control resumes with the first statement (line 44) after the last `catch` handler following the `try` block. This is known as the **termination model of exception handling**. Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the throw point. As with any other block of code, when a `try` block terminates, local variables defined in the block go out of scope.

Common Programming Error 16.4

Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.

Error-Prevention Tip 16.1

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what's called mission-critical computing or business-critical computing.

If the `try` block completes its execution successfully (i.e., no exceptions occur in the `try` block), then the program ignores the `catch` handlers and program control continues with the first statement after the last `catch` following that `try` block.

If an exception that occurs in a `try` block has no matching `catch` handler, or if an exception occurs in a statement that is not in a `try` block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function. This process is called **stack unwinding** and is discussed in Section 16.7.

Flow of Program Control When the User Enters a Nonzero Denominator

Consider the flow of control when the user inputs the numerator `100` and the denominator `7`. In line 13, function `quotient` determines that the `denominator` does not equal zero, so line 17 performs the division and returns the result (`14.2857`) to line 35 as a `double`. Program control then continues sequentially from line 35, so line 36 displays the division result—line 37 ends the `try` block. Because the `try` block completed successfully and did not throw an exception, the program does not execute the statements contained in the `catch` handler (lines 38–42), and control continues to line 44 (the first line of code after the `catch` handler), which prompts the user to enter two more integers.

Flow of Program Control When the User Enters a Denominator of Zero

Now consider the case in which the user inputs the numerator `100` and the denominator `0`. In line 13, `quotient` determines that the `denominator` equals zero, which indicates an attempt to divide by zero. Line 14 throws an exception, which we represent as an object of class `DivideByZeroException` (Fig. 16.1).

To throw an exception, line 14 uses keyword **`throw`** followed by an operand that represents the type of exception to throw. Normally, a `throw` statement specifies one operand. (In Section 16.4, we discuss how to use a `throw` statement with no operand.) The operand of a `throw` can be of any type. If the operand is an object, we call it an **exception object**—in this example, the exception object is an object of type `DivideByZeroException`. However, a `throw` operand also can assume other values, such as the value of an expression that does not result in an object of a class (e.g., `throw x > 5`) or the value of an `int` (e.g., `throw 5`). The examples in this chapter focus exclusively on throwing objects of exception classes.

Common Programming Error 16.5

Use caution when `throw`*ing the result of a conditional expression (*`?:`*)—promotion rules could cause the value to be of a type different from the one expected. For example, when throwing an* `int` *or a* `double` *from the same conditional expression, the* `int` *is promoted to a* `double`*. So, a* `catch` *handler that catches an* `int` *would never execute based on such a conditional expression.*

As part of throwing an exception, the `throw` operand is created and used to initialize the parameter in the `catch` handler, which we discuss momentarily. The `throw` statement in line 14 creates a `DivideByZeroException` object. When line 14 throws the exception, function `quotient` exits immediately. So, line 14 throws the exception *before* function `quotient` can perform the division in line 17. This is a central characteristic of exception handling: *A function should throw an exception before the error has an opportunity to occur.*

Because we enclosed the call to `quotient` (line 35) in a `try` block, program control enters the `catch` handler (lines 38–42) that immediately follows the `try` block. This `catch` handler serves as the exception handler for the divide-by-zero exception. In general, when an exception is thrown within a `try` block, the exception is caught by a `catch` handler that specifies the type matching the thrown exception. In this program, the `catch` handler specifies that it catches `DivideByZeroException` objects—this type matches the object type thrown in function `quotient`. Actually, the `catch` handler catches a *reference* to the `DivideByZeroException` object created by function `quotient`'s `throw` statement (line 14), so that the `catch` handler does not make a copy of the exception object.

The `catch`'s body (lines 40–41) prints the error message returned by function `what` of base-class `runtime_error`—i.e., the string that the `DivideByZeroException` constructor (lines 12–13 in Fig. 16.1) passed to the `runtime_error` base-class constructor.

Performance Tip 16.1

Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception.

Good Programming Practice 16.1

Associating each type of runtime error with an appropriately named exception object improves program clarity.

16.3 When to Use Exception Handling

Exception handling is designed to process synchronous errors, which occur when a statement executes, such as *out-of-range array subscripts*, *arithmetic overflow* (i.e., a value outside the representable range of values), *division by zero*, *invalid function parameters* and *unsuccessful memory allocation* (due to lack of memory). Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Software Engineering Observation 16.4

Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other's error-processing code.

Software Engineering Observation 16.5

Avoid using exception handling as an alternate form of flow of control. These "additional" exceptions can "get in the way" of genuine error-type exceptions.

Software Engineering Observation 16.6

Exception handling enables predefined software components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

Exception handling also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes. Such software elements often use exceptions to notify programs when problems occur. This enables you to implement *customized error handling* for each application.

Performance Tip 16.2

When no exceptions occur, exception-handling code incurs little or no performance penalty. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.

Software Engineering Observation 16.7

Functions with common error conditions should return `0` *or* `NULL` *(or other appropriate values, such as* `bool`*s) rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.*

Complex applications normally consist of predefined software components and application-specific components that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the *predefined component cannot know in advance how each application processes a problem that occurs.*

16.4 Rethrowing an Exception

It's possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially. In such cases, the exception handler can *defer the exception handling (or perhaps a portion of it) to another exception handler.* In either case, you achieve this by **rethrowing the exception** via the statement

```
throw;
```

Regardless of whether a handler can process an exception, the handler can *rethrow* the exception for further processing outside the handler. The next enclosing `try` block detects the rethrown exception, which a `catch` handler listed after that enclosing `try` block attempts to handle.

Common Programming Error 16.6

Executing an empty `throw` *statement outside a* `catch` *handler calls function* **`terminate`**, *which abandons exception processing and terminates the program immediately.*

The program of Fig. 16.3 demonstrates rethrowing an exception. In `main`'s `try` block (lines 29–34), line 32 calls function `throwException` (lines 8–24). The `throwException` function also contains a `try` block (lines 11–15), from which the `throw` statement in line 14 `throw`s an instance of standard-library-class `exception`. Function `throwException`'s `catch` handler (lines 16–21) catches this exception, prints an error message (lines 18–19) and rethrows the exception (line 20). This terminates function `throwException` and returns control to line 32 in the `try...catch` block in `main`. The `try` block terminates (so line 33 does not execute), and the `catch` handler in `main` (lines 35–38) catches this exception and prints an error message (line 37). Since we do not use the exception parameters in the `catch` handlers of this example, we omit the exception parameter names and specify only the type of exception to catch (lines 16 and 35).

```
1  // Fig. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
4  #include <exception>
5  using namespace std;
```

Fig. 16.3 | Rethrowing an exception. (Part 1 of 2.)

```

// throw, catch and rethrow exception
void throwException()
{
   // throw exception and catch it immediately
   try
   {
      cout << "  Function throwException throws an exception\n";
      throw exception(); // generate exception
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "  Exception handled in function throwException"
         << "\n  Function throwException rethrows exception";
      throw; // rethrow exception for further processing
   } // end catch

   cout << "This also should not print\n";
} // end function throwException

int main()
{
   // throw exception
   try
   {
      cout << "\nmain invokes function throwException\n";
      throwException();
      cout << "This should not print\n";
   } // end try
   catch ( exception & ) // handle exception
   {
      cout << "\n\nException handled in main\n";
   } // end catch

   cout << "Program control continues after catch in main\n";
} // end main
```

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Fig. 16.3 | Rethrowing an exception. (Part 2 of 2.)

16.5 Exception Specifications[1]

An optional exception specification (also called a throw list) enumerates a list of exceptions that a function can throw. For example, placing

1. Exception specifications are deprecated in the new C++ standard. We discuss exception-handling features of the new C++ standard in Chapter 23. Most current compilers ignore exception specifications.

```
throw ( ExceptionA, ExceptionB, ExceptionC )
```

immediately following the closing parenthesis of the function's parameter list, indicates that the function can throw exceptions of types `ExceptionA`, `ExceptionB` and `ExceptionC`, or their derived types. If the function `throw`s any other exception type, the exception-handling mechanism calls function **`unexpected`**, which terminates the program by default.

A function that does not provide an exception specification can `throw` *any* exception. Placing `throw()`—an empty exception specification—after a function's parameter list states that the function does not `throw` exceptions. If the function attempts to `throw` an exception, function `unexpected` is invoked.

16.6 Processing Unexpected Exceptions

Function `unexpected` calls the function registered with function **`set_unexpected`** (defined in header `<exception>`). If no function has been registered in this manner, function `terminate` is called by default. Cases in which function `terminate` is called include:

1. the exception mechanism cannot find a matching `catch` for a thrown exception
2. a destructor attempts to `throw` an exception during stack unwinding
3. an attempt is made to rethrow an exception when there's no exception currently being handled
4. a call to function `unexpected` defaults to calling function `terminate`

(Section 15.5.1 of the C++ Standard Document discusses several additional cases.) Function **`set_terminate`** can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls **`abort`**, *which terminates the program without calling the destructors of any remaining objects of automatic or static storage class.* This could lead to resource leaks when a program terminates prematurely.

Common Programming Error 16.7

*Aborting a program component due to an uncaught exception could leave a resource—such as a file stream or an I/O device—in a state in which other programs are unable to acquire the resource. This is known as a **resource leak**.*

Function `set_terminate` and function `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later. Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.

If the last action of a programmer-defined termination function is not to exit a program, function `abort` will be called to end program execution.

16.7 Stack Unwinding

When an exception is thrown but not caught in a particular scope, the function call stack is "unwound," and an attempt is made to `catch` the exception in the next outer `try…catch` block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a `try` block encloses that statement, an attempt is made to `catch` the exception. If a `try` block does not enclose that statement, stack unwinding occurs again. If no `catch` handler ever catches this exception, function `terminate` is called to terminate the program. The program of Fig. 16.4 demonstrates stack unwinding.

```cpp
// Fig. 16.4: Fig16_04.cpp
// Demonstrating stack unwinding.
#include <iostream>
#include <stdexcept>
using namespace std;

// function3 throws runtime error
void function3() throw ( runtime_error )
{
   cout << "In function 3" << endl;

   // no try block, stack unwinding occurs, return control to function2
   throw runtime_error( "runtime_error in function3" ); // no print
} // end function3

// function2 invokes function3
void function2() throw ( runtime_error )
{
   cout << "function3 is called inside function2" << endl;
   function3(); // stack unwinding occurs, return control to function1
} // end function2

// function1 invokes function2
void function1() throw ( runtime_error )
{
   cout << "function2 is called inside function1" << endl;
   function2(); // stack unwinding occurs, return control to main
} // end function1

// demonstrate stack unwinding
int main()
{
   // invoke function1
   try
   {
      cout << "function1 is called inside main" << endl;
      function1(); // call function1 which throws runtime_error
   } // end try
   catch ( runtime_error &error ) // handle runtime error
   {
      cout << "Exception occurred: " << error.what() << endl;
      cout << "Exception handled in main" << endl;
   } // end catch
} // end main
```

Fig. 16.4 | Stack unwinding. (Part 1 of 2.)

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Fig. 16.4 | Stack unwinding. (Part 2 of 2.)

In `main`, the `try` block (lines 34–38) calls `function1` (lines 24–28). Next, `function1` calls `function2` (lines 17–21), which in turn calls `function3` (lines 8–14). Line 13 of `function3` throws a `runtime_error` object. However, because no `try` block encloses the `throw` statement in line 13, stack unwinding occurs—`function3` terminates at line 13, then returns control to the statement in `function2` that invoked `function3` (i.e., line 20). Because no `try` block encloses line 20, stack unwinding occurs again—`function2` terminates at line 20 and returns control to the statement in `function1` that invoked `function2` (i.e., line 27). Because no `try` block encloses line 27, stack unwinding occurs one more time—`function1` terminates at line 27 and returns control to the statement in `main` that invoked `function1` (i.e., line 37). The `try` block of lines 34–38 encloses this statement, so the first matching `catch` handler located after this `try` block (line 39–43) catches and processes the exception. Line 41 uses function `what` to display the exception message. Recall that function `what` is a `virtual` function of class `exception` that can be overridden by a derived class to return an appropriate error message.

16.8 Constructors, Destructors and Exception Handling

First, let's discuss an issue that we've mentioned but not yet resolved satisfactorily: What happens when an error is detected in a *constructor*? For example, how should an object's constructor respond when `new` fails because it was unable to allocate required memory for storing that object's internal representation? Because the constructor cannot return a value to indicate an error, we must choose an alternative means of indicating that the object has not been constructed properly. One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it's in an inconsistent state. Another scheme is to set some variable outside the constructor. The preferred alternative is to require the constructor to `throw` an exception that contains the error information, thus offering an opportunity for the program to handle the failure.

Before an exception is thrown by a constructor, destructors are called for any member objects built as part of the object being constructed. Destructors are called for every automatic object constructed in a `try` block before an exception is thrown. Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing. If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.

If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed prior to the occurrence of the exception. If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed objects in the array will be called.

An exception could preclude the operation of code that would normally *release a resource* (such as memory or a file), thus causing a *resource leak*. One technique to resolve this problem is to initialize a local object to acquire the resource. When an exception occurs, the destructor for that object will be invoked and can free the resource.

Error-Prevention Tip 16.2

When an exception is thrown from the constructor for an object that's created in a `new` *expression, the dynamically allocated memory for that object is released.*

16.9 Exceptions and Inheritance

Various exception classes can be derived from a common base class, as we discussed in Section 16.2, when we created class `DivideByZeroException` as a derived class of class `exception`. If a `catch` handler catches a pointer or reference to an exception object of a base-class type, it also can `catch` a pointer or reference to all objects of classes publicly derived from that base class—this allows for polymorphic processing of related errors.

Error-Prevention Tip 16.3

Using inheritance with exceptions enables an exception handler to `catch` *related errors with concise notation. One approach is to* `catch` *each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to* `catch` *pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.*

16.10 Processing new Failures

The C++ standard specifies that, when operator `new` fails, it `throws` a **`bad_alloc`** exception (defined in header <new>).In this section, we present two examples of `new` failing. The first uses the version of `new` that `throws` a `bad_alloc` exception when `new` fails. The second uses function **`set_new_handler`** to handle `new` failures. [*Note:* The examples in Figs. 16.5–16.6 allocate large amounts of dynamic memory, which could cause your computer to become sluggish.]

`new` Throwing `bad_alloc` on Failure

Figure 16.5 demonstrates `new` throwing `bad_alloc` on failure to allocate the requested memory. The `for` statement (lines 16–20) inside the `try` block should loop 50 times and, on each pass, allocate an array of 50,000,000 `double` values. If `new` fails and throws a `bad_alloc` exception, the loop terminates, and the program continues in line 22, where the `catch` handler catches and processes the exception. Lines 24–25 print the message `"Exception occurred:"` followed by the message returned from the base-class-`exception` version of function `what` (i.e., an implementation-defined exception-specific message, such as `"Allocation Failure"` in Microsoft Visual C++). The output shows that the program performed only four iterations of the loop before `new` failed and threw the `bad_alloc` exception. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you're using.

```
// Fig. 16.5: Fig16_05.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>
#include <new> // bad_alloc class is defined here
using namespace std;

int main()
{
   double *ptr[ 50 ];

   // aim each ptr[i] at a big block of memory
   try
   {
      // allocate memory for ptr[ i ]; new throws bad_alloc on failure
      for ( int i = 0; i < 50; ++i )
      {
         ptr[ i ] = new double[ 50000000 ]; // may throw exception
         cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
      } // end for
   } // end try
   catch ( bad_alloc &memoryAllocationException )
   {
      cerr << "Exception occurred: "
         << memoryAllocationException.what() << endl;
   } // end catch
} // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
Exception occurred: bad allocation
```

Fig. 16.5 | `new` throwing `bad_alloc` on failure.

`new` Returning `0` on Failure

The C++ standard specifies that compilers can use an older version of `new` that returns `0` upon failure. For this purpose, header `<new>` defines object **`nothrow`** (of type `nothrow_t`), which is used as follows:

```
double *ptr = new( nothrow ) double[ 50000000 ];
```

The preceding statement uses the version of `new` that does *not* throw `bad_alloc` exceptions (i.e., `nothrow`) to allocate an array of 50,000,000 `double`s.

Software Engineering Observation 16.8

To make programs more robust, use the version of `new` that throws `bad_alloc` exceptions on failure.

Handling `new` Failures Using Function `set_new_handler`

An additional feature for handling `new` failures is function `set_new_handler` (prototyped in standard header `<new>`). This function takes as its argument a pointer to a function that takes no arguments and returns void. This pointer points to the function that will be called if new fails. This provides you with a uniform approach to handling all new failures, regardless of where a failure occurs in the program. Once set_new_handler registers a **new handler** in the program, operator new does not throw bad_alloc on failure; rather, it defers the error handling to the new-handler function.

If new allocates memory successfully, it returns a pointer to that memory. If new fails to allocate memory and set_new_handler did not register a new-handler function, new throws a bad_alloc exception. If new fails to allocate memory and a new-handler function has been registered, the new-handler function is called. The C++ standard specifies that the new-handler function should perform one of the following tasks:

1. Make more memory available by deleting other dynamically allocated memory (or telling the user to close other applications) and return to operator new to attempt to allocate memory again.
2. Throw an exception of type bad_alloc.
3. Call function abort or exit (both found in header <cstdlib>) to terminate the program.

Figure 16.6 demonstrates set_new_handler. Function customNewHandler (lines 9–13) prints an error message (line 11), then calls abort (line 12) to terminate the program. The output shows that the loop iterated four times before new failed and invoked function customNewHandler. Your output might differ based on the physical memory, disk space available for virtual memory on your system and your compiler.

```
1  // Fig. 16.6: Fig16_06.cpp
2  // Demonstrating set_new_handler.
3  #include <iostream>
4  #include <new> // set_new_handler function prototype
5  #include <cstdlib> // abort function prototype
6  using namespace std;
7
8  // handle memory allocation failure
9  void customNewHandler()
10 {
11    cerr << "customNewHandler was called";
12    abort();
13 } // end function customNewHandler
14
15 // using set_new_handler to handle failed memory allocation
16 int main()
17 {
18    double *ptr[ 50 ];
19
20    // specify that customNewHandler should be called on
21    // memory allocation failure
22    set_new_handler( customNewHandler );
23
```

Fig. 16.6 | set_new_handler specifying the function to call when new fails. (Part 1 of 2.)

```
   // aim each ptr[i] at a big block of memory; customNewHandler will be
   // called on failed memory allocation
   for ( int i = 0; i < 50; ++i )
   {
      ptr[ i ] = new double[ 50000000 ]; // may throw exception
      cout << "ptr[" << i << "] points to 50,000,000 new doubles\n";
   } // end for
} // end main
```

```
ptr[0] points to 50,000,000 new doubles
ptr[1] points to 50,000,000 new doubles
ptr[2] points to 50,000,000 new doubles
ptr[3] points to 50,000,000 new doubles
customNewHandler was called
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Fig. 16.6 | `set_new_handler` specifying the function to call when `new` fails. (Part 2 of 2.)

16.11 Class unique_ptr and Dynamic Memory Allocation[2]

A common programming practice is to allocate dynamic memory, assign the address of that memory to a pointer, use the pointer to manipulate the memory and deallocate the memory with `delete` when the memory is no longer needed. If an exception occurs after successful memory allocation but *before* the `delete` statement executes, a *memory leak* could occur. The C++ standard provides class template **`unique_ptr`** in header **`<memory>`** to deal with this situation.

An object of class `unique_ptr` maintains a pointer to dynamically allocated memory. When a `unique_ptr` object destructor is called (for example, when a `unique_ptr` object goes out of scope), it performs a `delete` operation on its pointer data member. Class template `unique_ptr` provides overloaded operators * and -> so that a `unique_ptr` object can be used just as a regular pointer variable is. Figure 16.9 demonstrates a `unique_ptr` object that points to a dynamically allocated object of class `Integer` (Figs. 16.7–16.8).

```
// Fig. 16.7: Integer.h
// Integer class definition.

class Integer
{
public:
   Integer( int i = 0 ); // Integer default constructor
   ~Integer(); // Integer destructor
   void setInteger( int i ); // functions to set Integer
   int getInteger() const; // function to return Integer
```

Fig. 16.7 | `Integer` class definition. (Part 1 of 2.)

---

2. Class `unique_ptr` is a part of the new C++ standard that's already implemented in Visual C++ 2010 and GNU C++. This class replaces the deprecated `auto_ptr` class. To compile this program in GNU C++, use the `-std=C++0x` compiler flag.

```
private:
   int value;
}; // end class Integer
```

Fig. 16.7 | Integer class definition. (Part 2 of 2.)

```
// Fig. 16.8: Integer.cpp
// Integer member function definitions.
#include <iostream>
#include "Integer.h"
using namespace std;

// Integer default constructor
Integer::Integer( int i )
   : value( i )
{
   cout << "Constructor for Integer " << value << endl;
} // end Integer constructor

// Integer destructor
Integer::~Integer()
{
   cout << "Destructor for Integer " << value << endl;
} // end Integer destructor

// set Integer value
void Integer::setInteger( int i )
{
   value = i;
} // end function setInteger

// return Integer value
int Integer::getInteger() const
{
   return value;
} // end function getInteger
```

Fig. 16.8 | Member function definitions of class Integer.

Line 15 of Fig. 16.9 creates `unique_ptr` object `ptrToInteger` and initializes it with a pointer to a dynamically allocated `Integer` object that contains the value `7`. Line 18 uses the `unique_ptr` overloaded `->` operator to invoke function `setInteger` on the `Integer` object that `ptrToInteger` manages. Line 21 uses the `unique_ptr` overloaded `*` operator to dereference `ptrToInteger`, then uses the dot (`.`) operator to invoke function `getInteger` on the `Integer` object. Like a regular pointer, a `unique_ptr`'s `->` and `*` overloaded operators can be used to access the object to which the `unique_ptr` points.

Because `ptrToInteger` is a local automatic variable in `main`, `ptrToInteger` is destroyed when `main` terminates. The `unique_ptr` destructor forces a `delete` of the `Integer` object pointed to by `ptrToInteger`, which in turn calls the `Integer` class destructor. The memory that `Integer` occupies is released, regardless of how control leaves the block (e.g., by a `return` statement or by an exception). Most importantly, using this

```
// Fig. 16.9: Fig16_09.cpp
// Demonstrating unique_ptr.
#include <iostream>
#include <memory>
using namespace std;

#include "Integer.h"

// use unique_ptr to manipulate Integer object
int main()
{
   cout << "Creating a unique_ptr object that points to an Integer\n";

   // "aim" unique_ptr at Integer object
   unique_ptr< Integer > ptrToInteger( new Integer( 7 ) );

   cout << "\nUsing the unique_ptr to manipulate the Integer\n";
   ptrToInteger->setInteger( 99 ); // use unique_ptr to set Integer value

   // use unique_ptr to get Integer value
   cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger()
} // end main
```

```
Creating a unique_ptr object that points to an Integer
Constructor for Integer 7

Using the unique_ptr to manipulate the Integer
Integer after setInteger: 99

Destructor for Integer 99
```

Fig. 16.9 | `unique_ptr` object manages dynamically allocated memory.

technique can *prevent memory leaks*. For example, suppose a function returns a pointer aimed at some object. Unfortunately, the function caller that receives this pointer might not `delete` the object, thus resulting in *a memory leak*. However, if the function returns a `unique_ptr` to the object, the object will be deleted automatically when the `unique_ptr` object's destructor gets called.

Only one `unique_ptr` at a time can own a dynamically allocated object and the object cannot be an array. By using its overloaded assignment operator or copy constructor, a `unique_ptr` can transfer ownership of the dynamic memory it manages. The last `unique_ptr` object that maintains the pointer to the dynamic memory will delete the memory. This makes `unique_ptr` an ideal mechanism for returning dynamically allocated memory to client code. When the `unique_ptr` goes out of scope in the client code, the `unique_ptr`'s destructor deletes the dynamic memory.

16.12 Standard Library Exception Hierarchy

Experience has shown that exceptions fall nicely into a number of categories. The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 16.10. As we first discussed in Section 16.2, this hierarchy is headed by base-class `ex-`

ception (defined in header <exception>), which contains virtual function what, which derived classes can override to issue appropriate error messages.

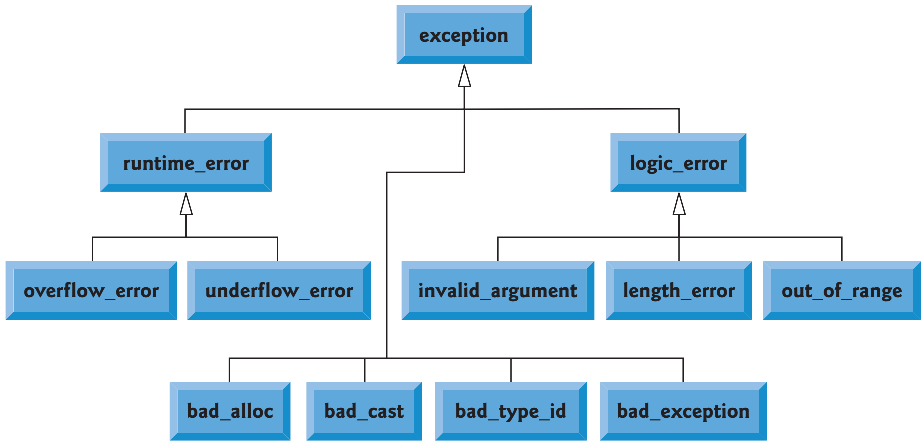


Fig. 16.10 | Some of the Standard Library exception classes.

Immediate derived classes of base-class exception include runtime_error and **logic_error** (both defined in header <stdexcept>), each of which has several derived classes. Also derived from exception are the exceptions thrown by C++ operators—for example, bad_alloc is thrown by new (Section 16.10), **bad_cast** is thrown by dynamic_cast (Chapter 13) and **bad_typeid** is thrown by typeid (Chapter 13). Including **bad_exception** in the throw list of a function means that, if an unexpected exception occurs, function unexpected can throw bad_exception rather than terminating the program's execution (by default) or calling another function specified by set_unexpected.

Common Programming Error 16.8

Placing a catch handler that catches a base-class object before a catch that catches an object of a class derived from that base class is a logic error. The base-class catch catches all objects of classes derived from that base class, so the derived-class catch will never execute.

Class logic_error is the base class of several standard exception classes that indicate errors in program logic. For example, class **invalid_argument** indicates that an invalid argument was passed to a function. (Proper coding can, of course, prevent invalid arguments from reaching a function.) Class **length_error** indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object. Class **out_of_range** indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Class runtime_error, which we used briefly in Section 16.7, is the base class of several other standard exception classes that indicate execution-time errors. For example, class **overflow_error** describes an **arithmetic overflow error** (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class

`underflow_error` describes an **arithmetic underflow error** (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

Common Programming Error 16.9

Exception classes need not be derived from class `exception`, so catching type `exception` is not guaranteed to `catch` all exceptions a program could encounter.

Error-Prevention Tip 16.4

To `catch` all exceptions potentially thrown in a `try` block, use `catch(...)`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there's no way to refer to the exception object inside the exception handler.

Software Engineering Observation 16.9

The standard `exception` hierarchy is a good starting point for creating exceptions. You can build programs that can `throw` standard exceptions, `throw` exceptions derived from the standard exceptions or `throw` your own exceptions not derived from the standard exceptions.

Software Engineering Observation 16.10

Use `catch(...)` to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing `catch` handlers.

16.13 Wrap-Up

In this chapter, you learned how to use exception handling to deal with errors in a program. You learned that exception handling enables you to remove error-handling code from the "main line" of the program's execution. We demonstrated exception handling in the context of a divide-by-zero example. We reviewed how to use `try` blocks to enclose code that may throw an exception, and how to use `catch` handlers to deal with exceptions that may arise. You learned how to throw and rethrow exceptions, and how to handle the exceptions that occur in constructors. The chapter continued with discussions of processing `new` failures, dynamic memory allocation with class `unique_ptr` and the standard library exception hierarchy. In the next chapter, you'll learn about file processing, including how persistent data is stored and how to manipulate it.

Summary

Section 16.1 Introduction

- An exception (p. 633) is an indication of a problem that occurs during a program's execution.
- Exception handling enables you to create programs that can resolve problems that occur at execution time—often allowing programs to continue executing as if no problems had been encountered. More severe problems may require a program to notify the user of the problem before terminating in a controlled manner.

Section 16.2 Example: Handling an Attempt to Divide by Zero

- Class `exception` is the standard base class for exceptions classes (p. 634). It provides virtual function `what` (p. 634) that returns an appropriate error message and can be overridden in derived classes.
- Class `runtime_error` (p. 634), which is defined in header <stdexcept> (p. 634), is the C++ standard base class for representing runtime errors.
- C++ uses the termination model (p. 637) of exception handling.
- A `try` block consists of keyword `try` followed by braces (`{}`) that define a block of code in which exceptions might occur. The `try` block encloses statements that might cause exceptions and statements that should not execute if exceptions occur.
- At least one `catch` handler must immediately follow a `try` block. Each `catch` handler specifies an exception parameter that represents the type of exception the `catch` handler can process.
- If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with a caught exception object (p. 638).
- The point in the program at which an exception occurs is called the throw point (p. 637).
- If an exception occurs in a `try` block, the `try` block expires and program control transfers to the first `catch` in which the exception parameter's type matches that of the thrown exception.
- When a `try` block terminates, local variables defined in the block go out of scope.
- When a `try` block terminates due to an exception, the program searches for the first `catch` handler that matches the type of exception that occurred. A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type. When a match occurs, the code contained within the matching `catch` handler executes.
- When a `catch` handler finishes processing, the `catch` parameter and local variables defined within the `catch` handler go out of scope. Any remaining `catch` handlers that correspond to the `try` block are ignored, and execution resumes at the first line of code after the `try…catch` sequence.
- If no exceptions occur in a `try` block, the program ignores the `catch` handler(s) for that block. Program execution resumes with the next statement after the `try…catch` sequence.
- If an exception that occurs in a `try` block has no matching `catch` handler, or if an exception occurs in a statement that is not in a `try` block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function. This process is called stack unwinding (p. 638).
- To throw an exception, use keyword `throw` followed by an operand that represents the type of exception to throw. The operand of a `throw` can be of any type.

Section 16.3 When to Use Exception Handling

- Exception handling is for synchronous errors (p. 639), which occur when a statement executes.
- Exception handling is not designed to process errors associated with asynchronous events (p. 639), which occur in parallel with, and independent of, the program's flow of control.

Section 16.4 Rethrowing an Exception

- The exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, the handler achieves this by rethrowing the exception (p. 640).
- Common examples of exceptions are out-of-range array subscripts, arithmetic overflow, division by zero, invalid function parameters and unsuccessful memory allocations.

Section 16.5 Exception Specifications

- An optional exception specification enumerates a list of exceptions that a function can throw. A function can throw only exceptions of the types indicated by the exception specification or exceptions of any type derived from these types. If the function throws any other type of exception, function `unexpected` (p. 642) is called and the program terminates.

- A function with no exception specification can throw any exception. The empty exception specification `throw()` indicates that a function does not throw exceptions. If a function with an empty exception specification attempts to throw an exception, function `unexpected` is invoked.

Section 16.6 Processing Unexpected Exceptions

- Function `unexpected` calls the function registered with function `set_unexpected` (p. 642). If no function has been registered in this manner, function `terminate` (p. 640) is called by default.
- Function `set_terminate` (p. 642) can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls `abort` (p. 642), which terminates the program without calling the destructors of objects that are declared `static` and `auto`.
- Functions `set_terminate` and `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables you to save the function pointer so it can be restored later.
- Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.
- If a programmer-defined termination function does not exit a program, function `abort` will be called after the programmer-defined termination function completes execution.

Section 16.7 Stack Unwinding

- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.

Section 16.8 Constructors, Destructors and Exception Handling

- Exceptions thrown by a constructor cause destructors to be called for any objects built as part of the object being constructed before the exception is thrown.
- Each automatic object constructed in a `try` block is destructed before an exception is thrown.
- Stack unwinding completes before an exception handler begins executing.
- If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.
- If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed before the exception occurs.
- If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed array element objects will be called.
- When an exception is thrown from the constructor for an object that is created in a `new` expression, the dynamically allocated memory for that object is released.

Section 16.9 Exceptions and Inheritance

- If a `catch` handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors.

Section 16.10 Processing `new` Failures

- The C++ standard document specifies that, when operator `new` fails, it throws a `bad_alloc` exception (p. 645), which is defined in header `<new>`.

- Function `set_new_handler` (p. 645) takes as its argument a pointer to a function that takes no arguments and returns `void`. This pointer points to the function that will be called if `new` fails.
- Once `set_new_handler` registers a `new` handler (p. 647) in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new`-handler function.
- If `new` allocates memory successfully, it returns a pointer to that memory.
- If an exception occurs after successful memory allocation but before the `delete` statement executes, a memory leak could occur.

Section 16.11 Class `unique_ptr` and Dynamic Memory Allocation

- The C++ Standard Library provides class template `unique_ptr` (p. 648) to deal with memory leaks.
- An object of class `unique_ptr` maintains a pointer to dynamically allocated memory. A `unique_ptr`'s destructor performs a `delete` operation on the `unique_ptr`'s pointer data member.
- Class template `unique_ptr` provides overloaded operators `*` and `->` so that a `unique_ptr` object can be used just as a regular pointer variable is. A `unique_ptr` also transfers ownership of the dynamic memory it manages via its copy constructor and overloaded assignment operator.

Section 16.12 Standard Library Exception Hierarchy

- The C++ Standard Library includes a hierarchy of exception classes. This hierarchy is headed by base-class `exception`.
- Immediate derived classes of base class `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes.
- Several operators throw standard exceptions—operator `new` throws `bad_alloc`, operator `dynamic_cast` throws `bad_cast` (p. 651) and operator `typeid` throws `bad_typeid` (p. 651).
- Including `bad_exception` (p. 651) in the throw list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program's execution or calling another function specified by `set_unexpected`.

Self-Review Exercises

16.1 List five common examples of exceptions.

16.2 Give several reasons why exception-handling techniques should not be used for conventional program control.

16.3 Why are exceptions appropriate for dealing with errors produced by library functions?

16.4 What's a "resource leak"?

16.5 If no exceptions are thrown in a `try` block, where does control proceed to after the `try` block completes execution?

16.6 What happens if an exception is thrown outside a `try` block?

16.7 Give a key advantage and a key disadvantage of using `catch(...)`.

16.8 What happens if no `catch` handler matches the type of a thrown object?

16.9 What happens if several handlers match the type of the thrown object?

16.10 Why would you specify a base-class type as the type of a `catch` handler, then `throw` objects of derived-class types?

16.11 Suppose a `catch` handler with a precise match to an exception object type is available. Under what circumstances might a different handler be executed for exception objects of that type?

16.12 Must throwing an exception cause program termination?

16.13 What happens when a `catch` handler `throws` an exception?

16.14 What does the statement `throw;` do?

Answers to Self-Review Exercises

16.1 Insufficient memory to satisfy a `new` request, array subscript out of bounds, arithmetic overflow, division by zero, invalid function parameters.

16.2 (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally. (b) Flow of control with conventional control structures generally is clearer and more efficient than with exceptions. (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception might not be freed. (d) The "additional" exceptions make it more difficult for you to handle the larger number of exception cases.

16.3 It's unlikely that a library function will perform error processing that will meet the unique needs of all users.

16.4 A program that terminates abruptly could leave a resource in a state in which other programs would not be able to acquire the resource, or the program itself might not be able to reacquire a "leaked" resource.

16.5 The exception handlers (in the `catch` handlers) for that `try` block are skipped, and the program resumes execution after the last `catch` handler.

16.6 An exception thrown outside a `try` block causes a call to `terminate`.

16.7 The form `catch(...)` catches any type of exception thrown in a `try` block. An advantage is that all possible exceptions will be caught. A disadvantage is that the `catch` has no parameter, so it cannot reference information in the thrown object and cannot know the cause of the exception.

16.8 This causes the search for a match to continue in the next enclosing `try` block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches the type of the thrown object; in this case, `terminate` is called, which by default calls `abort`. An alternative `terminate` function can be provided as an argument to `set_terminate`.

16.9 The first matching exception handler after the `try` block is executed.

16.10 This is a nice way to `catch` related types of exceptions.

16.11 A base-class handler would catch objects of all derived-class types.

16.12 No, but it does terminate the block in which the exception is thrown.

16.13 The exception will be processed by a `catch` handler (if one exists) associated with the `try` block (if one exists) enclosing the `catch` handler that caused the exception.

16.14 It rethrows the exception if it appears in a `catch` handler; otherwise, function `unexpected` is called.

Exercises

16.15 *(Exceptional Conditions)* List various exceptional conditions that have occurred throughout this text. List as many additional exceptional conditions as you can. For each of these exceptions, describe briefly how a program typically would handle the exception, using the exception-handling techniques discussed in this chapter. Some typical exceptions are division by zero, arithmetic overflow, array subscript out of bounds, exhaustion of the free store, etc.

16.16 *(Catch Parameter)* Under what circumstances would you not provide a parameter name when defining the type of the object that will be caught by a handler?

16.17 *(`throw` Statement)* A program contains the statement

```
throw;
```

Where would you normally expect to find such a statement? What if that statement appeared in a different part of the program?

16.18 *(Exception Handling vs. Other Schemes)* Compare and contrast exception handling with the various other error-processing schemes discussed in the text.

16.19 *(Exception Handling and Program Control)* Why should exceptions *not* be used as an alternate form of program control?

16.20 *(Handling Related Exceptions)* Describe a technique for handling related exceptions.

16.21 *(Throwing Exceptions from a `catch`)* Suppose a program `throw`s an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself `throw`s the same exception. Does this create infinite recursion? Write a program to check your observation.

16.22 *(Catching Derived-Class Exceptions)* Use inheritance to create various derived classes of `runtime_error`. Then show that a `catch` handler specifying the base class can `catch` derived-class exceptions.

16.23 *(Throwing the Result of a Conditional Expression)* Throw the result of a conditional expression that returns either a `double` or an `int`. Provide an `int catch` handler and a `double catch` handler. Show that only the `double catch` handler executes, regardless of whether the `int` or the `double` is returned.

16.24 *(Local Variable Destructors)* Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block.

16.25 *(Member Object Destructors)* Write a program illustrating that member object destructors are called for only those member objects that were constructed before an exception occurred.

16.26 *(Cathing All Exceptions)* Write a program that demonstrates several exception types being caught with the `catch(...)` exception handler.

16.27 *(Order of Exception Handlers)* Write a program illustrating that the order of exception handlers is important. The first matching handler is the one that executes. Attempt to compile and run your program two different ways to show that two different handlers execute with two different effects.

16.28 *(Constructors Throwing Exceptions)* Write a program that shows a constructor passing information about constructor failure to an exception handler after a `try` block.

16.29 *(Rethrowing Exceptions)* Write a program that illustrates rethrowing an exception.

16.30 *(Uncaught Exceptions)* Write a program that illustrates that a function with its own `try` block does not have to catch every possible error generated within the `try`. Some exceptions can slip through to, and be handled in, outer scopes.

16.31 *(Stack Unwinding)* Write a program that `throw`s an exception from a deeply nested function and still has the `catch` handler following the `try` block enclosing the initial call in `main` catch the exception.

17 File Processing

I read part of it all the way through.
—Samuel Goldwyn

A great memory does not make a philosopher,
any more than a dictionary can be called grammar.
—John Henry, Cardinal Newman

I can only assume that a "Do Not File" document is filed in a "Do Not File" file.
—Senator Frank Church
Senate Intelligence Subcommittee Hearing, 1975

Objectives

In this chapter you'll learn:

- To create, read, write and update files.
- Sequential file processing.
- Random-access file processing.
- To use high-performance unformatted I/O operations.
- The differences between formatted-data and raw-data file processing.
- To build a transaction-processing program using random-access file processing.
- To understand the concept of object serialization.

Outline

17.1 Introduction

Storage of data in memory is temporary. **Files** are used for **data persistence**—permanent retention of data. Computers store files on **secondary storage devices,** such as hard disks, CDs, DVDs, flash drives and tapes. In this chapter, we explain how to build C++ programs that create, update and process data files. We consider both sequential files and random-access files. We compare formatted-data file processing and raw-data file processing. We examine techniques for input of data from, and output of data to, `string` streams rather than files in Chapter 18, Class `string` and String Stream Processing.

17.2 Files and Streams

C++ views each file simply as *a sequence of bytes* (Fig. 17.1). Each file ends either with an **end-of-file marker** or at a specific byte number recorded in an operating-system-maintained, administrative data structure. When a file is *opened*, an object is created, and a stream is associated with the object. In Chapter 15, we saw that objects `cin`, `cout`, `cerr` and `clog` are created when `<iostream>` is included. The streams associated with these objects provide communication channels between a program and a particular file or device. For example, the `cin` object (standard input stream object) enables a program to input data from the keyboard or from other devices, the `cout` object (standard output stream object) enables a program to output data to the screen or other devices, and the `cerr` and `clog` objects (standard error stream objects) enable a program to output error messages to the screen or other devices.

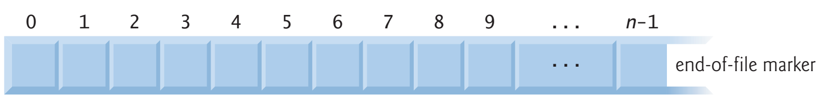


Fig. 17.1 | C++'s simple view of a file of *n* bytes.

To perform file processing in C++, headers `<iostream>` and `<fstream>` must be included. Header `<fstream>` includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input *and* output). Each class template has a predefined template specialization that enables `char` I/O. In addition, the `<fstream>` library provides `typedef` aliases for these template specializations. For example, the `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file. Similarly, `typedef ofstream` represents a specialization of `basic_ofstream` that enables `char` output to files. Also, `typedef fstream` represents a specialization of `basic_fstream` that enables `char` input from, *and* output to, files.

Files are *opened* by creating objects of these stream template specializations. These templates derive from the class templates `basic_istream`, `basic_ostream` and `basic_iostream`, respectively. Thus, all member functions, operators and manipulators that belong to these templates (which we described in Chapter 15) also can be applied to file streams. Figure 17.2 summarizes the inheritance relationships of the I/O classes that we've discussed to this point.

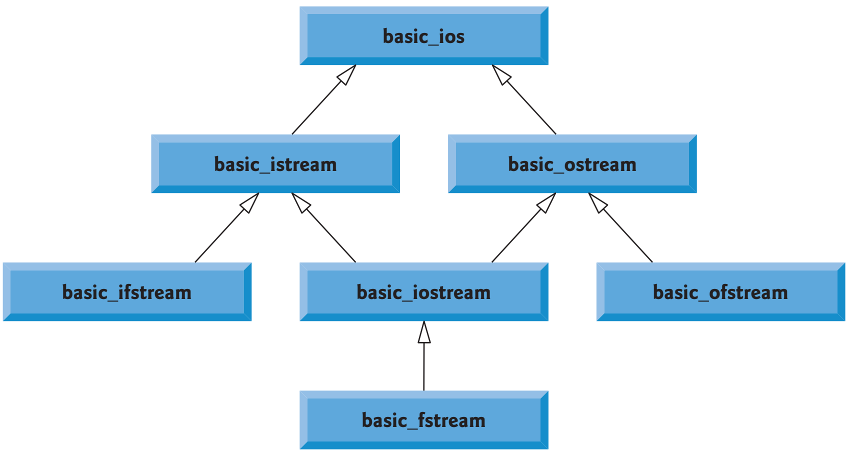


Fig. 17.2 | Portion of stream I/O template hierarchy.

17.3 Creating a Sequential File

C++ imposes no structure on a file. Thus, a concept like that of a "record" does not exist in a C++ file. You must structure files to meet the application's requirements. The following example shows how you can impose a simple record structure on a file.

Figure 17.3 creates a sequential file that might be used in an accounts-receivable system to help manage the money owed to a company by its credit clients. For each client, the program obtains the client's account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a *record* for that client. The account number serves as the record key; that is, the program creates and maintains the records of the file in account number order. This program assumes the user enters the records in account number order. In a comprehensive accounts receivable system, a sorting capability would be provided for the user to enter records *in any* order—the records then would be *sorted* and written to the file.

```
1  // Fig. 17.3: Fig17_03.cpp
2  // Create a sequential file.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // file stream
6  #include <cstdlib>
7  using namespace std;
8
9  int main()
10 {
11    // ofstream constructor opens file
12    ofstream outClientFile( "clients.txt", ios::out );
13
14    // exit program if unable to create file
15    if ( !outClientFile ) // overloaded ! operator
16    {
17       cerr << "File could not be opened" << endl;
18       exit( 1 );
19    } // end if
20
21    cout << "Enter the account, name, and balance." << endl
22       << "Enter end-of-file to end input.\n? ";
23
24    int account;
25    string name;
26    double balance;
27
28    // read account, name and balance from cin, then place in file
29    while ( cin >> account >> name >> balance )
30    {
31       outClientFile << account << ' ' << name << ' ' << balance << endl;
32       cout << "? ";
33    } // end while
34 } // end main
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 17.3 | Creating a sequential file.

Let's examine this program. As stated previously, files are opened by creating `ifstream`, `ofstream` or `fstream` objects. In Fig. 17.3, the file is to be opened for output, so an `ofstream` object is created. Two arguments are passed to the object's constructor—the **filename** and the **file-open mode** (line 12). For an `ofstream` object, the file-open mode can be either **`ios::out`** to *output* data to a file or **`ios::app`** to *append* data to the end of a file (without modifying any data already in the file). Existing files opened with mode `ios::out` are **truncated**—all data in the file is *discarded*. If the specified file does not yet exist, then the `ofstream` object *creates* the file, using that filename.

Line 12 creates an `ofstream` object named `outClientFile` associated with the file `clients.dat` that's opened for output. The arguments `"clients.dat"` and `ios::out` are passed to the `ofstream` constructor, which opens the file—this establishes a "line of communication" with the file. By default, `ofstream` objects are opened for output, so line 12 could have used the alternate statement

```
ofstream outClientFile( "clients.dat" );
```

to open `clients.dat` for output. Figure 17.4 lists the file-open modes. These modes can also be combined, as we discuss in Section 17.8.

Common Programming Error 17.1

Use caution when opening an existing file for output (`ios::out`), especially when you want to preserve the file's contents, which will be discarded without warning.

Mode	Description
`ios::app`	*Append* all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written *anywhere* in the file.
`ios::in`	Open a file for *input*.
`ios::out`	Open a file for *output*.
`ios::trunc`	*Discard* the file's contents (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary, i.e., *nontext*, input or output.

Fig. 17.4 | File open modes.

An `ofstream` object can be created *without* opening a specific file—a file can be attached to the object later. For example, the statement

```
ofstream outClientFile;
```

creates an `ofstream` object named `outClientFile`. The `ofstream` member function **`open`** opens a file and attaches it to an existing `ofstream` object as follows:

```
outClientFile.open( "clients.dat", ios::out );
```

After creating an `ofstream` object and attempting to open it, the program tests whether the open operation was successful. The `if` statement in lines 15–19 uses the overloaded `ios` member function `operator!` to determine whether the `open` operation succeeded. The condition returns a `true` value if either the `failbit` or the `badbit` is set for the stream on the `open` operation. Some possible errors are attempting to open a nonexistent file for reading, attempting to open a file for reading or writing from a directory that you don't have permission to access, and opening a file for writing when no disk space is available.

If the condition indicates an unsuccessful attempt to open the file, line 17 outputs the error message `"File could not be opened"`, and line 18 invokes function `exit` to terminate the program. The argument to `exit` is returned to the environment from which the program was invoked. Argument `0` indicates that the program terminated *normally*; any other value indicates that the program terminated due to an *error*. The calling environment (most likely the operating system) uses the value returned by `exit` to respond appropriately to the error.

Another overloaded `ios` member function—`operator void *`—converts the stream to a pointer, so it can be tested as `0` (i.e., the null pointer) or nonzero (i.e., any other pointer value). When a pointer value is used as a condition, C++ interprets a null pointer in a condition as the `bool` value `false` and interprets a non-null pointer as the `bool` value `true`. If the `failbit` or `badbit` (see Chapter 15) has been set for the stream, `0` (`false`) is returned. The condition in the `while` statement of lines 29–33 invokes the `operator void *` member function on `cin` *implicitly*. The condition remains `true` as long as neither the `failbit` nor the `badbit` has been set for `cin`. Entering the end-of-file indicator sets the `failbit` for `cin`. The `operator void *` function can be used to test an input object for end-of-file instead of calling the `eof` member function explicitly on the input object.

If line 12 opens the file successfully, the program begins processing data. Lines 21–22 prompt the user to enter either the various fields for each record or the end-of-file indicator when data entry is complete. Figure 17.5 lists the keyboard combinations for entering end-of-file for various computer systems.

Computer system	Keyboard combination
UNIX/Linux/Mac OS X	<*Ctrl-d*> (on a line by itself)
Microsoft Windows	<*Ctrl-z*> (sometimes followed by pressing *Enter*)

Fig. 17.5 | End-of-file key combinations for various popular computer systems.

Line 29 extracts each set of data and determines whether end-of-file has been entered. When end-of-file is encountered or bad data is entered, `operator void *` returns the null pointer (which converts to the `bool` value `false`) and the `while` statement terminates. The user enters end-of-file to inform the program to process no additional data. The end-of-file indicator is set when the user enters the end-of-file key combination. The `while` statement loops until the end-of-file indicator is set.

Line 31 writes a set of data to the file `clients.txt`, using the stream insertion operator `<<` and the `outClientFile` object associated with the file at the beginning of the program. The data may be retrieved by a program designed to read the file (see Section 17.4). The file created in Fig. 17.3 is simply a text file, so it can be viewed by any text editor.

Once the user enters the end-of-file indicator, `main` terminates. This implicitly invokes `outClientFile`'s destructor, which closes the `clients.txt` file. You also can close the `ofstream` object explicitly, using member function **`close`** in the statement

```
outClientFile.close();
```

In the sample execution for the program of Fig. 17.3, the user enters information for five accounts, then signals that data entry is complete by entering end-of-file (`^Z` is displayed for Microsoft Windows). This dialog window does *not* show how the data records appear in the file. To verify that the program created the file successfully, the next section shows how to create a program that reads this file and prints its contents.

17.4 Reading Data from a Sequential File

Files store data so it may be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. We now discuss how to read data sequentially from a file. Figure 17.6 reads and displays the records from the `clients.txt` file that we created using the program of Fig. 17.3. Creating an `ifstream` object opens a file for input. The `ifstream` constructor can receive the filename and the file open mode as arguments. Line 15 creates an `ifstream` object called `inClientFile` and associates it with the `clients.txt` file. The arguments in parentheses are passed to the `ifstream` constructor, which opens the file and establishes a "line of communication" with the file.

Good Programming Practice 17.1

Open a file for input only (using `ios::in`*) if the file's contents should* not *be modified. This prevents unintentional modification of the file's contents and is another example of the principle of least privilege.*

```
1  // Fig. 17.6: Fig17_06.cpp
2  // Reading and printing a sequential file.
3  #include <iostream>
4  #include <fstream> // file stream
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine( int, const string, double ); // prototype
11
12 int main()
13 {
14    // ifstream constructor opens the file
15    ifstream inClientFile( "clients.txt", ios::in );
16
17    // exit program if ifstream could not open file
18    if ( !inClientFile )
19    {
20       cerr << "File could not be opened" << endl;
21       exit( 1 );
22    } // end if
23
24    int account;
25    string name;
26    double balance;
27
28    cout << left << setw( 10 ) << "Account" << setw( 13 )
29       << "Name" << "Balance" << endl << fixed << showpoint;
30
31    // display each record in file
         inClientFile >> account >> name >> balance
33       outputLine( account, name, balance );
34 } // end main
```

Fig. 17.6 | Reading and printing a sequential file. (Part 1 of 2.)

```
35
36   // display single record from file
37   void outputLine( int account, const string name, double balance )
38   {
39      cout << left << setw( 10 ) << account << setw( 13 ) << name
40         << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
41   } // end function outputLine
```

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
```

Fig. 17.6 | Reading and printing a sequential file. (Part 2 of 2.)

Objects of class `ifstream` are opened for *input* by default, so the statement

```
ifstream inClientFile( "clients.txt" );
```

opens `clients.txt` for input. Just as with an `ofstream` object, an `ifstream` object can be created without opening a specific file, because a file can be attached to it later.

Before attempting to retrieve data from the file, the program uses the condition `!inClientFile` to determine whether the file was opened successfully. Line 32 reads a set of data (i.e., a record) from the file. After line 32 executes the first time, `account` has the value `100`, `name` has the value `"Jones"` and `balance` has the value `24.98`. Each time line 32 executes, it reads another record from the file into the variables `account`, `name` and `balance`. Line 33 displays the records, using function `outputLine` (lines 37–41), which uses parameterized stream manipulators to format the data for display. When the end of file has been reached, the *implicit call to* `operator void *` *in the* `while` *condition* returns the null pointer (which converts to the `bool` value `false`), the `ifstream` destructor closes the file and the program terminates.

To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Both `istream` and `ostream` provide member functions for *repositioning the* ***file-position pointer*** (the byte number of the next byte in the file to be read or written). These member functions are **`seekg`** ("seek get") for `istream` and **`seekp`** ("seek put") for `ostream`. Each `istream` object has a *get pointer*, which indicates the byte number in the file from which the next input is to occur, and each `ostream` object has a *put pointer*, which indicates the byte number in the file at which the next output should be placed. The statement

```
inClientFile.seekg( 0 );
```

repositions the file-position pointer to the beginning of the file (location `0`) attached to `inClientFile`. The argument to `seekg` is a `long` integer. A second argument can be specified to indicate the **seek direction**, which can be **`ios::beg`** (the default) for positioning relative to the *beginning* of a stream, **`ios::cur`** for positioning relative to the *current position* in a stream or **ios::end** for positioning relative to the end of a stream. The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location (this is also referred to as the **offset** from the beginning of the file). Some examples of positioning the *get* file-position pointer are

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

The same operations can be performed using `ostream` member function `seekp`. Member functions **tellg** and **tellp** are provided to return the current locations of the *get* and *put* pointers, respectively. The following statement assigns the *get* file-position pointer value to variable `location` of type `long`:

```
location = fileObject.tellg();
```

Figure 17.7 enables a credit manager to display the account information for those customers with zero balances (i.e., customers who do not owe the company any money), credit (negative) balances (i.e., customers to whom the company owes money), and debit (positive) balances (i.e., customers who owe the company money for goods and services received in the past). The program displays a menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 produces a list of accounts with zero balances. Option 2 produces a list of accounts with credit balances. Option 3 produces a list of accounts with debit balances. Option 4 terminates program execution. Entering an invalid option displays the prompt to enter another choice. Lines 65–66 enable the program to read from the beginning of the file after the EOF marker has been read.

```
1  // Fig. 17.7: Fig17_07.cpp
2  // Credit inquiry program.
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
11 int getRequest();
12 bool shouldDisplay( int, double );
13 void outputLine( int, const string, double );
14
15 int main()
16 {
17    // ifstream constructor opens the file
18    ifstream inClientFile( "clients.txt", ios::in );
```

Fig. 17.7 | Credit inquiry program. (Part 1 of 4.)

```

   // exit program if ifstream could not open file
   if ( !inClientFile )
   {
      cerr << "File could not be opened" << endl;
      exit( 1 );
   } // end if

   int request;
   int account;
   string name;
   double balance;

   // get user's request (e.g., zero, credit or debit balance)
   request = getRequest();

   // process user's request
   while ( request != END )
   {
      switch ( request )
      {
         case ZERO_BALANCE:
            cout << "\nAccounts with zero balances:\n";
            break;
         case CREDIT_BALANCE:
            cout << "\nAccounts with credit balances:\n";
            break;
         case DEBIT_BALANCE:
            cout << "\nAccounts with debit balances:\n";
            break;
      } // end switch

      // read account, name and balance from file
      inClientFile >> account >> name >> balance;

      // display file contents (until eof)
      while ( !inClientFile.eof() )
      {
         // display record
         if ( shouldDisplay( request, balance ) )
            outputLine( account, name, balance );

         // read account, name and balance from file
         inClientFile >> account >> name >> balance;
      } // end inner while

      inClientFile.clear(); // reset eof for next input
      inClientFile.seekg( 0 ); // reposition to beginning of file
      request = getRequest(); // get additional request from user
   } // end outer while

   cout << "End of run." << endl;
} // end main
```

Fig. 17.7 | Credit inquiry program. (Part 2 of 4.)

```

// obtain request from user
int getRequest()
{
   int request; // request from user

   // display request options
   cout << "\nEnter request" << endl
      << " 1 - List accounts with zero balances" << endl
      << " 2 - List accounts with credit balances" << endl
      << " 3 - List accounts with debit balances" << endl
      << " 4 - End of run" << fixed << showpoint;

   do // input user request
   {
      cout << "\n? ";
      cin >> request;
   } while ( request < ZERO_BALANCE && request > END );

   return request;
} // end function getRequest

// determine whether to display given record
bool shouldDisplay( int type, double balance )
{
   // determine whether to display zero balances
   if ( type == ZERO_BALANCE && balance == 0 )
      return true;

   // determine whether to display credit balances
   if ( type == CREDIT_BALANCE && balance < 0 )
      return true;

   // determine whether to display debit balances
   if ( type == DEBIT_BALANCE && balance > 0 )
      return true;

   return false;
} // end function shouldDisplay

// display single record from file
void outputLine( int account, const string name, double balance )
{
   cout << left << setw( 10 ) << account << setw( 13 ) << name
      << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
} // end function outputLine
```

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1
```

Fig. 17.7 | Credit inquiry program. (Part 3 of 4.)

```
Accounts with zero balances:
300       White           0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 2

Accounts with credit balances:
400       Stone         -42.16

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 4
End of run.
```

Fig. 17.7 | Credit inquiry program. (Part 4 of 4.)

17.5 Updating Sequential Files

Data that is formatted and written to a sequential file as shown in Section 17.3 cannot be modified without the risk of destroying other data in the file. For example, if the name "`White`" needs to be changed to "`Worthington`," the old name cannot be overwritten without corrupting the file. The record for `White` was written to the file as

```
300 White 0.00
```

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

```
300 Worthington 0.00
```

The new record contains six more characters than the original record. Therefore, the characters beyond the second "`o`" in "`Worthington`" would overwrite the beginning of the next sequential record in the file. The problem is that, in the formatted input/output model using the stream insertion operator << and the stream extraction operator >>, fields—and hence records—can vary in size. For example, values 7, 14, –117, 2074, and 27383 are all `int`s, which store the same number of "raw data" bytes internally (typically four bytes on today's popular 32-bit machines). However, these integers become different-sized fields when output as formatted text (character sequences). Therefore, the formatted input/output model usually is not used to update records in place.

Such updating can be done awkwardly. For example, to make the preceding name change, the records before `300 White 0.00` in a sequential file could be copied to a new file, the updated record then written to the new file, and the records after `300 White 0.00` copied to the new file. This requires processing *every* record in the file to update one record. If many records are being updated in one pass of the file, though, this technique can be acceptable.

17.6 Random-Access Files

So far, we've seen how to create sequential files and search them to locate information. Sequential files are inappropriate for **instant-access applications**, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of **transaction-processing systems** that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with **random-access files.** Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.

As we've said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the *same fixed length*. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. We'll soon see how this facilitates immediate access to specific records, even in large files.

Figure 17.8 illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long). A random-access file is like a railroad train with many same-size cars—some empty and some with contents.

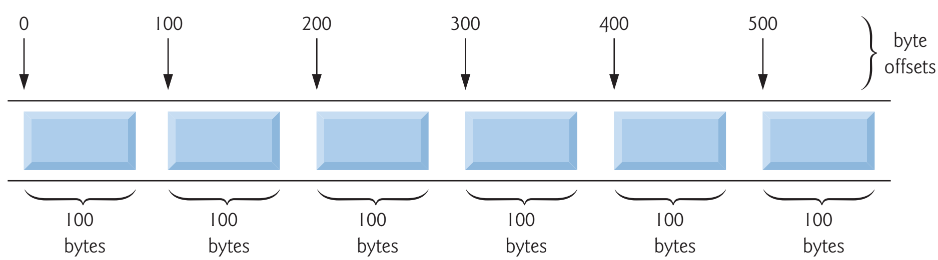


Fig. 17.8 | C++ view of a random-access file.

Data can be inserted into a random-access file without destroying other data in the file. Data stored previously also can be updated or deleted without rewriting the entire file. In the following sections, we explain how to create a random-access file, enter data into the file, read the data both sequentially and randomly, update the data and delete data that is no longer needed.

17.7 Creating a Random-Access File

The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. When the stream is associated with a file, function `write` writes the data *at the location in the file specified by the put file-position pointer*. The `istream` member function **`read`** inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function `read` inputs bytes at the location in the file specified by the "get" file-position pointer.

Writing Bytes with `ostream` *Member Function* `write`

When writing the integer `number` to a file, instead of using the statement

```
outFile << number;
```

which for a four-byte integer could print as few digits as one or as many as 11 (10 digits plus a sign, each requiring a single byte of storage), we can use the statement

```
outFile.write( reinterpret_cast< const char * >( &number ),
   sizeof( number ) );
```

which always writes the binary version of the integer `number`'s *four* bytes (on a machine with four-byte integers). Function `write` treats its first argument as a group of bytes by viewing the object in memory as a `const char *`, which is a pointer to a byte. Starting from that location, function `write` outputs the number of bytes specified by its second argument—an integer of type `size_t`. As we'll see, `istream` function `read` can subsequently be used to read the four bytes back into integer variable `number`.

Converting Between Pointer Types with the `reinterpret_cast` *Operator*

Unfortunately, most pointers that we pass to function `write` as the first argument are *not* of type `const char *`. To output objects of other types, we must convert the pointers to those objects to type `const char *`; otherwise, the compiler will not compile calls to function `write`. C++ provides the **`reinterpret_cast`** operator for cases like this in which a pointer of one type must be cast to an *unrelated* pointer type. Without a `reinterpret_cast`, the `write` statement that outputs the integer `number` will not compile because the compiler does not allow a pointer of type `int *` (the type returned by the expression `&number`) to be passed to a function that expects an argument of type `const char *`—as far as the compiler is concerned, these types are inconsistent.

A `reinterpret_cast` is performed at *compile time* and does *not* change the value of the object to which its operand points. Instead, it requests that the compiler reinterpret the operand as the target type (specified in the angle brackets following the keyword `reinterpret_cast`). In Fig. 17.11, we use `reinterpret_cast` to convert a `ClientData` pointer to a `const char *`, which reinterprets a `ClientData` object as bytes to be output to a file. Random-access file-processing programs rarely write a single field to a file. Typically, they write one object of a class at a time, as we show in the following examples.

Error-Prevention Tip 17.1

It's easy to use `reinterpret_cast` *to perform dangerous manipulations that could lead to serious execution-time errors.*

Portability Tip 17.1

Using `reinterpret_cast` *is compiler dependent and can cause programs to behave differently on different platforms. The* `reinterpret_cast` *operator should not be used unless absolutely necessary.*

Portability Tip 17.2

A program that reads unformatted data (written by `write`*) must be compiled and executed on a system compatible with the program that wrote the data, because different systems may represent internal data differently.*

Credit Processing Program

Consider the following problem statement:

> *Create a credit-processing program capable of storing at most 100 fixed-length records for a company that can have up to 100 customers. Each record should consist of an account number that acts as the record key, a last name, a first name and a balance. The program should be able to update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.*

The next several sections introduce the techniques for creating this credit-processing program. Figure 17.11 illustrates opening a random-access file, defining the record format using an object of class `ClientData` (Figs. 17.9–17.10) and writing data to the disk in *binary* format. This program initializes all 100 records of the file `credit.dat` with *empty* objects, using function `write`. Each empty object contains `0` for the account number, the null string (represented by empty quotation marks) for the last and first name and `0.0` for the balance. Each record is initialized with the amount of empty space in which the account data will be stored.

```
// Fig. 17.9: ClientData.h
// Class ClientData definition used in Fig. 17.11-Fig. 17.14.
#ifndef CLIENTDATA_H
#define CLIENTDATA_H

#include <string>
using namespace std;

class ClientData
{
public:
   // default ClientData constructor
   ClientData( int = 0, string = "", string = "", double = 0.0 );

   // accessor functions for accountNumber
   void setAccountNumber( int );
```

Fig. 17.9 | `ClientData` class header. (Part 1 of 2.)

```
   int getAccountNumber() const;

   // accessor functions for lastName
   void setLastName( string );
   string getLastName() const;

   // accessor functions for firstName
   void setFirstName( string );
   string getFirstName() const;

   // accessor functions for balance
   void setBalance( double );
   double getBalance() const;
private:
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   double balance;
}; // end class ClientData

#endif
```

Fig. 17.9 | `ClientData` class header. (Part 2 of 2.)

```
// Fig. 17.10: ClientData.cpp
// Class ClientData stores customer's credit information.
#include <string>
#include "ClientData.h"
using namespace std;

// default ClientData constructor
ClientData::ClientData( int accountNumberValue,
   string lastNameValue, string firstNameValue, double balanceValue )
{
   setAccountNumber( accountNumberValue );
   setLastName( lastNameValue );
   setFirstName( firstNameValue );
   setBalance( balanceValue );
} // end ClientData constructor

// get account-number value
int ClientData::getAccountNumber() const
{
   return accountNumber;
} // end function getAccountNumber

// set account-number value
void ClientData::setAccountNumber( int accountNumberValue )
{
   accountNumber = accountNumberValue; // should validate
} // end function setAccountNumber
```

Fig. 17.10 | `ClientData` class represents a customer's credit information. (Part 1 of 2.)

```
28
29 // get last-name value
30 string ClientData::getLastName() const
31 {
32    return lastName;
33 } // end function getLastName
34
35 // set last-name value
36 void ClientData::setLastName( string lastNameString )
37 {
38    // copy at most 15 characters from string to lastName
39    int length = lastNameString.size();
40    length = ( length < 15 ? length : 14 );
41    lastNameString.copy( lastName, length );
42    lastName[ length ] = '\0'; // append null character to lastName
43 } // end function setLastName
44
45 // get first-name value
46 string ClientData::getFirstName() const
47 {
48    return firstName;
49 } // end function getFirstName
50
51 // set first-name value
52 void ClientData::setFirstName( string firstNameString )
53 {
54    // copy at most 10 characters from string to firstName
55    int length = firstNameString.size();
56    length = ( length < 10 ? length : 9 );
57    firstNameString.copy( firstName, length );
58    firstName[ length ] = '\0'; // append null character to firstName
59 } // end function setFirstName
60
61 // get balance value
62 double ClientData::getBalance() const
63 {
64    return balance;
65 } // end function getBalance
66
67 // set balance value
68 void ClientData::setBalance( double balanceValue )
69 {
70    balance = balanceValue;
71 } // end function setBalance
```

Fig. 17.10 | `ClientData` class represents a customer's credit information. (Part 2 of 2.)

Objects of class `string` *do not have uniform size*, rather they use dynamically allocated memory to accommodate strings of various lengths. We must maintain fixed-length records, so class `ClientData` stores the client's first and last name in fixed-length `char` arrays (declared in Fig. 17.9, lines 32–33). Member functions `setLastName` (Fig. 17.10, lines 36–43) and `setFirstName` (Fig. 17.10, lines 52–59) each copy the characters of a `string` object into the corresponding `char` array. Consider function `setLastName`. Line

39 invokes `string` member function **`size`** to get the length of `lastNameString`. Line 40 ensures that `length` is fewer than 15 characters, then line 41 copies `length` characters from `lastNameString` into the `char` array `lastName` using `string` member function `copy`. Member function `setFirstName` performs the same steps for the first name.

In Fig. 17.11, line 11 creates an `ofstream` object for the file `credit.dat`. The second argument to the constructor—`ios::out | ios::binary`—indicates that we are opening the file for output in binary mode, which is required if we are to write fixed-length records. Lines 24–25 cause the `blankClient` to be written to the `credit.dat` file associated with `ofstream` object `outCredit`. Remember that operator `sizeof` returns the size in bytes of the object contained in parentheses (see Chapter 8). The first argument to function `write` at line 24 must be of type `const char *`. However, the data type of `&blankClient` is `ClientData *`. To convert `&blankClient` to `const char *`, line 24 uses the cast operator `reinterpret_cast`, so the call to `write` compiles without issuing a compilation error.

```
1  // Fig. 17.11: Fig17_11.cpp
2  // Creating a randomly accessed file.
3  #include <iostream>
4  #include <fstream>
5  #include <cstdlib>
6  #include "ClientData.h" // ClientData class definition
7  using namespace std;
8
9  int main()
10 {
11    ofstream outCredit( "credit.dat", ios::out | ios::binary );
12
13    // exit program if ofstream could not open file
14    if ( !outCredit )
15    {
16       cerr << "File could not be opened." << endl;
17       exit( 1 );
18    } // end if
19
20    ClientData blankClient; // constructor zeros out each data member
21
22    // output 100 blank records to file
23    for ( int i = 0; i < 100; ++i )
24       outCredit.write( reinterpret_cast< const char * >( &blankClient ),
25          sizeof( ClientData ) );
26 } // end main
```

Fig. 17.11 | Creating a random-access file with 100 blank records sequentially.

17.8 Writing Data Randomly to a Random-Access File

Figure 17.12 writes data to the file `credit.dat` and uses the combination of `fstream` functions `seekp` and `write` to store data at *exact* locations in the file. Function `seekp` sets the *put* file-position pointer to a specific position in the file, then function `write` outputs the data. Line 6 includes the header `ClientData.h` defined in Fig. 17.9, so the program can use `ClientData` objects.

```
// Fig. 17.12: Fig17_12.cpp
// Writing to a random-access file.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "ClientData.h" // ClientData class definition
using namespace std;

int main()
{
   int accountNumber;
   string lastName;
   string firstName;
   double balance;

   fstream outCredit( "credit.dat", ios::in | ios::out | ios::binary );

   // exit program if fstream cannot open file
   if ( !outCredit )
   {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   } // end if

   cout << "Enter account number (1 to 100, 0 to end input)\n? ";

   // require user to specify account number
   ClientData client;
   cin >> accountNumber;

   // user enters information, which is copied into file
   while ( accountNumber > 0 && accountNumber <= 100 )
   {
      // user enters last name, first name and balance
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> lastName;
      cin >> firstName;
      cin >> balance;

      // set record accountNumber, lastName, firstName and balance values
      client.setAccountNumber( accountNumber );
      client.setLastName( lastName );
      client.setFirstName( firstName );
      client.setBalance( balance );

      // seek position in file of user-specified record
      outCredit.seekp( ( client.getAccountNumber() - 1 ) *
         sizeof( ClientData ) );

      // write user-specified information in file
      outCredit.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );

```

Fig. 17.12 | Writing to a random-access file. (Part 1 of 2.)

```
54         // enable user to enter another account
55         cout << "Enter account number\n? ";
56         cin >> accountNumber;
57      } // end while
58   } // end main
```

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Fig. 17.12 | Writing to a random-access file. (Part 2 of 2.)

Lines 47–48 position the *put* file-position pointer for object `outCredit` to the byte location calculated by

```
( client.getAccountNumber() - 1 ) * sizeof( ClientData )
```

Because the account number is between 1 and 100, 1 is subtracted from the account number when calculating the byte location of the record. Thus, for record 1, the file-position pointer is set to byte 0 of the file. Line 16 uses the `fstream` object `outCredit` to open the existing `credit.dat` file. The file is opened for input and output in binary mode by combining the file-open modes `ios::in`, `ios::out` and `ios::binary`. Multiple file-open modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (`|`). Opening the existing `credit.dat` file in this manner ensures that this program can manipulate the records written to the file by the program of Fig. 17.11, rather than creating the file from scratch. Chapter 21, Bits, Characters, C Strings and `struct`s, discusses the bitwise inclusive OR operator in detail.

17.9 Reading from a Random-Access File Sequentially

In the previous sections, we created a random-access file and wrote data to that file. In this section, we develop a program that reads the file sequentially and prints only those records that contain data. These programs produce an additional benefit. See if you can determine what it is; we'll reveal it at the end of this section.

The istream function read inputs a specified number of bytes from the current position in the specified stream into an object. For example, lines 30–31 from Fig. 17.13 read the number of bytes specified by sizeof(ClientData) from the file associated with ifstream object inCredit and store the data in the client record. Function read requires a first argument of type char *. Since &client is of type ClientData *, &client must be cast to char * using the cast operator reinterpret_cast.

```
1  // Fig. 17.13: Fig17_13.cpp
2  // Reading a random-access file sequentially.
3  #include <iostream>
4  #include <iomanip>
5  #include <fstream>
6  #include <cstdlib>
7  #include "ClientData.h" // ClientData class definition
8  using namespace std;
9
10 void outputLine( ostream&, const ClientData & ); // prototype
11
12 int main()
13 {
14    ifstream inCredit( "credit.dat", ios::in | ios::binary );
15
16    // exit program if ifstream cannot open file
17    if ( !inCredit )
18    {
19       cerr << "File could not be opened." << endl;
20       exit( 1 );
21    } // end if
22
23    cout << left << setw( 10 ) << "Account" << setw( 16 )
24       << "Last Name" << setw( 11 ) << "First Name" << left
25       << setw( 10 ) << right << "Balance" << endl;
26
27    ClientData client; // create record
28
29    // read first record from file
30    inCredit.read( reinterpret_cast< char * >( &client ),
31       sizeof( ClientData ) );
32
33    // read all records from file
34    while ( inCredit && !inCredit.eof() )
35    {
36       // display record
37       if ( client.getAccountNumber() != 0 )
38          outputLine( cout, client );
39
40       // read next from file
41       inCredit.read( reinterpret_cast< char * >( &client ),
42          sizeof( ClientData ) );
43    } // end while
44 } // end main
```

Fig. 17.13 | Reading a random-access file sequentially. (Part 1 of 2.)

```
45
46 // display single record
47 void outputLine( ostream &output, const ClientData &record )
48 {
49    output << left << setw( 10 ) << record.getAccountNumber()
50       << setw( 16 ) << record.getLastName()
51       << setw( 11 ) << record.getFirstName()
52       << setw( 10 ) << setprecision( 2 ) << right << fixed
53       << showpoint << record.getBalance() << endl;
54 } // end function outputLine
```

```
Account   Last Name       First Name    Balance
29        Brown           Nancy          -24.54
33        Dunn            Stacey         314.33
37        Barker          Doug             0.00
88        Smith           Dave           258.34
96        Stone           Sam             34.98
```

Fig. 17.13 | Reading a random-access file sequentially. (Part 2 of 2.)

Figure 17.13 reads every record in the `credit.dat` file sequentially, checks each record to determine whether it contains data, and displays formatted outputs for records containing data. The condition in line 34 uses the `ios` member function `eof` to determine when the end of file is reached and causes execution of the `while` statement to terminate. Also, if an error occurs when reading from the file, the loop terminates, because `inCredit` evaluates to `false`. The data input from the file is output by function `outputLine` (lines 47–54), which takes two arguments—an `ostream` object and a `clientData` structure to be output. The `ostream` parameter type is interesting, because any `ostream` object (such as `cout`) or any object of a derived class of `ostream` (such as an object of type `ofstream`) can be supplied as the argument. This means that the *same* function can be used, for example, to perform output to the standard-output stream and to a file stream without writing separate functions.

What about that additional benefit we promised? If you examine the output window, you'll notice that the records are listed in *sorted order* (by account number). This is a consequence of how we stored these records in the file, using direct-access techniques. Compared to the insertion sort we used in Chapter 7, sorting using direct-access techniques is relatively fast. *The speed is achieved by making the file large enough to hold every possible record that might be created.* This, of course, means that the file could be occupied sparsely most of the time, resulting in a waste of storage. This is an example of the *space-time trade-off:* By using *large amounts of space*, we can develop *a much faster sorting algorithm*. Fortunately, the continuous reduction in price of storage units has made this less of an issue.

17.10 Case Study: A Transaction-Processing Program

We now present a substantial transaction-processing program (Fig. 17.14) using a random-access file to achieve instant-access processing. The program maintains a bank's account information. It updates existing accounts, adds new accounts, deletes accounts and stores a formatted listing of all current accounts in a text file. We assume that the program of Fig. 17.11 has been executed to create the file `credit.dat` and that the program of Fig. 17.12 has been executed to insert the initial data.

```
// Fig. 17.14: Fig17_14.cpp
// This program reads a random-access file sequentially, updates
// data previously written to the file, creates data to be placed
// in the file, and deletes data previously stored in the file.
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include "ClientData.h" // ClientData class definition
using namespace std;

int enterChoice();
void createTextFile( fstream& );
void updateRecord( fstream& );
void newRecord( fstream& );
void deleteRecord( fstream& );
void outputLine( ostream&, const ClientData & );
int getAccount( const char * const );

enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };

int main()
{
   // open file for reading and writing
   fstream inOutCredit( "credit.dat", ios::in | ios::out | ios::binary );

   // exit program if fstream cannot open file
   if ( !inOutCredit )
   {
      cerr << "File could not be opened." << endl;
      exit ( 1 );
   } // end if

   int choice; // store user choice

   // enable user to specify action
   while ( ( choice = enterChoice() ) != END )
   {
      switch ( choice )
      {
         case PRINT: // create text file from record file
            createTextFile( inOutCredit );
            break;
         case UPDATE: // update record
            updateRecord( inOutCredit );
            break;
         case NEW: // create record
            newRecord( inOutCredit );
            break;
         case DELETE: // delete existing record
            deleteRecord( inOutCredit );
            break;
```

Fig. 17.14 | Bank account program. (Part 1 of 5.)

```
         default: // display error if user does not select valid choice
            cerr << "Incorrect choice" << endl;
            break;
      } // end switch

      inOutCredit.clear(); // reset end-of-file indicator
   } // end while
} // end main

// enable user to input menu choice
int enterChoice()
{
   // display available options
   cout << "\nEnter your choice" << endl
      << "1 - store a formatted text file of accounts" << endl
      << "    called \"print.txt\" for printing" << endl
      << "2 - update an account" << endl
      << "3 - add a new account" << endl
      << "4 - delete an account" << endl
      << "5 - end program\n? ";

   int menuChoice;
   cin >> menuChoice; // input menu selection from user
   return menuChoice;
} // end function enterChoice

// create formatted text file for printing
void createTextFile( fstream &readFromFile )
{
   // create text file
   ofstream outPrintFile( "print.txt", ios::out );

   // exit program if ofstream cannot create file
   if ( !outPrintFile )
   {
      cerr << "File could not be created." << endl;
      exit( 1 );
   } // end if

   outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
      << "Last Name" << setw( 11 ) << "First Name" << right
      << setw( 10 ) << "Balance" << endl;

   // set file-position pointer to beginning of readFromFile
   readFromFile.seekg( 0 );

   // read first record from record file
   ClientData client;
   readFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

```

Fig. 17.14 | Bank account program. (Part 2 of 5.)

```
   // copy all records from record file into text file
   while ( !readFromFile.eof() )
   {
      // write single record to text file
      if ( client.getAccountNumber() != 0 ) // skip empty records
         outputLine( outPrintFile, client );

      // read next record from record file
      readFromFile.read( reinterpret_cast< char * >( &client ),
         sizeof( ClientData ) );
   } // end while
} // end function createTextFile

// update balance in record
void updateRecord( fstream &updateFile )
{
   // obtain number of account to update
   int accountNumber = getAccount( "Enter account to update" );

   // move file-position pointer to correct record in file
   updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read first record from file
   ClientData client;
   updateFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // update record
   if ( client.getAccountNumber() != 0 )
   {
      outputLine( cout, client ); // display the record

      // request user to specify transaction
      cout << "\nEnter charge (+) or payment (-): ";
      double transaction; // charge or payment
      cin >> transaction;

      // update record balance
      double oldBalance = client.getBalance();
      client.setBalance( oldBalance + transaction );
      outputLine( cout, client ); // display the record

      // move file-position pointer to correct record in file
      updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

      // write updated record over old record in file
      updateFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account does not exist
      cerr << "Account #" << accountNumber
         << " has no information." << endl;
} // end function updateRecord
```

Fig. 17.14 | Bank account program. (Part 3 of 5.)

```

// create and insert record
void newRecord( fstream &insertInFile )
{
   // obtain number of account to create
   int accountNumber = getAccount( "Enter new account number" );

   // move file-position pointer to correct record in file
   insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read record from file
   ClientData client;
   insertInFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // create record, if record does not previously exist
   if ( client.getAccountNumber() == 0 )
   {
      string lastName;
      string firstName;
      double balance;

      // user enters last name, first name and balance
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> setw( 15 ) >> lastName;
      cin >> setw( 10 ) >> firstName;
      cin >> balance;

      // use values to populate account values
      client.setLastName( lastName );
      client.setFirstName( firstName );
      client.setBalance( balance );
      client.setAccountNumber( accountNumber );

      // move file-position pointer to correct record in file
      insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

      // insert record in file
      insertInFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account already exists
      cerr << "Account #" << accountNumber
         << " already contains information." << endl;
} // end function newRecord

// delete an existing record
void deleteRecord( fstream &deleteFromFile )
{
   // obtain number of account to delete
   int accountNumber = getAccount( "Enter account to delete" );

```

Fig. 17.14 | Bank account program. (Part 4 of 5.)

```
   // move file-position pointer to correct record in file
   deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read record from file
   ClientData client;
   deleteFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // delete record, if record exists in file
   if ( client.getAccountNumber() != 0 )
   {
      ClientData blankClient; // create blank record

      // move file-position pointer to correct record in file
      deleteFromFile.seekp( ( accountNumber - 1 ) *
         sizeof( ClientData ) );

      // replace existing record with blank record
      deleteFromFile.write(
         reinterpret_cast< const char * >( &blankClient ),
         sizeof( ClientData ) );

      cout << "Account #" << accountNumber << " deleted.\n";
   } // end if
   else // display error if record does not exist
      cerr << "Account #" << accountNumber << " is empty.\n";
} // end deleteRecord

// display single record
void outputLine( ostream &output, const ClientData &record )
{
   output << left << setw( 10 ) << record.getAccountNumber()
      << setw( 16 ) << record.getLastName()
      << setw( 11 ) << record.getFirstName()
      << setw( 10 ) << setprecision( 2 ) << right << fixed
      << showpoint << record.getBalance() << endl;
} // end function outputLine

// obtain account-number value from user
int getAccount( const char * const prompt )
{
   int accountNumber;

   // obtain account-number value
   do
   {
      cout << prompt << " (1 - 100): ";
      cin >> accountNumber;
   } while ( accountNumber < 1 || accountNumber > 100 );

   return accountNumber;
} // end function getAccount
```

Fig. 17.14 | Bank account program. (Part 5 of 5.)

The program has five options (Option 5 is for terminating the program). Option 1 calls function `createTextFile` to store a formatted list of all the account information in a text file called `print.txt` that may be printed. Function `createTextFile` (lines 80–115) takes an `fstream` object as an argument to be used to input data from the `credit.dat` file. Function `createTextFile` invokes `istream` member function `read` (lines 101–102) and uses the sequential-file-access techniques of Fig. 17.13 to input data from `credit.dat`. Function `outputLine`, discussed in Section 17.9, outputs the data to file `print.txt`. Note that `createTextFile` uses `istream` member function `seekg` (line 97) to ensure that the file-position pointer is at the beginning of the file. After choosing Option 1, the `print.txt` file contains

```
Account    Last Name        First Name    Balance
29         Brown            Nancy          -24.54
33         Dunn             Stacey         314.33
37         Barker           Doug             0.00
88         Smith            Dave           258.34
96         Stone            Sam             34.98
```

Option 2 calls `updateRecord` (lines 118–156) to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 128–129 read data into object `client`, using `istream` member function `read`. Then line 132 compares the value returned by `getAccountNumber` of the `client` object to zero to determine whether the record contains information. If this value is zero, lines 154–155 print an error message indicating that the record is empty. If the record contains information, line 134 displays the record, using function `outputLine`, line 139 inputs the transaction amount and lines 142–151 calculate the new balance and rewrite the record to the file. A typical output for Option 2 is

```
Enter account to update (1 - 100): 37
37         Barker           Doug             0.00

Enter charge (+) or payment (-): +87.99
37         Barker           Doug            87.99
```

Option 3 calls function `newRecord` (lines 159–201) to add a new account to the file. If the user enters an account number for an existing account, `newRecord` displays an error message indicating that the account exists (lines 199–200). This function adds a new account in the same manner as the program of Fig. 17.11. A typical output for Option 3 is

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option 4 calls function `deleteRecord` (lines 204–235) to delete a record from the file. Line 207 prompts the user to enter the account number. Only an existing record may be deleted, so, if the specified account is empty, line 234 displays an error message. If the account exists, lines 227–229 reinitialize that account by copying an empty record (`blank-`

`Client`) to the file. Line 231 displays a message to inform the user that the record has been deleted. A typical output for Option 4 is

```
Enter account to delete (1 - 100): 29
Account #29 deleted.
```

Line 25 opens the `credit.dat` file by creating an `fstream` object for both reading and writing, using modes `ios::in` and `ios::out` "or-ed" together.

17.11 Object Serialization

This chapter and Chapter 15 introduced the object-oriented style of input/output. However, our examples concentrated on I/O of fundamental types rather than objects of user-defined types. In Chapter 11, we showed how to input and output objects using operator overloading. We accomplished object input by overloading the stream extraction operator, `>>`, for the appropriate `istream`. We accomplished object output by overloading the stream insertion operator, `<<`, for the appropriate `ostream`. In both cases, only an object's data members were input or output, and, in each case, they were in a format meaningful only for objects of that particular type. An object's member functions are *not* input or output with the object's data; rather, *one copy of the class's member functions remains available internally and is shared by all objects of the class.*

When object data members are output to a disk file, we lose the object's type information. We store only the values of the object's attributes, not type information, on the disk. If the program that reads this data knows the object type to which the data corresponds, the program can read the data into an object of that type as we did in our random-access file examples.

An interesting problem occurs when we store objects of different types in the same file. How can we distinguish them (or their collections of data members) as we read them into a program? The problem is that objects typically do *not* have type fields (we discussed this issue in Chapter 13).

One approach used by several programming languages is called **object serialization**. A so-called **serialized object** is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. After a serialized object has been written to a file, it can be read from the file and **deserialized**—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory. C++ does not provide a built-in serialization mechanism; however, there are third party and open source C++ libraries that support object serialization. The open source Boost C++ Libraries (`www.boost.org`) provide support for serializing objects in text, binary and extensible markup language (XML) formats (`www.boost.org/libs/serialization/doc/index.html`). We overview the Boost C++ Libraries in Chapter 23.

17.12 Wrap-Up

In this chapter, we presented various file-processing techniques to manipulate persistent data. You were introduced to the differences between character-based and byte-based streams, and to several file-processing class templates in header `<fstream>`. Then, you learned how to use sequential file processing to manipulate records stored in order, by a record-key field. You also learned how to use random-access files to "instantly" retrieve and manipulate fixed-length records. We presented a substantial transaction-processing program using a random-access file to achieve "instant-access" processing. Finally, we discussed the basic concepts of object serialization. In the next chapter, we discuss typical string-manipulation operations provided by class template `basic_string`. We also introduce string stream-processing capabilities that allow strings to be input from and output to memory.

Summary

Section 17.1 Introduction

- Files are used for data persistence (p. 659)—permanent retention of data.
- Computers store files on secondary storage devices (p. 659), such as hard disks, CDs, DVDs, flash memory and tapes.

Section 17.2 Files and Streams

- C++ views each file simply as a sequence of bytes.
- Each file ends either with an end-of-file marker (p. 659) or at a specific byte number recorded in a system-maintained, administrative data structure.
- When a file is opened, an object is created, and a stream is associated with the object.
- To perform file processing in C++, headers `<iostream>` and `<fstream>` must be included.
- Header `<fstream>` (p. 659) includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output).
- Each class template has a predefined template specialization that enables char I/O. The `<fstream>` library provides `typedef` aliases for these template specializations. The `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file. The `typedef ofstream` represents a specialization of `basic_ofstream` that enables `char` output to files. The `typedef fstream` (p. 659) represents a specialization of `basic_fstream` that enables `char` input from, and output to, files.
- The file-processing templates derive from class templates `basic_istream`, `basic_ostream` and `basic_iostream`, respectively. Thus, all member functions, operators and manipulators that belong to these templates also can be applied to file streams.

Section 17.3 Creating a Sequential File

- C++ imposes no structure on a file; you must structure files to meet the application's requirements.
- A file can be opened for output when an `ofstream` object is created. Two arguments are passed to the object's constructor—the filename (p. 661) and the file-open mode (p. 661).
- For an `ofstream` (p. 661) object, the file-open mode can be either `ios::out` (p. 661) to output data to a file or `ios::app` (p. 661) to append data to the end of a file. Existing files opened with mode `ios::out` are truncated (p. 661). If the specified file does not exist, the `ofstream` object creates the file using that filename.
- By default, `ofstream` objects are opened for output.

- An `ofstream` object can be created without opening a specific file—a file can be attached to the object later with member function `open` (p. 662).
- The `ios` member function `operator!` determines whether a stream was opened correctly. This operator can be used in a condition that returns a true value if either the `failbit` or the `badbit` is set for the stream on the open operation.
- The `ios` member function `operator void *` converts a stream to a pointer, so it can be compared to `0`. When a pointer value is used as a condition, a null pointer represents `false` and a non-null pointer represents `true`. If the `failbit` or `badbit` has been set for a stream, `0` (`false`) is returned.
- Entering the end-of-file indicator sets the `failbit` for `cin`.
- The `operator void *` function can be used to test an input object for end-of-file instead of calling the `eof` member function explicitly on the input object.
- When a stream object's destructor is called, the corresponding stream is closed. You also can close the stream object explicitly, using the stream's `close` member function.

Section 17.4 Reading Data from a Sequential File

- Files store data so it may be retrieved for processing when needed.
- Creating an `ifstream` object opens a file for input. The `ifstream` constructor can receive the filename and the file open mode as arguments.
- Open a file for input only if the file's contents should not be modified.
- Objects of class `ifstream` are opened for input by default.
- An `ifstream` object can be created without opening a specific file; a file can be attached to it later.
- To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found.
- The member functions for repositioning the file-position pointer (p. 665) are `seekg` ("seek get"; p. 665) for `istream` and `seekp` ("seek put"; p. 665) for `ostream`. Each `istream` has a "get pointer," which indicates the byte number in the file from which the next input is to occur, and each `ostream` has a "put pointer," which indicates the byte number in the file at which the next output should be placed.
- The argument to `seekg` (p. 665) is a long integer. A second argument can be specified to indicate the seek direction (p. 665), which can be `ios::beg` (the default; p. 665) for positioning relative to the beginning of a stream, `ios::cur` (p. 665) for positioning relative to the current position in a stream or `ios::end` (p. 666) for positioning relative to the end of a stream.
- The file-position pointer (p. 665) is an integer value that specifies the location in the file as a number of bytes from the file's starting location (i.e., the offset (p. 666) from the beginning of the file).
- Member functions `tellg` (p. 666) and `tellp` (p. 666) are provided to return the current locations of the "get" and "put" pointers, respectively.

Section 17.5 Updating Sequential Files

- Data that is formatted and written to a sequential file cannot be modified without the risk of destroying other data in the file. The problem is that records can vary in size.

Section 17.6 Random-Access Files

- Sequential files are inappropriate for instant-access applications (p. 670), in which a particular record must be located immediately.
- Instant access is made possible with random-access files (p. 670). Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.

- The easiest method to format files for random access is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file.
- Data can be inserted into a random-access file without destroying other data in the file.
- Data stored previously can be updated or deleted without rewriting the entire file.

Section 17.7 Creating a Random-Access File

- The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. Function `write` writes the data at the location in the file specified by the "put" file-position pointer.
- The `istream` member function `read` (p. 671) inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function read inputs bytes at the location in the file specified by the "get" file-position pointer.
- Function `write` treats its first argument as a group of bytes by viewing the object in memory as a `const char *`, which is a pointer to a byte (remember that a `char` is one byte). Starting from that location, function `write` outputs the number of bytes specified by its second argument. The `istream` function read can subsequently be used to read the bytes back into memory.
- The `reinterpret_cast` operator (p. 671) converts a pointer of one type to an unrelated pointer type.
- A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points.
- A program that reads unformatted data must be compiled and executed on a system compatible with the program that wrote the data—different systems may represent internal data differently.
- Objects of class `string` do not have uniform size, rather they use dynamically allocated memory to accommodate strings of various lengths.
- The `string` member function `data` returns an array containing the characters of the `string`. This array is not guaranteed to be null terminated.

Section 17.8 Writing Data Randomly to a Random-Access File

- Multiple file-open modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (|).
- The `string` member function `size` (p. 675) gets the length of a `string`.
- The file open mode `ios::binary` (p. 675) indicates that a file should be opened in binary mode.

Section 17.9 Reading from a Random-Access File Sequentially

- The `istream` function `read` inputs a specified number of bytes from the current position in the specified stream into an object.
- A function that receives an `ostream` parameter can receive any `ostream` object (such as `cout`) or any object of a derived class of `ostream` (such as an object of type `ofstream`) as an argument. This means that the same function can be used, for example, to perform output to the standard-output stream and to a file stream without writing separate functions.

Section 17.11 Object Serialization

- When object data members are output to a disk file, we lose the object's type information. We store only the values of the object's attributes, not type information, on the disk. If the program that reads this data knows the object type to which the data corresponds, the program can read the data into an object of that type.

- A so-called serialized object (p. 686) is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. A serialized object can be read from the file and deserialized (p. 686).
- The open source Boost Libraries provide support for serializing objects (p. 686) in text, binary and extensible markup language (XML) formats.

Self-Review Exercises

17.1 *(Fill in the Blanks)* Fill in the blanks in each of the following:

a) Member function ________ of the file streams `fstream`, `ifstream` and `ofstream` closes a file.
b) The `istream` member function ________ reads a character from the specified stream.
c) Member function ________ of the file streams `fstream`, `ifstream` and `ofstream` opens a file.
d) The `istream` member function ________ is normally used when reading data from a file in random-access applications.
e) Member functions ________ and ________ of `istream` and `ostream` set the file-position pointer to a specific location in an input or output stream, respectively.

17.2 *(True or False)* State which of the following are *true* and which are *false*. If *false*, explain why.

a) Member function `read` cannot be used to read data from the input object `cin`.
b) You must create the `cin`, `cout`, `cerr` and `clog` objects explicitly.
c) A program must call function `close` explicitly to close a file associated with an `ifstream`, `ofstream` or `fstream` object.
d) If the file-position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
e) The `ostream` member function `write` can write to standard-output stream `cout`.
f) Data in sequential files always is updated without overwriting nearby data.
g) Searching all records in a random-access file to find a specific record is unnecessary.
h) Records in random-access files must be of uniform length.
i) Member functions `seekp` and `seekg` must seek relative to the beginning of a file.

17.3 Assume that each of the following statements applies to the same program.

a) Write a statement that opens file `oldmast.dat` for input; use an `ifstream` object called `inOldMaster`.
b) Write a statement that opens file `trans.dat` for input; use an `ifstream` object called `inTransaction`.
c) Write a statement that opens file `newmast.dat` for output (and creation); use `ofstream` object `outNewMaster`.
d) Write a statement that reads a record from the file `oldmast.dat`. The record consists of integer `accountNumber`, string `name` and floating-point `currentBalance`; use `ifstream` object `inOldMaster`.
e) Write a statement that reads a record from the file `trans.dat`. The record consists of integer `accountNum` and floating-point `dollarAmount`; use `ifstream` object `inTransaction`.
f) Write a statement that writes a record to the file `newmast.dat`. The record consists of integer `accountNum`, string `name`, and floating-point `currentBalance`; use `ofstream` object `outNewMaster`.

17.4 Find the error(s) and show how to correct it (them) in each of the following.

a) File `payables.dat` referred to by `ofstream` object `outPayable` has not been opened.

```
outPayable << account << company << amount << endl;
```

b) The following statement should read a record from the file `payables.dat`. The `ifstream` object `inPayable` refers to this file, and `istream` object `inReceivable` refers to the file `receivables.dat`.

```
inReceivable >> account >> company >> amount;
```

c) The file `tools.dat` should be opened to add data to the file without discarding the current data.

```
ofstream outTools( "tools.dat", ios::out );
```

Answers to Self-Review Exercises

17.1 a) `close`. b) `get`. c) `open`. d) `read`. e) `seekg`, `seekp`.

17.2 a) False. Function `read` can read from any input stream object derived from `istream`.
b) False. These four streams are created automatically for you. The `<iostream>` header must be included in a file to use them. This header includes declarations for each stream object.
c) False. The files will be closed when destructors for `ifstream`, `ofstream` or `fstream` objects execute when the stream objects go out of scope or before program execution terminates, but it's a good programming practice to close all files explicitly with `close` once they're no longer needed.
d) False. Member function `seekp` or `seekg` can be used to reposition the "put" or "get" file-position pointer to the beginning of the file.
e) True.
f) False. In most cases, sequential file records are not of uniform length. Therefore, it's possible that updating a record will cause other data to be overwritten.
g) True.
h) False. Records in a random-access file normally are of uniform length.
i) False. It's possible to seek from the beginning of the file, from the end of the file and from the current position in the file.

17.3 a) `ifstream inOldMaster( "oldmast.dat", ios::in );`
b) `ifstream inTransaction( "trans.dat", ios::in );`
c) `ofstream outNewMaster( "newmast.dat", ios::out );`
d) `inOldMaster >> accountNumber >> name >> currentBalance;`
e) `inTransaction >> accountNum >> dollarAmount;`
f) `outNewMaster << accountNum << " " << name << " " << currentBalance;`

17.4 a) *Error:* The file `payables.dat` has not been opened before the attempt is made to output data to the stream.
Correction: Use `ostream` function `open` to open `payables.dat` for output.
b) *Error:* The incorrect `istream` object is being used to read a record from the file named `payables.dat`.
Correction: Use `istream` object `inPayable` to refer to `payables.dat`.
c) *Error:* The file's contents are discarded because the file is opened for output (`ios::out`).
Correction: To add data to the file, open the file either for updating (`ios::ate`) or for appending (`ios::app`).

Exercises

17.5 *(Fill in the Blanks)* Fill in the blanks in each of the following:
a) Computers store large amounts of data on secondary storage devices as ________.

b) The standard stream objects declared by header `<iostream>` are ________, ________, ________ and ________.
c) `ostream` member function ________ outputs a character to the specified stream.
d) `ostream` member function ________ is generally used to write data to a randomly accessed file.
e) `istream` member function ________ repositions the file-position pointer in a file.

17.6 *(File Matching)* Exercise 17.3 asked you to write a series of single statements. Actually, these statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it's common to have several files in each application system. In an accounts receivable system, for example, there is generally a master file containing detailed information about each customer, such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and, possibly, a condensed history of recent purchases and cash payments.

As transactions occur (e.g., sales are made and cash payments arrive), they're entered into a file. At the end of each business period (a month for some companies, a week for others and a day in some cases), the file of transactions (called `trans.dat` in Exercise 17.3) is applied to the master file (called `oldmast.dat` in Exercise 17.3), thus updating each account's record of purchases and payments. During an updating run, the master file is rewritten as a new file (`newmast.dat`), which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and therefore no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments may have just moved to this community, and the company may not have had a chance to create a master record for this customer.

Use the statements from Exercise 17.3 as a basis for writing a complete file-matching accounts receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing order by account number.

When a match occurs (i.e., records with the same account number appear on both the master and transaction files), add the dollar amount on the transaction file to the current balance on the master file, and write the `newmast.dat` record. (Assume purchases are indicated by positive amounts on the transaction file and payments are indicated by negative amounts.) When there is a master record for a particular account but no corresponding transaction record, merely write the master record to `newmast.dat`. When there is a transaction record but no corresponding master record, print the error message `"Unmatched transaction record for account number ..."` (fill in the account number from the transaction record).

17.7 *(File Matching Test Data)* After writing the program of Exercise 17.6, write a simple program to create some test data for checking out the program. Use the following sample account data:

Master file Account number	Name	Balance
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	–14.22

Transaction file Account number	Transaction amount
100	27.14
300	62.11
400	100.56
900	82.17

17.8 *(File Matching Test)* Run the program of Exercise 17.6, using the files of test data created in Exercise 17.7. Print the new master file. Check that the accounts have been updated correctly.

17.9 *(File Matching Enhancement)* It's common to have several transaction records with the same record key, because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts receivable file-matching program of Exercise 17.6 to provide for the possibility of handling several transaction records with the same record key. Modify the test data of Exercise 17.7 to include the following additional transaction records:

Account number	Dollar amount
300	83.89
700	80.78
700	1.53

17.10 Write a series of statements that accomplish each of the following. Assume that we've defined class `Person` that contains the `private` data members

```
char lastName[ 15 ];
char firstName[ 10 ];
int age;
int id;
```

and `public` member functions

```
// accessor functions for id
void setId( int );
int getId() const;

// accessor functions for lastName
void setLastName( string );
string getLastName() const;

// accessor functions for firstName
void setFirstName( string );
string getFirstName() const;

// accessor functions for age
void setAge( int );
int getAge() const;
```

Also assume that any random-access files have been opened properly.

a) Initialize `nameage.dat` with 100 records that store values `lastName ="unassigned"`, `firstName = ""` and `age = 0`.
b) Input 10 last names, first names and ages, and write them to the file.

c) Update a record that already contains information. If the record does not contain information, inform the user `"No info"`.
d) Delete a record that contains information by reinitializing that particular record.

17.11 *(Hardware Inventory)* You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the random-access file `hardware.dat` to 100 empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update *any* information in the file. The tool identification number should be the record number. Use the following information to start your file:

Record #	Tool name	Quantity	Cost
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	79.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

17.12 (*Telephone Number Word Generator*) Standard telephone keypads contain the digits 0 through 9. The numbers 2 through 9 each have three letters associated with them, as is indicated by the following table:

Digit	Letter	Digit	Letter
2	A B C	6	M N O
3	D E F	7	P Q R S
4	G H I	8	T U V
5	J K L	9	W X Y Z

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word “NUMBERS.”

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then no doubt the business will receive a few more calls. Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., “TAKEOUT”). Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It’s possible, however, that the owner of a barber shop would be pleased to know that the shop’s telephone number, 424-7288, corresponds to “HAIRCUT.” A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to “PETCARE.”

Write a program that, given a seven-digit number, writes to a file every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

17.13 *(`sizeof` Operator)* Write a program that uses the `sizeof` operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file `datasize.dat`, so that you may print the results later. The results should be displayed in two-column format with the type name in the left column and the size of the type in right column, as in:

```
char                   1
unsigned char          1
short int              2
unsigned short int     2
int                    4
unsigned int           4
long int               4
unsigned long int      4
float                  4
double                 8
long double           10
```

[*Note:* The sizes of the built-in data types on your computer might differ from those listed above.]

Making a Difference

17.14 *(Phishing Scanner)* Phishing is a form of identity theft in which, in an e-mail, a sender posing as a trustworthy source attempts to acquire private information, such as your user names, passwords, credit-card numbers and social security number. Phishing e-mails claiming to be from popular banks, credit-card companies, auction sites, social networks and online payment services may look quite legitimate. These fraudulent messages often provide links to spoofed (fake) websites where you're asked to enter sensitive information.

Visit McAfee® (`www.mcafee.com/us/threat_center/anti_phishing/phishing_top10.html`), Security Extra (`www.securityextra.com/`), `www.snopes.com` and other websites to find lists of the top phishing scams. Also check out the Anti-Phishing Working Group (`www.antiphishing.org/`), and the FBI's Cyber Investigations website (`www.fbi.gov/cyberinvest/cyberhome.htm`), where you'll find information about the latest scams and how to protect yourself.

Create a list of 30 words, phrases and company names commonly found in phishing messages. Assign a point value to each based on your estimate of its likeliness to be in a phishing message (e.g., one point if it's somewhat likely, two points if moderately likely, or three points if highly likely). Write a program that scans a file of text for these terms and phrases. For each occurrence of a keyword or phrase within the text file, add the assigned point value to the total points for that word or phrase. For each keyword or phrase found, output one line with the word or phrase, the number of occurrences and the point total. Then show the point total for the entire message. Does your program assign a high point total to some actual phishing e-mails you've received? Does it assign a high point total to some legitimate e-mails you've received.

18 Class `string` and String Stream Processing

Suit the action to the word, the word to the action; with this special observance, that you o'erstep not the modesty of nature.
—William Shakespeare

The difference between the almost-right word and the right word is really a large matter — it's the difference between the lightning bug and the lightning.
—Mark Twain

Mum's the word.
—Miguel de Cervantes

I have made this letter longer than usual, because I lack the time to make it short.
—Blaise Pascal

Objectives

In this chapter you'll learn:

- To assign, concatenate, compare, search and swap `string`s.
- To determine `string` characteristics.
- To find, replace and insert characters in `string`s.
- To convert `string`s to C-style strings and vice versa.
- To use `string` iterators.
- To perform input from and output to `string`s in memory.

Outline

18.1 Introduction

The class template **`basic_string`** provides typical string-manipulation operations such as copying, searching, etc. The template definition and all support facilities are defined in `namespace std`; these include the `typedef` statement

```
typedef basic_string< char > string;
```

that creates the alias type `string` for **`basic_string<char>`**. A `typedef` is also provided for the **`wchar_t`** type (`wstring`). Type `wchar_t`[1] stores characters (e.g., two-byte characters, four-byte characters, etc.) for supporting other character sets. We use `string` exclusively throughout this chapter. To use `string`s, include header `<string>`.

A `string` object can be initialized with a constructor argument as in

```
string text( "Hello" ); // creates a string from a const char *
```

which creates a `string` containing the characters in `"Hello"`, or with two constructor arguments as in

```
string name( 8, 'x' ); // string of 8 'x' characters
```

which creates a `string` containing eight `'x'` characters. Class `string` also provides a *default constructor* (which creates an empty string) and a *copy constructor*. A `string` also can be initialized in its definition as in

```
string month = "March"; // same as: string month( "March" );
```

Remember that = in the preceding declaration is *not* an assignment; rather it's an *implicit call to the `string` class constructor*, which does the conversion.

Class `string` provides no conversions from `int` or `char` to `string` in a `string` definition. For example, the definitions

```
string error1 = 'c';
string error2( 'u' );
string error3 = 22;
string error4( 8 );
```

1. Type `wchar_t` commonly is used to represent Unicode®. The Unicode Standard outlines a specification to produce consistent encoding of the world's characters and *symbols*. To learn more about the Unicode Standard, visit `www.unicode.org`.

result in syntax errors. Assigning a single character to a string object is permitted in an assignment statement as in

```
string1 = 'n';
```

Unlike C-style char * strings, strings are not necessarily null terminated. [*Note:* The C++ standard document provides only a description of the interface for class string—implementation is platform dependent.] The length of a string can be retrieved with member function **length** and with member function **size**. The subscript operator, [], can be used with strings to access and modify individual characters. Like C-style strings, strings have a first subscript of 0 and a last subscript of length() - 1.

Most string member functions take as arguments a starting subscript location and the number of characters on which to operate.

The stream extraction operator (>>) is overloaded to support strings. The statements

```
string stringObject;
cin >> stringObject;
```

declare a string object and read a string from cin. Input is delimited by white-space characters. When a delimiter is encountered, the input operation is terminated. Function **getline** also is overloaded for strings. Assuming string1 is a string, the statement

```
getline( cin, string1 );
```

reads a string from the keyboard into string1. Input is delimited by a newline ('\n'), so getLine can read a line of text into a string object. You can specify an alternate delimiter as the optional third argument to getline.

18.2 string Assignment and Concatenation

Figure 18.1 demonstrates string assignment and concatenation. Line 4 includes header <string> for class string. The strings string1, string2 and string3 are created in lines 9–11. Line 13 assigns the value of string1 to string2. After the assignment takes place, string2 is a copy of string1. Line 14 uses member function **assign** to copy string1 into string3. A separate copy is made (i.e., string1 and string3 are independent objects). Class string also provides an overloaded version of member function assign that copies a specified number of characters, as in

```
targetString.assign( sourceString, start, numberOfCharacters );
```

where sourceString is the string to be copied, start is the starting subscript and numberOfCharacters is the number of characters to copy.

```
1 // Fig. 18.1: Fig18_01.cpp
2 // Demonstrating string assignment and concatenation.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
```

Fig. 18.1 | Demonstrating string assignment and concatenation. (Part 1 of 2.)

```
9      string string1( "cat" );
10     string string2; // initialized to the empty string
11     string string3; // initialized to the empty string
12
13     string2 = string1; // assign string1 to string2
14     string3.assign( string1 ); // assign string1 to string3
15     cout << "string1: " << string1 << "\nstring2: " << string2
16        << "\nstring3: " << string3 << "\n\n";
17
18     // modify string2 and string3
19     string2[ 0 ] = string3[ 2 ] = 'r';
20
21     cout << "After modification of string2 and string3:\n" << "string1: "
22        << string1 << "\nstring2: " << string2 << "\nstring3: ";
23
24     // demonstrating member function at
25     for ( int i = 0; i < string3.length(); ++i )
26        cout << string3.at( i );
27
28     // declare string4 and string5
29     string string4( string1 + "apult" ); // concatenation
30     string string5; // initialized to the empty string
31
32     // overloaded +=
33     string3 += "pet"; // create "carpet"
34     string1.append( "acomb" ); // create "catacomb"
35
36     // append subscript locations 4 through end of string1 to
37     // create string "comb" (string5 was initially empty)
38     string5.append( string1, 4, string1.length() - 4 );
39
40     cout << "\n\nAfter concatenation:\nstring1: " << string1
41        << "\nstring2: " << string2 << "\nstring3: " << string3
42        << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
43  } // end main
```

```
string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
string1: cat
string2: rat
string3: car

After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
```

Fig. 18.1 | Demonstrating `string` assignment and concatenation. (Part 2 of 2.)

Line 19 uses the subscript operator to assign `'r'` to `string3[ 2 ]` (forming `"car"`) and to assign `'r'` to `string2[ 0 ]` (forming `"rat"`). The `string`s are then output.

Lines 25–26 output the contents of `string3` one character at a time using member function **`at`**. Member function `at` provides **checked access** (or **range checking**); i.e., going past the end of the `string` throws an `out_of_range` exception. *The subscript operator, `[]`, does not provide checked access.* This is consistent with its use on arrays.

Common Programming Error 18.1

Accessing an element beyond the size of the `string` using the subscript operator is an unreported logic error.

String `string4` is declared (line 29) and initialized to the result of concatenating `string1` and `"apult"` using the overloaded `+` operator, which for class `string` denotes concatenation. Line 33 uses the addition assignment operator, `+=`, to concatenate `string3` and `"pet"`. Line 34 uses member function **`append`** to concatenate `string1` and `"acomb"`.

Line 38 appends the string `"comb"` to empty `string` `string5`. This member function is passed the `string` (`string1`) to retrieve characters from, the starting subscript in the `string` (`4`) and the number of characters to append (the value returned by `string1.length() - 4`).

18.3 Comparing strings

Class `string` provides member functions for comparing `string`s. Figure 18.2 demonstrates class `string`'s comparison capabilities.

```
// Fig. 18.2: Fig18_02.cpp
// Demonstrating string comparison capabilities.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string string1( "Testing the comparison functions." );
   string string2( "Hello" );
   string string3( "stinger" );
   string string4( string2 );

   cout << "string1: " << string1 << "\nstring2: " << string2
      << "\nstring3: " << string3 << "\nstring4: " << string4 << "\n\n";

   // comparing string1 and string4
   if ( string1 == string4 )
      cout << "string1 == string4\n";
   else // string1 != string4
   {
      if ( string1 > string4 )
         cout << "string1 > string4\n";
      else // string1 < string4
         cout << "string1 < string4\n";
   } // end else
```

Fig. 18.2 | Comparing `string`s. (Part 1 of 3.)

```

   // comparing string1 and string2
   int result = string1.compare( string2 );

   if ( result == 0 )
      cout << "string1.compare( string2 ) == 0\n";
   else // result != 0
   {
      if ( result > 0 )
         cout << "string1.compare( string2 ) > 0\n";
      else // result < 0
         cout << "string1.compare( string2 ) < 0\n";
   } // end else

   // comparing string1 (elements 2-5) and string3 (elements 0-5)
   result = string1.compare( 2, 5, string3, 0, 5 );

   if ( result == 0 )
      cout << "string1.compare( 2, 5, string3, 0, 5 ) == 0\n";
   else // result != 0
   {
      if ( result > 0 )
         cout << "string1.compare( 2, 5, string3, 0, 5 ) > 0\n";
      else // result < 0
         cout << "string1.compare( 2, 5, string3, 0, 5 ) < 0\n";
   } // end else

   // comparing string2 and string4
   result = string4.compare( 0, string2.length(), string2 );

   if ( result == 0 )
      cout << "string4.compare( 0, string2.length(), "
         << "string2 ) == 0" << endl;
   else // result != 0
   {
      if ( result > 0 )
         cout << "string4.compare( 0, string2.length(), "
            << "string2 ) > 0" << endl;
      else // result < 0
         cout << "string4.compare( 0, string2.length(), "
            << "string2 ) < 0" << endl;
   } // end else

   // comparing string2 and string4
   result = string2.compare( 0, 3, string4 );

   if ( result == 0 )
      cout << "string2.compare( 0, 3, string4 ) == 0" << endl;
   else // result != 0
   {
      if ( result > 0 )
         cout << "string2.compare( 0, 3, string4 ) > 0" << endl;
```

Fig. 18.2 | Comparing strings. (Part 2 of 3.)

```
79         else // result < 0
80            cout << "string2.compare( 0, 3, string4 ) < 0" << endl;
81      } // end else
82   } // end main
```

```
string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

string1 > string4
string1.compare( string2 ) > 0
string1.compare( 2, 5, string3, 0, 5 ) == 0
string4.compare( 0, string2.length(), string2 ) == 0
string2.compare( 0, 3, string4 ) < 0
```

Fig. 18.2 | Comparing `string`s. (Part 3 of 3.)

The program declares four `string`s (lines 9–12) and outputs each (lines 14–15). Line 18 tests `string1` against `string4` for equality using the overloaded equality operator. If the condition is `true`, `"string1 == string4"` is output. If the condition is `false`, the condition in line 22 is tested. All the `string` class overloaded relational and equality operator functions return `bool` values.

Line 29 uses `string` member function **`compare`** to compare `string1` to `string2`. Variable `result` is assigned `0` if the `string`s are equivalent, a positive number if `string1` is **lexicographically** greater than `string2` or a negative number if `string1` is lexicographically less than `string2`. When we say that a `string` is lexicographically less than another, we mean that the `compare` method uses the numerical values of the characters (see Appendix B, ASCII Character Set) in each `string` to determine that the first `string` is less than the second. Because a `string` starting with `'T'` is considered lexicographically greater than a string starting with `'H'`, `result` is assigned a value greater than `0`, as confirmed by the output. A lexicon is a dictionary.

Line 42 compares portions of `string1` and `string3` using an overloaded version of member function `compare`. The first two arguments (`2` and `5`) specify the starting subscript and length of the portion of `string1` (`"sting"`) to compare with `string3`. The third argument is the comparison `string`. The last two arguments (`0` and `5`) are the starting subscript and length of the portion of the comparison `string` being compared (also `"sting"`). The value assigned to `result` is `0` for equality, a positive number if `string1` is lexicographically greater than `string3` or a negative number if `string1` is lexicographically less than `string3`. The two pieces being compared here are identical, so `result` is assigned `0`.

Line 55 uses another overloaded version of function `compare` to compare `string4` and `string2`. The first two arguments are the same—the starting subscript and length. The last argument is the comparison `string`. The value returned is also the same—`0` for equality, a positive number if `string4` is lexicographically greater than `string2` or a negative number if `string4` is lexicographically less than `string2`. Because the two pieces of `string`s being compared here are identical, `result` is assigned `0`.

Line 71 calls member function `compare` to compare the first `3` characters in `string2` to `string4`. Because `"Hel"` is less than `"Hello"`, a value less than zero is returned.

18.4 Substrings

Class `string` provides member function **`substr`** for retrieving a substring from a `string`. The result is a new `string` object that's copied from the source `string`. Figure 18.3 demonstrates `substr`. The program declares and initializes a `string` at line 9. Line 13 uses member function `substr` to retrieve a substring from `string1`. The first argument specifies the beginning subscript of the desired substring; the second argument specifies the substring's length.

```
1  // Fig. 18.3: Fig18_03.cpp
2  // Demonstrating string member function substr.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "The airplane landed on time." );
10
11    // retrieve substring "plane" which
12    // begins at subscript 7 and consists of 5 characters
13    cout << string1.substr( 7, 5 ) << endl;
14 } // end main
```

```
plane
```

Fig. 18.3 | Demonstrating `string` member function `substr`.

18.5 Swapping strings

Class `string` provides member function **`swap`** for swapping `string`s. Figure 18.4 swaps two `string`s. Lines 9–10 declare and initialize `string`s `first` and `second`. Each `string` is then output. Line 15 uses `string` member function `swap` to swap the values of `first` and `second`. The two `string`s are printed again to confirm that they were indeed swapped. The `string` member function `swap` is useful for implementing programs that sort strings.

```
1  // Fig. 18.4: Fig18_04.cpp
2  // Using the swap function to swap two strings.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string first( "one" );
10    string second( "two" );
11
12    // output strings
13    cout << "Before swap:\n first: " << first << "\nsecond: " << second;
```

Fig. 18.4 | Using function `swap` to swap two `string`s. (Part 1 of 2.)

```

   first.swap( second ); // swap strings

   cout << "\n\nAfter swap:\n first: " << first
      << "\nsecond: " << second << endl;
} // end main
```

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

Fig. 18.4 | Using function swap to swap two strings. (Part 2 of 2.)

18.6 string Characteristics

Class string provides member functions for gathering information about a string's size, length, capacity, maximum length and other characteristics. A string's size or length is the number of characters currently stored in the string. A string's **capacity** is the number of characters that can be stored in the string without allocating more memory. The capacity of a string must be at least equal to the current size of the string, though it can be greater. The exact capacity of a string depends on the implementation. The **maximum size** is the largest possible size a string can have. If this value is exceeded, a length_error exception is thrown. Figure 18.5 demonstrates string class member functions for determining various characteristics of strings.

```
// Fig. 18.5: Fig18_05.cpp
// Demonstrating member functions related to size and capacity.
#include <iostream>
#include <string>
using namespace std;

void printStatistics( const string & );

int main()
{
   string string1; // empty string

   cout << "Statistics before input:\n" << boolalpha;
   printStatistics( string1 );

   // read in only "tomato" from "tomato soup"
   cout << "\n\nEnter a string: ";
   cin >> string1; // delimited by whitespace
   cout << "The string entered was: " << string1;

```

Fig. 18.5 | Printing string characteristics. (Part 1 of 3.)

```
   cout << "\nStatistics after input:\n";
   printStatistics( string1 );

   // read in "soup"
   cin >> string1; // delimited by whitespace
   cout << "\n\nThe remaining string is: " << string1 << endl;
   printStatistics( string1 );

   // append 46 characters to string1
   string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
   cout << "\n\nstring1 is now: " << string1 << endl;
   printStatistics( string1 );

   // add 10 elements to string1
   string1.resize( string1.length() + 10 );
   cout << "\n\nStats after resizing by (length + 10):\n";
   printStatistics( string1 );
   cout << endl;
} // end main

// display string statistics
void printStatistics( const string &stringRef )
{
   cout << "capacity: " << stringRef.capacity() << "\nmax size: "
      << stringRef.max_size() << "\nsize: " << stringRef.size()
      << "\nlength: " << stringRef.length()
      << "\nempty: " << stringRef.empty();
} // end printStatistics
```

```
Statistics before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15

max size: 4294967293
size: 6
length: 6
empty: false

The remaining string is: soup
capacity: 15
max size: 4294967293
size: 4
length: 4
empty: false
```

Fig. 18.5 | Printing string characteristics. (Part 2 of 3.)

```
string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacity: 63
max size: 4294967293
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967293
size: 60
length: 60
empty: false
```

Fig. 18.5 | Printing `string` characteristics. (Part 3 of 3.)

The program declares empty `string string1` (line 11) and passes it to function `printStatistics` (line 14). Function `printStatistics` (lines 42–48) takes a reference to a `const string` as an argument and outputs the capacity (using member function **`capacity`**), maximum size (using member function **`max_size`**), size (using member function `size`), length (using member function `length`) and whether the `string` is empty (using member function `empty`). The initial call to `printStatistics` indicates that the initial values for the capacity, size and length of `string1` are `0`.

The size and length of `0` indicate that there are no characters stored in `string`. Because the initial capacity is `0`, when characters are placed in `string1`, memory is allocated to accommodate the new characters. Recall that the size and length are always identical. In this implementation, the maximum size is 4294967293. Object `string1` is an empty `string`, so function `empty` returns `true`.

Line 18 inputs a string. In this example, `"tomato soup"` is input. Because a space character is a delimiter, only `"tomato"` is stored in `string1`; however, `"soup"` remains in the input buffer. Line 22 calls function `printStatistics` to output statistics for `string1`. Notice in the output that the length is `6` and the capacity is `15`.

Line 25 reads `"soup"` from the input buffer and stores it in `string1`, thereby replacing `"tomato"`. Line 27 passes `string1` to `printStatistics`.

Line 30 uses the overloaded `+=` operator to concatenate a 46-character-long string to `string1`. Line 32 passes `string1` to `printStatistics`. The capacity has increased to `63` elements and the length is now `50`.

Line 35 uses member function **`resize`** to increase the length of `string1` by `10` characters. The additional elements are set to null characters. The output shows that the capacity has not changed and the length is now `60`.

18.7 Finding Substrings and Characters in a string

Class `string` provides `const` member functions for finding substrings and characters in a `string`. Figure 18.6 demonstrates the `find` functions.

String `string1` is declared and initialized in line 9. Line 14 attempts to find `"is"` in `string1` using function **`find`**. If `"is"` is found, the subscript of the starting location of that string is returned. If the `string` is not found, the value **`string::npos`** (a `public static` constant defined in class `string`) is returned. This value is returned by the `string` `find`-related functions to indicate that a substring or character was not found in the `string`.

```
// Fig. 18.6: Fig18_06.cpp
// Demonstrating the string find member functions.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string string1( "noon is 12 pm; midnight is not." );
   int location;

   // find "is" at location 5 and 24
   cout << "Original string:\n" << string1
      << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
      << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );

   // find 'o' at location 1
   location = string1.find_first_of( "misop" );
   cout << "\n\n(find_first_of) found '" << string1[ location ]
      << "' from the group \"misop\" at: " << location;

   // find 'o' at location 29
   location = string1.find_last_of( "misop" );
   cout << "\n\n(find_last_of) found '" << string1[ location ]
      << "' from the group \"misop\" at: " << location;

   // find '1' at location 8
   location = string1.find_first_not_of( "noi spm" );
   cout << "\n\n(find_first_not_of) '" << string1[ location ]
      << "' is not contained in \"noi spm\" and was found at: "
      << location;

   // find '.' at location 12
   location = string1.find_first_not_of( "12noi spm" );
   cout << "\n\n(find_first_not_of) '" << string1[ location ]
      << "' is not contained in \"12noi spm\" and was "
      << "found at: " << location << endl;

   // search for characters not in string1
   location = string1.find_first_not_of(
      "noon is 12 pm; midnight is not." );
   cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
      << " returned: " << location << endl;
} // end main
```

```
Original string:
noon is 12 pm; midnight is not.

(find) "is" was found at: 5
(rfind) "is" was found at: 24
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 1 of 2.)

```
(find_first_of) found 'o' from the group "misop" at: 1
(find_last_of) found 'o' from the group "misop" at: 29
(find_first_not_of) '1' is not contained in "noi spm" and was found at: 8
(find_first_not_of) '.' is not contained in "12noi spm" and was found at: 12
find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

Fig. 18.6 | Demonstrating the `string` `find` functions. (Part 2 of 2.)

Line 15 uses member function **`rfind`** to search `string1` backward (i.e., right-to-left). If `"is"` is found, the subscript location is returned. If the string is not found, `string::npos` is returned. [*Note:* The rest of the find functions presented in this section return the same type unless otherwise noted.]

Line 18 uses member function **`find_first_of`** to locate the first occurrence in `string1` of any character in `"misop"`. The searching is done from the beginning of `string1`. The character `'o'` is found in element 1.

Line 23 uses member function **`find_last_of`** to find the last occurrence in `string1` of any character in `"misop"`. The searching is done from the end of `string1`. The character `'o'` is found in element 29.

Line 28 uses member function **`find_first_not_of`** to find the first character in `string1` not contained in `"noi spm"`. The character `'1'` is found in element 8. Searching is done from the beginning of `string1`.

Line 34 uses member function `find_first_not_of` to find the first character not contained in `"12noi spm"`. The character `'.'` is found in element 12. Searching is done from the beginning of `string1`.

Lines 40–41 use member function `find_first_not_of` to find the first character not contained in `"noon is 12 pm; midnight is not."`. In this case, the `string` being searched contains every character specified in the string argument. Because a character was not found, `string::npos` (which has the value `-1` in this case) is returned.

18.8 Replacing Characters in a `string`

Figure 18.7 demonstrates `string` member functions for replacing and erasing characters. Lines 10–14 declare and initialize `string` `string1`. Line 20 uses `string` member function **`erase`** to erase everything from (and including) the character in position 62 to the end of `string1`. [*Note:* Each newline character occupies one character in the `string`.]

```
1  // Fig. 18.7: Fig18_07.cpp
2  // Demonstrating string member functions erase and replace.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 1 of 3.)

```
   // compiler concatenates all parts into one string
   string string1( "The values in any left subtree"
      "\nare less than the value in the"
      "\nparent node and the values in"
      "\nany right subtree are greater"
      "\nthan the value in the parent node" );

   cout << "Original string:\n" << string1 << endl << endl;

   // remove all characters from (and including) location 62
   // through the end of string1
   string1.erase( 62 );

   // output new string
   cout << "Original string after erase:\n" << string1
      << "\n\nAfter first replacement:\n";

   int position = string1.find( " " ); // find first space

   // replace all spaces with period
   while ( position != string::npos )
   {
      string1.replace( position, 1, "." );
      position = string1.find( " ", position + 1 );
   } // end while

   cout << string1 << "\n\nAfter second replacement:\n";

   position = string1.find( "." ); // find first period

   // replace all periods with two semicolons
   // NOTE: this will overwrite characters
   while ( position != string::npos )
   {
      string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
      position = string1.find( ".", position + 1 );
   } // end while

   cout << string1 << endl;
} // end main
```

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node

Original string after erase:
The values in any left subtree
are less than the value in the
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 2 of 3.)

```
After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the


After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

Fig. 18.7 | Demonstrating functions `erase` and `replace`. (Part 3 of 3.)

Lines 26–33 use `find` to locate each occurrence of the space character. Each space is then replaced with a period by a call to `string` member function **`replace`**. Function `replace` takes three arguments: the subscript of the character in the `string` at which replacement should begin, the number of characters to replace and the replacement string. Member function `find` returns `string::npos` when the search character is not found. In line 32, `1` is added to `position` to continue searching at the location of the next character.

Lines 37–45 use function `find` to find every period and another overloaded function `replace` to replace every period and its following character with two semicolons. The arguments passed to this version of `replace` are the subscript of the element where the replace operation begins, the number of characters to replace, a replacement character string from which a substring is selected to use as replacement characters, the element in the character string where the replacement substring begins and the number of characters in the replacement character string to use.

18.9 Inserting Characters into a `string`

Class `string` provides member functions for inserting characters into a `string`. Figure 18.8 demonstrates the `string` `insert` capabilities.

The program declares, initializes then outputs `string`s `string1`, `string2`, `string3` and `string4`. Line 19 uses `string` member function **`insert`** to insert `string2`'s content before element 10 of `string1`.

Line 22 uses `insert` to insert `string4` before `string3`'s element 3. The last two arguments specify the starting and last element of `string4` that should be inserted. Using `string::npos` causes the entire `string` to be inserted.

```
1  // Fig. 18.8: Fig18_08.cpp
2  // Demonstrating class string insert member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "beginning end" );
10    string string2( "middle " );
11    string string3( "12345678" );
12    string string4( "xx" );
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 1 of 2.)

```

   cout << "Initial strings:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << "\n\n";

   // insert "middle" at location 10 in string1
   string1.insert( 10, string2 );

   // insert "xx" at location 3 in string3
   string3.insert( 3, string4, 0, string::npos );

   cout << "Strings after insert:\nstring1: " << string1
      << "\nstring2: " << string2 << "\nstring3: " << string3
      << "\nstring4: " << string4 << endl;
} // end main
```

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

Fig. 18.8 | Demonstrating the `string` `insert` member functions. (Part 2 of 2.)

18.10 Conversion to C-Style Pointer-Based char * Strings

Class `string` provides member functions for converting `string` class objects to C-style pointer-based strings. As mentioned earlier, unlike pointer-based strings, `string`s are not necessarily null terminated. These conversion functions are useful when a given function takes a pointer-based string as an argument. Figure 18.9 demonstrates conversion of `string`s to pointer-based strings.

```
// Fig. 18.9: Fig18_09.cpp
// Converting to C-style strings.
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string string1( "STRINGS" ); // string constructor with char* arg
   const char *ptr1 = 0; // initialize *ptr1
   int length = string1.length();
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 1 of 2.)

```
12       char *ptr2 = new char[ length + 1 ]; // including null
13
14       // copy characters from string1 into allocated memory
15       string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char*
16       ptr2[ length ] = '\0'; // add null terminator
17
18       cout << "string string1 is " << string1
19          << "\nstring1 converted to a C-Style string is "
20          << string1.c_str() << "\nptr1 is ";
21
22       // Assign to pointer ptr1 the const char * returned by
23       // function data(). NOTE: this is a potentially dangerous
24       // assignment. If string1 is modified, pointer ptr1 can
25       // become invalid.
26       ptr1 = string1.data();
27
28       // output each character using pointer
29       for ( int i = 0; i < length; ++i )
30          cout << *( ptr1 + i ); // use pointer arithmetic
31
32       cout << "\nptr2 is " << ptr2 << endl;
33       delete [] ptr2; // reclaim dynamically allocated memory
34    } // end main
```

```
string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

Fig. 18.9 | Converting `string`s to C-style strings and character arrays. (Part 2 of 2.)

The program declares a `string`, an `int` and two `char` pointers (lines 9–12). The `string string1` is initialized to `"STRINGS"`, `ptr1` is initialized to `0` and `length` is initialized to the length of `string1`. Memory of sufficient size to hold a pointer-based string equivalent of `string string1` is allocated dynamically and attached to `char` pointer `ptr2`.

Line 15 uses `string` member function **`copy`** to copy object `string1` into the `char` array pointed to by `ptr2`. Line 16 manually places a terminating null character in the array pointed to by `ptr2`.

Line 20 uses function **`c_str`** to obtain a `const char *` that points to a null terminated C-style string with the same content as `string1`. The pointer is passed to the stream insertion operator for output.

Line 26 assigns the `const char * ptr1` a pointer returned by class `string` member function **`data`**. This member function returns a non-null-terminated C-style character array. We do not modify `string string1` in this example. If `string1` were to be modified (e.g., the `string`'s dynamic memory changes its address due to a member function call such as `string1.insert( 0, "abcd" );`), `ptr1` could become invalid—which could lead to unpredictable results.

Lines 29–30 use pointer arithmetic to output the character array pointed to by `ptr1`. In lines 32–33, the C-style string pointed to by `ptr2` is output and the memory allocated for `ptr2` is `delete`d to avoid a memory leak.

Common Programming Error 18.2

Not terminating the character array returned by `data` *with a null character can lead to execution-time errors.*

Good Programming Practice 18.1

Whenever possible, use the more robust `string` *class objects rather than C-style pointer-based strings.*

18.11 Iterators

Class `string` provides iterators for forward and backward traversal of `string`s. Iterators provide access to individual characters with syntax that's similar to pointer operations. *Iterators are not range checked.* In this section we provide "mechanical examples" to demonstrate the use of iterators. We discuss more robust uses of iterators in Chapter 22, Standard Template Library (STL). Figure 18.10 demonstrates iterators.

```
1  // Fig. 18.10: Fig18_10.cpp
2  // Using an iterator to output a string.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string string1( "Testing iterators" );
10    string::const_iterator iterator1 = string1.begin();
11
12    cout << "string1 = " << string1
13       << "\n(Using iterator iterator1) string1 is: ";
14
15    // iterate through string
16    while ( iterator1 != string1.end() )
17    {
18       cout << *iterator1; // dereference iterator to get char
19       ++iterator1; // advance iterator to next char
20    } // end while
21
22    cout << endl;
23 } // end main
```

```
string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators
```

Fig. 18.10 | Using an iterator to output a `string`.

Lines 9–10 declare `string` `string1` and **`string::const_iterator`** `iterator1`. A `const_iterator` is an iterator that cannot modify the `string`—in this case the `string` through which it's iterating. Iterator `iterator1` is initialized to the beginning of `string1` with the `string` class member function **`begin`**. Two versions of `begin` exist—one that returns an iterator for iterating through a non-const string and a const version that returns a const_iterator for iterating through a const string. Line 12 outputs string1.

Lines 16–20 use iterator iterator1 to "walk through" string1. Class string member function **end** returns an iterator (or a const_iterator) for the position past the last element of string1. Each element is printed by dereferencing the iterator much as you'd dereference a pointer, and the iterator is advanced one position using operator ++.

Class string provides member functions **rend** and **rbegin** for accessing individual string characters in reverse from the end of a string toward the beginning. Member functions rend and rbegin return **reverse_iterators** or **const_reverse_iterators** (based on whether the string is non-const or const). Exercise 18.8 asks you to write a program that demonstrates these capabilities.

Good Programming Practice 18.2

When the operations involving the iterator should not modify the data being processed, use a const_iterator*. This is another example of employing the principle of least privilege.*

18.12 String Stream Processing

In addition to standard stream I/O and file stream I/O, C++ stream I/O includes capabilities for inputting from, and outputting to, strings in memory. These capabilities often are referred to as **in-memory I/O** or **string stream processing**.

Input from a string is supported by class **istringstream**. Output to a string is supported by class **ostringstream**. The class names istringstream and ostringstream are actually aliases defined by the typedefs

```
typedef basic_istringstream< char > istringstream;
typedef basic_ostringstream< char > ostringstream;
```

Class templates basic_istringstream and basic_ostringstream provide the same functionality as classes istream and ostream plus other member functions specific to in-memory formatting. Programs that use in-memory formatting must include the **<sstream>** and <iostream> headers.

One application of these techniques is data validation. A program can read an entire line at a time from the input stream into a string. Next, a validation routine can scrutinize the contents of the string and correct (or repair) the data, if necessary. Then the program can proceed to input from the string, knowing that the input data is in the proper format.

Outputting to a string is a nice way to take advantage of the powerful output formatting capabilities of C++ streams. Data can be prepared in a string to mimic the edited screen format. That string could be written to a disk file to preserve the screen image.

An ostringstream object uses a string object to store the output data. The **str** member function of class ostringstream returns a copy of that string.

Figure 18.11 demonstrates an ostringstream object. The program creates ostringstream object outputString (line 10) and uses the stream insertion operator to output a series of strings and numerical values to the object.

Lines 22–23 output string string1, string string2, string string3, double double1, string string4, int integer, string string5 and the address of int integer—all to outputString in memory. Line 26 uses the stream insertion operator and the call outputString.str() to display a copy of the string created in lines 22–23. Line

```
1 // Fig. 18.11: Fig18_11.cpp
2 // Using an ostringstream object.
3 #include <iostream>
4 #include <string>
5 #include <sstream> // header for string stream processing
6 using namespace std;
7
8 int main()
9 {
10    ostringstream outputString; // create ostringstream instance
11
12    string string1( "Output of several data types " );
13    string string2( "to an ostringstream object:" );
14    string string3( "\n        double: " );
15    string string4( "\n           int: " );
16    string string5( "\naddress of int: " );
17
18    double double1 = 123.4567;
19    int integer = 22;
20
21    // output strings, double and int to ostringstream outputString
22    outputString << string1 << string2 << string3 << double1
23       << string4 << integer << string5 << &integer;
24
25    // call str to obtain string contents of the ostringstream
26    cout << "outputString contains:\n" << outputString.str();
27
28    // add additional characters and call str to output string
29    outputString << "\nmore characters added";
30    cout << "\n\nafter additional stream insertions,\n"
31       << "outputString contains:\n" << outputString.str() << endl;
32 } // end main
```

```
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540
more characters added
```

Fig. 18.11 | Using an `ostringstream` object.

29 demonstrates that more data can be appended to the `string` in memory by simply issuing another stream insertion operation to `outputString`. Lines 30–31 display `string` `outputString` after appending additional characters.

An `istringstream` object inputs data from a `string` in memory to program variables. Data is stored in an `istringstream` object as characters. Input from the `istringstream` object works identically to input from any file. The end of the `string` is interpreted by the `istringstream` object as end-of-file.

Figure 18.12 demonstrates input from an `istringstream` object. Lines 10–11 create `string input` containing the data and `istringstream` object `inputString` constructed to contain the data in `string input`. The `string input` contains the data

```
Input test 123 4.7 A
```

which, when read as input to the program, consist of two strings (`"Input"` and `"test"`), an `int` (`123`), a `double` (`4.7`) and a `char` (`'A'`). These characters are extracted to variables `string1`, `string2`, `integer`, `double1` and `character` in line 18.

The data is then output in lines 20–23. The program attempts to read from `inputString` again in line 27. The `if` condition in line 30 uses function `good` (Section 15.8) to test if any data remains. Because no data remains, the function returns `false` and the `else` part of the `if…else` statement is executed.

```
1  // Fig. 18.12: Fig18_12.cpp
2  // Demonstrating input from an istringstream object.
3  #include <iostream>
4  #include <string>
5  #include <sstream>
6  using namespace std;
7
8  int main()
9  {
10    string input( "Input test 123 4.7 A" );
11    istringstream inputString( input );
12    string string1;
13    string string2;
14    int integer;
15    double double1;
16    char character;
17
18    inputString >> string1 >> string2 >> integer >> double1 >> character;
19
20    cout << "The following items were extracted\n"
21       << "from the istringstream object:" << "\nstring: " << string1
22       << "\nstring: " << string2 << "\n   int: " << integer
23       << "\ndouble: " << double1 << "\n  char: " << character;
24
25    // attempt to read from empty stream
26    long value;
27    inputString >> value;
28
29    // test stream results
30    if ( inputString.good() )
31       cout << "\n\nlong value is: " << value << endl;
32    else
33       cout << "\n\ninputString is empty" << endl;
34 } // end main
```

Fig. 18.12 | Demonstrating input from an `istringstream` object. (Part 1 of 2.)

```
The following items were extracted
from the istringstream object:
string: Input
string: test
   int: 123
double: 4.7
  char: A

inputString is empty
```

Fig. 18.12 | Demonstrating input from an istringstream object. (Part 2 of 2.)

18.13 Wrap-Up

This chapter discussed the details of C++ Standard Library class string. We discussed assigning, concatenating, comparing, searching and swapping strings. We also introduced a number of methods to determine string characteristics, to find, replace and insert characters in a string, and to convert strings to C-style strings and vice versa. You also learned about string iterators and performing input from and output to strings in memory. In Chapter 19, Searching and Sorting, we discuss the binary search algorithm and the merge sort algorithm. We also use Big O notation to analyze and compare the efficiency of various searching and sorting algorithms.

Summary

Section 18.1 Introduction

- Class template basic_string provides typical string-manipulation operations.
- The typedef statement

```
typedef basic_string< char > string;
```

  creates the alias type string for basic_string<char> (p. 697). A typedef also is provided for the wchar_t type (wstring). Type wchar_t (p. 697) normally stores two-byte (16-bit) characters for supporting other character sets. The size of wchar_t is not fixed by the standard.
- To use strings, include C++ Standard Library header <string>.
- Assigning a single character to a string object is permitted in an assignment statement.
- strings are not necessarily null terminated.
- Most string member functions take as arguments a starting subscript location and the number of characters on which to operate.

Section 18.2 string Assignment and Concatenation

- Class string provides overloaded operator= and member function assign (p. 698) for assignments.
- The subscript operator, [], provides read/write access to any element of a string.
- string member function at (p. 700) provides checked access—going past either end of the string throws an out_of_range exception. The subscript operator, [], does not provide checked access.
- The overloaded + and += operators and member function append (p. 700) perform string concatenation.

Section 18.3 Comparing **strings**
- Class string provides overloaded ==, !=, <, >, <= and >= operators for string comparisons.
- string member function compare (p. 702) compares two strings (or substrings) and returns 0 if the strings are equal, a positive number if the first string is lexicographically (p. 702) greater than the second or a negative number if the first string is lexicographically less than the second.

Section 18.4 Substrings
- string member function substr (p. 703) retrieves a substring from a string.

Section 18.5 Swapping **strings**
- string member function swap (p. 703) swaps the contents of two strings.

Section 18.6 **string** *Characteristics*
- string member functions size and length (p. 698) return the number of characters currently stored in a string.
- string member function capacity (p. 706) returns the total number of characters that can be stored in a string without increasing the amount of memory allocated to the string.
- string member function max_size (p. 706) returns the maximum size a string can have.
- string member function resize (p. 706) changes the length of a string.
- string member function empty returns true if a string is empty.

Section 18.7 Finding Substrings and Characters in a **string**
- Class string find functions (p. 706) find, rfind, find_first_of, find_last_of and find_first_not_of locate substrings or characters in a string.

Section 18.8 Replacing Characters in a **string**
- string member function erase (p. 708) deletes elements of a string.x
- string member function replace (p. 710) replaces characters in a string.

Section 18.9 Inserting Characters into a **string**
- string member function insert (p. 710) inserts characters in a string.

Section 18.10 Conversion to C-Style Pointer-Based **char** ** Strings*
- string member function c_str (p. 712) returns a const char * pointing to a null-terminated C-style character string that contains all the characters in a string.
- string member function data (p. 712) returns a const char * pointing to a non-null-terminated C-style character array that contains all the characters in a string.

Section 18.11 Iterators
- Class string provides member functions begin and end (p. 713) to iterate through individual elements.
- Class string provides member functions rend and rbegin (p. 714) for accessing individual string characters in reverse from the end of a string toward the beginning.

Section 18.12 String Stream Processing
- Input from a string is supported by type istringstream (p. 714). Output to a string is supported by type ostringstream (p. 714).
- ostringstream member function str (p. 714) returns the string from the stream.

Self-Review Exercises

18.1 Fill in the blanks in each of the following:

a) Header ________ must be included for class `string`.
b) Class `string` belongs to the ________ `namespace`.
c) Function ________ deletes characters from a `string`.
d) Function ________ finds the first occurrence of one of a specified set of characters from a `string`.

18.2 State which of the following statements are *true* and which are *false*. If a statement is *false*, explain why.

a) Concatenation of `string` objects can be performed with the addition assignment operator, `+=`.
b) Characters within a `string` begin at index `0`.
c) The assignment operator, `=`, copies a `string`.
d) A C-style string is a `string` object.

18.3 Find the error(s) in each of the following, and explain how to correct it (them):

a)
```
string string1( 28 ); // construct string1
string string2( 'z' ); // construct string2
```
b)
```
// assume std namespace is known
const char *ptr = name.data(); // name is "joe bob"
ptr[ 3 ] = '-';
cout << ptr << endl;
```

Answers to Self-Review Exercises

18.1 a) `<string>`. b) `std`. c) `erase`. d) `find_first_of`.

18.2 a) True.
b) True.
c) True.
d) False. A `string` is an object that provides many different services. A C-style string does not provide any services. C-style strings are null terminated; `string`s are not necessarily null terminated. C-style strings are pointers and `string`s are objects.

18.3 a) Constructors for class `string` do not exist for integer and character arguments. Other valid constructors should be used—converting the arguments to `string`s if need be.
b) Function `data` does not add a null terminator. Also, the code attempts to modify a `const char`. Replace all of the lines with the code:
```
cout << name.substr( 0, 3 ) + "-" + name.substr( 4 ) << endl;
```

Exercises

18.4 ***(Fill in the Blanks)*** Fill in the blanks in each of the following:

a) Class `string` member function ________ converts a `string` to a C-style string.
b) Class `string` member function ________ is used for assignment.
c) ________ is the return type of function `rbegin`.
d) Class `string` member function ________ is used to retrieve a substring.

18.5 ***(True or False)*** State which of the following statements are *true* and which are *false*. If a statement is *false*, explain why.

a) `string`s are always null terminated.
b) Class `string` member function `max_size` returns the maximum size for a `string`.
c) Class `string` member function `at` can throw an `out_of_range` exception.
d) Class `string` member function `begin` returns an `iterator`.

18.6 *(Find Code Errors)* Find any errors in the following and explain how to correct them:

a)
```
std::cout << s.data() << std::endl; // s is "hello"
```
b)
```
erase( s.rfind( "x" ), 1 ); // s is "xenon"
```
c)
```
string& foo()
{
   string s( "Hello" );
   ...   // other statements
   return;
} // end function foo
```

18.7 (*Simple Encryption*) Some information on the Internet may be encrypted with a simple algorithm known as "rot13," which rotates each character by 13 positions in the alphabet. Thus, `'a'` corresponds to `'n'`, and `'x'` corresponds to `'k'`. rot13 is an example of **symmetric key encryption**. With symmetric key encryption, both the encrypter and decrypter use the same key.

a) Write a program that encrypts a message using rot13.
b) Write a program that decrypts the scrambled message using 13 as the key.
c) After writing the programs of part (a) and part (b), briefly answer the following question: If you did not know the key for part (b), how difficult do you think it would be to break the code? What if you had access to substantial computing power (e.g., supercomputers)? In Exercise 18.25 we ask you to write a program to accomplish this.

18.8 *(Using `string` Iterators)* Write a program using iterators that demonstrates the use of functions `rbegin` and `rend`.

18.9 *(Words Ending in "r" or "ay")* Write a program that reads in several `string`s and prints only those ending in "r" or "ay". Only lowercase letters should be considered.

18.10 *(`string` Concatenation)* Write a program that separately inputs a first name and a last name and concatenates the two into a new `string`. Show two techniques for accomplishing this task.

18.11 *(Hangman Game)* Write a program that plays the game of Hangman. The program should pick a word (which is either coded directly into the program or read from a text file) and display the following:

```
Guess the word:    XXXXXX
```

Each `X` represents a letter. The user tries to guess the letters in the word. The appropriate response `yes` or `no` should be displayed after each guess. After each incorrect guess, display the diagram with another body part filled. After seven incorrect guesses, the user should be hanged. The display should look as follows:

```
 O
/|\
 |
/ \
```

After each guess, display all user guesses. If the user guesses the word correctly, display

```
Congratulations!!! You guessed my word. Play again? yes/no
```

18.12 *(Printing a `string` Backward)* Write a program that inputs a `string` and prints the `string` backward. Convert all uppercase characters to lowercase and all lowercase characters to uppercase.

18.13 *(Alphabetizing Animal Names)* Write a program that uses the comparison capabilities introduced in this chapter to alphabetize a series of animal names. Only uppercase letters should be used for the comparisons.

18.14 *(Cryptograms)* Write a program that creates a cryptogram out of a `string`. A cryptogram is a message or word in which each letter is replaced with another letter. For example the `string`

```
The bird was named squawk
```

might be scrambled to form

```
cin vrjs otz ethns zxqtop
```

Spaces are not scrambled. In this particular case, `'T'` was replaced with `'x'`, each `'a'` was replaced with `'h'`, etc. Uppercase letters become lowercase letters in the cryptogram. Use techniques similar to those in Exercise 18.7.

18.15 *(Solving Cryptograms)* Modify Exercise 18.14 to allow the user to solve the cryptogram. The user should input two characters at a time: The first character specifies a letter in the cryptogram, and the second letter specifies the replacement letter. If the replacement letter is correct, replace the letter in the cryptogram with the replacement letter in uppercase.

18.16 *(Counting Palindromes)* Write a program that inputs a sentence and counts the number of palindromes in it. A palindrome is a word that reads the same backward and forward. For example, `"tree"` is not a palindrome, but `"noon"` is.

18.17 *(Counting Vowels)* Write a program that counts the total number of vowels in a sentence. Output the frequency of each vowel.

18.18 *(String Insertion)* Write a program that inserts the characters `"******"` in the exact middle of a `string`.

18.19 *(Erasing Characters from a `string`)* Write a program that erases the sequences `"by"` and `"BY"` from a `string`.

18.20 *(Replacing Punctuation and Tokenizing `string`s)* Write a program that inputs a line of text, replaces all punctuation marks with spaces and uses the C-string library function `strtok` to tokenize the `string` into individual words.

18.21 *(Reversing a `string` with Iterators)* Write a program that inputs a line of text and prints the text backward. Use iterators in your solution.

18.22 *(Reversing a `string` with Iterators using Recursion)* Write a recursive version of Exercise 18.21.

18.23 *(Using the `erase` Functions with Iterator Arguments)* Write a program that demonstrates the use of the `erase` functions that take `iterator` arguments.

18.24 *(Letter Pyramid)* Write a program that generates the following from the `string` `"abcdefghijklmnopqrstuvwxyz"`:

```
             a
            bcb
           cdedc
          defgfed
         efghihgfe
        fghijkjihgf
       ghijklmlkjihg
      hijklmnonmlkjih
     ijklmnopqponmlkji
    jklmnopqrsrqponmlkj
   klmnopqrstutsrqponmlk
  lmnopqrstuvwvutsrqponml
 mnopqrstuvwxyxwvutsrqponm
nopqrstuvwxyz{zyxwvutsrqpon
```

18.25 *(Simple Decryption)* In Exercise 18.7, we asked you to write a simple encryption algorithm. Write a program that will attempt to decrypt a “rot13” message using simple frequency substitution. (Assume that you do not know the key.) The most frequent letters in the encrypted phrase should be replaced with the most commonly used English letters (a, e, i, o, u, s, t, r, etc.). Write the possibilities to a file. What made the code breaking easy? How can the encryption mechanism be improved?

18.26 *(Sorting `strings`)* Write a version of the selection sort routine (Fig. 8.20) that sorts `strings`. Use function `swap` in your solution.

18.27 *(Enhanced `Employee` Class)* Modify class `Employee` in Figs. 13.2–13.3 by adding a `private` utility function called `isValidSocialSecurityNumber`. This member function should validate the format of a social security number (e.g., `###-##-####`, where `#` is a digit). If the format is valid, return `true`; otherwise return `false`.

Making a Difference

18.28 *(Cooking with Healthier Ingredients)* Obesity in the United States is increasing at an alarming rate. Check the map from the Centers for Disease Control and Prevention (CDC) at `www.cdc.gov/nccdphp/dnpa/Obesity/trend/maps/index.htm`, which shows obesity trends in the United States over the last 20 years. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Write a program that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The program should read a recipe from the user and suggest healthier replacements for some of the ingredients. For simplicity, your program should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in Fig. 18.13. Your program should display a warning such as, "Always consult your physician before making significant changes to your diet."

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup margarine or yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 of a large banana (mashed)
1 cup milk	1 cup soy milk
1/4 cup oil	1/4 cup applesauce
white bread	whole-grain bread

Fig. 18.13 | Common ingredient substitutions.

Your program should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at websites such as:

```
chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
www.pioneerthinking.com/eggsub.html
www.gourmetsleuth.com/conversions.htm
```

Your program should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the program should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

18.29 *(Spam Scanner)* Spam (or junk e-mail) costs U.S. organizations billions of dollars a year in spam-prevention software, equipment, network resources, bandwidth, and lost productivity. Research online some of the most common spam e-mail messages and words, and check your own junk e-mail folder. Create a list of 30 words and phrases commonly found in spam messages. Write an application in which the user enters an e-mail message. Then, scan the message for each of the 30 keywords or phrases. For each occurrence of one of these within the message, add a point to the message's "spam score." Next, rate the likelihood that the message is spam, based on the number of points it received.

18.30 *(SMS Language)* Short Message Service (SMS) is a communications service that allows sending text messages of 160 or fewer characters between mobile phones. With the proliferation of mobile phone use worldwide, SMS is being used in many developing nations for political purposes (e.g., voicing opinions and opposition), reporting news about natural disasters, and so on. For example, check out `comunica.org/radio2.0/archives/87`. Since the length of SMS messages is limited, SMS Language—abbreviations of common words and phrases in mobile text messages, e-mails, instant messages, etc.—is often used. For example, "in my opinion" is "IMO" in SMS Language. Research SMS Language online. Write a program in which the user can enter a message using SMS Language; the program should translate it into English (or your own language). Also provide a mechanism to translate text written in English (or your own language) into SMS Language. One potential problem is that one SMS abbreviation could expand into a variety of phrases. For example, IMO (as used above) could also stand for "International Maritime Organization," "in memory of," "in my opinion," etc.

19 Searching and Sorting

With sobs and tears
he sorted out
Those of the largest size …
—Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
—Robert Herrick

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
—William Shakespeare

It is an immutable law in business that words are words, explanations are explanations, promises are promises — but only performance is reality.
—Harold S. Green

Objectives

In this chapter you'll learn:

- To search for a given value in a vector using binary search.
- To use Big O notation to express the efficiency of searching and sorting algorithms and to compare their performance.
- To sort a vector using the recursive merge sort algorithm.
- To understand the nature of algorithms of constant, linear and quadratic runtime.

Outline

19.1 Introduction

Searching data involves determining whether a value (referred to as the **search key**) is present in the data and, if so, finding the value's location. Two popular search algorithms are the simple linear search (introduced in Section 7.7) and the faster but more complex binary search, which is introduced in this chapter.

Sorting places data in ascending or descending order, based on one or more **sort keys**. A list of names could be sorted alphabetically, bank accounts could be sorted by account number, employee payroll records could be sorted by social security number, and so on. Previously, you learned about insertion sort (Section 7.8) and selection sort (Section 8.6). Here we introduce the more efficient, but more complex merge sort. Figure 19.1 summarizes the searching and sorting algorithms discussed in the book's examples and exercises. This chapter also introduces **Big O notation**, which is used to characterize an algorithm's worst-case runtime—that is, how hard an algorithm may have to work to solve a problem.

Algorithm	Location	Algorithm	Location
Searching Algorithms		*Sorting Algorithms*	
Linear search	Section 7.7	Insertion sort	Section 7.8
Binary search	Section 19.2.2	Selection sort	Section 8.6
Recursive linear search	Exercise 19.8	Recursive merge sort	Section 19.3.3
Recursive binary search	Exercise 19.9	Bubble sort	Exercises 19.5–19.6
Binary tree search	Section 20.7	Bucket sort	Exercise 19.7
Linear search (linked list)	Exercise 20.21	Recursive quicksort	Exercise 19.10
`binary_search` standard library function	Section 22.8.6	Binary tree sort	Section 20.7
		`sort` standard library function	Section 22.8.6
		Heap sort	Section 22.8.12

Fig. 19.1 | Searching and sorting algorithms in this text.

19.2 Searching Algorithms

Looking up a phone number, accessing a website and checking a word's definition in a dictionary all involve searching through large amounts of data. A searching algorithm finds an element that matches a given search key, if such an element does, in fact, exist. There are, however, a number of things that differentiate search algorithms from one another.

The major difference is the amount of effort they require to complete the search. One way to describe this *effort* is with Big O notation. For searching and sorting algorithms, this is particularly dependent on the number of data elements.

In Chapter 7, we discussed the simple linear search algorithm. We'll now discuss the efficiency of the linear search algorithm as measured by Big O notation. Then, we'll introduce a searching algorithm that's relatively efficient but more complex to implement.

19.2.1 Efficiency of Linear Search

Suppose an algorithm simply tests whether the first element of a vector is equal to the second element. If the vector has 10 elements, this algorithm requires only one comparison. If the vector has 1000 elements, the algorithm still requires only one comparison. In fact, the algorithm is independent of the number of vector elements. This algorithm is said to have a **constant runtime**, which is represented in Big O notation as ***O*(1)**. An algorithm that's $O(1)$ does not necessarily require only one comparison. $O(1)$ just means that the number of comparisons is *constant*—it does *not* grow as the size of the vector increases. An algorithm that tests whether the first element of a vector is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered $O(1)$. $O(1)$ is often pronounced "on the order of 1" or more simply "**order 1**."

An algorithm that tests whether the first element of a vector is equal to *any* of the other elements of the vector requires at most $n - 1$ comparisons, where n is the number of elements in the vector. If the vector has 10 elements, the algorithm requires up to nine comparisons. If the vector has 1000 elements, the algorithm requires up to 999 comparisons. As n grows larger, the n part of the expression "dominates," and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described in this paragraph) is said to be ***O*(*n*)**. An $O(n)$ algorithm is referred to as having a **linear runtime**. $O(n)$ is often pronounced "on the order of n" or more simply "**order *n***."

Now suppose you have an algorithm that tests whether *any* element of a vector is duplicated elsewhere in the vector. The first element must be compared with *all the other elements*. The second element must be compared with all the other elements except the first (it was already compared to the first). The third element must be compared with all the other elements except the first two. In the end, this algorithm will end up making $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates and the n term becomes inconsequential. Again, Big O notation highlights the n^2 term, leaving $n^2/2$. As we'll soon see, even constant factors are omitted in Big O notation.

Big O is concerned with how an algorithm's runtime grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons. With this algorithm, *doubling* the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons. Again, doubling the number of elements quadruples the number of comparisons. Both of these algorithms grow as the square of n, so Big O ignores the constant, and both algorithms are considered to be **$O(n^2)$**, which is referred to as **quadratic runtime** and pronounced "on the order of n-squared" or more simply "**order *n*-squared**."

When n is small, $O(n^2)$ algorithms (running on today's billion-operation-per-second personal computers) will not noticeably affect performance. But as n grows, you'll start to notice the performance degradation. An $O(n^2)$ algorithm running on a million-element vector would require a trillion "operations" (where each could actually require several machine instructions to execute). This could require a few hours to execute. A billion-element vector would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, $O(n^2)$ algorithms tend to be easy to write. In this chapter, you'll see algorithms with more favorable Big O measures. Such efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be worth the extra effort, especially as n gets large.

The linear search algorithm runs in $O(n)$ time. The worst case in this algorithm is that *every* element must be checked to determine whether the search key exists in the vector. If the size of the vector is *doubled*, the number of comparisons that the algorithm must perform is also *doubled*. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the vector. But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the vector. If a program needs to perform many searches on large vectors, it may be better to implement a different, more efficient algorithm, such as the binary search which we consider in the next section.

Performance Tip 19.1

Sometimes the simplest algorithms perform poorly. Their virtue is that they're easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.

19.2.2 Binary Search

The **binary search algorithm** is more efficient than the linear search algorithm, but it requires that the vector first be *sorted*. This is only worthwhile when the vector, once sorted, will be searched a great many times—or when the searching application has stringent performance requirements. The first iteration of this algorithm tests the middle element in the vector. If this matches the search key, the algorithm ends. Assuming the vector is sorted in ascending order, then if the search key is *less* than the middle element, the search key cannot match any element in the second half of the vector so the algorithm continues with only the first half of the vector (i.e., the first element up to, but *not* including, the middle element). If the search key is *greater* than the middle element, the search key cannot match any element in the first half of the vector so the algorithm continues with only the second half of the vector (i.e., the element *after* the middle element through the last element). Each iteration tests the *middle value* of the remaining portion of the vector. If the element does not match the search key, the algorithm eliminates half of the remaining elements. The algorithm ends either by finding an element that matches the search key or by reducing the subvector to zero size.

As an example, consider the sorted 15-element vector

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

and the search key 65. A binary search would first check whether 51 is the search key (because 51 is the middle element of the vector). The search key (65) is larger than 51, so 51 is eliminated from consideration along with the first half of the vector (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remainder of the vector) matches the search key. The search key (65) is smaller than 81, so 81 is eliminated from consideration along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of elements to check to three (56, 65 and 77). The algorithm then checks 65 (which matches the search key), and returns the index (9) of the vector element containing 65. In this case, the algorithm required just three comparisons to determine whether a vector element matched the search key. Using a linear search algorithm would have required 10 comparisons. [*Note:* In this example, we've chosen to use a vector with 15 elements, so that there will always be an obvious middle element in the vector. With an even number of elements, the middle of the vector lies between two elements. We implement the algorithm to choose the larger of those two elements.]

Figures 19.2–19.3 define class `BinarySearch` and its member functions, respectively. Class `BinarySearch` is similar to `LinearSearch` (Section 7.7)—it has a constructor, a search function (`binarySearch`), a `displayElements` function, two `private` data members and a `private` utility function (`displaySubElements`). Lines 11–21 of Fig. 19.3 define the constructor. After initializing the vector with random `int`s from `10–99` (lines 17–18), line 20 calls the Standard Library function `sort` on the vector `data`. Recall that the binary search algorithm will work only on a sorted vector. Function **`sort`** requires two arguments that specify the range of elements to sort. These arguments are specified with *iterators* (discussed in detail in Chapter 22, Standard Template Library (STL)). The `vector` member functions `begin` and `end` return iterators that can be used with function `sort` to indicate that all the elements from the beginning to the end should be sorted.

```
// Fig 19.2: BinarySearch.h
// Class that contains a vector of random integers and a function
// that uses binary search to find an integer.
#include <vector>
using namespace std;

class BinarySearch
{
public:
   BinarySearch( int ); // constructor initializes vector
   int binarySearch( int ) const; // perform a binary search on vector
   void displayElements() const; // display vector elements
private:
   int size; // vector size
   vector< int > data; // vector of ints
   void displaySubElements( int, int ) const; // display range of values
}; // end class BinarySearch
```

Fig. 19.2 | `BinarySearch` class definition.

```
1 // Fig 19.3: BinarySearch.cpp
2 // BinarySearch class member-function definition.
3 #include <iostream>
```

Fig. 19.3 | `BinarySearch` class member-function definition. (Part 1 of 3.)

```
#include <cstdlib> // prototypes for functions srand and rand
#include <ctime> // prototype for function time
#include <algorithm> // prototype for sort function
#include "BinarySearch.h" // class BinarySearch definition
using namespace std;

// constructor initializes vector with random ints and sorts the vector
BinarySearch::BinarySearch( int vectorSize )
{
   size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
   srand( time( 0 ) ); // seed using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; ++i )
      data.push_back( 10 + rand() % 90 ); // 10-99

   sort( data.begin(), data.end() ); // sort the data
} // end BinarySearch constructor

// perform a binary search on the data
int BinarySearch::binarySearch( int searchElement ) const
{
   int low = 0; // low end of the search area
   int high = size - 1; // high end of the search area
   int middle = ( low + high + 1 ) / 2; // middle element
   int location = -1; // return value; -1 if not found

   do // loop to search for element
   {
      // print remaining elements of vector to be searched
      displaySubElements( low, high );

      // output spaces for alignment
      for ( int i = 0; i < middle; ++i )
         cout << "   ";

      cout << " * " << endl; // indicate current middle

      // if the element is found at the middle
      if ( searchElement == data[ middle ] )
         location = middle; // location is the current middle
      else if ( searchElement < data[ middle ] ) // middle is too high
         high = middle - 1; // eliminate the higher half
      else // middle element is too low
         low = middle + 1; // eliminate the lower half

      middle = ( low + high + 1 ) / 2; // recalculate the middle
   } while ( ( low <= high ) && ( location == -1 ) );

   return location; // return location of search key
} // end function binarySearch

```

Fig. 19.3 | BinarySearch class member-function definition. (Part 2 of 3.)

```
56 // display values in vector
57 void BinarySearch::displayElements() const
58 {
59    displaySubElements( 0, size - 1 );
60 } // end function displayElements
61
62 // display certain values in vector
63 void BinarySearch::displaySubElements( int low, int high ) const
64 {
65    for ( int i = 0; i < low; ++i ) // output spaces for alignment
66       cout << "   ";
67
68    for ( int i = low; i <= high; ++i ) // output elements left in vector
69       cout << data[ i ] << " ";
70
71    cout << endl;
72 } // end function displaySubElements
```

Fig. 19.3 | BinarySearch class member-function definition. (Part 3 of 3.)

Lines 24–54 define function `binarySearch`. The search key is passed into parameter `searchElement` (line 24). Lines 26–28 calculate the `low` end index, `high` end index and `middle` index of the portion of the vector that the program is currently searching. At the beginning of the function, the `low` end is `0`, the `high` end is the size of the vector minus `1` and the `middle` is the average of these two values. Line 29 initializes the `location` of the found element to `-1`—the value that will be returned if the search key is *not* found. Lines 31–51 loop until `low` is greater than `high` (this occurs when the element is not found) or `location` does not equal `-1` (indicating that the search key was found). Line 43 tests whether the value in the `middle` element is equal to `searchElement`. If this is `true`, line 44 assigns `middle` to `location`. Then the loop terminates and `location` is returned to the caller. Each iteration of the loop tests a single value (line 43) and eliminates half of the remaining values in the vector (line 46 or 48).

Lines 22–38 of Fig. 19.4 loop until the user enters the value `-1`. For each other number the user enters, the program performs a binary search on the data to determine whether it matches an element in the vector. The first line of output from this program is the vector of `int`s, in increasing order. When the user instructs the program to search for `38`, the program first tests the middle element, which is `67` (as indicated by `*`). The search key is less than `67`, so the program eliminates the second half of the vector and tests the middle element from the first half of the vector. The search key equals `38`, so the program returns the index `3`.

```
1 // Fig 19.4: Fig19_04.cpp
2 // BinarySearch test program.
3 #include <iostream>
4 #include "BinarySearch.h" // class BinarySearch definition
5 using namespace std;
6
```

Fig. 19.4 | BinarySearch test program. (Part 1 of 3.)

```
int main()
{
   int searchInt; // search key
   int position; // location of search key in vector

   // create vector and output it
   BinarySearch searchVector ( 15 );
   searchVector.displayElements();

   // get input from user
   cout << "\nPlease enter an integer value (-1 to quit): ";
   cin >> searchInt; // read an int from user
   cout << endl;

   // repeatedly input an integer; -1 terminates the program
   while ( searchInt != -1 )
   {
      // use binary search to try to find integer
      position = searchVector.binarySearch( searchInt );

      // return value of -1 indicates integer was not found
      if ( position == -1 )
         cout << "The integer " << searchInt << " was not found.\n";
      else
         cout << "The integer " << searchInt
            << " was found in position " << position << ".\n";

      // get input from user
      cout << "\n\nPlease enter an integer value (-1 to quit): ";
      cin >> searchInt; // read an int from user
      cout << endl;
   } // end while
} // end main
```

```
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
The integer 38 was found in position 3.


Please enter an integer value (-1 to quit): 91

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
                        73 74 82 89 90 91 95
                                  *
                                    90 91 95
                                        *
The integer 91 was found in position 13.
```

Fig. 19.4 | BinarySearch test program. (Part 2 of 3.)

```
Please enter an integer value (-1 to quit): 25

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
26 31 33
    *
26
 *
The integer 25 was not found.

Please enter an integer value (-1 to quit): -1
```

Fig. 19.4 | `BinarySearch` test program. (Part 3 of 3.)

Efficiency of Binary Search

In the worst-case scenario, searching a sorted vector of 1023 elements will take only 10 comparisons when using a binary search. Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration half of the remaining vector) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 ($2^{10} - 1$) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, a vector of 1,048,575 ($2^{20} - 1$) elements takes a maximum of 20 comparisons to find the key, and a vector of about one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous improvement in performance over the linear search. For a one-billion-element vector, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted vector is the exponent of the first power of 2 greater than the number of elements in the vector, which is represented as $\log_2 n$. All logarithms grow at roughly the same rate, so in Big O notation the base can be omitted. This results in a Big O of **$O(\log n)$** for a binary search, which is also known as **logarithmic runtime** and pronounced “on the order of log n” or more simply “**order log n**.”

19.3 Sorting Algorithms

Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications. A bank sorts all of its checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their lists of accounts by last name and, further, by first name to make it easy to find phone numbers. Virtually every organization must sort some data, and often, massive amounts of it. Sorting data is an intriguing, computer-intensive problem that has attracted intense research efforts.

An important point to understand about sorting is that the end result—the sorted vector—will be the same no matter which algorithm you use to sort the vector. The choice of algorithm affects only the runtime and memory use of the program. In previous chapters, we introduced the selection sort and insertion sort—simple algorithms to implement, but inefficient. The next section examines the efficiency of these two algorithms using Big O notation. The last algorithm—merge sort, which we introduce in this chapter—is much faster but is more difficult to implement.

19.3.1 Efficiency of Selection Sort

Selection sort is an easy-to-implement, but inefficient, sorting algorithm. Its first iteration selects the smallest element in the vector and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element, leaving the largest element in the last index. After the i^{th} iteration, the smallest i elements of the vector will be sorted into increasing order in the first i elements of the vector.

The selection sort algorithm iterates $n - 1$ times, each time swapping the smallest remaining element into its sorted position. Locating the smallest remaining element requires $n - 1$ comparisons during the first iteration, $n - 2$ during the second iteration, then $n - 3, \ldots, 3, 2, 1$. This results in a total of $n(n - 1)/2$ or $(n^2 - n)/2$ comparisons. In Big O notation, smaller terms drop out and constants are ignored, leaving a final Big O of $O(n^2)$.

19.3.2 Efficiency of Insertion Sort

Insertion sort is another simple, but inefficient, sorting algorithm. The algorithm's first iteration takes the second element in the vector and, if it's less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the i^{th} iteration of this algorithm, the first i elements in the original vector will be sorted.

Insertion sort iterates $n - 1$ times, inserting an element into the appropriate position in the elements sorted so far. For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements—$n - 1$ comparisons in the worst case. Each individual repetition statement runs in $O(n)$ time. For determining Big O notation, *nested* statements mean that you must *multiply* the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each $O(n)$ iteration of the outer loop, there will be $O(n)$ iterations of the inner loop, resulting in a Big O of $O(n * n)$ or $O(n^2)$.

19.3.3 Merge Sort (A Recursive Implementation)

Merge sort is an efficient sorting algorithm but is conceptually more complex than selection sort and insertion sort. The merge sort algorithm sorts a vector by splitting it into two equal-sized subvectors, sorting each subvector then *merging* them into one larger vector. With an odd number of elements, the algorithm creates the two subvectors such that one has one more element than the other.

Merge sort performs the merge by looking at the first element in each vector, which is also the smallest element in the vector. Merge sort takes the smallest of these and places it in the first element of the larger, sorted vector. If there are still elements in the subvector, merge sort looks at the second element in that subvector (which is now the smallest element remaining) and compares it to the first element in the other subvector. Merge sort continues this process until the larger vector is filled.

The implementation of merge sort in this example is *recursive*. The *base case* is a vector with one element. A one-element vector is, of course, sorted, so merge sort immediately returns when it's called with a one-element vector. The *recursion step* splits a vector of two or more elements into two equal-sized subvectors, recursively sorts each subvector, then merges them into one larger, sorted vector. [Again, if there is an odd number of elements, one subvector is one element larger than the other.]

Suppose the algorithm has already merged smaller vectors to create sorted vectors A:

```
4    10    34    56    77
```

and B:

```
5    30    51    52    93
```

Merge sort combines these two vectors into one larger, sorted vector. The smallest value in A is 4 (located in the zeroth element of A). The smallest value in B is 5 (located in the zeroth element of B). In order to determine the smallest element in the larger vector, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the value of the first element in the merged vector. The algorithm continues by comparing 10 (the value of the second element in A) to 5 (the value of the first element in B). The value from B is smaller, so 5 becomes the value of the second element in the larger vector. The algorithm continues by comparing 10 to 30, with 10 becoming the value of the third element in the vector, and so on.

Figure 19.5 defines class `MergeSort`, and lines 22–25 of Fig. 19.6 define the `sort` function. Line 24 calls function `sortSubVector` with `0` and `size - 1` as the arguments. These arguments correspond to the beginning and ending indices of the vector to be sorted, causing `sortSubVector` to operate on the entire vector. Function `sortSubVector` is defined in lines 28–52. Line 31 tests the base case. If the size of the vector is `0`, the vector is already sorted, so the function simply returns immediately. If the size of the vector is greater than or equal to `1`, the function splits the vector in two, recursively calls function `sortSubVector` to sort the two subvectors, then merges them. Line 46 recursively calls function `sortSubVector` on the first half of the vector, and line 47 recursively calls function `sortSubVector` on the second half of the vector. When these two function calls return, each half of the vector has been sorted. Line 50 calls function `merge` (lines 55–99) on the two halves of the vector to combine the two sorted vectors into one larger sorted vector.

```
// Fig 19.5: MergeSort.h
// Class that creates a vector filled with random integers.
// Provides a function to sort the vector with merge sort.
#include <vector>
using namespace std;

// MergeSort class definition
class MergeSort
{
public:
   MergeSort( int ); // constructor initializes vector
   void sort(); // sort vector using merge sort
   void displayElements() const; // display vector elements
```

Fig. 19.5 | `MergeSort` class definition. (Part 1 of 2.)

```
private:
   int size; // vector size
   vector< int > data; // vector of ints
   void sortSubVector( int, int ); // sort subvector
   void merge( int, int, int, int ); // merge two sorted vectors
   void displaySubVector( int, int ) const; // display subvector
}; // end class SelectionSort
```

Fig. 19.5 | MergeSort class definition. (Part 2 of 2.)

```
// Fig 19.6: MergeSort.cpp
// Class MergeSort member-function definition.
#include <iostream>
#include <vector>
#include <cstdlib> // prototypes for functions srand and rand
#include <ctime> // prototype for function time
#include "MergeSort.h" // class MergeSort definition
using namespace std;

// constructor fill vector with random integers
MergeSort::MergeSort( int vectorSize )
{
   size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
   srand( time( 0 ) ); // seed random number generator using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; ++i )
      data.push_back( 10 + rand() % 90 );
} // end MergeSort constructor

// split vector, sort subvectors and merge subvectors into sorted vector
void MergeSort::sort()
{
   sortSubVector( 0, size - 1 ); // recursively sort entire vector
} // end function sort

// recursive function to sort subvectors
void MergeSort::sortSubVector( int low, int high )
{
   // test base case; size of vector equals 1
   if ( ( high - low ) >= 1 ) // if not base case
   {
      int middle1 = ( low + high ) / 2; // calculate middle of vector
      int middle2 = middle1 + 1; // calculate next element over

      // output split step
      cout << "split:   ";
      displaySubVector( low, high );
      cout << endl << "          ";
      displaySubVector( low, middle1 );
      cout << endl << "          ";
```

Fig. 19.6 | MergeSort class member-function definition. (Part 1 of 3.)

```
      displaySubVector( middle2, high );
      cout << endl << endl;

      // split vector in half; sort each half (recursive calls)
      sortSubVector( low, middle1 ); // first half of vector
      sortSubVector( middle2, high ); // second half of vector

      // merge two sorted vectors after split calls return
      merge( low, middle1, middle2, high );
   } // end if
} // end function sortSubVector

// merge two sorted subvectors into one sorted subvector
void MergeSort::merge( int left, int middle1, int middle2, int right )
{
   int leftIndex = left; // index into left subvector
   int rightIndex = middle2; // index into right subvector
   int combinedIndex = left; // index into temporary working vector
   vector< int > combined( size ); // working vector

   // output two subvectors before merging
   cout << "merge:   ";
   displaySubVector( left, middle1 );
   cout << endl << "          ";
   displaySubVector( middle2, right );
   cout << endl;

   // merge vectors until reaching end of either
   while ( leftIndex <= middle1 && rightIndex <= right )
   {
      // place smaller of two current elements into result
      // and move to next space in vector
      if ( data[ leftIndex ] <= data[ rightIndex ] )
         combined[ combinedIndex++ ] = data[ leftIndex++ ];
      else
         combined[ combinedIndex++ ] = data[ rightIndex++ ];
   } // end while

   if ( leftIndex == middle2 ) // if at end of left vector
   {
      while ( rightIndex <= right ) // copy in rest of right vector
         combined[ combinedIndex++ ] = data[ rightIndex++ ];
   } // end if
   else // at end of right vector
   {
      while ( leftIndex <= middle1 ) // copy in rest of left vector
         combined[ combinedIndex++ ] = data[ leftIndex++ ];
   } // end else

   // copy values back into original vector
   for ( int i = left; i <= right; ++i )
      data[ i ] = combined[ i ];
```

Fig. 19.6 | MergeSort class member-function definition. (Part 2 of 3.)

```
94
95        // output merged vector
96        cout << "           ";
97        displaySubVector( left, right );
98        cout << endl << endl;
99     } // end function merge
100
101    // display elements in vector
102    void MergeSort::displayElements() const
103    {
104       displaySubVector( 0, size - 1 );
105    } // end function displayElements
106
107    // display certain values in vector
108    void MergeSort::displaySubVector( int low, int high ) const
109    {
110       // output spaces for alignment
111       for ( int i = 0; i < low; ++i )
112          cout << "   ";
113
114       // output elements left in vector
115       for ( int i = low; i <= high; ++i )
116          cout << " " << data[ i ];
117    } // end function displaySubVector
```

Fig. 19.6 | MergeSort class member-function definition. (Part 3 of 3.)

Lines 70–78 in function `merge` loop until the program reaches the end of either subvector. Line 74 tests which element at the beginning of the vectors is smaller. If the element in the left vector is smaller, line 75 places it in position in the combined vector. If the element in the right vector is smaller, line 77 places it in position in the combined vector. When the `while` loop has completed (line 78), one entire subvector is placed in the combined vector, but the other subvector still contains data. Line 80 tests whether the left vector has reached the end. If so, lines 82–83 fill the combined vector with the elements of the right vector. If the left vector has not reached the end, then the right vector must have reached the end, and lines 87–88 fill the combined vector with the elements of the left vector. Finally, lines 92–93 copy the combined vector into the original vector. Figure 19.7 creates and uses a `MergeSort` object. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

```
// Fig 19.7: Fig19_07.cpp
// MergeSort test program.
#include <iostream>
#include "MergeSort.h" // class MergeSort definition
using namespace std;

int main()
{
```

Fig. 19.7 | MergeSort test program. (Part 1 of 3.)

```
   // create object to perform merge sort
   MergeSort sortVector( 10 );

   cout << "Unsorted vector:" << endl;
   sortVector.displayElements(); // print unsorted vector
   cout << endl << endl;

   sortVector.sort(); // sort vector

   cout << "Sorted vector:" << endl;
   sortVector.displayElements(); // print sorted vector
   cout << endl;
} // end main
```

```
Unsorted vector:
 30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                         18 60 78 26 54

split:    30 47 22 67 79
          30 47 22
                   67 79

split:    30 47 22
          30 47
                22

split:    30 47
          30
             47

merge:    30
             47
          30 47

merge:    30 47
                22
          22 30 47

split:             67 79
                   67
                      79

merge:             67
                      79
                   67 79

merge:    22 30 47
                   67 79
          22 30 47 67 79

split:                   18 60 78 26 54
                         18 60 78
                                  26 54
```

Fig. 19.7 | MergeSort test program. (Part 2 of 3.)

```
split:                    18 60 78
                          18 60
                                78

split:                    18 60
                          18
                             60

merge:                    18
                             60
                          18 60

merge:                    18 60
                                78
                          18 60 78

split:                             26 54
                                   26
                                      54

merge:                             26
                                      54
                                   26 54

merge:                    18 60 78
                                   26 54
                          18 26 54 60 78

merge:     22 30 47 67 79
                          18 26 54 60 78
           18 22 26 30 47 54 60 67 78 79

Sorted vector:
 18 22 26 30 47 54 60 67 78 79
```

Fig. 19.7 | `MergeSort` test program. (Part 3 of 3.)

Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort (although that may be difficult to believe when looking at the rather busy output in Fig. 19.7). Consider the first (nonrecursive) call to function `sortSubVector` (line 24). This results in two recursive calls to function `sortSubVector` with subvectors each approximately half the size of the original vector, and a single call to function `merge`. This call to function `merge` requires, at worst, $n - 1$ comparisons to fill the original vector, which is $O(n)$. (Recall that each vector element is chosen by comparing one element from each of the subvectors.) The two calls to function `sortSubVector` result in four more recursive calls to function `sortSubVector`—each with a subvector approximately one-quarter the size of the original vector—and two calls to function `merge`. These two calls to function `merge` each require, at worst, $n/2 - 1$ comparisons, for a total number of comparisons of $O(n)$. This process continues, each call to `sortSubVector` generating two additional calls to `sortSubVector` and a call to `merge`, until the algorithm has split the vector into one-element subvectors. At each level, $O(n)$ comparisons are required to merge the subvectors. Each level splits the size of the vectors in half, so doubling the size of the vector requires one more level. Quadrupling the size of the vector requires two more levels. This pattern is logarithmic and results in $\log_2 n$ levels. This results in a total efficiency of $\boldsymbol{O(n \log n)}$.

Figure 19.8 summarizes the searching and sorting algorithms we cover in this book and lists the Big O for each. Figure 19.9 lists the Big O categories we've covered in this chapter along with a number of values for n to highlight the differences in the growth rates.

Algorithm	Location	Big O
Searching Algorithms		
Linear search	Section 7.7	$O(n)$
Binary search	Section 19.2.2	$O(\log n)$
Recursive linear search	Exercise 19.8	$O(n)$
Recursive binary search	Exercise 19.9	$O(\log n)$
Sorting Algorithms		
Insertion sort	Section 7.8	$O(n^2)$
Selection sort	Section 8.6	$O(n^2)$
Merge sort	Section 19.3.3	$O(n \log n)$
Bubble sort	Exercises 19.5 and 19.6	$O(n^2)$
Quicksort	Exercise 19.10	Worst case: $O(n^2)$ Average case: $O(n \log n)$

Fig. 19.8 | Searching and sorting algorithms with Big O values.

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$2^{10} \cdot 10$	2^{20}
2^{20}	1,000,000	20	2^{20}	$2^{20} \cdot 20$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$2^{30} \cdot 30$	2^{60}

Fig. 19.9 | Approximate number of comparisons for common Big O notations.

19.4 Wrap-Up

This chapter discussed searching and sorting data. We discussed the binary search algorithm, which is faster but more complex than linear search (Section 7.7). The binary search algorithm will work only on a sorted array; each iteration of binary search eliminates from consideration half of the elements in the array. You learned the merge sort algorithm, which is more efficient than either the insertion sort (Section 7.8) or the selection sort (Section 8.6). We also introduced Big O notation, which helps you express the efficiency of an algorithm. Big O notation measures the worst-case runtime for an algorithm. The Big O value is useful for comparing algorithms in order to choose the most efficient one. In the next chapter, you'll learn about dynamic data structures that can grow or shrink at execution time.

Summary

Section 19.1 Introduction

- Searching data involves determining whether a search key (p. 725) is present in the data and, if so, finding its location.
- Sorting (p. 725) involves arranging data into order.
- One way to describe the efficiency of an algorithm is with Big O notation (p. 725), which indicates how hard an algorithm may have to work to solve a problem.

Section 19.2 Searching Algorithms

- A key difference among searching algorithms is the amount of effort they require to return a result.

Section 19.2.1 Efficiency of Linear Search

- For searching and sorting algorithms, Big O describes how the amount of effort of a particular algorithm varies depending on how many elements are in the data.
- An algorithm that's $O(1)$ has a constant runtime (p. 726)—the number of comparisons does not grow as the size of the vector increases.
- An $O(n)$ algorithm is referred to as having a linear runtime (p. 726).
- Big O highlights dominant factors and ignores terms that are unimportant with high values of n.
- Big O notation represents the growth rate of algorithm runtimes, so constants are ignored.
- The linear search algorithm runs in $O(n)$ time.
- In the worst case for linear search every element must be checked to determine whether the search element exists. This occurs if the search key is the last element in the vector or is not present.

Section 19.2.2 Binary Search

- Binary search (p. 727) is more efficient than linear search, but it requires that the vector first be sorted. This is worthwhile only when the vector, once sorted, will be searched many times.
- The first iteration of binary search tests the middle element. If this is the search key, the algorithm returns its location. If the search key is less than the middle element, binary search continues with the first half of the vector. If the search key is greater than the middle element, binary search continues with the second half. Each iteration tests the middle value of the remaining vector and, if the element is not found, eliminates from consideration half of the remaining elements.
- Binary search is more efficient than linear search, because with each comparison it eliminates from consideration half of the elements in the vector.
- Binary search runs in $O(\log n)$ (p. 732) time.
- If the size of the vector is doubled, binary search requires only one extra comparison to complete.

Section 19.3.1 Efficiency of Selection Sort

- Selection sort is a simple, but inefficient, sorting algorithm.
- The first iteration of selection sort selects the smallest element and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest remaining element) and swaps it with the second element. This continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. At the i^{th} iteration, the smallest i elements are sorted into the first i elements.

Section 19.3.2 Efficiency of Insertion Sort

- The selection sort algorithm runs in $O(n^2)$ time (p. 726).

- The first iteration of insertion sort takes the second element value and, if it's less than the first, swaps it with the first. The second iteration looks at the third element value and inserts it in the correct position with respect to the first two element values. After the i^{th} iteration of insertion sort, the first i element values in the original vector are sorted. Only $n - 1$ iterations are required.
- The insertion sort algorithm runs in $O(n^2)$ time.

Section 19.3.3 Merge Sort (A Recursive Implementation)

- Merge sort (p. 733) is faster, but more complex to implement, than selection sort and insertion sort.
- The merge sort algorithm sorts a vector by splitting the vector into two equal-sized subvectors, sorting each subvector and merging the subvectors into one larger vector.
- Merge sort's base case is a vector with one element. A one-element vector is already sorted, so merge sort immediately returns when it's called with a one-element vector. The merge part of merge sort takes two sorted vectors (these could be one-element vectors) and combines them into one larger sorted vector.
- Merge sort performs the merge by looking at the first element in each vector, which is also the smallest element in the vector. Merge sort takes the smallest of these and places it in the first element of the larger, sorted vector. If there are still elements in the subvector, merge sort looks at the second element in that subvector (which is now the smallest element remaining) and compares it to the first element in the other subvector. Merge sort continues this process until the larger vector is filled.
- In the worst case, the first call to merge sort has to make $O(n)$ comparisons to fill the n slots in the final vector.
- The merging portion of the merge sort algorithm is performed on two subvectors, each of approximately size $n/2$. Creating each of these subvectors requires $n/2 - 1$ comparisons for each subvector, or $O(n)$ comparisons total. This pattern continues, as each level works on twice as many vectors, but each is half the size of the previous vector.
- Similar to binary search, this halving results in $\log n$ levels, each level requiring $O(n)$ comparisons, for a total efficiency of $O(n \log n)$ (p. 740).

Self-Review Exercises

19.1 Fill in the blanks in each of the following statements:

a) A selection sort application would take approximately ________ times as long to run on a 128-element vector as on a 32-element vector.

b) The efficiency of merge sort is ________.

19.2 What key aspect of both the binary search and the merge sort accounts for the logarithmic portion of their respective Big Os?

19.3 In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

19.4 In the text, we say that after the merge sort splits the vector into two subvectors, it then sorts these two subvectors and merges them. Why might someone be puzzled by our statement that "it then sorts these two subvectors"?

Answers to Self-Review Exercises

19.1 a) 16, because an $O(n^2)$ algorithm takes 16 times as long to sort four times as much information. b) $O(n \log n)$.

19.2 Both of these algorithms incorporate "halving"—somehow reducing something by half. The binary search eliminates from consideration half of the vector after each comparison. The merge sort splits the vector in half each time it's called.

19.3 The insertion sort is easier to understand and to implement than the merge sort. The merge sort is far more efficient ($O(n \log n)$) than the insertion sort ($O(n^2)$).

19.4 In a sense, it does not really sort these two subvectors. It simply keeps splitting the original vector in half until it provides a one-element subvector, which is, of course, sorted. It then builds up the original two subvectors by merging these one-element vectors to form larger subvectors, which are then merged, and so on.

Exercises

[*Note:* Most of the exercises shown here are duplicates of exercises from Chapters 7–8. We include the exercises again here as a convenience for readers studying searching and sorting in this chapter.]

19.5 ***(Bubble Sort)*** Implement bubble sort—another simple yet inefficient sorting technique. It's called bubble sort or sinking sort because smaller values gradually "bubble" their way to the top of the vector (i.e., toward the first element) like air bubbles rising in water, while the larger values sink to the bottom (end) of the vector. The technique uses nested loops to make several passes through the vector. Each pass compares successive pairs of elements. If a pair is in increasing order (or the values are equal), the bubble sort leaves the values as they are. If a pair is in decreasing order, the bubble sort swaps their values in the vector.

The first pass compares the first two element values of the vector and swaps them if necessary. It then compares the second and third element values in the vector. The end of this pass compares the last two element values in the vector and swaps them if necessary. After one pass, the largest value will be in the last element. After two passes, the largest two values will be in the last two elements. Explain why bubble sort is an $O(n^2)$ algorithm.

19.6 ***(Enhanced Bubble Sort)*** Make the following simple modifications to improve the performance of the bubble sort you developed in Exercise 19.5:

a) After the first pass, the largest value is guaranteed to be in the highest-numbered element of the vector; after the second pass, the two highest values are "in place"; and so on. Instead of making nine comparisons (for a 10-element vector) on every pass, modify the bubble sort to make only the eight necessary comparisons on the second pass, seven on the third pass, and so on.
b) The data in the vector may already be in the proper order or near-proper order, so why make nine passes (of a 10-element vector) if fewer will suffice? Modify the sort to check at the end of each pass whether any swaps have been made. If none have been made, the data must already be in the proper order, so the program should terminate. If swaps have been made, at least one more pass is needed.

19.7 (***Bucket Sort***) A bucket sort begins with a one-dimensional vector of positive integers to be sorted and a two-dimensional vector of integers with rows indexed from 0 to 9 and columns indexed from 0 to $n - 1$, where n is the number of values to be sorted. Each row of the two-dimensional vector is referred to as a *bucket*. Write a class named `BucketSort` containing a function called `sort` that operates as follows:

a) Place each value of the one-dimensional vector into a row of the bucket vector, based on the value's "ones" (rightmost) digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a *distribution pass*.
b) Loop through the bucket vector row by row, and copy the values back to the original vector. This procedure is called a *gathering pass*. The new order of the preceding values in the one-dimensional vector is 100, 3 and 97.

c) Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second (tens digit) pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional vector is 100, 3 and 97. On the third (hundreds digit) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original vector is in sorted order.

Note that the two-dimensional vector of buckets is 10 times the length of the integer vector being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory—the bubble sort requires space for only one additional element of data. This comparison is an example of the space–time trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort requires copying all the data back to the original vector on each pass. Another possibility is to create a second two-dimensional bucket vector and repeatedly swap the data between the two bucket vectors.

19.8 (***Recursive Linear Search)*** Modify Exercise 7.33 to use recursive function `recursiveLinearSearch` to perform a linear search of the vector. The function should receive the search key and starting index as arguments. If the search key is found, return its index in the vector; otherwise, return `-1`. Each call to the recursive function should check one element value in the vector.

19.9 (***Recursive Binary Search***) Modify Fig. 19.3 to use recursive function `recursiveBinarySearch` to perform a binary search of the vector. The function should receive the search key, starting index and ending index as arguments. If the search key is found, return its index in the vector. If the search key is not found, return `-1`.

19.10 ***(Quicksort***) The recursive sorting technique called quicksort uses the following basic algorithm for a one-dimensional vector of values:

a) *Partitioning Step*: Take the first element of the unsorted vector and determine its final location in the sorted vector (i.e., all values to the left of the element in the vector are less than the element's value, and all values to the right of the element in the vector are greater than the element's value—we show how to do this below). We now have one value in its proper location and two unsorted subvectors.

b) *Recursion Step*: Perform the *Partitioning Step* on each unsorted subvector. Each time the *Partitioning Step* is performed on a subvector, another value is placed in its final location of the sorted vector, and two unsorted subvectors are created. When a subvector consists of one element, that element's value is in its final location (because a one-element vector is already sorted).

The basic algorithm seems simple enough, but how do we determine the final position of the first element value of each subvector? As an example, consider the following set of values (the value in bold is for the partitioning element—it will be placed in its final location in the sorted vector):

37 2 6 4 89 8 10 12 68 45

Starting from the rightmost element of the vector, compare each element value with **37** until an element value less than **37** is found; then swap **37** and that element's value. The first element value less than **37** is 12, so **37** and 12 are swapped. The new vector is

12 2 6 4 89 8 10 **37** 68 45

Element value 12 is in italics to indicate that it was just swapped with **37**.

Starting from the left of the vector, but beginning with the element value after 12, compare each element value with **37** until an element value greater than **37** is found—then swap **37** and that element value. The first element value greater than **37** is 89, so **37** and 89 are swapped. The new vector is

12 2 6 4 **37** 8 10 *89* 68 45

Starting from the right, but beginning with the element value before 89, compare each element value with **37** until an element value less than **37** is found—then swap **37** and that element value. The first element value less than **37** is 10, so **37** and 10 are swapped. The new vector is

12 2 6 4 *10* 8 **37** 89 68 45

Starting from the left, but beginning with the element value after 10, compare each element value with **37** until an element value greater than **37** is found—then swap **37** and that element value. There are no more element values greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted vector. Every value to the left of **37** is smaller than it, and every value to the right of **37** is larger than it.

Once the partition has been applied on the previous vector, there are two unsorted subvectors. The subvector with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subvector with values greater than 37 contains 89, 68 and 45. The sort continues recursively, with both subvectors being partitioned in the same manner as the original vector.

Based on the preceding discussion, write recursive function `quickSortHelper` to sort a one-dimensional integer vector. The function should receive as arguments a starting index and an ending index on the original vector being sorted.

20 Custom Templatized Data Structures

Much that I bound,
I could not free;
Much that I freed
returned to me.
—Lee Wilson Dodd

'Will you walk a little faster?'
said a whiting to a snail,
'There's a porpoise close behind
us, and he's treading on my tail.'
—Lewis Carroll

There is always room at the top.
—Daniel Webster

Push on—keep moving.
—Thomas Morton

I'll turn over a new leaf.
—Miguel de Cervantes

Objectives

In this chapter you'll learn:

- To form linked data structures using pointers, self-referential classes and recursion.
- To create and manipulate dynamic data structures such as linked lists, queues, stacks and binary trees.
- To use binary search trees for high-speed searching and sorting.
- To understand important applications of linked data structures.
- To understand how to create reusable data structures with class templates, inheritance and composition.

Outline

20.1 Introduction

We've studied fixed-size **data structures** such as one-dimensional arrays and two-dimensional arrays. This chapter introduces **dynamic data structures** that grow and shrink during execution. **Linked lists** are collections of data items logically "lined up in a row"—insertions and removals are made *anywhere* in a linked list. **Stacks** are important in compilers and operating systems: Insertions and removals are made *only at one end* of a stack—its **top**. **Queues** represent *waiting lines*; insertions are made at the *back* (also referred to as the **tail**) of a queue and removals are made from the *front* (also referred to as the **head**) of a queue. **Binary trees** facilitate high-speed searching and sorting of data, efficient *elimination of duplicate data items*, representation of *file-system directories* and *compilation* of expressions into machine language. These data structures have many other interesting applications.

We discuss several popular and important data structures and implement programs that create and manipulate them. We use classes, class templates, inheritance and composition to create and package these data structures for reusability and maintainability.

This chapter is solid preparation for Chapter 22, Standard Template Library (STL). The STL is a major portion of the C++ Standard Library. The STL provides *containers*, *iterators* for traversing those containers and *algorithms* for processing the containers' elements. You'll see that the STL has taken each of the data structures we discuss in this chapter and packaged them into templatized classes. The STL code is carefully written to be portable, efficient and extensible. Once you understand the principles and construction of data structures, you'll be able to make the best use of the prepackaged data structures, iterators and algorithms in the STL, a world-class set of reusable components.

The chapter examples are practical programs that you'll be able to use in more advanced courses and in industry applications. The programs employ extensive pointer manipulation. The exercises include a rich collection of useful applications.

We encourage you to attempt the optional major project described in the Special Section: Building Your Own Compiler. You've been using a C++ compiler to translate your programs to machine language so that you could execute these programs on your computer. In this project, you'll actually build your own compiler. It will read a file of statements written in a simple, yet powerful, high-level language similar to early versions of the popular language BASIC. Your compiler will translate these statements into a file of Simpletron Machine Language (SML) instructions—SML is the language you learned in the Chapter 8 Special Section, Building: Your Own Computer. Your Simpletron Simulator program will then execute the SML program produced by your compiler! The special section carefully walks you through the specifications of the high-level language and describes the algorithms you'll need to convert each type of high-level language statement into machine-language instructions. This chapter's exercises suggest many enhancements to both the compiler and the Simpletron Simulator.

20.2 Self-Referential Classes

A **self-referential class** contains a member that points to a class object of the same class type. For example, the definition

```
class Node
{
public:
   Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
   void setNextPtr( Node * ); // set pointer to next Node
   Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data stored in this Node
   Node *nextPtr; // pointer to another object of same type
}; // end class Node
```

defines a type, `Node`. Type `Node` has two `private` data members—integer member `data` and pointer member `nextPtr`. Member `nextPtr` points to an object of type `Node`—another object of the same type as the one being declared here, hence the term "self-referential class." Member `nextPtr` is referred to as a **link**—i.e., `nextPtr` can "tie" an object of type `Node` to another object of the *same* type. Type `Node` also has five member functions—a constructor that receives an integer to initialize member `data`, a `setData` function to set the value of member `data`, a `getData` function to return the value of member `data`, a `setNextPtr` function to set the value of member `nextPtr` and a `getNextPtr` function to return the value of member `nextPtr`.

Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees. Figure 20.1 illustrates two self-referential class objects linked together to form a list. Note that a slash—representing a null (`0`) pointer—is placed in the link member of the second self-referential class object to indicate that the link does not point to another object. The slash is only for illustration purposes; it does *not* correspond to the backslash character in C++. A null pointer normally indicates the *end of a data structure* just as the null character (`'\0'`) indicates the end of a string.

Common Programming Error 20.1

Not setting the link in the last node of a linked data structure to null (`0`) is a (possibly fatal) logic error.

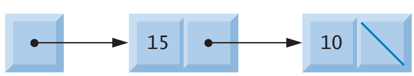


Fig. 20.1 | Two self-referential class objects linked together.

20.3 Dynamic Memory Allocation and Data Structures

Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes. When that memory is no longer needed by the program, the memory can be *released* so that it can be reused to allocate other objects in the future. The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system. Often, the limits are much smaller, because available memory must be shared among many programs.

The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type. For example, the statement

```
Node *newPtr = new Node( 10 ); // create Node with data 10
```

allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`. If no memory is available, `new` throws a `bad_alloc` exception. The value `10` is passed to the `Node` constructor which initializes the `Node`'s `data` member to 10.

The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`—the memory is returned to the system so that the memory can be reallocated in the future. To free memory dynamically allocated by the preceding `new`, use the statement

```
delete newPtr;
```

Note that `newPtr` itself is *not* deleted; rather the destructor of the `Node` object that `newPtr` points to is called and the object's memory is freed. If pointer `newPtr` has the null pointer value `0`, the preceding statement has no effect. It is *not* an error to `delete` a null pointer.

The following sections discuss lists, stacks, queues and trees. The data structures presented in this chapter are created and maintained with dynamic memory allocation, self-referential classes, class templates and function templates.

20.4 Linked Lists

A linked list is a linear collection of self-referential class objects, called **nodes**, connected by **pointer links**—hence, the term "linked" list. A linked list is accessed via a pointer to the list's first node. Each subsequent node is accessed via the link-pointer member stored in the previous node. By *convention*, the link pointer in the last node of a list is set to null (`0`) to mark the end of the list. Data is stored in a linked list dynamically—each node is created as necessary. A node can contain data of any type, including objects of other classes. If nodes contain base-class pointers to base-class and derived-class objects related by inheritance, we can have a linked list of such nodes and process them *polymorphically* using `virtual` function calls. Stacks and queues are also **linear data structures** and, as we'll see, can be viewed as constrained versions of linked lists. Trees are **nonlinear data structures**.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented at one time is *unpredictable*. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of a "conventional" C++ array, however, cannot be altered, because the array size is fixed at compile time. "Conventional" arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy additional dynamic storage allocation requests.

Performance Tip 20.1

An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Class template `vector` *(Section 7.11) implements a dynamically resizable array-based data structure.*

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list. Existing list elements do *not* need to be moved. Pointers merely need to be updated to point to the correct node.

Performance Tip 20.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

Performance Tip 20.3

The elements of an array are stored contiguously in memory. This allows immediate access *to any element, because an element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate* direct access *to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it's typically more efficient to insert an item in a sorted linked list than a sorted array.*

Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous. Figure 20.2 illustrates a linked list with several nodes.

Performance Tip 20.4

Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

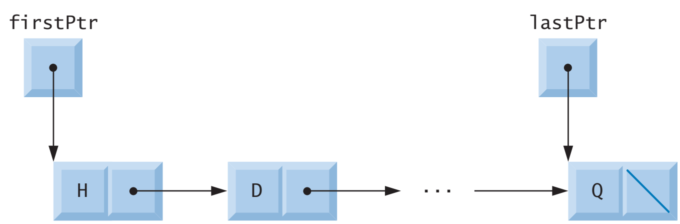


Fig. 20.2 | A graphical representation of a list.

Linked List Implementation

The program of Figs. 20.3–20.5 uses a `List` class template (see Chapter 14 for information on class templates) to manipulate a list of integer values and a list of floating-point values. The driver program (Fig. 20.5) provides five options: 1) Insert a value at the beginning of the list, 2) insert a value at the end of the list, 3) delete a value from the beginning of the list, 4) delete a value from the end of the list and 5) end the list processing. A detailed discussion of the program follows. Exercise 20.20 asks you to implement a recursive function that prints a linked list backward, and Exercise 20.21 asks you to implement a recursive function that searches a linked list for a particular data item.

The program uses class templates `ListNode` (Fig. 20.3) and `List` (Fig. 20.4). Encapsulated in each `List` object is a linked list of `ListNode` objects. Class template `ListNode` (Fig. 20.3) contains `private` members `data` and `nextPtr` (lines 19–20), a constructor to initialize these members and function `getData` to return the data in a node. Member `data` stores a value of type `NODETYPE`, the type parameter passed to the class template. Member `nextPtr` stores a pointer to the next `ListNode` object in the linked list. Line 13 of the `ListNode` class template definition declares class `List< NODETYPE >` as a `friend`. This makes all member functions of a given specialization of class template `List` friends of the corresponding specialization of class template `ListNode`, so they can access the `private` members of `ListNode` objects of that type. Because the `ListNode` template parameter `NODETYPE` is used as the template argument for `List` in the `friend` declaration, `ListNode`s specialized with a particular type can be processed only by a `List` specialized with the same type (e.g., a `List` of `int` values manages `ListNode` objects that store `int` values).

```
1  // Fig. 20.3: ListNode.h
2  // Template ListNode class definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List required to announce that class
7  // List exists so it can be used in the friend declaration at line 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    ListNode( const NODETYPE & ); // constructor
17    NODETYPE getData() const; // return data in node
18 private:
19    NODETYPE data; // data
20    ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
23 // constructor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // empty body
29 } // end ListNode constructor
30
```

Fig. 20.3 | `ListNode` class-template definition. (Part 1 of 2.)

```
// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData

#endif
```

Fig. 20.3 | `ListNode` class-template definition. (Part 2 of 2.)

Lines 23–24 of the `List` class template (Fig. 20.4) declare `private` data members `firstPtr` (a pointer to the first `ListNode` in a `List`) and `lastPtr` (a pointer to the last `ListNode` in a `List`). The default constructor (lines 31–36) initializes both pointers to `0` (null). The destructor (lines 39–59) ensures that all `ListNode` objects in a `List` object are destroyed when that `List` object is destroyed. The primary `List` functions are `insertAtFront` (lines 62–74), `insertAtBack` (lines 77–89), `removeFromFront` (lines 92–110) and `removeFromBack` (lines 113–140).

Function `isEmpty` (lines 143–147) is called a *predicate function*—it does not alter the `List`; rather, it determines whether the `List` is empty (i.e., the pointer to the first node of the `List` is null). If the `List` is empty, `true` is returned; otherwise, `false` is returned. Function `print` (lines 158–178) displays the `List`'s contents. Utility function `getNewNode` (lines 150–155) returns a dynamically allocated `ListNode` object. This function is called from functions `insertAtFront` and `insertAtBack`.

Error-Prevention Tip 20.1

Assign null (0) to the link member of a new node. Pointers must be initialized before they're used.

```
1  // Fig. 20.4: List.h
2  // Template List class definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  #include "ListNode.h" // ListNode class definition
8  using namespace std;
9
10 template< typename NODETYPE >
11 class List
12 {
13 public:
14    List(); // constructor
15    ~List(); // destructor
16    void insertAtFront( const NODETYPE & );
17    void insertAtBack( const NODETYPE & );
18    bool removeFromFront( NODETYPE & );
19    bool removeFromBack( NODETYPE & );
```

Fig. 20.4 | `List` class-template definition. (Part 1 of 5.)

```
      bool isEmpty() const;
      void print() const;
   private:
      ListNode< NODETYPE > *firstPtr; // pointer to first node
      ListNode< NODETYPE > *lastPtr; // pointer to last node

      // utility function to allocate new node
      ListNode< NODETYPE > *getNewNode( const NODETYPE & );
   }; // end class List

   // default constructor
   template< typename NODETYPE >
   List< NODETYPE >::List()
      : firstPtr( 0 ), lastPtr( 0 )
   {
      // empty body
   } // end List constructor

   // destructor
   template< typename NODETYPE >
   List< NODETYPE >::~List()
   {
      if ( !isEmpty() ) // List is not empty
      {
         cout << "Destroying nodes ...\n";

         ListNode< NODETYPE > *currentPtr = firstPtr;
         ListNode< NODETYPE > *tempPtr;

         while ( currentPtr != 0 ) // delete remaining nodes
         {
            tempPtr = currentPtr;
            cout << tempPtr->data << '\n';
            currentPtr = currentPtr->nextPtr;
            delete tempPtr;
         } // end while
      } // end if

      cout << "All nodes destroyed\n\n";
   } // end List destructor

   // insert node at front of list
   template< typename NODETYPE >
   void List< NODETYPE >::insertAtFront( const NODETYPE &value )
   {
      ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

      if ( isEmpty() ) // List is empty
         firstPtr = lastPtr = newPtr; // new list has only one node
      else // List is not empty
      {
         newPtr->nextPtr = firstPtr; // point new node to previous 1st node
```

Fig. 20.4 | List class-template definition. (Part 2 of 5.)

```
         firstPtr = newPtr; // aim firstPtr at new node
      } // end else
   } // end function insertAtFront

   // insert node at back of list
   template< typename NODETYPE >
   void List< NODETYPE >::insertAtBack( const NODETYPE &value )
   {
      ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

      if ( isEmpty() ) // List is empty
         firstPtr = lastPtr = newPtr; // new list has only one node
      else // List is not empty
      {
         lastPtr->nextPtr = newPtr; // update previous last node
         lastPtr = newPtr; // new last node
      } // end else
   } // end function insertAtBack

   // delete node from front of list
   template< typename NODETYPE >
   bool List< NODETYPE >::removeFromFront( NODETYPE &value )
   {
      if ( isEmpty() ) // List is empty
         return false; // delete unsuccessful
      else
      {
         ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete

         if ( firstPtr == lastPtr )
            firstPtr = lastPtr = 0; // no nodes remain after removal
         else
            firstPtr = firstPtr->nextPtr; // point to previous 2nd node

         value = tempPtr->data; // return data being removed
         delete tempPtr; // reclaim previous front node
         return true; // delete successful
      } // end else
   } // end function removeFromFront

   // delete node from back of list
   template< typename NODETYPE >
   bool List< NODETYPE >::removeFromBack( NODETYPE &value )
   {
      if ( isEmpty() ) // List is empty
         return false; // delete unsuccessful
      else
      {
         ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete

         if ( firstPtr == lastPtr ) // List has one element
            firstPtr = lastPtr = 0; // no nodes remain after removal
```

Fig. 20.4 | List class-template definition. (Part 3 of 5.)

```
      else
      {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node

         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = 0; // this is now the last node
      } // end else

      value = tempPtr->data; // return value from old last node
      delete tempPtr; // reclaim former last node
      return true; // delete successful
   } // end else
} // end function removeFromBack

// is List empty?
template< typename NODETYPE >
bool List< NODETYPE >::isEmpty() const
{
   return firstPtr == 0;
} // end function isEmpty

// return pointer to newly allocated node
template< typename NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
   const NODETYPE &value )
{
   return new ListNode< NODETYPE >( value );
} // end function getNewNode

// display contents of List
template< typename NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) // List is empty
   {
      cout << "The list is empty\n\n";
      return;
   } // end if

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) // get element data
   {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   } // end while

```

Fig. 20.4 | List class-template definition. (Part 4 of 5.)

```
   cout << "\n\n";
} // end function print

#endif
```

Fig. 20.4 | List class-template definition. (Part 5 of 5.)

In Fig. 20.5, Lines 69 and 73 create `List` objects for types `int` and `double`, respectively. Lines 70 and 74 invoke the `testList` function template to manipulate objects.

```
1  // Fig. 20.5: Fig20_05.cpp
2  // List class test program.
3  #include <iostream>
4  #include <string>
5  #include "List.h" // List class definition
6  using namespace std;
7
8  // display program instructions to user
9  void instructions()
10 {
11    cout << "Enter one of the following:\n"
12       << "  1 to insert at beginning of list\n"
13       << "  2 to insert at end of list\n"
14       << "  3 to delete from beginning of list\n"
15       << "  4 to delete from end of list\n"
16       << "  5 to end list processing\n";
17 } // end function instructions
18
19 // function to test a List
20 template< typename T >
21 void testList( List< T > &listObject, const string &typeName )
22 {
23    cout << "Testing a List of " << typeName << " values\n";
24    instructions(); // display instructions
25
26    int choice; // store user choice
27    T value; // store input value
28
29    do // perform user-selected actions
30    {
31       cout << "? ";
32       cin >> choice;
33
34       switch ( choice )
35       {
36          case 1: // insert at beginning
37             cout << "Enter " << typeName << ": ";
38             cin >> value;
39             listObject.insertAtFront( value );
40             listObject.print();
41             break;
```

Fig. 20.5 | Manipulating a linked list. (Part 1 of 4.)

```
            case 2: // insert at end
               cout << "Enter " << typeName << ": ";
               cin >> value;
               listObject.insertAtBack( value );
               listObject.print();
               break;
            case 3: // remove from beginning
               if ( listObject.removeFromFront( value ) )
                  cout << value << " removed from list\n";

               listObject.print();
               break;
            case 4: // remove from end
               if ( listObject.removeFromBack( value ) )
                  cout << value << " removed from list\n";

               listObject.print();
               break;
         } // end switch
      } while ( choice < 5 ); // end do...while

      cout << "End list test\n\n";
   } // end function testList

   int main()
   {
      // test List of int values
      List< int > integerList;
      testList( integerList, "integer" );

      // test List of double values
      List< double > doubleList;
      testList( doubleList, "double" );
   } // end main
```

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3
```

Fig. 20.5 | Manipulating a linked list. (Part 2 of 4.)

```
? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3
```

Fig. 20.5 | Manipulating a linked list. (Part 3 of 4.)

```
? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Fig. 20.5 | Manipulating a linked list. (Part 4 of 4.)

Member Function `insertAtFront`

Over the next several pages, we discuss each of the member functions of class `List` in detail. Function `insertAtFront` (Fig. 20.4, lines 62–74) places a new node at the front of the list. The function consists of several steps:

1. Call function `getNewNode` (line 65), passing it `value`, which is a constant reference to the node value to be inserted.
2. Function `getNewNode` (lines 150–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtFront` (line 65).
3. If the list is empty (line 67), `firstPtr` and `lastPtr` are set to `newPtr` (line 68)—i.e., the first and last node are the same node.
4. If the list is not empty (line 69), then the node pointed to by `newPtr` is threaded into the list by copying `firstPtr` to `newPtr->nextPtr` (line 71), so that the new node points to what used to be the first node of the list, and copying `newPtr` to `firstPtr` (line 72), so that `firstPtr` now points to the new first node of the list.

Figure 20.6 illustrates function `insertAtFront`. Part (a) shows the list and the new node before calling `insertAtFront`. The dashed arrows in part (b) illustrate *Step 4* of the `insertAtFront` operation that enables the node containing `12` to become the new list front.

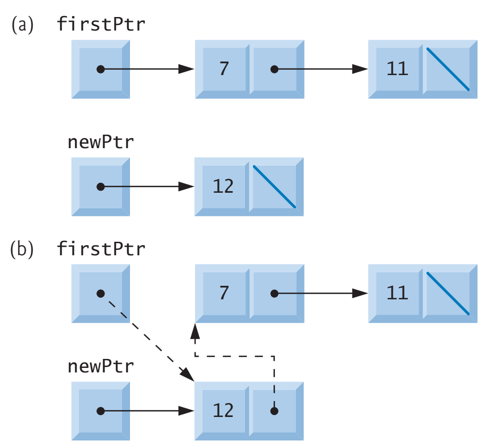


Fig. 20.6 | Operation `insertAtFront` represented graphically.

Member Function `insertAtBack`

Function `insertAtBack` (Fig. 20.4, lines 77–89) places a new node at the back of the list. The function consists of several steps:

1. Call function `getNewNode` (line 80), passing it `value`, which is a constant reference to the node value to be inserted.
2. Function `getNewNode` (lines 150–155) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtBack` (line 80).
3. If the list is empty (line 82), then both `firstPtr` and `lastPtr` are set to `newPtr` (line 83).
4. If the list is not empty (line 84), then the node pointed to by `newPtr` is threaded into the list by copying `newPtr` into `lastPtr->nextPtr` (line 86), so that the new node is pointed to by what used to be the last node of the list, and copying `newPtr` to `lastPtr` (line 87), so that `lastPtr` now points to the new last node of the list.

Figure 20.7 illustrates an `insertAtBack` operation. Part (a) of the figure shows the list and the new node before the operation. The dashed arrows in part (b) illustrate *Step 4* of function `insertAtBack` that enables a new node to be added to the end of a list that's not empty.

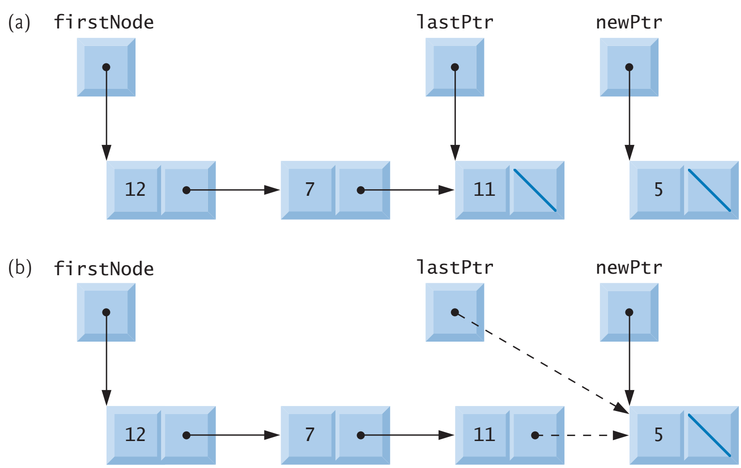


Fig. 20.7 | Operation `insertAtBack` represented graphically.

Member Function `removeFromFront`

Function `removeFromFront` (Fig. 20.4, lines 92–110) removes the front node of the list and copies the node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 95–96) and returns `true` if the removal is successful. The function consists of several steps:

1. Assign `tempPtr` the address to which `firstPtr` points (line 99). Eventually, `tempPtr` will be used to delete the node being removed.

2. If `firstPtr` is equal to `lastPtr` (line 101), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 102) to dethread that node from the list (leaving the list empty).
3. If the list has more than one node prior to removal, then leave `lastPtr` as is and set `firstPtr` to `firstPtr->nextPtr` (line 104); i.e., modify `firstPtr` to point to what was the second node prior to removal (and is now the new first node).
4. After all these pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 106).
5. Now `delete` the node pointed to by `tempPtr` (line 107).
6. Return `true`, indicating successful removal (line 108).

Figure 20.8 illustrates function `removeFromFront`. Part (a) illustrates the list before the removal operation. Part (b) shows the actual pointer manipulations for removing the front node from a nonempty list.

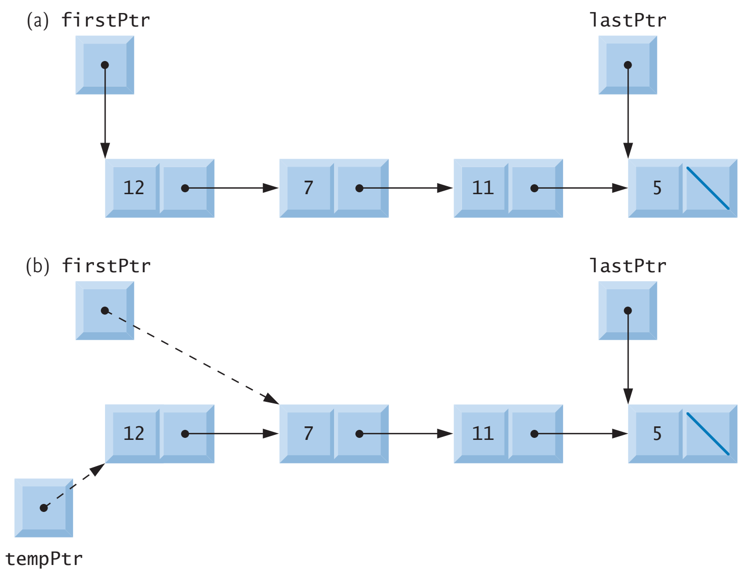


Fig. 20.8 | Operation `removeFromFront` represented graphically.

Member Function `removeFromBack`

Function `removeFromBack` (Fig. 20.4, lines 113–140) removes the back node of the list and copies the node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 116–117) and returns `true` if the removal is successful. The function consists of several steps:

1. Assign to `tempPtr` the address to which `lastPtr` points (line 120). Eventually, `tempPtr` will be used to delete the node being removed.
2. If `firstPtr` is equal to `lastPtr` (line 122), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 123) to dethread that node from the list (leaving the list empty).

3. If the list has more than one node prior to removal, then assign `currentPtr` the address to which `firstPtr` points (line 126) to prepare to "walk the list."
4. Now "walk the list" with `currentPtr` until it points to the node before the last node. This node will become the last node after the remove operation completes. This is done with a `while` loop (lines 129–130) that keeps replacing `currentPtr` by `currentPtr->nextPtr`, while `currentPtr->nextPtr` is not `lastPtr`.
5. Assign `lastPtr` to the address to which `currentPtr` points (line 132) to dethread the back node from the list.
6. Set `currentPtr->nextPtr` to zero (line 133) in the new last node of the list.
7. After all the pointer manipulations are complete, copy to reference parameter `value` the `data` member of the node being removed (line 136).
8. Now `delete` the node pointed to by `tempPtr` (line 137).
9. Return `true` (line 138), indicating successful removal.

Figure 20.9 illustrates `removeFromBack`. Part (a) of the figure illustrates the list before the removal operation. Part (b) of the figure shows the actual pointer manipulations.

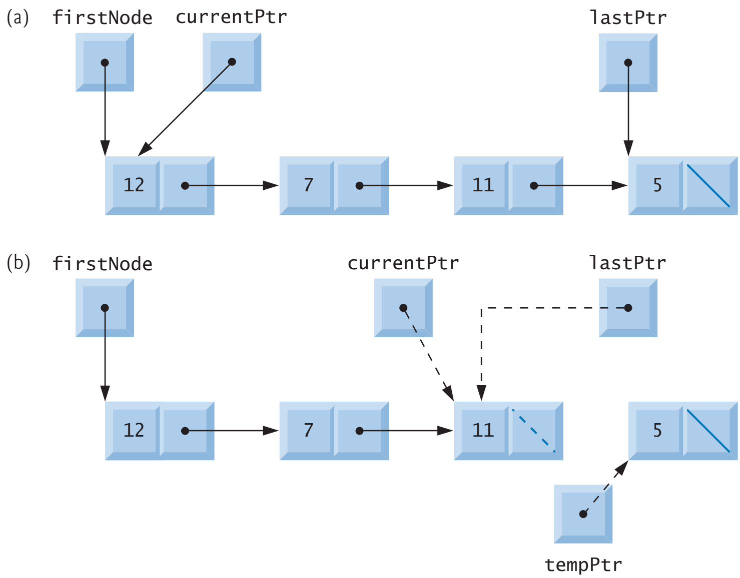


Fig. 20.9 | Operation `removeFromBack` represented graphically.

Member Function `print`

Function `print` (lines 158–178) first determines whether the list is empty (line 161). If so, it prints `"The list is empty"` and returns (lines 163–164). Otherwise, it iterates through the list and outputs the value in each node. The function initializes `currentPtr` as a copy of `firstPtr` (line 167), then prints the string `"The list is: "` (line 169). While `currentPtr` is not null (line 171), `currentPtr->data` is printed (line 173) and `currentPtr` is assigned the value of `currentPtr->nextPtr` (line 174). Note that if the link in the last node of the list is not null, the printing algorithm will erroneously attempt to print past the end of the list. The printing algorithm is identical for linked lists, stacks and queues (because we base each of these data structures on the same linked list infrastructure).

Circular Linked Lists and Double Linked Lists

The kind of linked list we've been discussing is a **singly linked list**—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence." This list terminates with a node whose pointer member has the value 0. A singly linked list may be traversed in only *one* direction.

A **circular, singly linked list** (Fig. 20.10) begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

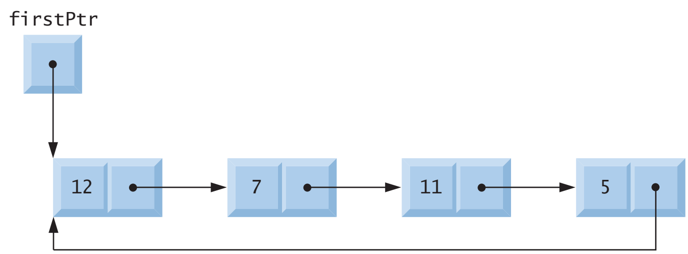


Fig. 20.10 | Circular, singly linked list.

A **doubly linked list** (Fig. 20.11) allows traversals *both forward and backward*. Such a list is often implemented with two "start pointers"—one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal. Each node has *both* a forward pointer to the next node in the list in the forward direction *and* a backward pointer to the next node in the list in the backward direction. If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list. Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

In a **circular, doubly linked list** (Fig. 20.12), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

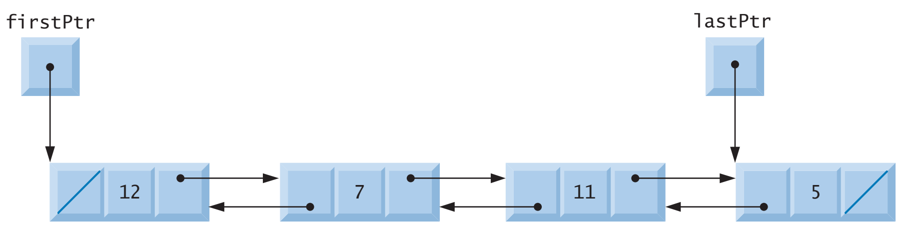


Fig. 20.11 | Doubly linked list.

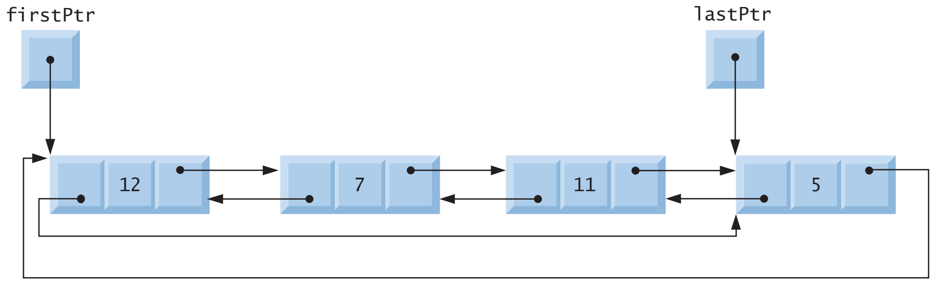


Fig. 20.12 | Circular, doubly linked list.

20.5 Stacks

Chapter 14, Templates, explained the notion of a stack class template with an underlying array implementation. In this section, we use an underlying pointer-based linked-list implementation. We also discuss stacks in Chapter 22.

A stack data structure allows nodes to be added to the stack and removed from the stack only at the *top*. For this reason, a stack is referred to as a *last-in, first-out (LIFO)* data structure. One way to implement a stack is as a constrained version of a linked list. In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the *bottom* of the stack.

The primary member functions used to manipulate a stack are `push` and `pop`. Function `push` *inserts* a new node at the top of the stack. Function `pop` *removes* a node from the top of the stack, stores the popped value in a reference variable that's passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

Stacks have many interesting applications. For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls. Section 6.11 discusses the function call stack in detail.

Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function. When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.

Stacks are used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several applications of stacks, including using them to develop your own complete working compiler.

We'll take advantage of the close relationship between lists and stacks to implement a stack class primarily by reusing a list class. First, we implement the stack class through `private` inheritance of the list class. Then we implement an identically performing stack class through composition by including a list object as a `private` member of a stack class. Of course, all of the data structures in this chapter, including these two stack classes, are implemented as templates to encourage further reusability.

The program of Figs. 20.13–20.14 creates a `Stack` class template (Fig. 20.13) primarily through `private` inheritance (line 9) of the `List` class template of Fig. 20.4. We want the Stack to have member functions push (lines 13–16), pop (lines 19–22), isStackEmpty (lines 25–28) and printStack (lines 31–34). Note that these are essentially the insertAtFront, removeFromFront, isEmpty and print functions of the List class template. Of course, the List class template contains other member functions (i.e., insertAtBack and removeFromBack) that we would not want to make accessible through the public interface to the Stack class. So when we indicate that the Stack class template is to inherit from the List class template, we specify private inheritance. This makes all the List class template's member functions private in the Stack class template. When we implement the Stack's member functions, we then have each of these call the appropriate member function of the List class—push calls insertAtFront (line 15), pop calls removeFromFront (line 21), isStackEmpty calls isEmpty (line 27) and printStack calls print (line 33)—this is referred to as **delegation**.

```
1  // Fig. 20.13: Stack.h
2  // Template Stack class definition derived from class List.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
11 public:
12    // push calls the List function insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16    } // end function push
17
18    // pop calls the List function removeFromFront
19    bool pop( STACKTYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function pop
23
24    // isStackEmpty calls the List function isEmpty
25    bool isStackEmpty() const
26    {
27       return this->isEmpty();
28    } // end function isStackEmpty
29
30    // printStack calls the List function print
31    void printStack() const
32    {
33       this->print();
34    } // end function print
35 }; // end class Stack
36
37 #endif
```

Fig. 20.13 | Stack class-template definition.

The explicit use of `this` on lines 27 and 33 is required so the compiler can properly resolve identifiers in template definitions. A **dependent name** is an identifier that depends on a template parameter. For example, the call to `removeFromFront` (line 21) depends on the argument `data` which has a type that's dependent on the template parameter `STACKTYPE`. Resolution of dependent names occurs when the template is instantiated. In contrast, the identifier for a function that takes no arguments like `isEmpty` or `print` in the `List` superclass is a **non-dependent name**. Such identifiers are normally resolved at the point where the template is defined. If the template has not yet been instantiated, then the code for the function with the non-dependent name does not yet exist and some compilers will generate compilation errors. Adding the explicit use of `this->` in lines 27 and 33 makes the calls to the base class's member functions dependent on the template parameter and ensures that the code will compile properly.

The stack class template is used in `main` (Fig. 20.14) to instantiate integer stack `intStack` of type `Stack< int >` (line 9). Integers 0 through 2 are pushed onto `intStack` (lines 14–18), then popped off `intStack` (lines 23–28). The program uses the `Stack` class template to create `doubleStack` of type `Stack< double >` (line 30). Values 1.1, 2.2 and 3.3 are pushed onto `doubleStack` (lines 36–41), then popped off `doubleStack` (lines 46–51).

```
1  // Fig. 20.14: Fig20_14.cpp
2  // Template Stack class test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class definition
5  using namespace std;
6
7  int main()
8  {
9     Stack< int > intStack; // create Stack of ints
10
11    cout << "processing an integer Stack" << endl;
12
13    // push integers onto intStack
14    for ( int i = 0; i < 3; ++i )
15    {
16       intStack.push( i );
17       intStack.printStack();
18    } // end for
19
20    int popInteger; // store int popped from stack
21
22    // pop integers from intStack
23    while ( !intStack.isStackEmpty() )
24    {
25       intStack.pop( popInteger );
26       cout << popInteger << " popped from stack" << endl;
27       intStack.printStack();
28    } // end while
29
30    Stack< double > doubleStack; // create Stack of doubles
31    double value = 1.1;
32
```

Fig. 20.14 | A simple stack program. (Part 1 of 2.)

```
   cout << "processing a double Stack" << endl;

   // push floating-point values onto doubleStack
   for ( int j = 0; j < 3; ++j )
   {
      doubleStack.push( value );
      doubleStack.printStack();
      value += 1.1;
   } // end for

   double popDouble; // store double popped from stack

   // pop floating-point values from doubleStack
   while ( !doubleStack.isStackEmpty() )
   {
      doubleStack.pop( popDouble );
      cout << popDouble << " popped from stack" << endl;
      doubleStack.printStack();
   } // end while
} // end main
```

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 20.14 | A simple stack program. (Part 2 of 2.)

Another way to implement a `Stack` class template is by reusing the `List` class template through *composition*. Figure 20.15 is a new implementation of the `Stack` class template that contains a `List< STACKTYPE >` object called `stackList` (line 38). This version of the `Stack` class template uses class `List` from Fig. 20.4. To test this class, use the driver program in Fig. 20.14, but include the new header—`Stackcomposition.h` in line 6 of that file. The output of the program is identical for both versions of class `Stack`.

```
1  // Fig. 20.15: Stackcomposition.h
2  // Template Stack class definition with composed List object.
3  #ifndef STACKCOMPOSITION_H
4  #define STACKCOMPOSITION_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack
10 {
11 public:
12    // no constructor; List constructor does initialization
13
14    // push calls stackList object's insertAtFront member function
15    void push( const STACKTYPE &data )
16    {
17       stackList.insertAtFront( data );
18    } // end function push
19
20    // pop calls stackList object's removeFromFront member function
21    bool pop( STACKTYPE &data )
22    {
23       return stackList.removeFromFront( data );
24    } // end function pop
25
26    // isStackEmpty calls stackList object's isEmpty member function
27    bool isStackEmpty() const
28    {
29       return stackList.isEmpty();
30    } // end function isStackEmpty
31
32    // printStack calls stackList object's print member function
33    void printStack() const
34    {
35       stackList.print();
36    } // end function printStack
37 private:
38    List< STACKTYPE > stackList; // composed List object
39 }; // end class Stack
40
41 #endif
```

Fig. 20.15 | `Stack` class template with a composed `List` object.

20.6 Queues

A **queue** is similar to a supermarket checkout line—the *first* person in line is serviced *first*, and other customers enter the line at the *end* and *wait* to be serviced. Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue. For this reason, a queue is referred to as a first-in, *first-out (FIFO)* data structure. The insert and remove operations are known as **enqueue** and **dequeue**.

Queues have many applications in computer systems. Computers that have a single processor can service only one user at a time. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

Queues are also used to support **print spooling**. For example, a single printer might be shared by all users of a network. Many users can send print jobs to the printer, even when the printer is already busy. These print jobs are placed in a queue until the printer becomes available. A program called a **spooler** manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

The program of Figs. 20.16–20.17 creates a `Queue` class template (Fig. 20.16) through `private` inheritance (line 9) of the `List` class template (Fig. 20.4). The `Queue` has member functions `enqueue` (lines 13–16), `dequeue` (lines 19–22), `isQueueEmpty` (lines 25–28) and `printQueue` (lines 31–34). These are essentially the `insertAtBack`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template. Of course, the `List` class template contains other member functions that we do *not* want to make accessible through the `public` interface to the `Queue` class. So when we indicate that the `Queue` class template is to inherit the `List` class template, we specify `private` inheritance. This makes all the `List` class template's member functions `private` in the `Queue` class template. When we implement the `Queue`'s member functions, we have each of these call the appropriate member function of the list class—`enqueue` calls `insertAtBack` (line 15), `dequeue` calls `removeFromFront` (line 21), `isQueueEmpty` calls `isEmpty` (line 27) and `printQueue` calls `print` (line 33). As with the `Stack` example in Fig. 20.13, this *delegation* requires explicit use of the `this` pointer in `isQueueEmpty` and `printQueue` to avoid compilation errors.

```
1  // Fig. 20.16: Queue.h
2  // Template Queue class definition derived from class List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // List class definition
7
8  template< typename QUEUETYPE >
9  class Queue : private List< QUEUETYPE >
10 {
11 public:
```

Fig. 20.16 | Queue class-template definition. (Part 1 of 2.)

```
   // enqueue calls List member function insertAtBack
   void enqueue( const QUEUETYPE &data )
   {
      insertAtBack( data );
   } // end function enqueue

   // dequeue calls List member function removeFromFront
   bool dequeue( QUEUETYPE &data )
   {
      return removeFromFront( data );
   } // end function dequeue

   // isQueueEmpty calls List member function isEmpty
   bool isQueueEmpty() const
   {
      return this->isEmpty();
   } // end function isQueueEmpty

   // printQueue calls List member function print
   void printQueue() const
   {
      this->print();
   } // end function printQueue
}; // end class Queue

#endif
```

Fig. 20.16 | Queue class-template definition. (Part 2 of 2.)

Figure 20.17 uses the `Queue` class template to instantiate integer queue `intQueue` of type `Queue< int >` (line 9). Integers 0 through 2 are enqueued to `intQueue` (lines 14–18), then dequeued from `intQueue` in first-in, first-out order (lines 23–28). Next, the program instantiates queue `doubleQueue` of type `Queue< double >` (line 30). Values 1.1, 2.2 and 3.3 are enqueued to `doubleQueue` (lines 36–41), then dequeued from `doubleQueue` in first-in, first-out order (lines 46–51).

```
1  // Fig. 20.17: Fig20_17.cpp
2  // Template Queue class test program.
3  #include <iostream>
4  #include "Queue.h" // Queue class definition
5  using namespace std;
6
7  int main()
8  {
9     Queue< int > intQueue; // create Queue of integers
10
11    cout << "processing an integer Queue" << endl;
12
13    // enqueue integers onto intQueue
14    for ( int i = 0; i < 3; ++i )
15    {
```

Fig. 20.17 | Queue-processing program. (Part 1 of 3.)

```
         intQueue.enqueue( i );
         intQueue.printQueue();
      } // end for

      int dequeueInteger; // store dequeued integer

      // dequeue integers from intQueue
      while ( !intQueue.isQueueEmpty() )
      {
         intQueue.dequeue( dequeueInteger );
         cout << dequeueInteger << " dequeued" << endl;
         intQueue.printQueue();
      } // end while

      Queue< double > doubleQueue; // create Queue of doubles
      double value = 1.1;

      cout << "processing a double Queue" << endl;

      // enqueue floating-point values onto doubleQueue
      for ( int j = 0; j < 3; ++j )
      {
         doubleQueue.enqueue( value );
         doubleQueue.printQueue();
         value += 1.1;
      } // end for

      double dequeueDouble; // store dequeued double

      // dequeue floating-point values from doubleQueue
      while ( !doubleQueue.isQueueEmpty() )
      {
         doubleQueue.dequeue( dequeueDouble );
         cout << dequeueDouble << " dequeued" << endl;
         doubleQueue.printQueue();
      } // end while
   } // end main
```

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1
```

Fig. 20.17 | Queue-processing program. (Part 2 of 3.)

```
The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 20.17 | Queue-processing program. (Part 3 of 3.)

20.7 Trees

Linked lists, stacks and queues are linear data structures. *A tree is a nonlinear, two-dimensional data structure.* Tree nodes contain two or more links. This section discusses **binary trees** (Fig. 20.18)—trees whose nodes all contain two links (none, one or both of which may be null).

Basic Terminology

For this discussion, refer to nodes `A`, `B`, `C` and `D` in Fig. 20.18. The **root node** (node `B`) is the first node in a tree. Each link in the root node refers to a **child** (nodes `A` and `D`). The **left child** (node `A`) is the root node of the **left subtree** (which contains only node `A`), and the **right child** (node `D`) is the root node of the **right subtree** (which contains nodes `D` and `C`). The children of a given node are called **siblings** (e.g., nodes `A` and `D` are siblings). A node with no children is a **leaf node** (e.g., nodes `A` and `C` are leaf nodes). Computer scientists normally draw trees from the root node down—the opposite of how trees grow in nature.

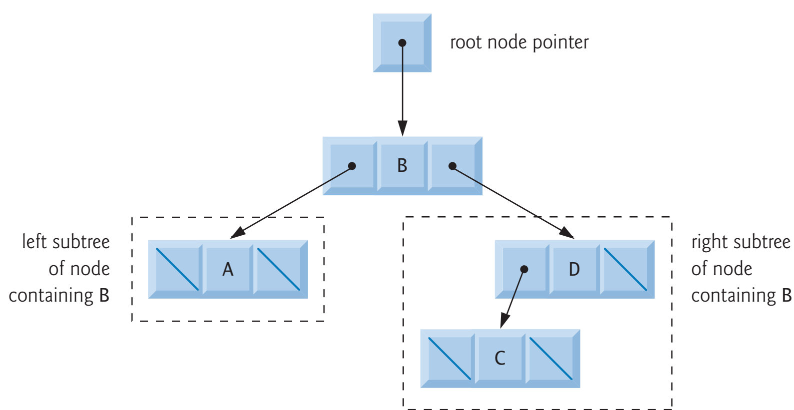


Fig. 20.18 | A graphical representation of a binary tree.

Binary Search Trees

A **binary search tree** (with no duplicate node values) has the characteristic that the values in any left subtree are *less than* the value in its **parent node**, and the values in any right subtree are *greater than* the value in its parent node. Figure 20.19 illustrates a binary search tree with 9 values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

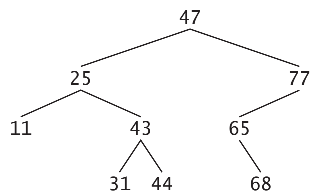


Fig. 20.19 | A binary search tree.

Implementing the Binary Search Tree Program

The program of Figs. 20.20–20.22 creates a binary search tree and traverses it (i.e., walks through all its nodes) three ways—using *recursive* **inorder**, **preorder** and **postorder traversals**. We explain these traversal algorithms shortly.

```
// Fig. 20.20: TreeNode.h
// Template TreeNode class definition.
#ifndef TREENODE_H
#define TREENODE_H

// forward declaration of class Tree
template< typename NODETYPE > class Tree;

// TreeNode class-template definition
template< typename NODETYPE >
class TreeNode
{
   friend class Tree< NODETYPE >;
public:
   // constructor
   TreeNode( const NODETYPE &d )
      : leftPtr( 0 ), // pointer to left subtree
        data( d ), // tree node data
        rightPtr( 0 ) // pointer to right substree
   {
      // empty body
   } // end TreeNode constructor

   // return copy of node's data
   NODETYPE getData() const
   {
```

Fig. 20.20 | TreeNode class-template definition. (Part 1 of 2.)

```
      return data;
   } // end getData function
private:
   TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
   NODETYPE data;
   TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
}; // end class TreeNode

#endif
```

Fig. 20.20 | TreeNode class-template definition. (Part 2 of 2.)

```
// Fig. 20.21: Tree.h
// Template Tree class definition.
#ifndef TREE_H
#define TREE_H

#include <iostream>
#include "TreeNode.h"
using namespace std;

// Tree class-template definition
template< typename NODETYPE > class Tree
{
public:
   Tree(); // constructor
   void insertNode( const NODETYPE & );
   void preOrderTraversal() const;
   void inOrderTraversal() const;
   void postOrderTraversal() const;
private:
   TreeNode< NODETYPE > *rootPtr;

   // utility functions
   void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE & );
   void preOrderHelper( TreeNode< NODETYPE > * ) const;
   void inOrderHelper( TreeNode< NODETYPE > * ) const;
   void postOrderHelper( TreeNode< NODETYPE > * ) const;
}; // end class Tree

// constructor
template< typename NODETYPE >
Tree< NODETYPE >::Tree()
{
   rootPtr = 0; // indicate tree is initially empty
} // end Tree constructor

// insert node in Tree
template< typename NODETYPE >
void Tree< NODETYPE >::insertNode( const NODETYPE &value )
{
```

Fig. 20.21 | Tree class-template definition. (Part 1 of 3.)

```
   insertNodeHelper( &rootPtr, value );
} // end function insertNode

// utility function called by insertNode; receives a pointer
// to a pointer so that the function can modify pointer's value
template< typename NODETYPE >
void Tree< NODETYPE >::insertNodeHelper(
   TreeNode< NODETYPE > **ptr, const NODETYPE &value )
{
   // subtree is empty; create new TreeNode containing value
   if ( *ptr == 0 )
      *ptr = new TreeNode< NODETYPE >( value );
   else // subtree is not empty
   {
      // data to insert is less than data in current node
      if ( value < ( *ptr )->data )
         insertNodeHelper( &( ( *ptr )->leftPtr ), value );
      else
      {
         // data to insert is greater than data in current node
         if ( value > ( *ptr )->data )
            insertNodeHelper( &( ( *ptr )->rightPtr ), value );
         else // duplicate data value ignored
            cout << value << " dup" << endl;
      } // end else
   } // end else
} // end function insertNodeHelper

// begin preorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::preOrderTraversal() const
{
   preOrderHelper( rootPtr );
} // end function preOrderTraversal

// utility function to perform preorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      cout << ptr->data << ' '; // process node
      preOrderHelper( ptr->leftPtr ); // traverse left subtree
      preOrderHelper( ptr->rightPtr ); // traverse right subtree
   } // end if
} // end function preOrderHelper

// begin inorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderTraversal() const
{
   inOrderHelper( rootPtr );
} // end function inOrderTraversal
```

Fig. 20.21 | Tree class-template definition. (Part 2 of 3.)

```
// utility function to perform inorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      inOrderHelper( ptr->leftPtr ); // traverse left subtree
      cout << ptr->data << ' '; // process node
      inOrderHelper( ptr->rightPtr ); // traverse right subtree
   } // end if
} // end function inOrderHelper

// begin postorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderTraversal() const
{
   postOrderHelper( rootPtr );
} // end function postOrderTraversal

// utility function to perform postorder traversal of Tree
template< typename NODETYPE >
void Tree< NODETYPE >::postOrderHelper(
   TreeNode< NODETYPE > *ptr ) const
{
   if ( ptr != 0 )
   {
      postOrderHelper( ptr->leftPtr ); // traverse left subtree
      postOrderHelper( ptr->rightPtr ); // traverse right subtree
      cout << ptr->data << ' '; // process node
   } // end if
} // end function postOrderHelper

#endif
```

Fig. 20.21 | Tree class-template definition. (Part 3 of 3.)

```
// Fig. 20.22: Fig20_22.cpp
// Tree class test program.
#include <iostream>
#include <iomanip>
#include "Tree.h" // Tree class definition
using namespace std;

int main()
{
   Tree< int > intTree; // create Tree of int values
   int intValue;

   cout << "Enter 10 integer values:\n";

```

Fig. 20.22 | Creating and traversing a binary tree. (Part 1 of 3.)

```
   // insert 10 integers to intTree
   for ( int i = 0; i < 10; ++i )
   {
      cin >> intValue;
      intTree.insertNode( intValue );
   } // end for

   cout << "\nPreorder traversal\n";
   intTree.preOrderTraversal();

   cout << "\nInorder traversal\n";
   intTree.inOrderTraversal();

   cout << "\nPostorder traversal\n";
   intTree.postOrderTraversal();

   Tree< double > doubleTree; // create Tree of double values
   double doubleValue;

   cout << fixed << setprecision( 1 )
      << "\n\n\nEnter 10 double values:\n";

   // insert 10 doubles to doubleTree
   for ( int j = 0; j < 10; ++j )
   {
      cin >> doubleValue;
      doubleTree.insertNode( doubleValue );
   } // end for

   cout << "\nPreorder traversal\n";
   doubleTree.preOrderTraversal();

   cout << "\nInorder traversal\n";
   doubleTree.inOrderTraversal();

   cout << "\nPostorder traversal\n";
   doubleTree.postOrderTraversal();
   cout << endl;
} // end main
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50


Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 2 of 3.)

```
Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inorder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

Fig. 20.22 | Creating and traversing a binary tree. (Part 3 of 3.)

We begin our discussion with the driver program (Fig. 20.22), then continue with the implementations of classes `TreeNode` (Fig. 20.20) and `Tree` (Fig. 20.21). Function `main` (Fig. 20.22) begins by instantiating integer tree `intTree` of type `Tree< int >` (line 10). The program prompts for 10 integers, each of which is inserted in the binary tree by calling `insertNode` (line 19). The program then performs preorder, inorder and postorder traversals (these are explained shortly) of `intTree` (lines 23, 26 and 29, respectively). The program then instantiates floating-point tree `doubleTree` of type `Tree< double >` (line 31). The program prompts for 10 `double` values, each of which is inserted in the binary tree by calling `insertNode` (line 41). The program then performs preorder, inorder and postorder traversals of `doubleTree` (lines 45, 48 and 51, respectively).

The `TreeNode` class template (Fig. 20.20) definition declares `Tree<NODETYPE>` as its `friend` (line 13). This makes all member functions of a given specialization of class template `Tree` (Fig. 20.21) friends of the corresponding specialization of class template `TreeNode`, so they can access the `private` members of `TreeNode` objects of that type. Because the `TreeNode` template parameter `NODETYPE` is used as the template argument for `Tree` in the `friend` declaration, `TreeNodes` specialized with a particular type can be processed only by a `Tree` specialized with the same type (e.g., a `Tree` of `int` values manages `TreeNode` objects that store `int` values).

Lines 30–32 declare a `TreeNode`'s `private` data—the node's `data` value, and pointers `leftPtr` (to the node's left subtree) and `rightPtr` (to the node's right subtree). The constructor (lines 16–22) sets `data` to the value supplied as a constructor argument and sets pointers `leftPtr` and `rightPtr` to zero (thus initializing this node to be a leaf node). Member function `getData` (lines 25–28) returns the `data` value.

Class template `Tree` (Fig. 20.21) has as `private` data `rootPtr` (line 20), a pointer to the tree's root node. Lines 15–18 declare the `public` member functions `insertNode` (that inserts a new node in the tree) and `preOrderTraversal`, `inOrderTraversal` and `postOrderTraversal`, each of which walks the tree in the designated manner. Each of these member functions calls its own recursive utility function to perform the appropriate operations on the internal representation of the tree, so the program is not required to access the underlying `private` data to perform these functions. Remember that the recursion requires us to pass in a pointer that represents the next subtree to process. The `Tree` constructor initializes `rootPtr` to zero to indicate that the tree is initially empty.

The `Tree` class's utility function `insertNodeHelper` (lines 45–66) is called by `insertNode` (lines 37–41) to recursively insert a node into the tree. *A node can only be inserted as a leaf node in a binary search tree.* If the tree is empty, a new `TreeNode` is created, initialized and inserted in the tree (lines 51–52).

If the tree is not empty, the program compares the value to be inserted with the `data` value in the root node. If the insert value is smaller (line 55), the program recursively calls `insertNodeHelper` (line 56) to insert the value in the left subtree. If the insert value is larger (line 60), the program recursively calls `insertNodeHelper` (line 61) to insert the value in the right subtree. If the value to be inserted is identical to the data value in the root node, the program prints the message `" dup"` (line 63) and returns *without inserting the duplicate value into the tree*. Note that `insertNode` passes the address of `rootPtr` to `insertNodeHelper` (line 40) so it can modify the value stored in `rootPtr` (i.e., the address of the root node). To receive a pointer to `rootPtr` (which is also a pointer), `insertNode-Helper`'s first argument is declared as a pointer to a pointer to a `TreeNode`.

Member functions `inOrderTraversal` (lines 88–92), `preOrderTraversal` (lines 69–73) and `postOrderTraversal` (lines 107–111) traverse the tree and print the node values. For the purpose of the following discussion, we use the binary search tree in Fig. 20.23.

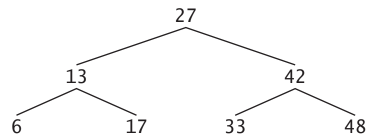


Fig. 20.23 | A binary search tree.

Inorder Traversal Algorithm

Function `inOrderTraversal` invokes utility function `inOrderHelper` to perform the inorder traversal of the binary tree. The steps for an inorder traversal are:

1. Traverse the left subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 100.)
2. Process the value in the node—i.e., print the node value (line 101).
3. Traverse the right subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 102.)

The value in a node is not processed until the values in its left subtree are processed, because each call to `inOrderHelper` immediately calls `inOrderHelper` again with the pointer to the left subtree. The inorder traversal of the tree in Fig. 20.23 is

```
6 13 17 27 33 42 48
```

Note that the inorder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually *sorts* the data—thus, this process is called the **binary tree sort**.

Preorder Traversal Algorithm

Function `preOrderTraversal` invokes utility function `preOrderHelper` to perform the preorder traversal of the binary tree. The steps for an preorder traversal are:

1. Process the value in the node (line 81).
2. Traverse the left subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 82.)
3. Traverse the right subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 83.)

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed. Then the values in the right subtree are processed. The preorder traversal of the tree in Fig. 20.23 is

```
27 13 6 17 42 33 48
```

Postorder Traversal Algorithm

Function `postOrderTraversal` invokes utility function `postOrderHelper` to perform the postorder traversal of the binary tree. The steps for a postorder traversal are:

1. Traverse the left subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 120.)
2. Traverse the right subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 121.)
3. Process the value in the node (line 122).

The value in each node is not printed until the values of its children are printed. The `postOrderTraversal` of the tree in Fig. 20.23 is

```
6 17 13 33 48 42 27
```

Duplicate Elimination

The binary search tree facilitates **duplicate elimination**. As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same “go left” or “go right” decisions on each comparison as the original value did when it was inserted in the tree. Thus, the duplicate will eventually be compared with a node containing the same value. The duplicate value may be *discarded* at this point.

Searching a binary tree for a value that matches a key value is also fast. If the tree is balanced, then each branch contains about half the number of nodes in the tree. Each comparison of a node to the search key eliminates half the nodes.This is called an $O(\log n)$ algorithm (Big O notation is discussed in Chapter 19). So a binary search tree with n elements would require a maximum of $\log_2 n$ comparisons either to find a match or to determine that no match exists. This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because $2^{10} > 1000$. When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because $2^{20} > 1{,}000{,}000$.

Overview of the Binary Tree Exercises

In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a **level-order traversal** of a binary tree. The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right. Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

20.8 Wrap-Up

In this chapter, you learned that linked lists are collections of data items that are “linked up in a chain.” You also learned that a program can perform insertions and deletions anywhere in a linked list (though our implementation only performed insertions and deletions at the ends of the list). We demonstrated that the stack and queue data structures are constrained versions of lists. For stacks, you saw that insertions and deletions are made only at the top. For queues, you saw that insertions are made at the tail and deletions are made from the head. We also presented the binary tree data structure. You saw a binary search tree that facilitated high-speed searching and sorting of data and efficient duplicate elimination. You learned how to create these data structures for reusability (as templates) and maintainability. In the next chapter, we introduce `struct`s, which are similar to classes, and discuss the manipulation of bits, characters and C-style strings.

Summary

Section 20.1 Introduction

- Dynamic data structures (p. 747) grow and shrink during execution.
- Linked lists (p. 747) are collections of data items "lined up in a row"—insertions and removals are made anywhere in a linked list.
- Stacks (p. 747) are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top (p. 747).
- Queues (p. 747) represent waiting lines; insertions are made at the back (also referred to as the tail; p. 747) of a queue and removals are made from the front (also referred to as the head; p. 747).
- Binary trees (p. 747) facilitate high-speed searching and sorting of data, efficient duplicate elimination, representation of file-system directories and compilation of expressions into machine language.

Section 20.2 Self-Referential Classes

- A self-referential class (p. 748) contains a pointer that points to an object of the same class type.
- Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees.

Section 20.3 Dynamic Memory Allocation and Data Structures

- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system.

Section 20.4 Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links (p. 749)—hence, the term "linked" list.
- A linked list is accessed via a pointer to the first node of the list. Each subsequent node is accessed via the link-pointer member stored in the previous node.
- Linked lists, stacks and queues are linear data structures (p. 749). Trees are nonlinear data structures (p. 749).
- A linked list is appropriate when the number of data elements to be represented is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- A singly linked list () begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence."
- A circular, singly linked list (p. 763) begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain a null pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

- A doubly linked list (p. 763) allows traversals both forward and backward.
- A doubly linked list is often implemented with two “start pointers”—one that points to the first element to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal. Each node has a pointer to both the next and previous nodes.
- In a circular, doubly linked list (p. 763), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the “circle.”

Section 20.5 Stacks

- A stack data structure allows nodes to be added to and removed from the stack only at the top.
- A stack is referred to as a last-in, first-out (LIFO) data structure.
- Function `push` (p. 764) inserts a new node at the top of the stack. Function `pop` removes a node from the top of the stack.
- A dependent name (p. 766) is an identifier that depends on the value of a template parameter. Resolution of dependent names occurs when the template is instantiated.
- Non-dependent names (p. 766) are resolved at the point where the template is defined.

Section 20.6 Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from a queue’s head and are inserted only at its tail (p. 769).
- A queue is referred to as a first-in, first-out (FIFO) data structure. The insert and remove operations are known as `enqueue` and `dequeue` (p. 769).

Section 20.7 Trees

- Binary trees are trees whose nodes all contain two links (none, one or both of which may be null).
- The root node (p. 772) is the first node in a tree.
- Each link in the root node refers to a child. The left child is the root node of the left subtree, and the right child is the root node of the right subtree.
- The children of a single node are called siblings (p. 772). A node with no children is called a leaf node (p. 772).
- A binary search tree (p. 773) (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node (p. 773), and the values in any right subtree are greater than the value in its parent node.
- A node can only be inserted as a leaf node in a binary search tree.
- An inorder traversal (p. 779) of a binary tree traverses the left subtree, processes the value in the root node then traverses the right subtree. The value in a node is not processed until the values in its left subtree are processed.
- A preorder traversal (p. 773) processes the value in the root node, traverses the left subtree, then traverses the right subtree. The value in each node is processed as the node is encountered.
- A postorder traversal (p. 773) traverses the left subtree, traverses the right subtree, then processes the root node’s value. The value in each node is not processed until the values in both subtrees are processed.
- The binary search tree helps eliminate duplicate data (p. 780). As the tree is being created, an attempt to insert a duplicate value will be recognized and the duplicate value may be discarded.
- The level-order traversal (p. 780) of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right.

Self-Review Exercises

20.1 Fill in the blanks in each of the following:

a) A self-________ class is used to form dynamic data structures that can grow and shrink at execution time

b) The ________ operator is used to dynamically allocate memory and construct an object; this operator returns a pointer to the object.

c) A(n) ________ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list and node values are returned in last-in, first-out order.

d) A function that does not alter a linked list, but looks at the list to determine whether it's empty, is an example of a(n) ________ function.

e) A queue is referred to as a(n) ________ data structure, because the first nodes inserted are the first nodes removed.

f) The pointer to the next node in a linked list is referred to as a(n) ________.

g) The ________ operator is used to destroy an object and release dynamically allocated memory.

h) A(n) ________ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.

i) A(n) ________ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.

j) A stack is referred to as a(n) ________ data structure, because the last node inserted is the first node removed.

k) The nodes of a(n) ________ tree contain two link members.

l) The first node of a tree is the ________ node.

m) Each link in a tree node points to a(n) ________ or ________ of that node.

n) A tree node that has no children is called a(n) ________ node.

o) The four traversal algorithms we mentioned in the text for binary search trees are ________, ________, ________ and ________.

20.2 What are the differences between a linked list and a stack?

20.3 What are the differences between a stack and a queue?

20.4 Perhaps a more appropriate title for this chapter would have been "Reusable Data Structures." Comment on how each of the following entities or concepts contributes to the reusability of data structures:

a) classes

b) class templates

c) inheritance

d) `private` inheritance

e) composition

20.5 Provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 20.24.

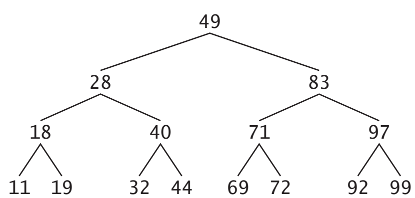


Fig. 20.24 | A 15-node binary search tree.

Answers to Self-Review Exercises

20.1 a) referential. b) `new`. c) stack. d) predicate. e) first-in, first-out (FIFO). f) link. g) `delete`. h) queue. i) tree. j) last-in, first-out (LIFO). k) binary. l) root. m) child or subtree. n) leaf. o) inorder, preorder, postorder and level order.

20.2 It's possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may only be inserted at the top of the stack and removed from the top of a stack.

20.3 A queue data structure allows nodes to be removed only from the head of the queue and inserted only at the tail of the queue. A queue is referred to as a first-in, first-out (FIFO) data structure. A stack data structure allows nodes to be added to the stack and removed from the stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure.

20.4
a) Classes allow us to instantiate as many data structure objects of a certain type (i.e., class) as we wish.
b) Class templates enable us to instantiate related classes, each based on different type parameters—we can then generate as many objects of each template class as we like.
c) Inheritance enables us to reuse code from a base class in a derived class, so that the derived-class data structure is also a base-class data structure (with `public` inheritance, that is).
d) Private inheritance enables us to reuse portions of the code from a base class to form a derived-class data structure; because the inheritance is `private`, all `public` base-class member functions become `private` in the derived class. This enables us to prevent clients of the derived-class data structure from accessing base-class member functions that do not apply to the derived class.
e) Composition enables us to reuse code by making a class object data structure a member of a composed class; if we make the class object a `private` member of the composed class, then the class object's `public` member functions are not available through the composed object's interface.

20.5 The inorder traversal is

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

The preorder traversal is

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

The postorder traversal is

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Exercises

20.6 *(Concatenating Lists)* Write a program that concatenates two linked list objects of characters. The program should include function `concatenate`, which takes references to both list objects as arguments and concatenates the second list to the first list.

20.7 *(Merging Ordered Lists)* Write a program that merges two ordered list objects of integers into a single ordered list object of integers. Function `merge` should receive references to each of the list objects to be merged and reference to a list object into which the merged elements will be placed.

20.8 *(Summing and Averaging Elements in a List)* Write a program that inserts 25 random integers from 0 to 100 in order in a linked list object. The program should calculate the sum of the elements and the floating-point average of the elements.

20.9 *(Copying a List in Reverse Order)* Write a program that creates a linked list object of 10 characters and creates a second list object containing a copy of the first list, but in reverse order.

20.10 *(Printing a Sentence in Reverse Order with a Stack)* Write a program that inputs a line of text and uses a stack object to print the line reversed.

20.11 *(Palindrome Testing with Stacks)* Write a program that uses a stack object to determine if a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.

20.12 *(Infix-to-Postfix Conversion)* Stacks are used by compilers to help in the process of evaluating expressions and generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators and parentheses.

Humans generally write expressions like `3 + 4` and `7 / 9` in which the operator (`+` or `/` here) is written between its operands—this is called **infix notation**. Computers "prefer" **postfix notation** in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation and evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm the stack is used for a different purpose.

In this exercise, you'll write a C++ version of the infix-to-postfix conversion algorithm. In the next exercise, you'll write a C++ version of the postfix expression evaluation algorithm. Later in the chapter, you'll discover that code you write in this exercise can help you implement a complete working compiler.

Write a program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into `string infix` and use modified versions of the stack functions implemented in this chapter to help create the postfix expression in `string postfix`. The algorithm for creating a postfix expression is as follows:

1) Push a left parenthesis `'('` onto the stack.
2) Append a right parenthesis `')'` to the end of `infix`.
3) While the stack is not empty, read `infix` from left to right and do the following:
   - If the current character in `infix` is a digit, copy it to the next element of `postfix`.
   - If the current character in `infix` is a left parenthesis, push it onto the stack.
   - If the current character in `infix` is an operator,
     - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in `postfix`.
     - Push the current character in `infix` onto the stack.
   - If the current character in `infix` is a right parenthesis
     - Pop operators from the top of the stack and insert them in `postfix` until a left parenthesis is at the top of the stack.
     - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `^` exponentiation
- `%` modulus

[*Note:* We assume left-to-right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes, each containing a data member and a pointer to the next stack node.

Some of the functional capabilities you may want to provide are:

a) function `convertToPostfix` that converts the infix expression to postfix notation
b) function `isOperator` that determines whether `c` is an operator
c) function `precedence` that determines whether the precedence of `operator1` is greater than or equal to the precedence of `operator2`, and, if so, returns `true`.
d) function `push` that pushes a value onto the stack
e) function `pop` that pops a value off the stack
f) function `stackTop` that returns the top value of the stack without popping the stack
g) function `isEmpty` that determines if the stack is empty
h) function `printStack` that prints the stack

20.13 *(Postfix Evaluation)* Write a program that evaluates a postfix expression (assume it's valid) such as

```
6 2 + 5 * 8 4 / -
```

The program should read a postfix expression consisting of digits and operators into a `string`. Using modified versions of the stack functions implemented earlier in this chapter, the program should scan the expression and evaluate it. The algorithm is as follows:

1) While you have not reached the end of the `string`, read the expression from left to right.
   If the current character is a digit,
      Push its integer value onto the stack (the integer value of a digit character is its value in the computer's character set minus the value of `'0'` in the computer's character set).
   Otherwise, if the current character is an *operator*,
      Pop the two top elements of the stack into variables `x` and `y`.
      Calculate `y` *operator* `x`.
      Push the result of the calculation onto the stack.
2) When you reach the end of the `string`, pop the top value of the stack. This is the result of the postfix expression.

[*Note:* In *Step 2* above, if the operator is `'/'`, the top of the stack is `2` and the next element in the stack is `8`, then pop `2` into `x`, pop `8` into `y`, evaluate `8 / 2` and push the result, `4`, back onto the stack. This note also applies to operator `'-'`.] The arithmetic operations allowed in an expression are

`+` addition
`-` subtraction
`*` multiplication
`/` division
`^` exponentiation
`%` modulus

[*Note:* We assume left-to-right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes that contain an `int` data member and a pointer to the next stack node. You may want to provide the following functional capabilities:

a) function `evaluatePostfixExpression` that evaluates the postfix expression
b) function `calculate` that evaluates the expression `op1 operator op2`
c) function `push` that pushes a value onto the stack
d) function `pop` that pops a value off the stack
e) function `isEmpty` that determines if the stack is empty
f) function `printStack` that prints the stack

20.14 *(Postfix Evaluation Enhanced)* Modify the postfix evaluator program of Exercise 20.13 so that it can process integer operands larger than 9.

20.15 *(Supermarket Simulation)* Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of 1–4 minutes. Also, each customer is served in random integer intervals of 1–4 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with "balanced" rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes) using the following algorithm:

1) Choose a random integer from 1 to 4 to determine the minute at which the first customer arrives.
2) At the first customer's arrival time:
   Determine customer's service time (random integer from 1 to 4);
   Begin servicing the customer;
   Schedule arrival time of next customer (random integer 1 to 4 added to the current time).
3) For each minute of the day:
   If the next customer arrives,
      Say so, enqueue the customer, and schedule the arrival time of the next customer;
   If service was completed for the last customer;
      Say so, dequeue next customer to be serviced and determine customer's service completion time (random integer from 1 to 4 added to the current time).

Now run your simulation for 720 minutes, and answer each of the following:

a) What's the maximum number of customers in the queue at any time?
b) What's the longest wait any one customer experiences?
c) What happens if the arrival interval is changed from 1–4 minutes to 1–3 minutes?

20.16 *(Allowing Duplicates in Binary Trees)* Modify the program of Figs. 20.20–20.22 to allow the binary tree object to contain duplicates.

20.17 *(Binary Tree of Strings)* Write a program based on Figs. 20.20–20.22 that inputs a line of text, tokenizes the sentence into separate words (you may want to use the `istringstream` library class), inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree. Use an OOP approach.

20.18 *(Duplicate Elimination)* In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you'd perform duplicate elimination using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

20.19 *(Depth of a Binary Tree)* Write a function `depth` that receives a binary tree and determines how many levels it has.

20.20 (*Recursively Print a List Backward*) Write a member function `printListBackward` that recursively outputs the items in a linked list object in reverse order. Write a test program that creates a sorted list of integers and prints the list in reverse order.

20.21 (*Recursively Search a List*) Write a member function `searchList` that recursively searches a linked list object for a specified value. The function should return a pointer to the value if it's found; otherwise, null should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

20.22 (*Binary Tree Delete*) Deleting items from binary search trees is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item—the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the pointer in the parent node is set to null.

If the item to be deleted is contained in a node with one child, the pointer in the parent node is set to point to the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the pointer in the parent node cannot be assigned to point to one of the children of the node to be deleted. In most cases, the resulting binary search tree would not adhere to the following characteristic of binary search trees (with no duplicate values): *The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node*.

Which node is used as a *replacement node* to maintain this characteristic? Either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let's consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the pointer to the right child of the current node is null. We are now pointing to the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable (this pointer is used to delete the dynamically allocated memory).
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to null.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

1) Store the pointer to the node to be deleted in a temporary pointer variable.
2) Set the pointer in the parent of the node being deleted to point to the replacement node.
3) Set the pointer in the parent of the replacement node to point to the left child of the replacement node.
4) Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.
5) Delete the node to which the temporary pointer variable points.

Write member function `deleteNode`, which takes as its arguments a pointer to the root node of the tree object and the value to be deleted. The function should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. The function should print a message that indicates whether the value is deleted. Modify the program of Figs. 20.20–20.22 to use this function. After deleting an item, call the `inOrder`, `preOrder` and `postOrder` traversal functions to confirm that the delete operation was performed correctly.

20.23 (***Binary Tree Search***) Write member function `binaryTreeSearch`, which attempts to locate a specified value in a binary search tree object. The function should take as arguments a pointer to the binary tree's root node and a search key to locate. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a null pointer.

20.24 (***Level-Order Binary Tree Traversal***) The program of Figs. 20.20–20.22 illustrated three recursive methods of traversing a binary tree—inorder, preorder and postorder traversals. This exercise presents the *level-order traversal* of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The level-order traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

1) Insert the root node in the queue
2) While there are nodes left in the queue,
   Get the next node in the queue
   Print the node's value
   If the pointer to the left child of the node is not null
      Insert the left child node in the queue
   If the pointer to the right child of the node is not null
      Insert the right child node in the queue.

Write member function `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Figs. 20.20–20.22 to use this function. [*Note:* You'll also need to modify and incorporate the queue-processing functions of Fig. 20.16 in this program.]

20.25 (***Printing Trees***) Write a recursive member function `outputTree` to display a binary tree object on the screen. The function should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 20.24 is output as shown in Fig. 20.25. Note that the rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output (this variable should start at zero, so the root node is output at the left of the screen). The function uses a modified inorder traversal to output the tree—it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the pointer to the current node is not null
   Recursively call `outputTree` with the current node's right subtree and `totalSpaces` + 5
   Use a `for` structure to count from 1 to `totalSpaces` and output spaces
   Output the value in the current node
   Set the pointer to the current node to point to the left subtree of the current node
   Increment `totalSpaces` by 5.

```
                    99
               97
                    92
          83
                    72
               71
                    69
     49
                    44
               40
                    32
          28
                    19
               18
                    11
```

Fig. 20.25 | Displaying a binary tree

20.26 *(Insert/Delete Anywhere in a Linked List)* Our linked list class template allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used `private` inheritance and composition to produce a stack class template and a queue class template with a minimal amount of code by reusing the list class template. Actually, linked lists are more general than those we provided. Modify the linked list class template we developed in this chapter to handle insertions and deletions anywhere in the list.

20.27 *(List and Queues without Tail Pointers)* Our implementation of a linked list (Figs. 20.3–20.5) used both a `firstPtr` and a `lastPtr`. The `lastPtr` was useful for the `insertAtBack` and `removeFromBack` member functions of the `List` class. The `insertAtBack` function corresponds to the `enqueue` member function of the `Queue` class. Rewrite the `List` class so that it does not use a `lastPtr`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class (Fig. 20.16)?

20.28 Use the composition version of the stack program (Fig. 20.15) to form a complete working stack program. Modify this program to `inline` the member functions. Compare the two approaches. Summarize the advantages and disadvantages of inlining member functions.

20.29 *(Performance of Binary Tree Sorting and Searching)* One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree—for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data were inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?

20.30 *(Indexed Lists)* As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list searching performance is to create and maintain an index to the list. An index is a set of pointers to various key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries—one for each letter of the alphabet. A search operation for a last name beginning with `"Y"` would first search the index to determine where the `"Y"` entries begin and "jump into" the list at that point and search linearly until the desired name was found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Figs. 20.3–20.5 as the basis of an `IndexedList` class. Write a program that demonstrates the operation of indexed lists. Be sure to include member functions `insertInIndexedList`, `searchInIndexedList` and `deleteFromIndexedList`.

Special Section: Building Your Own Compiler

In Exercises 8.18–8.19 and 8.20, we introduced Simpletron Machine Language (SML), and you implemented a Simpletron computer simulator to execute SML programs. In Exercises 20.31—20.35, we build a compiler that converts programs written in a high-level programming language to SML. This section "ties" together the entire programming process. You'll write programs in this new high-level language, compile them on the compiler you build and run them on the simulator you built in Exercise 8.19. You should make every effort to implement your compiler in an object-oriented manner. [*Note:* Due to the size of the descriptions for Exercises 20.31—20.35, we've posted them in a PDF document located at `www.deitel.com/books/cpphtp8/`.]

Bits, Characters, C Strings and `structs`

21

The same old charitable lie
Repeated as the years scoot by
Perpetually makes a hit—
"You really haven't changed a bit!"
—Margaret Fishback

The chief defect of Henry King
Was chewing little bits of string.
—Hilaire Belloc

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences.
—William Strunk, Jr.

Objectives

In this chapter you'll learn:

- To create and use `struct`s and to understand their near equivalence with classes.
- To use `typedef` to create aliases for data types.
- To manipulate data with the bitwise operators and to create bit fields for storing data compactly.
- To use the functions of the character-handling library `<cctype>`.
- To use the string-conversion functions of the general-utilities library `<cstdlib>`.
- To use the string-processing functions of the string-handling library `<cstring>`.

Outline

21.1 Introduction

We now discuss structures, their near equivalence with classes, and the manipulation of bits, characters and C-style strings. Many of the techniques we present here are included for the benefit of those who will work with legacy C and C++ code.

C++'s designers evolved structures into the notion of a class. Like classes, C++ structures may contain access specifiers, member functions, constructors and destructors. In fact, *the only differences between structures and classes in C++ is that structure members default to* `public` *access and class members default to* `private` *access when no access specifiers are used, and that structures default to* `public` *inheritance, whereas classes default to* `private` *inheritance.* Classes have been covered thoroughly in the book, so there is really no need for us to discuss structures in detail. Our presentation of structures in this chapter focuses on their use in a C-like manner, where they contain only `public` data members. This use of structures is typical of the legacy C code and early C++ code you'll see in industry.

We present a high-performance card shuffling and dealing simulation in which we use structure objects containing C++ `string` objects to represent the cards. We discuss the bitwise operators that allow you to access and manipulate the individual bits in bytes of data. We also present bitfields—special structures that can be used to specify the exact number of bits a variable occupies in memory. These bit manipulation techniques are common in C and C++ programs that interact directly with hardware devices that have limited memory. The chapter finishes with examples of many character and C string manipulation functions—some of which are designed to process blocks of memory as arrays of bytes. The detailed C string treatment in this chapter is mostly for reasons of legacy code support and because there are still remnants of C string use in C++, such as command-line arguments (Appendix F). *New development should use C++* `string` *objects rather than C strings.*

21.2 Structure Definitions

Consider the following structure definition:

```
struct Card
{
   string face;
   string suit;
}; // end struct Card
```

Keyword **struct** introduces the definition for structure Card. The identifier Card is the **structure name** and is used in C++ to declare variables of the **structure type** (in C, the type name of the preceding structure is struct Card). Data (and possibly functions—just as with classes) declared within the braces of the structure definition are the structure's members. Card's definition contains two string members—face and suit.

The following declarations

```
Card oneCard;
Card deck[ 52 ];
Card *cardPtr;
```

declare oneCard to be a structure variable of type Card, deck to be an array with 52 elements of type Card and cardPtr to be a pointer to a Card structure. Variables of a given structure type can also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding declarations could have been incorporated into the Card structure definition as follows:

```
struct Card
{
   string face;
   string suit;
} oneCard, deck[ 52 ], *cardPtr;
```

As with classes, structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are "holes" in a structure, because some computers store specific data types only on certain memory boundaries for performance reasons, such as half-word, word or double-word boundaries. A word is a standard memory unit used to store data in a computer—usually two bytes or four bytes and typically four bytes on 32-bit systems. Consider the following structure definition in which structure objects sample1 and sample2 of type Example are declared:

```
struct Example
{
   char c;
   int i;
} sample1, sample2;
```

A computer with two-byte words might require that each of the members of Example be aligned on a word boundary (i.e., at the beginning of a word—this is machine dependent). Figure 21.1 shows a sample storage alignment for an object of type Example that's been assigned the character 'a' and the integer 97 (the bit representations of the values are shown). If the members are stored beginning at word boundaries, there is a one-byte hole (byte 1 in the figure) in the storage for objects of type Example. The value in the one-byte hole is undefined. If the member values of sample1 and sample2 are in fact equal, the structure objects are not necessarily equal, because the undefined one-byte holes are not likely to contain identical values.

Common Programming Error 21.1

Comparing variables of structure types is a compilation error.

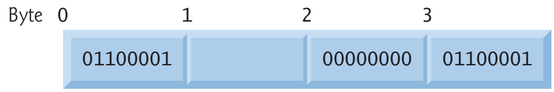


Fig. 21.1 | Possible storage alignment for a variable of type `Example`, showing an undefined area in memory.

Portability Tip 21.1
Because the size of data items of a particular type is machine dependent, and because storage alignment considerations are machine dependent, so too is the representation of a structure.

21.3 typedef

Keyword **`typedef`** provides a mechanism for creating synonyms (or aliases) for previously defined data types. Names for structure types are often defined with `typedef` to create shorter, simpler or more readable type names. For example, the statement

```
typedef Card *CardPtr;
```

defines the new type name `CardPtr` as a synonym for type `Card *`.

Creating a new name with `typedef` does not create a new type; `typedef` simply creates a new type name that can then be used in the program as an alias for an existing type name.

Portability Tip 21.2
Synonyms for built-in data types can be created with `typedef` to make programs more portable. For example, a program can use `typedef` to create alias `Integer` for four-byte integers. `Integer` can then be aliased to `int` on systems with four-byte integers and can be aliased to `long int` on systems with two-byte integers where `long int` values occupy four bytes. Then, you simply declare all four-byte integer variables to be of type `Integer`.

21.4 Example: Card Shuffling and Dealing Simulation

The card shuffling and dealing program in Figs. 21.2–21.4 is similar to the one described in Exercise 10.10. This program represents the deck of cards as a `vector` of structures and uses high-performance shuffling and dealing algorithms.

```
// Fig. 21.2: DeckOfCards.h
// Definition of class DeckOfCards that
// represents a deck of playing cards.
#include <string>
#include <vector>
using namespace std;

// Card structure definition
struct Card
{
```

Fig. 21.2 | Header for `DeckOfCards` class. (Part 1 of 2.)

```
   string face;
   string suit;
}; // end structure Card

// DeckOfCards class definition
class DeckOfCards
{
public:
   static const int numberOfCards = 52;
   static const int faces = 13;
   static const int suits = 4;

   DeckOfCards(); // constructor initializes deck
   void shuffle(); // shuffles cards in deck
   void deal() const; // deals cards in deck

private:
   vector< Card > deck; // represents deck of cards
}; // end class DeckOfCards
```

Fig. 21.2 | Header for `DeckOfCards` class. (Part 2 of 2.)

The constructor (lines 12–32 of Fig. 21.3) initializes the `vector` in order with character strings representing Ace through King of each suit. Function `shuffle` implements the high-performance shuffling algorithm. The function loops through all 52 cards (subscripts 0 to 51). For each card, a number between 0 and 51 is picked randomly. Next, the current `Card` and the randomly selected `Card` are swapped in the `vector`. A total of 52 swaps are made in a single pass of the entire `vector`, and the `vector` is shuffled. Because the `Card` structures were swapped in place in the `vector`, the dealing algorithm implemented in function `deal` requires only one pass of the `vector` to deal the shuffled cards.

```
1  // Fig. 21.3: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  #include <iomanip>
6  #include <cstdlib> // prototypes for rand and srand
7  #include <ctime> // prototype for time
8  #include "DeckOfCards.h" // DeckOfCards class definition
9  using namespace std;
10
11 // no-argument DeckOfCards constructor intializes deck
12 DeckOfCards::DeckOfCards()
13    : deck( numberOfCards )
14 {
15    // initialize suit array
16    static string suit[ suits ] =
17       { "Hearts", "Diamonds", "Clubs", "Spades" };
18
```

Fig. 21.3 | Class file for `DeckOfCards`. (Part 1 of 2.)

```
   // initialize face array
   static string face[ faces ] =
      { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
      "Eight", "Nine", "Ten", "Jack", "Queen", "King" };

   // set values for deck of 52 Cards
   for ( int i = 0; i < numberOfCards; ++i )
   {
      deck[ i ].face = face[ i % faces ];
      deck[ i ].suit = suit[ i / faces ];
   } // end for

   srand( time( 0 ) ); // seed random number generator
} // end no-argument DeckOfCards constructor

// shuffle cards in deck
void DeckOfCards::shuffle()
{
   // shuffle cards randomly
   for ( int i = 0; i < numberOfCards; ++i )
   {
      int j = rand() % numberOfCards;
      Card temp = deck[ i ];
      deck[ i ] = deck[ j ];
      deck[ j ] = temp;
   } // end for
} // end function shuffle

// deal cards in deck
void DeckOfCards::deal() const
{
   // display each card's face and suit
   for ( int i = 0; i < numberOfCards; ++i )
      cout << right << setw( 5 ) << deck[ i ].face << " of "
         << left << setw( 8 ) << deck[ i ].suit
         << ( ( i + 1 ) % 2 ? '\t' : '\n' );
} // end function deal
```

Fig. 21.3 | Class file for DeckOfCards. (Part 2 of 2.)

```
// Fig. 21.4: fig21_04.cpp
// Card shuffling and dealing program.
#include "DeckOfCards.h" // DeckOfCards class definition

int main()
{
   DeckOfCards deckOfCards; // create DeckOfCards object
   deckOfCards.shuffle(); // shuffle the cards in the deck
   deckOfCards.deal(); // deal the cards in the deck
} // end main
```

Fig. 21.4 | High-performance card shuffling and dealing simulation. (Part 1 of 2.)

```
 King of Clubs           Ten of Diamonds
 Five of Diamonds       Jack of Clubs
Seven of Spades         Five of Clubs
Three of Spades         King of Hearts
  Ten of Clubs         Eight of Spades
Eight of Hearts          Six of Hearts
 Nine of Diamonds       Nine of Clubs
Three of Diamonds      Queen of Hearts
  Six of Clubs         Seven of Hearts
Seven of Diamonds       Jack of Diamonds
 Jack of Spades         King of Diamonds
Deuce of Diamonds       Four of Clubs
Three of Clubs          Five of Hearts
Eight of Clubs           Ace of Hearts
Deuce of Spades          Ace of Clubs
  Ten of Spades        Eight of Diamonds
  Ten of Hearts          Six of Spades
Queen of Diamonds       Nine of Hearts
Seven of Clubs         Queen of Clubs
Deuce of Clubs         Queen of Spades
Three of Hearts         Five of Spades
Deuce of Hearts         Jack of Hearts
 Four of Hearts          Ace of Diamonds
 Nine of Spades         Four of Diamonds
  Ace of Spades          Six of Diamonds
 Four of Spades         King of Spades
```

Fig. 21.4 | High-performance card shuffling and dealing simulation. (Part 2 of 2.)

21.5 Bitwise Operators

C++ provides extensive bit-manipulation capabilities for getting down to the so-called "bits-and-bytes" level. Operating systems, test-equipment software, networking software and many other kinds of software require that you communicate "directly with the hardware." This and the next several sections discuss *bit manipulation*. We introduce each of C++'s many *bitwise operators*, and we discuss how to save memory by using *bit fields*.

All data is represented internally by computers as sequences of bits. Each bit can assume the value `0` or the value `1`. On most systems, a sequence of eight bits forms a byte—the standard storage unit for a variable of type `char`. Other data types are stored in larger numbers of bytes. Bitwise operators are used to manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both `signed` and `unsigned`). Unsigned integers are normally used with the bitwise operators.

Portability Tip 21.3

Bitwise data manipulations are machine dependent.

The bitwise operator discussions in this section show the binary representations of the integer operands. For a detailed explanation of the binary (also called base-2) number system, see Appendix D. Because of the machine-dependent nature of bitwise manipulations, some of these programs might not work on your system without modification.

The bitwise operators are: **bitwise AND (`&`)**, **bitwise inclusive OR (`|`)**, **bitwise exclusive OR (`^`)**, **left shift (`<<`)**, **right shift (`>>`)** and **bitwise complement (`~`)**—also known as the **one's complement**. (Note that we've been using `&`, `<<` and `>>` for other purposes. This is a classic example of operator overloading.) The bitwise AND, bitwise inclusive OR and bitwise exclusive OR operators compare their two operands bit by bit. The *bitwise AND* operator sets each bit in the result to 1 if the corresponding bit in *both* operands is 1. The *bitwise inclusive OR* operator sets each bit in the result to 1 if the corresponding bit in *either (or both)* operand(s) is 1. The *bitwise exclusive OR* operator sets each bit in the result to 1 if the corresponding bit in *either* operand—*but not both*—is 1. The *left-shift* operator shifts the bits of its left operand to the left by the number of bits specified in its right operand. The *right-shift* operator shifts the bits in its left operand to the right by the number of bits specified in its right operand. The *bitwise complement* operator sets all `0` bits in its operand to `1` in the result and sets all `1` bits in its operand to `0` in the result. Detailed discussions of each bitwise operator appear in the following examples. The bitwise operators are summarized in Fig. 21.5.

Operator	Name	Description
`&`	bitwise AND	The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`.
`|`	bitwise inclusive OR	The bits in the result are set to `1` if one or both of the corresponding bits in the two operands is `1`.
`^`	bitwise exclusive OR	The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with `0` bits.
`>>`	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
`~`	bitwise complement	All `0` bits are set to `1` and all `1` bits are set to `0`.

Fig. 21.5 | Bitwise operators.

Printing a Binary Representation of an Integral Value

When using the bitwise operators, it's useful to illustrate their precise effects by printing values in their binary representation. The program of Fig. 21.6 prints an `unsigned` integer in its binary representation in groups of eight bits each.

```
// Fig. 21.6: fig21_06.cpp
// Printing an unsigned integer in bits.
#include <iostream>
#include <iomanip>
using namespace std;

void displayBits( unsigned ); // prototype

```

Fig. 21.6 | Printing an unsigned integer in bits. (Part 1 of 2.)

```
 9 int main()
10 {
11    unsigned inputValue; // integral value to print in binary
12
13    cout << "Enter an unsigned integer: ";
14    cin >> inputValue;
15    displayBits( inputValue );
16 } // end main
17
18 // display bits of an unsigned integer value
19 void displayBits( unsigned value )
20 {
21    const int SHIFT = 8 * sizeof( unsigned ) - 1;
22    const unsigned MASK = 1 << SHIFT;
23
24    cout << setw( 10 ) << value << " = ";
25
26    // display bits
27    for ( unsigned i = 1; i <= SHIFT + 1; ++i )
28    {
29       cout << ( value & MASK ? '1' : '0' );
30       value <<= 1; // shift value left by 1
31
32       if ( i % 8 == 0 ) // output a space after 8 bits
33          cout << ' ';
34    } // end for
35
36    cout << endl;
37 } // end function displayBits
```

```
Enter an unsigned integer: 65000
     65000 = 00000000 00000000 11111101 11101000
```

```
Enter an unsigned integer: 29
        29 = 00000000 00000000 00000000 00011101
```

Fig. 21.6 | Printing an unsigned integer in bits. (Part 2 of 2.)

Function `displayBits` (lines 19–37) uses the bitwise AND operator to combine variable `value` with constant `MASK`. Often, the bitwise AND operator is used with an operand called a **mask**—an integer value with specific bits set to 1. Masks are used to *hide* some bits in a value while *selecting* other bits. In `displayBits`, line 22 assigns constant `MASK` the value `1 << SHIFT`. The value of constant `SHIFT` was calculated in line 21 with the expression

```
8 * sizeof( unsigned ) - 1
```

which multiplies the number of bytes an `unsigned` object requires in memory by `8` (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1. The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is

```
10000000 00000000 00000000 00000000
```

The left-shift operator shifts the value 1 from the low-order (rightmost) bit to the high-order (leftmost) bit in MASK, and fills in 0 bits from the right. Line 29 prints a 1 or a 0 for the current leftmost bit of variable value. Assume that variable value contains 65000 (00000000 00000000 11111101 11101000). When value and MASK are combined using &, all the bits except the high-order bit in variable value are "masked off" (hidden), because any bit "ANDed" with 0 yields 0. If the leftmost bit is 1, value & MASK evaluates to

```
00000000 00000000 11111101 11101000    (value)
10000000 00000000 00000000 00000000    (MASK)
-----------------------------------
00000000 00000000 00000000 00000000    (value & MASK)
```

which is interpreted as false, and 0 is printed. Then line 30 shifts variable value left by one bit with the expression value <<= 1 (i.e., value = value << 1). These steps are repeated for each bit variable value. Eventually, a bit with a value of 1 is shifted into the leftmost bit position, and the bit manipulation is as follows:

```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
-----------------------------------
10000000 00000000 00000000 00000000    (value & MASK)
```

Because both left bits are 1s, the expression's result is nonzero (true) and 1 is printed. Figure 21.7 summarizes the results of combining two bits with the bitwise AND operator.

Common Programming Error 21.2

Using the logical AND operator (&&) for the bitwise AND operator (&) and vice versa is a logic error.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 21.7 | Results of combining two bits with the bitwise AND operator (&).

The program of Fig. 21.8 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. Function displayBits (lines 53–71) prints the unsigned integer values.

```
1  // Fig. 21.8: fig21_08.cpp
2  // Bitwise AND, inclusive OR,
3  // exclusive OR and complement operators.
4  #include <iostream>
```

Fig. 21.8 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 1 of 3.)

```
#include <iomanip>
using namespace std;

void displayBits( unsigned ); // prototype

int main()
{
   unsigned number1;
   unsigned number2;
   unsigned mask;
   unsigned setBits;

   // demonstrate bitwise &
   number1 = 2179876355;
   mask = 1;
   cout << "The result of combining the following\n";
   displayBits( number1 );
   displayBits( mask );
   cout << "using the bitwise AND operator & is\n";
   displayBits( number1 & mask );

   // demonstrate bitwise |
   number1 = 15;
   setBits = 241;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( setBits );
   cout << "using the bitwise inclusive OR operator | is\n";
   displayBits( number1 | setBits );

   // demonstrate bitwise exclusive OR
   number1 = 139;
   number2 = 199;
   cout << "\nThe result of combining the following\n";
   displayBits( number1 );
   displayBits( number2 );
   cout << "using the bitwise exclusive OR operator ^ is\n";
   displayBits( number1 ^ number2 );

   // demonstrate bitwise complement
   number1 = 21845;
   cout << "\nThe one's complement of\n";
   displayBits( number1 );
   cout << "is" << endl;
   displayBits( ~number1 );
} // end main

// display bits of an unsigned integer value
void displayBits( unsigned value )
{
   const int SHIFT = 8 * sizeof( unsigned ) - 1;
   const unsigned MASK = 1 << SHIFT;

```

Fig. 21.8 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 2 of 3.)

```
58      cout << setw( 10 ) << value << " = ";
59
60      // display bits
61      for ( unsigned i = 1; i <= SHIFT + 1; ++i )
62      {
63         cout << ( value & MASK ? '1' : '0' );
64         value <<= 1; // shift value left by 1
65
66         if ( i % 8 == 0 ) // output a space after 8 bits
67            cout << ' ';
68      } // end for
69
70      cout << endl;
71   } // end function displayBits
```

```
The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Fig. 21.8 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 3 of 3.)

Bitwise AND Operator (&)

In Fig. 21.8, line 18 assigns `2179876355` (10000001 11101110 01000110 00000011) to variable `number1`, and line 19 assigns `1` (00000000 00000000 00000000 00000001) to variable `mask`. When `mask` and `number1` are combined using the bitwise AND operator (&) in the expression `number1 & mask` (line 24), the result is 00000000 00000000 00000000 00000001. All the bits except the low-order bit in variable `number1` are "masked off" (hidden) by "ANDing" with constant `MASK`.

Bitwise Inclusive OR Operator (|)

The bitwise inclusive OR operator is used to set specific bits to 1 in an operand. In Fig. 21.8, line 27 assigns `15` (00000000 00000000 00000000 00001111) to variable `number1`, and line 28 assigns `241` (00000000 00000000 00000000 11110001) to variable `setBits`. When `number1` and `setBits` are combined using the bitwise OR operator in the expression `number1 | set-`

`Bits` (line 33), the result is `255` (`00000000 00000000 00000000 11111111`). Figure 21.9 summarizes the results of combining two bits with the bitwise inclusive-OR operator.

Common Programming Error 21.3

Using the logical OR operator (`||`) for the bitwise OR operator (`|`) and vice versa is a logic error.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Fig. 21.9 | Combining two bits with the bitwise inclusive-OR operator (`|`).

Bitwise Exclusive OR (^)

The bitwise exclusive OR operator (`^`) sets each bit in the result to 1 if *exactly* one of the corresponding bits in its two operands is 1. In Fig. 21.8, lines 36–37 assign variables `number1` and `number2` the values `139` (`00000000 00000000 00000000 10001011`) and `199` (`00000000 00000000 00000000 11000111`), respectively. When these variables are combined with the exclusive OR operator in the expression `number1 ^ number2` (line 42), the result is `00000000 00000000 00000000 01001100`. Figure 21.10 summarizes the results of combining two bits with the bitwise exclusive OR operator.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 21.10 | Combining two bits with the bitwise exclusive OR operator (`^`).

Bitwise Complement (~)

The bitwise complement operator (`~`) sets all `1` bits in its operand to `0` in the result and sets all `0` bits to `1` in the result—otherwise referred to as "taking the *one's complement* of the value." In Fig. 21.8, line 45 assigns variable `number1` the value `21845` (`00000000 00000000 01010101 01010101`). When the expression `~number1` evaluates, the result is (`11111111 11111111 10101010 10101010`).

Figure 21.11 demonstrates the left-shift operator (`<<`) and the right-shift operator (`>>`). Function `displayBits` (lines 27–45) prints the `unsigned` integer values.

```cpp
// Fig. 21.11: fig21_11.cpp
// Using the bitwise shift operators.
#include <iostream>
#include <iomanip>
using namespace std;

void displayBits( unsigned ); // prototype

int main()
{
   unsigned number1 = 960;

   // demonstrate bitwise left shift
   cout << "The result of left shifting\n";
   displayBits( number1 );
   cout << "8 bit positions using the left-shift operator is\n";
   displayBits( number1 << 8 );

   // demonstrate bitwise right shift
   cout << "\nThe result of right shifting\n";
   displayBits( number1 );
   cout << "8 bit positions using the right-shift operator is\n";
   displayBits( number1 >> 8 );
} // end main

// display bits of an unsigned integer value
void displayBits( unsigned value )
{
   const int SHIFT = 8 * sizeof( unsigned ) - 1;
   const unsigned MASK = 1 << SHIFT;

   cout << setw( 10 ) << value << " = ";

   // display bits
   for ( unsigned i = 1; i <= SHIFT + 1; ++i )
   {
      cout << ( value & MASK ? '1' : '0' );
      value <<= 1; // shift value left by 1

      if ( i % 8 == 0 ) // output a space after 8 bits
         cout << ' ';
   } // end for

   cout << endl;
} // end function displayBits
```

```
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left-shift operator is
    245760 = 00000000 00000011 11000000 00000000
```

Fig. 21.11 | Bitwise shift operators. (Part 1 of 2.)

```
The result of right shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the right-shift operator is
         3 = 00000000 00000000 00000000 00000011
```

Fig. 21.11 | Bitwise shift operators. (Part 2 of 2.)

Left-Shift Operator

The left-shift operator (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with `0`s; bits shifted off the left are lost. In Fig. 21.11, line 11 assigns variable `number1` the value `960` (`00000000 00000000 00000011 11000000`). The result of left-shifting variable `number1` eight bits in the expression `number1 << 8` (line 17) is `245760` (`00000000 00000011 11000000 00000000`).

Right-Shift Operator

The right-shift operator (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an `unsigned` integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost. In the program of Fig. 21.11, the result of right-shifting `number1` in the expression `number1 >> 8` (line 23) is `3` (`00000000 00000000 00000000 00000011`).

Common Programming Error 21.4

The result of shifting a value is undefined *if the right operand is negative or if the right operand is greater than or equal to the number of bits in which the left operand is stored.*

Portability Tip 21.4

The result of right-shifting a signed value is machine dependent. *Some machines fill with zeros and others use the sign bit.*

Bitwise Assignment Operators

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These **bitwise assignment operators** are shown in Fig. 21.12; they're used in a similar manner to the arithmetic assignment operators introduced in Chapter 2.

Bitwise assignment operators	
`&=`	Bitwise AND assignment operator.
`|=`	Bitwise inclusive OR assignment operator.
`^=`	Bitwise exclusive OR assignment operator.
`<<=`	Left-shift assignment operator.
`>>=`	Right-shift with sign extension assignment operator.

Fig. 21.12 | Bitwise assignment operators.

Figure 21.13 shows the precedence and associativity of the operators introduced up to this point in the text. They're shown top to bottom in decreasing order of precedence.

<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>:: (unary; right to left) :: (binary; left to right)
() (grouping parentheses)</td><td>[See parentheses caution in Fig. 2.10]</td><td>highest</td></tr>
<tr><td>() [] . -> ++ -- static_cast< type >()</td><td>left to right</td><td>postfix</td></tr>
<tr><td>++ -- + - ! delete sizeof
* ~ & new</td><td>right to left</td><td>prefix</td></tr>
<tr><td>* / %</td><td>left to right</td><td>multiplicative</td></tr>
<tr><td>+ -</td><td>left to right</td><td>additive</td></tr>
<tr><td><< >></td><td>left to right</td><td>shifting</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>relational</td></tr>
<tr><td>== !=</td><td>left to right</td><td>equality</td></tr>
<tr><td>&</td><td>left to right</td><td>bitwise AND</td></tr>
<tr><td>^</td><td>left to right</td><td>bitwise XOR</td></tr>
<tr><td>|</td><td>left to right</td><td>bitwise OR</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= %= &= |= ^= <<= >>=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

Fig. 21.13 | Operator precedence and associativity.

21.6 Bit Fields

C++ provides the ability to specify the number of bits in which an integral type or `enum` type member of a class or a structure is stored. Such a member is referred to as a **bit field.** Bit fields enable *better memory utilization* by storing data in the minimum number of bits required. Bit field members *must* be declared as an integral or `enum` type.

Performance Tip 21.1

Bit fields help conserve storage.

Consider the following structure definition:

```
struct BitCard
{
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
}; // end struct BitCard
```

The definition contains three `unsigned` bit fields—`face`, `suit` and `color`—used to represent a card from a deck of 52 cards. A bit field is declared by following an integral type or `enum` type member with a colon (`:`) and an integer constant representing the **width of the bit field** (i.e., the number of bits in which the member is stored). The width must be an integer constant.

The preceding structure definition indicates that member `face` is stored in four bits, member `suit` in 2 bits and member `color` in one bit. The number of bits is based on the desired range of values for each structure member. Member `face` stores values between `0` (Ace) and `12` (King)—four bits can store a value between 0 and 15. Member `suit` stores values between `0` and `3` (`0` = Diamonds, `1` = Hearts, `2` = Clubs, `3` = Spades)—two bits can store a value between `0` and `3`. Finally, member `color` stores either `0` (Red) or `1` (Black)—one bit can store either `0` or `1`.

The program in Figs. 21.14–21.16 creates `vector deck` containing `BitCard` structures (line 27 of Fig. 21.14). The constructor inserts the 52 cards in the `deck vector`, and function `deal` prints the 52 cards. Notice that bit fields are accessed exactly as any other structure member is (lines 15–17 and 26–31 of Fig. 21.15). The member `color` is included as a means of indicating the card color.

```
1  // Fig. 21.14: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4  #include <vector>
5  using namespace std;
6
7  // BitCard structure definition with bit fields
8  struct BitCard
9  {
10    unsigned face : 4; // 4 bits; 0-15
11    unsigned suit : 2; // 2 bits; 0-3
12    unsigned color : 1; // 1 bit; 0-1
13 }; // end struct BitCard
14
15 // DeckOfCards class definition
16 class DeckOfCards
17 {
18 public:
19    static const int faces = 13;
20    static const int colors = 2; // black and red
21    static const int numberOfCards = 52;
22
23    DeckOfCards(); // constructor initializes deck
24    void deal(); // deals cards in deck
25
26 private:
27    vector< BitCard > deck; // represents deck of cards
28 }; // end class DeckOfCards
```

Fig. 21.14 | Header for class `DeckOfCards`.

```
1  // Fig. 21.15: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  #include <iomanip>
```

Fig. 21.15 | Class file for `DeckOfCards`. (Part 1 of 2.)

```
#include "DeckOfCards.h" // DeckOfCards class definition
using namespace std;

// no-argument DeckOfCards constructor intializes deck
DeckOfCards::DeckOfCards()

{
   for ( int i = 0; i < numberOfCards; ++i )
   {
      deck[ i ].face = i % faces; // faces in order
      deck[ i ].suit = i / faces; // suits in order
      deck[ i ].color = i / ( faces * colors ); // colors in order
   } // end for
} // end no-argument DeckOfCards constructor

// deal cards in deck
void DeckOfCards::deal()
{
   for ( int k1 = 0, k2 = k1 + numberOfCards / 2;
      k1 < numberOfCards / 2 - 1; ++k1, ++k2 )
      cout << "Card:" << setw( 3 ) << deck[ k1 ].face
         << "  Suit:" << setw( 2 ) << deck[ k1 ].suit
         << "  Color:" << setw( 2 ) << deck[ k1 ].color
         << "   " << "Card:" << setw( 3 ) << deck[ k2 ].face
         << "  Suit:" << setw( 2 ) << deck[ k2 ].suit
         << "  Color:" << setw( 2 ) << deck[ k2 ].color << endl;
} // end function deal
```

Fig. 21.15 | Class file for DeckOfCards. (Part 2 of 2.)

```
// Fig. 21.16: fig21_16.cpp
// Card shuffling and dealing program.
#include "DeckOfCards.h" // DeckOfCards class definition

int main()
{
   DeckOfCards deckOfCards; // create DeckOfCards object
   deckOfCards.deal(); // deal the cards in the deck
} // end main
```

```
Card:  0  Suit: 0  Color: 0   Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0   Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0   Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0   Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0   Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0   Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0   Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0   Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0   Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0   Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0   Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0   Card: 11  Suit: 2  Color: 1
```

Fig. 21.16 | Bit fields used to store a deck of cards. (Part 1 of 2.)

```
Card: 12  Suit: 0  Color: 0   Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0   Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0   Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0   Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0   Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0   Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0   Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0   Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0   Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0   Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0   Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0   Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0   Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0   Card: 12  Suit: 3  Color: 1
```

Fig. 21.16 | Bit fields used to store a deck of cards. (Part 2 of 2.)

It's possible to specify an **unnamed bit field**, in which case the field is used as **padding** in the structure. For example, the structure definition uses an unnamed three-bit field as padding—nothing can be stored in those three bits. Member `b` is stored in another storage unit.

```
struct Example
{
   unsigned a : 13;
   unsigned   : 3; // align to next storage-unit boundary
   unsigned b : 4;
}; // end struct Example
```

An **unnamed bit field with a zero width** is used to align the next bit field on a new storage-unit boundary. For example, the structure definition

```
struct Example
{
   unsigned a : 13;
   unsigned   : 0; // align to next storage-unit boundary
   unsigned b : 4;
}; // end struct Example
```

uses an unnamed `0`-bit field to *skip* the remaining bits (as many as there are) of the storage unit in which `a` is stored and align `b` on the *next storage-unit boundary*.

Portability Tip 21.5

Bit-field manipulations are machine dependent. For example, some computers allow bit fields to cross word boundaries, whereas others do not.

Common Programming Error 21.5

Attempting to access individual bits of a bit field with subscripting as if they were elements of an array is a compilation error. Bit fields are not *"arrays of bits."*

Common Programming Error 21.6

Attempting to take the address of a bit field (the `&` *operator may not be used with bit fields because a pointer can designate only a particular byte in memory and bit fields can start in the middle of a byte) is a compilation error.*

Performance Tip 21.2

Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine-language operations to access only portions of an addressable storage unit. This is one of many examples of the space–time trade-offs that occur in computer science.

21.7 Character-Handling Library

Most data is entered into computers as characters—including letters, digits and various special symbols. In this section, we discuss C++'s capabilities for examining and manipulating individual characters. In the remainder of the chapter, we continue the discussion of character-string manipulation that we began in Chapter 8.

The character-handling library includes several functions that perform useful tests and manipulations of character data. Each function receives a character—represented as an `int`—or `EOF` as an argument. *Characters are often manipulated as integers.* Remember that `EOF` normally has the value `-1` and that some hardware architectures do not allow negative values to be stored in `char` variables. Therefore, the character-handling functions manipulate characters as integers. Figure 21.17 summarizes the functions of the character-handling library. When using functions from the character-handling library, include the `<cctype>` header.

Prototype	Description
`int isdigit( int c )`	Returns `1` if `c` is a digit and `0` otherwise.
`int isalpha( int c )`	Returns `1` if `c` is a letter and `0` otherwise.
`int isalnum( int c )`	Returns `1` if `c` is a digit or a letter and `0` otherwise.
`int isxdigit( int c )`	Returns `1` if `c` is a hexadecimal digit character and `0` otherwise. (See Appendix D, Number Systems, for a detailed explanation of binary, octal, decimal and hexadecimal numbers.)
`int islower( int c )`	Returns `1` if `c` is a lowercase letter and `0` otherwise.
`int isupper( int c )`	Returns `1` if `c` is an uppercase letter; `0` otherwise.
`int tolower( int c )`	If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c )`	If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c )`	Returns `1` if `c` is a white-space character—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), or vertical tab (`'\v'`)—and `0` otherwise.
`int iscntrl( int c )`	Returns `1` if `c` is a control character, such as newline (`'\n'`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`), vertical tab (`'\v'`), alert (`'\a'`), or backspace (`'\b'`)—and `0` otherwise.
`int ispunct( int c )`	Returns `1` if `c` is a printing character other than a space, a digit, or a letter and `0` otherwise.

Fig. 21.17 | Character-handling library functions. (Part 1 of 2.)

Prototype	Description
`int isprint( int c )`	Returns `1` if `c` is a printing character including space (`' '`) and `0` otherwise.
`int isgraph( int c )`	Returns `1` if `c` is a printing character other than space (`' '`) and `0` otherwise.

Fig. 21.17 | Character-handling library functions. (Part 2 of 2.)

Figure 21.18 demonstrates functions **`isdigit`**, **`isalpha`**, **`isalnum`** and **`isxdigit`**. Function `isdigit` determines whether its argument is a digit (`0`–`9`). Function `isalpha` determines whether its argument is an uppercase letter (`A`-`Z`) or a lowercase letter (`a`–`z`). Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function `isxdigit` determines whether its argument is a hexadecimal digit (`A`–`F`, `a`–`f`, `0`–`9`).

```
// Fig. 21.18: fig21_18.cpp
// Character-handling functions isdigit, isalpha, isalnum and isxdigit.
#include <iostream>
#include <cctype> // character-handling function prototypes
using namespace std;

int main()
{
   cout << "According to isdigit:\n"
      << ( isdigit( '8' ) ? "8 is a" : "8 is not a" ) << " digit\n"
      << ( isdigit( '#' ) ? "# is a" : "# is not a" ) << " digit\n";

   cout << "\nAccording to isalpha:\n"
      << ( isalpha( 'A' ) ? "A is a" : "A is not a" ) << " letter\n"
      << ( isalpha( 'b' ) ? "b is a" : "b is not a" ) << " letter\n"
      << ( isalpha( '&' ) ? "& is a" : "& is not a" ) << " letter\n"
      << ( isalpha( '4' ) ? "4 is a" : "4 is not a" ) << " letter\n";

   cout << "\nAccording to isalnum:\n"
      << ( isalnum( 'A' ) ? "A is a" : "A is not a" )
      << " digit or a letter\n"
      << ( isalnum( '8' ) ? "8 is a" : "8 is not a" )
      << " digit or a letter\n"
      << ( isalnum( '#' ) ? "# is a" : "# is not a" )
      << " digit or a letter\n";

   cout << "\nAccording to isxdigit:\n"
      << ( isxdigit( 'F' ) ? "F is a" : "F is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( 'J' ) ? "J is a" : "J is not a" )
      << " hexadecimal digit\n"
      << ( isxdigit( '7' ) ? "7 is a" : "7 is not a" )
```

Fig. 21.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 2.)

```
33          << " hexadecimal digit\n"
34          << ( isxdigit( '$' ) ? "$ is a" : "$ is not a" )
35          << " hexadecimal digit\n"
36          << ( isxdigit( 'f' ) ? "f is a" : "f is not a" )
37          << " hexadecimal digit" << endl;
38    } // end main
```

```
According to isdigit:
8 is a digit
# is not a digi

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 21.18 | Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 2 of 2.)

Figure 21.18 uses the conditional operator (`?:`) with each function to determine whether the string `" is a "` or the string `" is not a "` should be printed in the output for each character tested. For example, line 10 indicates that if `'8'` is a digit—i.e., if `isdigit` returns a true (nonzero) value—the string `"8 is a "` is printed. If `'8'` is not a digit (i.e., if `isdigit` returns `0`), the string `"8 is not a "` is printed.

Figure 21.19 demonstrates functions **`islower`**, **`isupper`**, **`tolower`** and **`toupper`**. Function `islower` determines whether its argument is a *lowercase letter* (`a`–`z`). Function `isupper` determines whether its argument is an *uppercase letter* (`A`–`Z`). Function `tolower` converts an uppercase letter to lowercase and returns the lowercase letter—if the argument is not an uppercase letter, `tolower` returns the argument value unchanged. Function `toupper` converts a lowercase letter to uppercase and returns the uppercase letter—if the argument is not a lowercase letter, `toupper` returns the argument value *unchanged*.

```
// Fig. 21.19: fig21_19.cpp
// Character-handling functions islower, isupper, tolower and toupper.
#include <iostream>
#include <cctype> // character-handling function prototypes
using namespace std;
```

Fig. 21.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 1 of 2.)

```

int main()
{
   cout << "According to islower:\n"
      << ( islower( 'p' ) ? "p is a" : "p is not a" )
      << " lowercase letter\n"
      << ( islower( 'P' ) ? "P is a" : "P is not a" )
      << " lowercase letter\n"
      << ( islower( '5' ) ? "5 is a" : "5 is not a" )
      << " lowercase letter\n"
      << ( islower( '!' ) ? "! is a" : "! is not a" )
      << " lowercase letter\n";

   cout << "\nAccording to isupper:\n"
      << ( isupper( 'D' ) ? "D is an" : "D is not an" )
      << " uppercase letter\n"
      << ( isupper( 'd' ) ? "d is an" : "d is not an" )
      << " uppercase letter\n"
      << ( isupper( '8' ) ? "8 is an" : "8 is not an" )
      << " uppercase letter\n"
      << ( isupper( '$' ) ? "$ is an" : "$ is not an" )
      << " uppercase letter\n";

   cout << "\nu converted to uppercase is "
      << static_cast< char >( toupper( 'u' ) )
      << "\n7 converted to uppercase is "
      << static_cast< char >( toupper( '7' ) )
      << "\n$ converted to uppercase is "
      << static_cast< char >( toupper( '$' ) )
      << "\nL converted to lowercase is "
      << static_cast< char >( tolower( 'L' ) ) << endl;
} // end main
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Fig. 21.19 | Character-handling functions `islower`, `isupper`, `tolower` and `toupper`. (Part 2 of 2.)

Figure 21.20 demonstrates functions **`isspace`**, **`iscntrl`**, **`ispunct`**, **`isprint`** and **`isgraph`**. Function `isspace` determines whether its argument is a white-space character, such as space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v'). Function iscntrl determines whether its argument is a control character such as horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r') or newline ('\n'). Function ispunct determines whether its argument is a printing character other than a space, digit or letter, such as $, #, (,), [,], {, }, ;, : or %. Function isprint determines whether its argument is a character that can be displayed on the screen (including the space character). Function isgraph tests for the same characters as isprint, but the space character is *not* included.

```cpp
// Fig. 21.20: fig21_20.cpp
// Using functions isspace, iscntrl, ispunct, isprint and isgraph.
#include <iostream>
#include <cctype> // character-handling function prototypes
using namespace std;

int main()
{
   cout << "According to isspace:\nNewline "
      << ( isspace( '\n' ) ? "is a" : "is not a" )
      << " whitespace character\nHorizontal tab "
      << ( isspace( '\t' ) ? "is a" : "is not a" )
      << " whitespace character\n"
      << ( isspace( '%' ) ? "% is a" : "% is not a" )
      << " whitespace character\n";

   cout << "\nAccording to iscntrl:\nNewline "
      << ( iscntrl( '\n' ) ? "is a" : "is not a" )
      << " control character\n"
      << ( iscntrl( '$' ) ? "$ is a" : "$ is not a" )
      << " control character\n";

   cout << "\nAccording to ispunct:\n"
      << ( ispunct( ';' ) ? "; is a" : "; is not a" )
      << " punctuation character\n"
      << ( ispunct( 'Y' ) ? "Y is a" : "Y is not a" )
      << " punctuation character\n"
      << ( ispunct( '#' ) ? "# is a" : "# is not a" )
      << " punctuation character\n";

   cout << "\nAccording to isprint:\n"
      << ( isprint( '$' ) ? "$ is a" : "$ is not a" )
      << " printing character\nAlert "
      << ( isprint( '\a' ) ? "is a" : "is not a" )
      << " printing character\nSpace "
      << ( isprint( ' ' ) ? "is a" : "is not a" )
      << " printing character\n";

   cout << "\nAccording to isgraph:\n"
      << ( isgraph( 'Q' ) ? "Q is a" : "Q is not a" )
      << " printing character other than a space\nSpace "
```

Fig. 21.20 | Character-handling functions isspace, iscntrl, ispunct, isprint and isgraph. (Part 1 of 2.)

```
        << ( isgraph( ' ' ) ? "is a" : "is not a" )
        << " printing character other than a space" << endl;
} // end main
```

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character
Space is a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Fig. 21.20 | Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. (Part 2 of 2.)

21.8 Pointer-Based String Manipulation Functions

The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces such as the separate words in a sentence) and determining the length of strings. This section presents some common string-manipulation functions of the string-handling library (from the C++ standard library). The functions are summarized in Fig. 21.21; then each is used in a live-code example. The prototypes for these functions are located in header `<cstring>`.

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 );`	
	Copies the string `s2` into the character array `s1`. The value of `s1` is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	
	Copies at most `n` characters of the string `s2` into the character array `s1`. The value of `s1` is returned.

Fig. 21.21 | String-manipulation functions of the string-handling library. (Part 1 of 2.)

Function prototype	Function description
`char *strcat( char *s1, const char *s2 );`	Appends the string `s2` to `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most `n` characters of string `s2` to string `s1`. The first character of `s2` overwrites the terminating null character of `s1`. The value of `s1` is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string `s1` with the string `s2`. The function returns a value of zero, less than zero or greater than zero if `s1` is equal to, less than or greater than `s2`, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to `n` characters of the string `s1` with the string `s2`. The function returns zero, less than zero or greater than zero if the `n`-character portion of `s1` is equal to, less than or greater than the corresponding `n`-character portion of `s2`, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to `strtok` breaks string `s1` into "tokens"—logical pieces such as words in a line of text. The string is broken up based on the characters contained in string `s2`. For instance, if we were to break the string `"this:is:a:string"` into tokens based on the character `':'`, the resulting tokens would be `"this"`, `"is"`, `"a"` and `"string"`. Function `strtok` returns only one token at a time—the first call contains `s1` as the first argument, and subsequent calls to continue tokenizing the same string contain `NULL` as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, `NULL` is returned.
`size_t strlen( const char *s );`	Determines the length of string `s`. The number of characters preceding the terminating null character is returned.

Fig. 21.21 | String-manipulation functions of the string-handling library. (Part 2 of 2.)

Several functions in Fig. 21.21 contain parameters with data type `size_t`. This type is defined in the header `<cstring>` to be an unsigned integral type such as `unsigned int` or `unsigned long`.

Common Programming Error 21.7

Forgetting to include the `<cstring>` header when using functions from the string-handling library causes compilation errors.

Copying Strings with `strcpy` and `strncpy`

Function **`strcpy`** copies its second argument—a string—into its first argument—a character array that must be large enough to store the string and its terminating null character, (which is also copied). Function **strncpy** is much like strcpy, except that strncpy specifies the number of characters to be copied from the string into the array. Function strncpy does not necessarily copy the terminating null character of its second argument—a terminating null character is written only if the number of characters to be copied is at least one more than the length of the string. For example, if "test" is the second argument, a terminating null character is written only if the third argument to strncpy is at least 5 (four characters in "test" plus one terminating null character). If the third argument is larger than 5, null characters are appended to the array until the total number of characters specified by the third argument is written.

Common Programming Error 21.8

*When using strncpy, the terminating null character of the second argument (a char * string) will not be copied if the number of characters specified by strncpy's third argument is not greater than the second argument's length. In that case, a fatal error may occur if you do not manually terminate the resulting char * string with a null character.*

Figure 21.22 uses strcpy (line 13) to copy the entire string in array x into array y and uses strncpy (line 19) to copy the first 14 characters of array x into array z. Line 20 appends a null character ('\0') to array z, because the call to strncpy in the program does not write a terminating null character. (The third argument is less than the string length of the second argument plus one.)

```
1  // Fig. 21.22: fig21_22.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  #include <cstring> // prototypes for strcpy and strncpy
5  using namespace std;
6
7  int main()
8  {
9     char x[] = "Happy Birthday to You"; // string length 21
10    char y[ 25 ];
11    char z[ 15 ];
12
13    strcpy( y, x ); // copy contents of x into y
14
15    cout << "The string in array x is: " << x
16       << "\nThe string in array y is: " << y << '\n';
17
18    // copy first 14 characters of x into z
19    strncpy( z, x, 14 ); // does not copy null character
20    z[ 14 ] = '\0'; // append '\0' to z's contents
21
22    cout << "The string in array z is: " << z << endl;
23 } // end main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 21.22 | strcpy and strncpy.

Concatenating Strings with `strcat` *and* `strncat`

Function **strcat** appends its second argument (a string) to its first argument (a character array containing a string). The first character of the second argument replaces the null character ('\0') that terminates the string in the first argument. You must ensure that the array used to store the first string is *large enough* to store the combination of the first string, the second string and the terminating null character (copied from the second string). Function **strncat** appends a specified number of characters from the second string to the first string and appends a terminating null character to the result. The program of Fig. 21.23 demonstrates function strcat (lines 15 and 25) and function strncat (line 20).

```
1  // Fig. 21.23: fig23_23.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  #include <cstring> // prototypes for strcat and strncat
5  using namespace std;
6
7  int main()
8  {
9     char s1[ 20 ] = "Happy "; // length 6
10    char s2[] = "New Year "; // length 9
11    char s3[ 40 ] = "";
12
13    cout << "s1 = " << s1 << "\ns2 = " << s2;
14
15    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
16
17    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
18
19    // concatenate first 6 characters of s1 to s3
20    strncat( s3, s1, 6 ); // places '\0' after last character
21
22    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
23       << "\ns3 = " << s3;
24
25    strcat( s3, s1 ); // concatenate s1 to s3
26    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
27       << "\ns3 = " << s3 << endl;
28 } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy
```

Fig. 21.23 | strcat and strncat. (Part 1 of 2.)

```
After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Fig. 21.23 | strcat and strncat. (Part 2 of 2.)

Comparing Strings with `strcmp` *and* `strncmp`

Figure 21.24 compares three strings using `strcmp` (lines 15–17) and `strncmp` (lines 20–22). Function `strcmp` compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string. Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares up to a specified number of characters. Function `strncmp` stops comparing characters if it reaches the null character in one of its string arguments. The program prints the integer value returned by each function call.

Common Programming Error 21.9

Assuming that `strcmp` and `strncmp` return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the `strcmp` or `strncmp` function should be compared with zero to determine whether the strings are equal.

```
1  // Fig. 21.24: fig21_24.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstring> // prototypes for strcmp and strncmp
6  using namespace std;
7
8  int main()
9  {
10    char *s1 = "Happy New Year";
11    char *s2 = "Happy New Year";
12    char *s3 = "Happy Holidays";
13
14    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
15       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
16       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
17       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
18
19    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
20       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
21       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
22       << strncmp( s3, s1, 7 ) << endl;
23 } // end main
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays
```

Fig. 21.24 | strcmp and strncmp. (Part 1 of 2.)

```
strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Fig. 21.24 | `strcmp` and `strncmp`. (Part 2 of 2.)

To understand what it means for one string to be "greater than" or "less than" another, consider the process of alphabetizing last names. You'd, no doubt, place "Jones" before "Smith," because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet. But the alphabet is more than just a list of 26 letters—it's an *ordered* list of characters. Each letter occurs in a specific position within the list. "Z" is more than just a letter of the alphabet; "Z" is specifically the 26th letter of the alphabet.

How does the computer know that one letter comes before another? All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

[*Note:* With some compilers, functions `strcmp` and `strncmp` always return `-1`, `0` or `1`, as in the sample output of Fig. 21.24. With other compilers, these functions return `0` or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s3` are compared, the first characters that differ between them are the first character of the second word in each string—`N` (numeric code 78) in `s1` and `H` (numeric code 72) in `s3`, respectively. In this case, the return value will be `6` (or `-6` if `s3` is compared to `s1`).]

Tokenizing a String with `strtok`

Function **`strtok`** breaks a string into a series of **tokens**. A token is a sequence of characters separated by **delimiting characters** (usually spaces or punctuation marks). For example, in a line of text, each word can be considered a token, and the spaces separating the words can be considered delimiters.

Multiple calls to `strtok` are required to break a string into tokens (assuming that the string contains more than one token). The first call to `strtok` contains two arguments, a string to be tokenized and a string containing characters that separate the tokens (i.e., delimiters). Line 16 in Fig. 21.25 assigns to `tokenPtr` a pointer to the first token in `sentence`. The second argument, `" "`, indicates that tokens in `sentence` are separated by spaces. Function `strtok` searches for the first character in `sentence` that's not a delimiting character (space). This begins the first token. The function then finds the next delimiting character in the string and replaces it with a null (`'\0'`) character. This terminates the current token. Function `strtok` saves (in a `static` variable) a pointer to the next character following the token in `sentence` and returns a pointer to the current token.

Subsequent calls to `strtok` to continue tokenizing `sentence` contain `NULL` as the first argument (line 22). The `NULL` argument indicates that the call to `strtok` should continue tokenizing from the location in `sentence` saved by the last call to `strtok`. Function `strtok` maintains this saved information in a manner that's not visible to you. If no tokens remain when `strtok` is called, `strtok` returns `NULL`. The program of Fig. 21.25 uses `strtok` to

```
1  // Fig. 21.25: fig21_25.cpp
2  // Using strtok to tokenize a string.
3  #include <iostream>
4  #include <cstring> // prototype for strtok
5  using namespace std;
6
7  int main()
8  {
9     char sentence[] = "This is a sentence with 7 tokens";
10    char *tokenPtr;
11
12    cout << "The string to be tokenized is:\n" << sentence
13       << "\n\nThe tokens are:\n\n";
14
15    // begin tokenization of sentence
16    tokenPtr = strtok( sentence, " " );
17
18    // continue tokenizing sentence until tokenPtr becomes NULL
19    while ( tokenPtr != NULL )
20    {
21       cout << tokenPtr << '\n';
22       tokenPtr = strtok( NULL, " " ); // get next token
23    } // end while
24
25    cout << "\nAfter strtok, sentence = " << sentence << endl;
26 } // end main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

Fig. 21.25 | Using `strtok` to tokenize a string.

tokenize the string `"This is a sentence with 7 tokens"`. The program prints each token on a separate line. Line 25 outputs `sentence` after tokenization. Note that *`strtok` modifies the input string*; therefore, a copy of the string should be made if the program requires the original after the calls to `strtok`. When `sentence` is output after tokenization, only the word “`This`” prints, because `strtok` replaced each blank in `sentence` with a null character (`'\0'`) during the tokenization process.

Common Programming Error 21.10

Not realizing that `strtok` modifies *the string being tokenized, then attempting to use that string as if it were the original unmodified string is a logic error.*

Determining String Lengths

Function **strlen** takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length. The length is also the index of the null character. The program of Fig. 21.26 demonstrates function strlen.

```
// Fig. 21.26: fig21_26.cpp
// Using strlen.
#include <iostream>
#include <cstring> // prototype for strlen
using namespace std;

int main()
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "four";
   char *string3 = "Boston";

   cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
      << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
      << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
      << endl;
} // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fig. 21.26 | strlen returns the length of a char * string.

21.9 Pointer-Based String-Conversion Functions

In Section 21.8, we discussed several of C++'s most popular pointer-based string-manipulation functions. In the next several sections, we cover the remaining functions, including functions for converting strings to numeric values, functions for searching strings and functions for manipulating, comparing and searching blocks of memory.

This section presents the pointer-based **string-conversion functions** from the **general-utilities library <cstdlib>**. These functions convert pointer-based strings of characters to integer and floating-point values. In new code, C++ programmers typically use the string stream processing capabilities (Chapter 18) to perform such conversions. Figure 21.27 summarizes the pointer-based string-conversion functions. When using functions from the general-utilities library, include the <cstdlib> header.

Prototype	Description
double atof(const char *nPtr)	Converts the string nPtr to double. If the string cannot be converted, 0 is returned.

Fig. 21.27 | Pointer-based string-conversion functions of the general-utilities library. (Part 1 of 2.)

Prototype	Description
`int atoi( const char *nPtr )`	Converts the string `nPtr` to `int`. If the string cannot be converted, `0` is returned.
`long atol( const char *nPtr )`	Converts the string `nPtr` to `long int`. If the string cannot be converted, `0` is returned.
`double strtod( const char *nPtr, char **endPtr )`	Converts the string `nPtr` to `double`. `endPtr` is the address of a pointer to the rest of the string after the `double`. If the string cannot be converted, `0` is returned.
`long strtol( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `long`. `endPtr` is the address of a pointer to the rest of the string after the `long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converts the string `nPtr` to `unsigned long`. `endPtr` is the address of a pointer to the rest of the string after the `unsigned long`. If the string cannot be converted, `0` is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.

Fig. 21.27 | Pointer-based string-conversion functions of the general-utilities library. (Part 2 of 2.)

Function **atof** (Fig. 21.28, line 9) converts its argument—a string that represents a floating-point number—to a `double` value. The function returns the `double` value. If the string cannot be converted—for example, if the first character of the string is not a digit—function `atof` returns zero.

```
1  // Fig. 21.28: fig21_28.cpp
2  // Using atof.
3  #include <iostream>
4  #include <cstdlib> // atof prototype
5  using namespace std;
6
7  int main()
8  {
9     double d = atof( "99.0" ); // convert string to double
10
11    cout << "The string \"99.0\" converted to double is " << d
12       << "\nThe converted value divided by 2 is " << d / 2.0 << endl;
13 } // end main
```

Fig. 21.28 | String-conversion function `atof`. (Part 1 of 2.)

```
The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5
```

Fig. 21.28 | String-conversion function atof. (Part 2 of 2.)

Function **atoi** (Fig. 21.29, line 9) converts its argument—a string of digits that represents an integer—to an int value. The function returns the int value. If the string cannot be converted, function atoi returns zero.

```
1  // Fig. 21.29: Fig21_29.cpp
2  // Using atoi.
3  #include <iostream>
4  #include <cstdlib> // atoi prototype
5  using namespace std;
6
7  int main()
8  {
9     int i = atoi( "2593" ); // convert string to int
10
11    cout << "The string \"2593\" converted to int is " << i
12       << "\nThe converted value minus 593 is " << i - 593 << endl;
13 } // end main
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Fig. 21.29 | String-conversion function atoi.

Function **atol** (Fig. 21.30, line 9) converts its argument—a string of digits representing a long integer—to a long value. The function returns the long value. If the string cannot be converted, function atol returns zero. If int and long are both stored in four bytes, function atoi and function atol work identically.

```
1  // Fig. 21.30: fig21_30.cpp
2  // Using atol.
3  #include <iostream>
4  #include <cstdlib> // atol prototype
5  using namespace std;
6
7  int main()
8  {
9     long x = atol( "1000000" ); // convert string to long
10
11    cout << "The string \"1000000\" converted to long is " << x
12       << "\nThe converted value divided by 2 is " << x / 2 << endl;
13 } // end main
```

Fig. 21.30 | String-conversion function atol. (Part 1 of 2.)

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Fig. 21.30 | String-conversion function `atol`. (Part 2 of 2.)

Function **`strtod`** (Fig. 21.31) converts a sequence of characters representing a floating-point value to `double`. Function `strtod` receives two arguments—a string (`char *`) and the address of a `char *` pointer (i.e., a `char **`). The string contains the character sequence to be converted to `double`. The second argument enables `strtod` to modify a `char *` pointer in the calling function, such that the pointer points to the location of the first character after the converted portion of the string. Line 13 indicates that `d` is assigned the `double` value converted from `string` and that `stringPtr` is assigned the location of the first character after the converted value (`51.2`) in `string`.

```
1  // Fig. 21.31: fig21_31.cpp
2  // Using strtod.
3  #include <iostream>
4  #include <cstdlib> // strtod prototype
5  using namespace std;
6
7  int main()
8  {
9     double d;
10    const char *string1 = "51.2% are admitted";
11    char *stringPtr;
12
13    d = strtod( string1, &stringPtr ); // convert characters to double
14
15    cout << "The string \"" << string1
16       << "\" is converted to the\ndouble value " << d
17       << " and the string \"" << stringPtr << "\"" << endl;
18 } // end main
```

```
The string "51.2% are admitted" is converted to the
double value 51.2 and the string "% are admitted"
```

Fig. 21.31 | String-conversion function `strtod`.

Function **`strtol`** (Fig. 21.32) converts to `long` a sequence of characters representing an integer. The function receives a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to convert. The second argument is assigned the location of the first character after the converted portion of the string. The integer specifies the *base* of the value being converted. Line 13 indicates that `x` is assigned the `long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (`-1234567`) in `string1`. Using a null pointer for the second argument causes the remainder of the string to be ignored. The third argument, `0`, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16). This is determined by the initial characters in the string—`0` indicates an octal number, `0x` indicates hexadecimal and a number from `1` to `9` indicates decimal.

```
// Fig. 21.32: fig21_32.cpp
// Using strtol.
#include <iostream>
#include <cstdlib> // strtol prototype
using namespace std;

int main()
{
   long x;
   const char *string1 = "-1234567abc";
   char *remainderPtr;

   x = strtol( string1, &remainderPtr, 0 ); // convert characters to long

   cout << "The original string is \"" << string1
      << "\"\nThe converted value is " << x
      << "\nThe remainder of the original string is \"" << remainderPtr
      << "\"\nThe converted value plus 567 is " << x + 567 << endl;
} // end main
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Fig. 21.32 | String-conversion function `strtol`.

In a call to function `strtol`, the base can be specified as zero or as any value between 2 and 36. (See Appendix D for a detailed explanation of the octal, decimal, hexadecimal and binary number systems.) Numeric representations of integers from base 11 to base 36 use the characters A–Z to represent the values 10 to 35. For example, hexadecimal values can consist of the digits 0–9 and the characters A–F. A base-11 integer can consist of the digits 0–9 and the character A. A base-24 integer can consist of the digits 0–9 and the characters A–N. A base-36 integer can consist of the digits 0–9 and the characters A–Z. [*Note:* The case of the letter used is ignored.]

Function **`strtoul`** (Fig. 21.33) converts to `unsigned long` a sequence of characters representing an `unsigned long` integer. The function works identically to `strtol`. Line 14 indicates that `x` is assigned the `unsigned long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (`1234567`) in `string1`. The third argument, `0`, indicates that the value to be converted can be in octal, decimal or hexadecimal format, depending on the initial characters.

```
1  // Fig. 21.33: fig21_33.cpp
2  // Using strtoul.
3  #include <iostream>
4  #include <cstdlib> // strtoul prototype
5  using namespace std;
6
```

Fig. 21.33 | String-conversion function `strtoul`. (Part 1 of 2.)

```
int main()
{
   unsigned long x;
   const char *string1 = "1234567abc";
   char *remainderPtr;

   // convert a sequence of characters to unsigned long
   x = strtoul( string1, &remainderPtr, 0 );

   cout << "The original string is \"" << string1
      << "\"\nThe converted value is " << x
      << "\nThe remainder of the original string is \"" << remainderPtr
      << "\"\nThe converted value minus 567 is " << x - 567 << endl;
} // end main
```

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Fig. 21.33 | String-conversion function `strtoul`. (Part 2 of 2.)

21.10 Search Functions of the Pointer-Based String-Handling Library

This section presents the functions of the string-handling library used to search strings for characters and other strings. The functions are summarized in Fig. 21.34. Functions `strcspn` and `strspn` specify return type `size_t`. Type `size_t` is a type defined by the standard as the integral type of the value returned by operator `sizeof`.

Function **`strchr`** searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer. The program of Fig. 21.35 uses `strchr` (lines 14 and 22) to search for the first occurrences of `'a'` and `'z'` in the string `"This is a test"`.

Prototype	Description
`char *strchr( const char *s, int c )`	
	Locates the first occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in `s` is returned. Otherwise, a null pointer is returned.
`char *strrchr( const char *s, int c )`	
	Searches from the end of string `s` and locates the last occurrence of character `c` in string `s`. If `c` is found, a pointer to `c` in string `s` is returned. Otherwise, a null pointer is returned.
`size_t strspn( const char *s1, const char *s2 )`	
	Determines and returns the length of the initial segment of string `s1` consisting only of characters contained in string `s2`.

Fig. 21.34 | Search functions of the pointer-based string-handling library. (Part 1 of 2.)

Prototype	Description
`char *strpbrk( const char *s1, const char *s2 )`	
	Locates the first occurrence in string s1 of any character in string s2. If a character from string s2 is found, a pointer to the character in string s1 is returned. Otherwise, a null pointer is returned.
`size_t strcspn( const char *s1, const char *s2 )`	
	Determines and returns the length of the initial segment of string s1 consisting of characters not contained in string s2.
`char *strstr( const char *s1, const char *s2 )`	
	Locates the first occurrence in string s1 of string s2. If the string is found, a pointer to the string in s1 is returned. Otherwise, a null pointer is returned.

Fig. 21.34 | Search functions of the pointer-based string-handling library. (Part 2 of 2.)

```
// Fig. 21.35: fig21_35.cpp
// Using strchr.
#include <iostream>
#include <cstring> // strchr prototype
using namespace std;

int main()
{
   const char *string1 = "This is a test";
   char character1 = 'a';
   char character2 = 'z';

   // search for character1 in string1
   if ( strchr( string1, character1 ) != NULL )
      cout << '\'' << character1 << "' was found in \""
         << string1 << "\".\n";
   else
      cout << '\'' << character1 << "' was not found in \""
         << string1 << "\".\n";

   // search for character2 in string1
   if ( strchr( string1, character2 ) != NULL )
      cout << '\'' << character2 << "' was found in \""
         << string1 << "\".\n";
   else
      cout << '\'' << character2 << "' was not found in \""
         << string1 << "\"." << endl;
} // end main
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Fig. 21.35 | String-search function `strchr`.

Function **strcspn** (Fig. 21.36, line 15) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.

```
1  // Fig. 21.36: fig21_36.cpp
2  // Using strcspn.
3  #include <iostream>
4  #include <cstring> // strcspn prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "The value is 3.14159";
10    const char *string2 = "1234567890";
11
12    cout << "string1 = " << string1 << "\nstring2 = " << string2
13       << "\n\nThe length of the initial segment of string1"
14       << "\ncontaining no characters from string2 = "
15       << strcspn( string1, string2 ) << endl;
16 } // end main
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Fig. 21.36 | String-search function `strcspn`.

Function **strpbrk** searches for the first occurrence in its first string argument of any character in its second string argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character in the first argument; otherwise, `strpbrk` returns a null pointer. Line 13 of Fig. 21.37 locates the first occurrence in `string1` of any character from `string2`.

```
1  // Fig. 21.37: fig21_37.cpp
2  // Using strpbrk.
3  #include <iostream>
4  #include <cstring> // strpbrk prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "This is a test";
10    const char *string2 = "beware";
11
12    cout << "Of the characters in \"" << string2 << "\"\n'"
13       << *strpbrk( string1, string2 ) << "\' is the first character "
14       << "to appear in\n\"" << string1 << '\"' << endl;
15 } // end main
```

Fig. 21.37 | String-search function `strpbrk`. (Part 1 of 2.)

```
Of the characters in "beware"
'a' is the first character to appear in
"This is a test"
```

Fig. 21.37 | String-search function `strpbrk`. (Part 2 of 2.)

Function **`strrchr`** searches for the last occurrence of the specified character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, `strrchr` returns 0. Line 15 of Fig. 21.38 searches for the last occurrence of the character `'z'` in the string `"A zoo has many animals including zebras"`.

```
1  // Fig. 21.38: fig21_38.cpp
2  // Using strrchr.
3  #include <iostream>
4  #include <cstring> // strrchr prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "A zoo has many animals including zebras";
10    char c = 'z';
11
12    cout << "string1 = " << string1 << "\n" << endl;
13    cout << "The remainder of string1 beginning with the\n"
14       << "last occurrence of character '"
15       << c << "' is: \"" << strrchr( string1, c ) << '\"' << endl;
16 } // end main
```

```
string1 = A zoo has many animals including zebras

The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Fig. 21.38 | String-search function `strrchr`.

Function **`strspn`** (Fig. 21.39, line 15) determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument. The function returns the length of the segment.

```
1  // Fig. 21.39: fig21_39.cpp
2  // Using strspn.
3  #include <iostream>
4  #include <cstring> // strspn prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "The value is 3.14159";
```

Fig. 21.39 | String-search function `strspn`. (Part 1 of 2.)

```
   const char *string2 = "aehils Tuv";

   cout << "string1 = " << string1 << "\nstring2 = " << string2
      << "\n\nThe length of the initial segment of string1\n"
      << "containing only characters from string2 = "
      << strspn( string1, string2 ) << endl;
} // end main
```

```
string1 = The value is 3.14159
string2 = aehils Tuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Fig. 21.39 | String-search function `strspn`. (Part 2 of 2.)

Function **`strstr`** searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned; otherwise, it returns `0`. Line 15 of Fig. 21.40 uses `strstr` to find the string `"def"` in the string `"abcdefabcdef"`.

```
1  // Fig. 21.40: fig21_40.cpp
2  // Using strstr.
3  #include <iostream>
4  #include <cstring> // strstr prototype
5  using namespace std;
6
7  int main()
8  {
9     const char *string1 = "abcdefabcdef";
10    const char *string2 = "def";
11
12    cout << "string1 = " << string1 << "\nstring2 = " << string2
13       << "\n\nThe remainder of string1 beginning with the\n"
14       << "first occurrence of string2 is: "
15       << strstr( string1, string2 ) << endl;
16 } // end main
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Fig. 21.40 | String-search function `strstr`.

21.11 Memory Functions of the Pointer-Based String-Handling Library

The string-handling library functions presented in this section facilitate manipulating, comparing and searching blocks of memory. The functions treat blocks of memory as arrays of bytes. These functions can manipulate any block of data. Figure 21.41 summarizes the memory functions of the string-handling library. In the function discussions, "object" refers to a block of data. [*Note:* The string-processing functions in prior sections operate on null-terminated strings. The ones in this section operate on arrays of bytes. The null-character value (i.e., a byte containing 0) has *no* significance with the functions in this section.]

Prototype	Description
`void *memcpy( void *s1, const void *s2, size_t n )`	
	Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. A pointer to the resulting object is returned. The area from which characters are copied is not allowed to overlap the area to which characters are copied.
`void *memmove( void *s1, const void *s2, size_t n )`	
	Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The copy is performed as if the characters were first copied from the object pointed to by `s2` into a temporary array, then copied from the temporary array into the object pointed to by `s1`. A pointer to the resulting object is returned. The area from which characters are copied is allowed to overlap the area to which characters are copied.
`int memcmp( const void *s1, const void *s2, size_t n )`	
	Compares the first `n` characters of the objects pointed to by `s1` and `s2`. The function returns `0`, less than `0`, or greater than `0` if `s1` is equal to, less than or greater than `s2`, respectively.
`void *memchr( const void *s, int c, size_t n )`	
	Locates the first occurrence of `c` (converted to `unsigned char`) in the first `n` characters of the object pointed to by `s`. If `c` is found, a pointer to `c` in the object is returned. Otherwise, `0` is returned.
`void *memset( void *s, int c, size_t n )`	
	Copies `c` (converted to `unsigned char`) into the first `n` characters of the object pointed to by `s`. A pointer to the result is returned.

Fig. 21.41 | Memory functions of the string-handling library.

The pointer parameters to these functions are declared `void *`. In Chapter 8, we saw that *a pointer to any data type can be assigned directly to a pointer of type* `void *`. For this reason, these functions can receive pointers to any data type. Remember that *a pointer of type* `void *` *cannot be assigned directly to a pointer of any other data type*. Because a `void *` pointer cannot be dereferenced, each function receives a size argument that specifies the number of characters (bytes) the function will process. For simplicity, the examples in this section manipulate character arrays (blocks of characters).

Function **`memcpy`** copies a specified number of characters (bytes) from the object pointed to by its second argument into the object pointed to by its first argument. The function can receive a pointer to any type of object. The result of this function is undefined if the two objects overlap in memory (i.e., are parts of the same object). The program of Fig. 21.42 uses `memcpy` (line 14) to copy the string in array `s2` to array `s1`.

```cpp
// Fig. 21.42: fig21_42.cpp
// Using memcpy.
#include <iostream>
#include <cstring> // memcpy prototype
using namespace std;

int main()
{
   char s1[ 17 ];

   // 17 total characters (includes terminating null)
   char s2[] = "Copy this string";

   memcpy( s1, s2, 17 ); // copy 17 characters from s2 to s1

   cout << "After s2 is copied into s1 with memcpy,\n"
      << "s1 contains \"" << s1 << '\"' << endl;
} // end main
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Fig. 21.42 | Memory-handling function `memcpy`.

Function **`memmove`**, like `memcpy`, copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument to a temporary array of characters, then copied from the temporary array to the first argument. This allows characters from one part of a string to be copied into another part of the same string.

Common Programming Error 21.11

String-manipulation functions other than `memmove` *that copy characters have undefined results when copying takes place between parts of the same string.*

The program in Fig. 21.43 uses `memmove` (line 13) to copy the last 10 bytes of array `x` into the first 10 bytes of array `x`.

```cpp
1  // Fig. 21.43: fig21_43.cpp
2  // Using memmove.
3  #include <iostream>
4  #include <cstring> // memmove prototype
5  using namespace std;
6
7  int main()
8  {
9     char x[] = "Home Sweet Home";
10
11    cout << "The string in array x before memmove is: " << x;
12    cout << "\nThe string in array x after memmove is:  "
```

Fig. 21.43 | Memory-handling function `memmove`. (Part 1 of 2.)

```
      << static_cast< char * >( memmove( x, &x[ 5 ], 10 ) ) << endl;
} // end main
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is:  Sweet Home Home
```

Fig. 21.43 | Memory-handling function `memmove`. (Part 2 of 2.)

Function **`memcmp`** (Fig. 21.44, lines 14–16) compares the specified number of characters of its first argument with the corresponding characters of its second argument. The function returns a value greater than zero if the first argument is greater than the second argument, zero if the arguments are equal, and a value less than zero if the first argument is less than the second argument. [*Note:* With some compilers, function `memcmp` returns `-1`, `0` or `1`, as in the sample output of Fig. 21.44. With other compilers, this function returns `0` or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s2` are compared, the first character that differs between them is the fifth character of each string—`E` (numeric code 69) for `s1` and `X` (numeric code 72) for `s2`. In this case, the return value will be `19` (or `-19` when `s2` is compared to `s1`).]

```
1   // Fig. 21.44: fig21_44.cpp
2   // Using memcmp.
3   #include <iostream>
4   #include <iomanip>
5   #include <cstring> // memcmp prototype
6   using namespace std;
7
8   int main()
9   {
10     char s1[] = "ABCDEFG";
11     char s2[] = "ABCDXYZ";
12
13     cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
14        << "\nmemcmp(s1, s2, 4) = " << setw( 3 ) << memcmp( s1, s2, 4 )
15        << "\nmemcmp(s1, s2, 7) = " << setw( 3 ) << memcmp( s1, s2, 7 )
16        << "\nmemcmp(s2, s1, 7) = " << setw( 3 ) << memcmp( s2, s1, 7 )
17        << endl;
18  } // end main
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =   0
memcmp(s1, s2, 7) =  -1
memcmp(s2, s1, 7) =   1
```

Fig. 21.44 | Memory-handling function `memcmp`.

Function **`memchr`** searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found in the object, a pointer to it is returned; otherwise, the function returns a null pointer. Line 13 of Fig. 21.45 searches for the character (byte) `'r'` in the string `"This is a string"`.

```
// Fig. 21.45: fig21_45.cpp
// Using memchr.
#include <iostream>
#include <cstring> // memchr prototype
using namespace std;

int main()
{
   char s[] = "This is a string";

   cout << "s = " << s << "\n" << endl;
   cout << "The remainder of s after character 'r' is found is \""
      << static_cast< char * >( memchr( s, 'r', 16 ) ) << '\"' << endl;
} // end main
```

```
s = This is a string

The remainder of s after character 'r' is found is "ring"
```

Fig. 21.45 | Memory-handling function `memchr`.

Function **`memset`** copies the value of the byte in its second argument into a specified number of bytes of the object pointed to by its first argument. Line 13 in Fig. 21.46 uses `memset` to copy `'b'` into the first 7 bytes of `string1`.

```
1  // Fig. 21.46: fig21_46.cpp
2  // Using memset.
3  #include <iostream>
4  #include <cstring> // memset prototype
5  using namespace std;
6
7  int main()
8  {
9     char string1[ 15 ] = "BBBBBBBBBBBBBB";
10
11    cout << "string1 = " << string1 << endl;
12    cout << "string1 after memset = "
13       << static_cast< char * >( memset( string1, 'b', 7 ) ) << endl;
14 } // end main
```

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

Fig. 21.46 | Memory-handling function `memset`.

21.12 Wrap-Up

This chapter introduced `struct` definitions, initializing `struct`s and using them with functions. We discussed `typedef`, using it to create aliases to help promote portability. We also introduced bitwise operators to manipulate data and bit fields for storing data compactly. You learned about the string-conversion functions in `<cstlib>` and the string-processing functions in `<cstring>`. In the next chapter, we continue our discussion of data structures by discussing *containers*—data structures defined in the C++ Standard Template Library. We also present the many algorithms defined in the STL as well.

Summary

Section 21.2 Structure Definitions

- Keyword `struct` (p. 793) begins every structure definition. Between the braces of the structure definition are the structure member declarations.
- A structure definition creates a new data type (p. 793) that can be used to declare variables.

Section 21.3 `typedef`

- Creating a new type name with `typedef` (p. 794) does not create a new type; it creates a name that's synonymous with a type defined previously.

Section 21.5 Bitwise Operators

- The bitwise AND operator (`&`; p. 797) takes two integral operands. A bit in the result is set to one if the corresponding bits in each of the operands are one.
- Masks (p. 799) are used with bitwise AND to hide some bits while preserving others.
- The bitwise inclusive OR operator (`|`; p. 797) takes two operands. A bit in the result is set to one if the corresponding bit in either operand is set to one.
- Each of the bitwise operators (except complement) has a corresponding assignment operator.
- The bitwise exclusive OR operator (`^`; p. 797) takes two operands. A bit in the result is set to one if exactly one of the corresponding bits in the two operands is set to one.
- The left-shift operator (`<<`) shifts the bits of its left operand left by the number of bits specified by its right operand. Bits vacated to the right are replaced with `zeros`.
- The right-shift operator (`>>`) shifts the bits of its left operand right by the number of bits specified in its right operand. Right shifting an unsigned integer causes bits vacated at the left to be replaced by zeros. Vacated bits in signed integers can be replaced with zeros or ones.
- The bitwise complement operator (`~`; p. 797) takes one operand and inverts its bits—this produces the one's complement of the operand.

Section 21.6 Bit Fields

- Bit fields (p. 806) reduce storage use by storing data in the minimum number of bits required. Bit-field members must be declared as `int` or `unsigned`.
- A bit field is declared by following an `unsigned` or `int` member name with a colon and the width of the bit field.
- The bit-field width must be an integer constant.
- If a bit field is specified without a name, the field is used as padding (p. 809) in the structure.
- An unnamed bit field (p. 809) with width `0` aligns the next bit field on a new machine-word boundary.

Section 21.7 Character-Handling Library

- Function `islower` (p. 812) determines if its argument is a lowercase letter (`a`–`z`). Function `isupper` (p. 812) determines whether its argument is an uppercase letter (`A`–`Z`).
- Function `isdigit` (p. 811) determines if its argument is a digit (`0`–`9`).

- Function `isalpha` (p. 811) determines if its argument is an uppercase (`A–Z`) or lowercase letter (`a–z`).
- Function `isalnum` (p. 811) determines if its argument is an uppercase letter (`A–Z`), a lowercase letter (`a–z`), or a digit (`0–9`).
- Function `isxdigit` (p. 811) determines if its argument is a hexadecimal digit (`A–F`, `a–f`, `0–9`).
- Function `toupper` (p. 812) converts a lowercase letter to an uppercase letter. Function `tolower` (p. 812) converts an uppercase letter to a lowercase letter.
- Function `isspace` (p. 813) determines if its argument is one of the following white-space characters: `' '` (space), `'\f'`, `'\n'`, `'\r'`, `'\t'` or `'\v'`.
- Function `iscntrl` (p. 813) determines if its argument is a control character, such as `'\t'`, `'\v'`, `'\f'`, `'\a'`, `'\b'`, `'\r'` or `'\n'`.
- Function `ispunct` (p. 813) determines if its argument is a printing character other than a space, a digit or a letter.
- Function `isprint` (p. 813) determines if its argument is any printing character, including space.
- Function `isgraph` (p. 813) determines if its argument is a printing character other than space.

Section 21.8 Pointer-Based String Manipulation Functions

- Function `strcpy` (p. 816) copies its second argument into its first argument. You must ensure that the target array is large enough to store the string and its terminating null character.
- Function `strncpy` (p. 817) is equivalent to `strcpy`, but it specifies the number of characters to be copied from the string into the array. The terminating null character will be copied only if the number of characters to be copied is at least one more than the length of the string.
- Function `strcat` (p. 818) appends its second string argument—including the terminating null character—to its first string argument. The first character of the second string replaces the null (`'\0'`) character of the first string. You must ensure that the target array used to store the first string is large enough to store both the first string and the second string.
- Function `strncat` (p. 818) is equivalent to `strcat`, but it appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.
- Function `strcmp` compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string.
- Function `strncmp` is equivalent to `strcmp`, but it compares a specified number of characters. If the number of characters in one of the strings is less than the number of characters specified, `strncmp` compares characters until the null character in the shorter string is encountered.
- A sequence of calls to `strtok` (p. 820) breaks a string into tokens that are separated by characters contained in a second string argument. The first call specifies the string to be tokenized as the first argument, and subsequent calls to continue tokenizing the same string specify `NULL` as the first argument. The function returns a pointer to the current token from each call. If there are no more tokens when `strtok` is called, `NULL` is returned.
- Function `strlen` (p. 822) takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length of the string.

Section 21.9 Pointer-Based String-Conversion Functions

- Function `atof` (p. 823) converts its argument—a string beginning with a series of digits that represents a floating-point number—to a `double` value.
- Function `atoi` (p. 824) converts its argument—a string beginning with a series of digits that represents an integer—to an `int` value.

- Function `atol` (p. 824) converts its argument—a string beginning with a series of digits that represents a long integer—to a `long` value.
- Function `strtod` (p. 825) converts a sequence of characters representing a floating-point value to `double`. The function receives two arguments—a string (`char *`) and the address of a `char *` pointer. The string contains the character sequence to be converted, and the pointer to `char *` is assigned the remainder of the string after the conversion.
- Function `strtol` (p. 825) converts a sequence of characters representing an integer to `long`. It receives a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned the location of the first character after the converted value and the integer specifies the base of the value being converted.
- Function `strtoul` (p. 826) converts a sequence of characters representing an integer to `unsigned long`. It receives a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned the location of the first character after the converted value and the integer specifies the base of the value being converted.

Section 21.10 Search Functions of the Pointer-Based String-Handling Library

- Function `strchr` (p. 827) searches for the first occurrence of a character in a string. If found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer.
- Function `strcspn` (p. 829) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.
- Function `strpbrk` (p. 829) searches for the first occurrence in its first argument of any character that appears in its second argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character; otherwise, `strpbrk` returns a null pointer.
- Function `strrchr` (p. 830) searches for the last occurrence of a character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, it returns a null pointer.
- Function `strspn` (p. 830) determines the length of the initial part of its first argument that contains only characters from the string in its second argument and returns the length of the segment.
- Function `strstr` (p. 831) searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned; otherwise it returns 0.

Section 21.11 Memory Functions of the Pointer-Based String-Handling Library

- Function `memcpy` (p. 832) copies a specified number of characters from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any object. The pointers are received as `void` pointers and converted to `char` pointers for use in the function. Function `memcpy` manipulates the bytes of its argument as characters.
- Function `memmove` (p. 833) copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary character array, then copied from the temporary array to the first argument.
- Function `memcmp` (p. 834) compares the specified number of characters of its first and second arguments.
- Function `memchr` (p. 834) searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found, a pointer to it is returned; otherwise, a null pointer is returned.

- Function `memset` (p. 835) copies its second argument, treated as an `unsigned char`, to a specified number of bytes of the object pointed to by the first argument.

Self-Review Exercises

21.1 Fill in the blanks in each of the following:

a) The bits in the result of an expression using the ________ operator are set to one if the corresponding bits in each operand are set to one. Otherwise, the bits are set to zero.
b) The bits in the result of an expression using the ________ operator are set to one if at least one of the corresponding bits in either operand is set to one. Otherwise, the bits are set to zero.
c) Keyword ________ introduces a structure declaration.
d) Keyword ________ is used to create a synonym for a previously defined data type.
e) Each bit in the result of an expression using the ________ operator is set to one if exactly one of the corresponding bits in either operand is set to one.
f) The bitwise AND operator `&` is often used to ________ bits (i.e., to select certain bits from a bit string while zeroing others).
g) The ________ and ________ operators are used to shift the bits of a value to the left or to the right, respectively.

21.2 Write a single statement or a set of statements to accomplish each of the following:

a) Define a structure called `Part` containing `int` variable `partNumber` and `char` array `partName`, whose values may be as long as 25 characters.
b) Define `PartPtr` to be a synonym for the type `Part *`.
c) Use separate statements to declare variable `a` to be of type `Part`, array `b[ 10 ]` to be of type `Part` and variable `ptr` to be of type pointer to `Part`.
d) Read a part number and a part name from the keyboard into the members of variable `a`.
e) Assign the member values of variable `a` to element three of array `b`.
f) Assign the address of array `b` to the pointer variable `ptr`.
g) Print the member values of element three of array `b`, using the variable `ptr` and the structure pointer operator to refer to the members.

21.3 Write a single statement to accomplish each of the following. Assume that variables `c` (which stores a character), `x`, `y` and `z` are of type `int`; variables `d`, `e` and `f` are of type `double`; variable `ptr` is of type `char *` and arrays `s1[ 100 ]` and `s2[ 100 ]` are of type `char`.

a) Convert the character stored in variable `c` to an uppercase letter. Assign the result to variable `c`.
b) Determine if the value of variable `c` is a digit. Use the conditional operator as shown in Figs. 21.18–21.20 to print `" is a "` or `" is not a "` when the result is displayed.
c) Convert the string `"1234567"` to `long`, and print the value.
d) Determine whether the value of variable `c` is a control character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.
e) Assign to `ptr` the location of the last occurrence of `c` in `s1`.
f) Convert the string `"8.63582"` to `double`, and print the value.
g) Determine whether the value of `c` is a letter. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.
h) Assign to `ptr` the location of the first occurrence of `s2` in `s1`.
i) Determine whether the value of variable `c` is a printing character. Use the conditional operator to print `" is a "` or `" is not a "` when the result is displayed.
j) Assign to `ptr` the location of the first occurrence in `s1` of any character from `s2`.
k) Assign to `ptr` the location of the first occurrence of `c` in `s1`.
l) Convert the string `"-21"` to `int`, and print the value.

Answers to Self-Review Exercises

21.1 a) bitwise AND (&). b) bitwise inclusive OR (|). c) `struct`. d) `typedef`. e) bitwise exclusive OR (^). f) mask. g) left-shift operator (<<), right-shift operator (>>).

21.2 a)
```
struct Part
{
   int partNumber;
   char partName[ 26 ];
};
```
b) `typedef Part * PartPtr;`

c)
```
Part a;
Part b[ 10 ];
Part *ptr;
```
d) `cin >> a.partNumber >> a.partName;`

e) `b[ 3 ] = a;`

f) `ptr = b;`

g)
```
cout << ( ptr + 3 )->partNumber << ' '
     << ( ptr + 3 )->partName << endl;
```

21.3 a) `c = toupper( c );`

b)
```
cout << '\'' << c << "\' "
     << ( isdigit( c ) ? "is a" : "is not a" )
      << " digit" << endl;
```
c) `cout << atol( "1234567" ) << endl;`

d)
```
cout << '\'' << c << "\' "
     << ( iscntrl( c ) ? "is a" : "is not a" )
      << " control character" << endl;
```
e) `ptr = strrchr( s1, c );`

f) `out << atof( "8.63582" ) << endl;`

g)
```
cout << '\'' << c << "\' "
      << ( isalpha( c ) ? "is a" : "is not a" )
      << " letter" << endl;
```
h) `ptr = strstr( s1, s2 );`

i)
```
cout << '\'' << c << "\' "
      << ( isprint( c ) ? "is a" : "is not a" )
      << " printing character" << endl;
```
j) `ptr = strpbrk( s1, s2 );`

k) `ptr = strchr( s1, c );`

l) `cout << atoi( "-21" ) << endl;`

Exercises

21.4 *(Defining Structures)* Provide the definition for each of the following structures:

a) Structure `Inventory`, containing character array `partName[ 30 ]`, integer `partNumber`, floating-point `price`, integer `stock` and integer `reorder`.

b) A structure called `Address` that contains character arrays `streetAddress[25]`, `city[20]`, `state[3]` and `zipCode[6]`.

c) Structure `Student`, containing arrays `firstName[ 15 ]` and `lastName[ 15 ]` and variable `homeAddress` of type `struct Address` from part (b).

d) Structure `Test`, containing 16 bit fields with widths of 1 bit. The names of the bit fields are the letters `a` to `p`.

21.5 *(Card Shufflling and Dealing)* Modify Fig. 21.14 to shuffle the cards using a high-performance shuffle, as shown in Fig. 21.3. Print the resulting deck in two-column format. Precede each card with its color.

21.6 *(Shifting and Printing an Integer)* Write a program that right-shifts an integer variable four bits. The program should print the integer in bits before and after the shift operation. Does your system place zeros or ones in the vacated bits?

21.7 *(Multiplication Via Bit Shifting)* Left-shifting an `unsigned` integer by one bit is equivalent to multiplying the value by 2. Write function `power2` that takes two integer arguments, `number` and `pow`, and calculates

$$\texttt{number * }2^{\texttt{pow}}$$

Use a shift operator to calculate the result. The program should print the values as integers and as bits.

21.8 *(Packing Characters into Unsigned Integers)* The left-shift operator can be used to pack two character values into a two-byte unsigned integer variable. Write a program that inputs two characters from the keyboard and passes them to function `packCharacters`. To pack two characters into an `unsigned` integer variable, assign the first character to the `unsigned` variable, shift the `unsigned` variable left by eight bit positions and combine the `unsigned` variable with the second character using the bitwise inclusive-OR operator. The program should output the characters in their bit format before and after they're packed into the `unsigned` integer to prove that they're in fact packed correctly in the `unsigned` variable.

21.9 *(Unpacking Characters from Unsigned Integers)* Using the right-shift operator, the bitwise AND operator and a mask, write function `unpackCharacters` that takes the `unsigned` integer from Exercise 21.8 and unpacks it into two characters. To unpack two characters from an `unsigned` two-byte integer, combine the unsigned integer with the mask `65280` (`11111111 00000000`) and right-shift the result eight bits. Assign the resulting value to a `char` variable. Then, combine the `unsigned` integer with the mask `255` (`00000000 11111111`). Assign the result to another `char` variable. The program should print the `unsigned` integer in bits before it's unpacked, then print the characters in bits to confirm that they were unpacked correctly.

21.10 *(Packing Characters into Unsigned Integers)* If your system uses four-byte integers, rewrite the program of Exercise 21.8 to pack four characters.

21.11 *(Unpacking Characters from Unsigned Integers)* If your system uses four-byte integers, rewrite the function `unpackCharacters` of Exercise 21.9 to unpack four characters. Create the masks you need to unpack the four characters by left-shifting the value 255 in the mask variable by eight bits 0, 1, 2 or 3 times (depending on the byte you are unpacking).

21.12 *(Reversing Bits)* Write a program that reverses the order of the bits in an `unsigned` integer value. The program should input the value from the user and call function `reverseBits` to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.

21.13 *(Testing Characters with the `<cctype>` Functions)* Write a program that inputs a character from the keyboard and tests the character with each function in the character-handling library. Print the value returned by each function.

21.14 The following program uses function `multiple` to determine whether the integer entered from the keyboard is a multiple of some integer `X`. Examine function `multiple`, then determine the value of `X`.

```cpp
// Exercise 21.14: ex21_14.cpp
// This program determines if a value is a multiple of X.
#include <iostream>
using namespace std;

bool multiple( int );

int main()
{
   int y;

   cout << "Enter an integer between 1 and 32000: ";
   cin >> y;

   if ( multiple( y ) )
      cout << y << " is a multiple of X" << endl;
   else
      cout << y << " is not a multiple of X" << endl;
} // end main

// determine if num is a multiple of X
bool multiple( int num )
{
   bool mult = true;

   for ( int i = 0, mask = 1; i < 10; ++i, mask <<= 1 )
      if ( ( num & mask ) != 0 )
      {
         mult = false;
         break;
      } // end if

   return mult;
} // end function multiple
```

21.15 What does the following program do?

```cpp
// Exercise 21.15: ex21_15.cpp
#include <iostream>
using namespace std;

bool mystery( unsigned );

int main()
{
   unsigned x;

   cout << "Enter an integer: ";
   cin >> x;
   cout << boolalpha
      << "The result is " << mystery( x ) << endl;
} // end main

// What does this function do?
bool mystery( unsigned bits )
{
   const int SHIFT = 8 * sizeof( unsigned ) - 1;
   const unsigned MASK = 1 << SHIFT;
```

```
    unsigned total = 0;

    for ( int i = 0; i < SHIFT + 1; ++i, bits <<= 1 )
       if ( ( bits & MASK ) == MASK )
          ++total;

    return !( total % 2 );
 } // end function mystery
```

21.16 Write a program that inputs a line of text with `istream` member function `getline` (as in Chapter 15) into character array `s[100]`. Output the line in uppercase letters and lowercase letters.

21.17 *(Converting Strings to Integers)* Write a program that inputs four strings that represent integers, converts the strings to integers, sums the values and prints the total of the four values. Use only the C-style string-processing techniques shown in this chapter.

21.18 *(Converting Strings to Floating-Point Numbers)* Write a program that inputs four strings that represent floating-point values, converts the strings to double values, sums the values and prints the total of the four values. Use only the C-style string-processing techniques shown in this chapter.

21.19 *(Searching for Substrings)* Write a program that inputs a line of text and a search string from the keyboard. Using function `strstr`, locate the first occurrence of the search string in the line of text, and assign the location to variable `searchPtr` of type `char *`. If the search string is found, print the remainder of the line of text beginning with the search string. Then use `strstr` again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [*Hint:* The second call to `strstr` should contain the expression `searchPtr + 1` as its first argument.]

21.20 *(Searching for Substrings)* Write a program based on the program of Exercise 21.19 that inputs several lines of text and a search string, then uses function `strstr` to determine the total number of occurrences of the string in the lines of text. Print the result.

21.21 *(Searching for Characters)* Write a program that inputs several lines of text and a search character and uses function `strchr` to determine the total number of occurrences of the character in the lines of text.

21.22 *(Searching for Characters)* Write a program based on the program of Exercise 21.21 that inputs several lines of text and uses function `strchr` to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format after the totals have been determined.

21.23 *(ASCII Character Set)* The chart in Appendix B shows the numeric code representations for the characters in the ASCII character set. Study this chart, then state whether each of the following is *true* or *false*:

a) The letter “`A`” comes before the letter “`B`.”
b) The digit “`9`” comes before the digit “`0`.”
c) The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.
d) The digits come before the letters.
e) If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.

21.24 *(Strings Beginning with `b`)* Write a program that reads a series of strings and prints only those strings beginning with the letter “`b`.”

21.25 *(Strings Ending with* **ED***)* Write a program that reads a series of strings and prints only those strings that end with the letters "ED."

21.26 *(Displaying Characters for Given ASCII Codes)* Write a program that inputs an ASCII code and prints the corresponding character. Modify this program so that it generates all possible three-digit codes in the range 000–255 and attempts to print the corresponding characters. What happens when this program is run?

21.27 *(Write Your Own Character Handling Functions)* Using the ASCII character chart in Appendix B as a guide, write your own versions of the character-handling functions in Fig. 21.17.

21.28 *(Write Your Own String Conversion Functions)* Write your own versions of the functions in Fig. 21.27 for converting strings to numbers.

21.29 *(Write Your Own String Searching Functions)* Write your own versions of the functions in Fig. 21.34 for searching strings.

21.30 *(Write Your Own Memory Handling Functions)* Write your own versions of the functions in Fig. 21.41 for manipulating blocks of memory.

21.31 *(What Does the Program Do?)* What does this program do?

```
// Ex. 21.31: ex21_31.cpp
// What does this program do?
#include <iostream>
using namespace std;

bool mystery3( const char *, const char * ); // prototype

int main()
{
   char string1[ 80 ], string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   cout << "The result is " << mystery3( string1, string2 ) << endl;
} // end main

// What does this function do?
bool mystery3( const char *s1, const char *s2 )
{
   for ( ; *s1 != '\0' && *s2 != '\0'; ++s1, ++s2 )

      if ( *s1 != *s2 )
         return false;

   return true;
} // end function mystery3
```

21.32 *(Comparing Strings)* Write a program that uses function `strcmp` to compare two strings input by the user. The program should state whether the first string is less than, equal to or greater than the second string.

21.33 *(Comparing Strings)* Write a program that uses function `strncmp` to compare two strings input by the user. The program should input the number of characters to compare. The program should state whether the first string is less than, equal to or greater than the second string.

21.34 *(Randomly Creating Sentences)* Write a program that uses random number generation to create sentences. The program should use four arrays of pointers to `char` called `article`, `noun`, `verb` and `preposition`. The program should create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, it should be concatenated to the previous words in a character array that's large enough to hold the entire sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 such sentences.

The arrays should be filled as follows: The `article` array should contain the articles `"the"`, `"a"`, `"one"`, `"some"` and `"any"`; the `noun` array should contain the nouns `"boy"`, `"girl"`, `"dog"`, `"town"` and `"car"`; the `verb` array should contain the verbs `"drove"`, `"jumped"`, `"ran"`, `"walked"` and `"skipped"`; the `preposition` array should contain the prepositions `"to"`, `"from"`, `"over"`, `"under"` and `"on"`.

After completing the program, modify it to produce a short story consisting of several of these sentences. (How about a random term-paper writer!)

21.35 *(Limericks)* A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 21.34, write a C++ program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

21.36 *(Pig Latin)* Write a program that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm: To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function `strtok`. To translate each English word into a pig-Latin word, place the first letter of the English word at the end of the English word and add the letters "ay." Thus, the word "`jump`" becomes "`umpjay`," the word "`the`" becomes "`hetay`" and the word "`computer`" becomes "`omputercay`." Blanks between words remain as blanks. Assume that the English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. [*Hint:* Each time a token is found in a call to `strtok`, pass the token pointer to function `printLatinWord` and print the pig-Latin word.]

21.37 *(Tokenizing Phone Numbers)* Write a program that inputs a telephone number as a string in the form `(555) 555-5555`. The program should use function `strtok` to extract the area code as a token, the first three digits of the phone number as a token, and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. Both the area code and the phone number should be printed.

21.38 *(Tokenizing and Reversing a Sentence)* Write a program that inputs a line of text, tokenizes the line with function `strtok` and outputs the tokens in reverse order.

21.39 *(Alphabetizing Strings)* Use the string-comparison functions discussed in Section 21.8 and the techniques for sorting arrays developed in Chapter 7 to write a program that alphabetizes a list of strings. Use the names of 10 towns in your area as data for your program.

21.40 *(Write Your Own String Copy and Concatenation Functions)* Write two versions of each string-copy and string-concatenation function in Fig. 21.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

21.41 *(Write Your Own String Comparison Functions)* Write two versions of each string-comparison function in Fig. 21.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

21.42 *(Write Your Own String Length Function)* Write two versions of function `strlen` in Fig. 21.21. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

Special Section: Advanced String-Manipulation Exercises

The preceding exercises are keyed to the text and designed to test your understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. You should find these problems challenging, yet enjoyable. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

21.43 *(Text Analysis)* The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence that Francis Bacon, Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these authors. This exercise examines three methods for analyzing texts with a computer. Thousands of texts, including Shakespeare, are available online at `www.gutenberg.org`.

a) Write a program that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one “a,” two “b’s,” no “c’s,” etc.

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

```
Whether 'tis nobler in the mind to suffer
```

contains the following word lengths and occurrences:

Word length	Occurrences
1	0
2	2
3	1
4	2 (including `'tis`)
5	0
6	2
7	1

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the word “to” three times, the word “be” two times, the word “or” once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

21.44 *(Word Processing)* One important function in word-processing systems is *type justification*—the alignment of words to both the left and right margins of a page. This generates a professional-looking document that gives the appearance of being set in type rather than prepared on a typewriter. Type justification can be accomplished on computer systems by inserting blank characters between the words in a line so that the rightmost word aligns with the right margin.

Write a program that reads several lines of text and prints this text in type-justified format. Assume that the text is to be printed on paper 8-1/2 inches wide and that one-inch margins are to be allowed on both the left and right sides. Assume that the computer prints 10 characters to the horizontal inch. Therefore, your program should print 6-1/2 inches of text, or 65 characters per line.

21.45 *(Printing Dates in Various Formats)* Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

```
07/21/1955
July 21, 1955
```

Write a program that reads a date in the first format and prints that date in the second format.

21.46 *(Check Protection)* Computers are frequently employed in check-writing systems such as payroll and accounts-payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems, because of human error or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose that a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example,

```
1,230.60  (check amount)
--------
12345678  (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces would ordinarily be left blank. For example,

```
   99.87
--------
12345678
```

contains three blank spaces. If a check is printed with blank spaces, it's easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87
--------
12345678
```

Write a program that inputs a dollar amount to be printed on a check then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

21.47 *(Writing the Word Equivalent of a Check Amount)* Continuing the discussion of the previous example, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it's extremely difficult to change the amount in words.

Write a program that inputs a numeric check amount and writes the word equivalent of the amount. Your program should be able to handle check amounts as large as $99.99. For example, the amount 112.43 should be written as

```
ONE HUNDRED TWELVE and 43/100
```

21.48 *(Morse Code)* Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound, and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 21.47.

Write a program that reads an English-language phrase and encodes it in Morse code. Also write a program that reads a phrase in Morse code and converts it into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

Character	Code	Character	Code	Character	Code
A	.-	N	-.	*Digits*	
B	-...	O	---	1	.----
C	-.-.	P	.--.	2	..---
D	-..	Q	--.-	3	...--
E	.	R	.-.	4	....-
F	..-.	S	...	5	.....
G	--.	T	-	6	-....
H	....	U	..-	7	--...
I	..	V	...-	8	---..
J	.---	W	.--	9	----.
K	-.-	X	-..-	0	-----
L	.-..	Y	-.--		
M	--	Z	--..		

Fig. 21.47 | Letters and digits as expressed in international Morse code.

21.49 *(Metric Conversion Program)* Write a program that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc., for the metric system and inches, quarts, pounds, etc., for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the question

```
"How many feet are in 5 kilograms?"
```

is not meaningful, because `"feet"` are units of length, while `"kilograms"` are units of weight.

Challenging String-Manipulation Projects

21.50 *(Crossword Puzzle Generator)* Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It's suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues that you must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or should two-dimensional arrays be used? You need a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle, in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

21.51 *(Spelling Checker)* Many popular word-processing software packages have built-in spell checkers. We used spell-checking capabilities in preparing this book and discovered that, no matter how careful we thought we were in writing a chapter, the software was always able to find a few more spelling errors than we were able to catch manually.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. You might find it helpful to use a computerized dictionary as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it's because we simply do not know the correct spelling, so we make a "best guess." In some cases, it's because we transpose two letters (e.g., "defualt" instead of "default"). Sometimes we double-type a letter accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"). And so on.

Design and implement a spell-checker program. Your program maintains an array `wordList` of character strings. You can either enter these strings or obtain them from a computerized dictionary.

Your program asks a user to enter a word. The program then looks up that word in the `wordList` array. If the word is present in the array, your program should print "`Word is spelled correctly.`"

If the word is not present in the array, your program should print "`Word is not spelled correctly.`" Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word "default" is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as "edfault," "dfeault," "deafult," "defalut" and "defautl." When you find a new word that matches one in `wordList`, print that word in a message such as "`Did you mean "default?".`"

Implement other tests, such as the replacing of each double letter with a single letter and any other tests you can develop to improve the value of your spell checker.

22 Standard Template Library (STL)

The shapes a bright container can contain!
—Theodore Roethke

Journey over all the universe in a map.
—Miguel de Cervantes

The historian is a prophet in reverse.
—Friedrich von Schlegel

Attempt the end, and never stand to doubt; Nothing's so hard but search will find it out.
—Robert Herrick

Objectives

In this chapter you'll learn:

- To use the STL containers, container adapters and "near containers."
- To program with many dozens of the STL algorithms.
- To use iterators to access the elements of STL containers.

22.1 Introduction to the Standard Template Library (STL)

The **Standard Template Library (STL)** defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. The STL offers proof of concept for generic programming with templates—introduced in Chapter 14, Templates, and used extensively in Chapter 20, Custom Templatized Data Structures. In industry, the features presented in this chapter are often referred to as the Standard Template Library or STL. However, these terms are not used in the C++ standard document, because these features are simply considered to be part of the C++ Standard Library.

The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and is based on their generic programming research, with significant contributions from David Musser. The STL was conceived and designed for performance and flexibility.

This chapter introduces the STL and discusses its three key components—**containers** (popular templatized data structures), **iterators** and **algorithms**. The STL containers are data structures capable of storing objects of almost any data type (there are some restrictions). We'll see that there are three styles of container classes—**first-class containers**, **adapters** and **near containers**.

Each STL container has associated member functions. A subset of these member functions is defined in *all* STL containers. We illustrate most of this common functionality in our examples of STL containers `vector` (a dynamically resizable array which we introduced in Chapter 7), **`list`** (a doubly linked list) and **`deque`** (a double-ended queue, pronounced "deck").

STL iterators, which have properties similar to those of pointers, are used by programs to manipulate the STL-container elements. Standard arrays also can be manipulated by STL algorithms, using standard pointers as iterators. We'll see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement. There are five categories of iterators, each of which we discuss in Section 22.3 and use throughout this chapter.

STL algorithms are functions that perform such common data manipulations as *searching*, *sorting* and *comparing elements or entire containers*. The STL provides scores of algorithms. Most of them use iterators to access container elements. Each algorithm has minimum requirements for the types of iterators that can be used with it. We'll see that each first-class container supports specific iterator types, some more powerful than others. A container's supported iterator type determines whether the container can be used with a specific algorithm. Iterators encapsulate the mechanism used to access container elements. This encapsulation enables many of the STL algorithms to be applied to various containers without regard for the underlying container implementation. As long as a container's iterators support the minimum requirements of the algorithm, then the algorithm can process that container's elements. This also enables you to create new algorithms that can process the elements of multiple container types.

Software Engineering Observation 22.1

The STL approach allows programs to be written so that the code does not depend on the underlying container. Such a programming style is called generic programming.

In Chapter 20, we studied data structures. We built linked lists, queues, stacks and trees. We carefully wove linked objects together with pointers. Pointer-based code is complex, and the slightest omission or oversight can lead to serious *memory-access violations* and *memory-leak* errors with no compiler complaints. Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work. In addition, if many programmers on a large project implement similar containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug. An advantage of the STL is that you can reuse the STL containers, iterators and algorithms to implement common data structures and manipulations project-wide. This reuse can save substantial development time, money and effort.

Software Engineering Observation 22.2

Avoid reinventing the wheel; program with the reusable components of the C++ Standard Library.

Error-Prevention Tip 22.1

The prepackaged, templatized containers of the STL are sufficient for most applications. Using the STL helps you reduce testing and debugging time.

22.2 Introduction to Containers

The STL container types are shown in Fig. 22.1. The containers are divided into three major categories—**sequence containers**, **associative containers** and **container adapters**.

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 22.1 | Standard Library container classes.

STL Containers Overview

The *sequence containers* represent *linear* data structures, such as vectors and linked lists. *Associative containers* are *nonlinear* containers that typically can locate elements stored in the containers quickly. Such containers can store sets of values or **key/value pairs**. The sequence containers and associative containers are collectively referred to as the *first-class containers*. As we saw in Chapter 20, stacks and queues actually are constrained versions of sequential containers. For this reason, STL implements stacks and queues as *container adapters* that enable a program to view a sequential container in a constrained manner. There are other container types that are considered "near containers"—C-like pointer-based arrays (discussed in Chapter 7), `bitset`s for maintaining sets of flag values and `valarray`s for performing high-speed mathematical vector operations (this last class is optimized for computation performance and is not as flexible as the first-class containers). These types are considered "near containers" because they exhibit capabilities similar to those of the first-class containers, but do not support all the first-class-container capabilities. Type `string` supports the same functionality as a sequence container, but stores only character data.

STL Container Common Functions

Most STL containers provide similar functionality. Many generic operations, such as member function `size`, apply to all containers, and other operations apply to subsets of similar containers. This encourages extensibility of the STL with new classes. Figure 22.2 describes the many functions common to all Standard Library containers. [*Note:* Overloaded operators <, <=, >, >=, == and != are not provided for priority_queues.]

Member function	Description
default constructor	A constructor that initializes an empty container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.
empty	Returns true if there are no elements in the container; otherwise, returns false.
insert	Inserts an item in the container.
size	Returns the number of elements currently in the container.
operator=	Assigns one container to another.
operator<	Returns true if the contents of the first container is less than the second; otherwise, returns false.
operator<=	Returns true if the contents of the first container is less than or equal to the second; otherwise, returns false.
operator>	Returns true if the contents of the first container is greater than the second; otherwise, returns false.
operator>=	Returns true if the contents of the first container is greater than or equal to the second; otherwise, returns false.
operator==	Returns true if the contents of the first container is equal to the second; otherwise, returns false.
operator!=	Returns true if the contents of the first container is not equal to the second; otherwise, returns false.
swap	Swaps the elements of two containers.
Functions found only in first-class containers	
max_size	Returns the maximum number of elements for a container.
begin	The two versions of this function return either an iterator or a const_iterator that refers to the first element of the container.
end	The two versions of this function return either an iterator or a const_iterator that refers to the next position after the end of the container.
rbegin	The two versions of this function return either a reverse_iterator or a const_reverse_iterator that refers to the last element of the container.
rend	The two versions of this function return either a reverse_iterator or a const_reverse_iterator that refers to next position after the last element of the container.

Fig. 22.2 | Common member functions for most STL containers. (Part 1 of 2.)

Member function	Description
`erase`	Erases one or more elements from the container.
`clear`	Erases all elements from the container.

Fig. 22.2 | Common member functions for most STL containers. (Part 2 of 2.)

STL Container Headers

The headers for each of the Standard Library containers are shown in Fig. 22.3. The contents of these headers are all in `namespace std`.

First-Class Container Common `typedef`s

Figure 22.4 shows the common `typedef`s (to create synonyms or aliases for lengthy type names) found in first-class containers. These `typedef`s are used in generic declarations of variables, parameters to functions and return values from functions. For example, `value_type` in each container is always a `typedef` that represents the type of elements stored in the container.

Standard Library container headers	
`<vector>`	
`<list>`	
`<deque>`	
`<queue>`	Contains both `queue` and `priority_queue`.
`<stack>`	
`<map>`	Contains both `map` and `multimap`.
`<set>`	Contains both `set` and `multiset`.
`<valarray>`	
`<bitset>`	

Fig. 22.3 | Standard Library container headers.

`typedef`	Description
`allocator_type`	The type of the object used to allocate the container's memory.
`value_type`	The type of element stored in the container.
`reference`	A reference for the container's element type.
`const_reference`	A constant reference for the container's element type. Such a reference can be used only for *reading* elements in the container and for performing `const` operations.
`pointer`	A pointer for the container's element type.

Fig. 22.4 | `typedef`s found in first-class containers. (Part 1 of 2.)

`typedef`	Description
`const_pointer`	A pointer for a constant of the container's element type.
`iterator`	An iterator that points to an element of the container's element type.
`const_iterator`	A constant iterator that points to an element of the container's element type and can be used only to *read* elements.
`reverse_iterator`	A reverse iterator that points to an element of the container's element type. This type of iterator is for iterating through a container in reverse.
`const_reverse_iterator`	A constant reverse iterator that points to an element of the container's element type and can be used only to *read* elements. This type of iterator is for iterating through a container in reverse.
`difference_type`	The type of the result of subtracting two iterators that refer to the same container (operator `-` is not defined for iterators of `list`s and associative containers).
`size_type`	The type used to count items in a container and index through a sequence container (cannot index through a `list`).

Fig. 22.4 | `typedef`s found in first-class containers. (Part 2 of 2.)

When preparing to use an STL container, it's important to ensure that the type of element being stored in the container supports a minimum set of functionality. When an element is inserted into a container, a copy of that element is made. For this reason, the element type should provide its own *copy constructor* and *assignment operator*. [*Note:* This is required only if *default memberwise copy* and *default memberwise assignment* do not perform proper copy and assignment operations for the element type.] Also, the associative containers and many algorithms require elements to be *compared*. For this reason, the element type should provide an *equality operator (==) and a less-than operator (<)*.

22.3 Introduction to Iterators

Iterators have many similarities to *pointers* and are used to point to first-class container elements. Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, the *dereferencing operator (*)* dereferences an iterator so that you can use the element to which it points. The *++ operation on an iterator* moves it to the container's *next element* (much as incrementing a pointer into an array aims the pointer at the next array element).

STL first-class containers provide member functions `begin` and `end`. Function **`begin`** returns an iterator pointing to the first element of the container. Function **`end`** returns an iterator pointing to the *first element past the end of the container* (an element that doesn't exist). If iterator `i` points to a particular element, then `++i` points to the "next" element and `*i` refers to the element pointed to by `i`. The iterator resulting from `end` is typically used in an equality or inequality comparison to determine whether the "moving iterator" (`i` in this case) has reached the end of the container.

An object of type `iterator` refers to a container element that can be modified. An object of type `const_iterator` refers to a container element that *cannot* be modified.

Using `istream_iterator` for Input and `ostream_iterator` for Output

We use iterators with **sequences** (also called **ranges**). These sequences can be in containers, or they can be **input sequences** or **output sequences**. The program of Fig. 22.5 demonstrates input from the standard input (a sequence of data for input into a program), using an **`istream_iterator`**, and output to the standard output (a sequence of data for output from a program), using an **`ostream_iterator`**. The program inputs two integers from the user at the keyboard and displays the sum of the integers. As you'll see later in this chapter, the `istream_iterator` and `ostream_iterator` can be used with the STL algorithms to create powerful statements. For example, you can use an `ostream_iterator` with the `copy` algorithm to copy a container's contents to the standard output stream with a single statement.

```
1  // Fig. 22.5: Fig22_05.cpp
2  // Demonstrating input and output with iterators.
3  #include <iostream>
4  #include <iterator> // ostream_iterator and istream_iterator
5  using namespace std;
6
7  int main()
8  {
9     cout << "Enter two integers: ";
10
11    // create istream_iterator for reading int values from cin
12    istream_iterator< int > inputInt( cin );
13
14    int number1 = *inputInt; // read int from standard input
15    ++inputInt; // move iterator to next input value
16    int number2 = *inputInt; // read int from standard input
17
18    // create ostream_iterator for writing int values to cout
19    ostream_iterator< int > outputInt( cout );
20
21    cout << "The sum is: ";
22    *outputInt = number1 + number2; // output result to cout
23    cout << endl;
24 } // end main
```

```
Enter two integers: 12 25
The sum is: 37
```

Fig. 22.5 | Input and output stream iterators.

Line 12 creates an `istream_iterator` that's capable of extracting (inputting) `int` values in a type-safe manner from the standard input object `cin`. Line 14 dereferences iterator `inputInt` to read the first integer from `cin` and assigns that integer to `number1`. The dereferencing operator `*` applied to iterator `inputInt` gets the value from the stream associated with `inputInt`; this is similar to dereferencing a pointer. Line 15 positions iterator `inputInt` to the next value in the input stream. Line 16 inputs the next integer from `inputInt` and assigns it to `number2`.

Line 19 creates an `ostream_iterator` that's capable of inserting (outputting) `int` values in the standard output object `cout`. Line 22 outputs an integer to `cout` by assigning to `*outputInt` the sum of `number1` and `number2`. Notice the use of the dereferencing operator `*` to use `*outputInt` as an *lvalue* in the assignment statement. If you want to output another value using `outputInt`, the iterator must be incremented with `++` (both the prefix and postfix increment can be used, but the prefix form should be preferred for performance reasons because it does not create a temporary object).

Error-Prevention Tip 22.2

The * *(dereferencing) operator of any* `const` *iterator returns a* `const` *reference to the container element, disallowing the use of non-*`const` *member functions.*

Common Programming Error 22.1

Attempting to create a non-`const` *iterator for a* `const` *container results in a compilation error.*

Iterator Categories and Iterator Category Hierarchy

Figure 22.6 shows the categories of STL iterators. Each category provides a specific set of functionality. Figure 22.7 illustrates the hierarchy of iterator categories. As you follow the hierarchy from top to bottom, each iterator category supports all the functionality of the categories above it in the figure. Thus the "weakest" iterator types are at the top and the most powerful one is at the bottom. Note that this is not an inheritance hierarchy.

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms—the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms—the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of *input and output iterators* and retains their position in the container (as state information).
bidirectional	Combines the capabilities of a *forward iterator* with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a *bidirectional iterator* with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

Fig. 22.6 | Iterator categories.

The iterator category that each container supports determines whether that container can be used with specific algorithms in the STL. *Containers that support random-access iterators can be used with all algorithms in the STL.* As we'll see, pointers into arrays can be used in place of iterators in most STL algorithms, including those that require random-access iterators. Figure 22.8 shows the iterator category of each of the STL containers. The first-class containers (`vector`s, `deque`s, `list`s, `set`s, `multiset`s, `map`s and `multimap`s), `string`s and arrays are all traversable with iterators.

Software Engineering Observation 22.3

Using the "weakest iterator" that yields acceptable performance helps produce maximally reusable components. For example, if an algorithm requires only forward iterators, it can be used with any container that supports forward iterators, bidirectional iterators or random-access iterators. However, an algorithm that requires random-access iterators can be used only with containers that have random-access iterators.

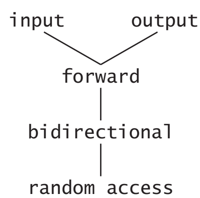


Fig. 22.7 | Iterator category hierarchy.

Container	Type of iterator supported
Sequence containers (first class)	
`vector`	random access
`deque`	random access
`list`	bidirectional
Associative containers (first class)	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Container adapters	
`stack`	no iterators supported
`queue`	no iterators supported
`priority_queue`	no iterators supported

Fig. 22.8 | Iterator types supported by each container.

Predefined Iterator `typedef`*s*

Figure 22.9 shows the predefined iterator `typedef`s that are found in the class definitions of the STL containers. Not every `typedef` is defined for every container. We use `const` versions of the iterators for traversing read-only containers. We use reverse iterators to traverse containers in the reverse direction.

Predefined `typedef`s for iterator types	Direction of `++`	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

Fig. 22.9 | Iterator `typedef`s.

Error-Prevention Tip 22.3

Operations performed on a `const_iterator` *return* `const` *references to prevent modification to elements of the container being manipulated. Using* `const_iterator`*s where appropriate is another example of the* principle of least privilege.

Iterator Operations

Figure 22.10 shows some operations that can be performed on each iterator type. The operations for each iterator type include all operations preceding that type in the figure. For input iterators and output iterators, it's not possible to save the iterator then use the saved value later.

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
`p == p1`	Compare iterators for equality.
`p != p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator.
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 1 of 2.)

Iterator operation	Description
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance between two elements in the same container.
`p[ i ]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator `p` is *before* iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is *before* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is *after* iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is *after* iterator `p1` or *at the same location* as iterator `p1` in the container); otherwise, return `false`.

Fig. 22.10 | Iterator operations for each type of iterator. (Part 2 of 2.)

22.4 Introduction to Algorithms

STL algorithms can be used *generically* across a variety of containers. STL provides many algorithms you'll use frequently to manipulate containers. Inserting, deleting, searching, sorting and others are appropriate for some or all of the STL containers.

The STL includes scores of standard algorithms. We show many of these. *The algorithms operate on container elements only indirectly through iterators.* Many algorithms operate on sequences of elements defined by pairs of iterators—one pointing to the first element of the sequence and one pointing to one element past the last element. Also, it's possible to *create your own new algorithms* that operate in a similar fashion so they can be used with the STL containers and iterators.

Algorithms often return iterators that indicate the results of the algorithms. Algorithm `find`, for example, locates an element and returns an iterator to that element. If the element is not found, `find` returns the "one past the end" iterator that was passed in to define the end of the range to be searched, which can be tested to determine whether an element was not found. The `find` algorithm can be used with any first-class STL container. STL algorithms create yet another opportunity for *reuse*—using the rich collection of popular algorithms can save you much time and effort.

An algorithm can be used with containers that support at least the algorithm's minimum iterator requirements. Some algorithms demand powerful iterators; for example, `sort` demands random-access iterators.

Software Engineering Observation 22.4

The STL is extensible. It's straightforward to add new algorithms and to do so without *changes to STL containers.*

Software Engineering Observation 22.5

The STL is implemented concisely. The algorithms are separated from the containers and operate on elements of the containers only indirectly through iterators. This separation makes it easier to write generic algorithms applicable to many container classes.

Software Engineering Observation 22.6

STL algorithms can operate on STL containers and on pointer-based, C-like arrays.

Portability Tip 22.1

Because STL algorithms process containers only indirectly *through iterators, one algorithm can often be used with many different containers.*

Figure 22.11 shows many of the **mutating-sequence algorithms**—i.e., the algorithms that result in *modifications* of the containers to which the algorithms are applied.

Mutating-sequence algorithms			
`copy`	`partition`	`replace_copy`	`stable_partition`
`copy_backward`	`random_shuffle`	`replace_copy_if`	`swap`
`fill`	`remove`	`replace_if`	`swap_ranges`
`fill_n`	`remove_copy`	`reverse`	`transform`
`generate`	`remove_copy_if`	`reverse_copy`	`unique`
`generate_n`	`remove_if`	`rotate`	`unique_copy`
`iter_swap`	`replace`	`rotate_copy`	

Fig. 22.11 | Mutating-sequence algorithms.

Figure 22.12 shows many of the nonmodifying sequence algorithms—i.e., the algorithms that do not result in modifications of the containers to which they're applied. Figure 22.13 shows the numerical algorithms of the header **`<numeric>`**.

Nonmodifying sequence algorithms			
`adjacent_find`	`equal`	`find_end`	`mismatch`
`count`	`find`	`find_first_of`	`search`
`count_if`	`find_each`	`find_if`	`search_n`

Fig. 22.12 | Nonmodifying sequence algorithms.

Numerical algorithms from header `<numeric>`	
accumulate	partial_sum
inner_product	adjacent_difference

Fig. 22.13 | Numerical algorithms from header `<numeric>`.

22.5 Sequence Containers

The C++ Standard Template Library provides three sequence containers—`vector`, `list` and `deque`. Class template `vector` and class template `deque` both are based on arrays. Class template `list` implements a linked-list data structure similar to our `List` class presented in Chapter 20, but more robust.

One of the most popular containers in the STL is `vector`. Recall that we introduced class template `vector` in Chapter 7 as a more robust type of array. A `vector` changes size dynamically. Unlike C and C++ "raw" arrays (see Chapter 7), `vector`s can be assigned to one another. This is *not* possible with pointer-based, C-like arrays, because those array names are *constant pointers* and thus cannot be the targets of assignments. Just as with C arrays, `vector` subscripting does not perform automatic range checking, but class template `vector` does provide this capability via member function `at` (also discussed in Chapter 7).

Performance Tip 22.1

Insertion at the back of a `vector` *is efficient. The* `vector` *simply grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a* `vector`*—the entire portion of the* `vector` after *the insertion (or deletion) point must be moved, because* `vector` *elements occupy contiguous cells in memory just as C or C++ "raw" arrays do.*

Figure 22.2 presented the operations common to *all* the STL containers. Beyond these operations, each container typically provides a variety of other capabilities. Many of these capabilities are common to several containers, but they're not always equally efficient for each container. You must choose the container most appropriate for the application.

Performance Tip 22.2

Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` *rather than a* `vector`*. Although we can insert and delete elements at the front and back of both a* `vector` *and a* `deque`*, class* `deque` *is more efficient than* `vector` *for doing insertions and deletions at the front.*

Performance Tip 22.3

Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a `list`*, due to its efficient implementation of insertion and deletion anywhere in the data structure.*

In addition to the common operations described in Fig. 22.2, the sequence containers have several other common operations—**`front`** to return a reference to the first element in a non-empty container, **`back`** to return a reference to the last element in a non-empty container, push_back to insert a new element at the end of the container and pop_back to remove the last element of the container.

22.5.1 vector Sequence Container

Class template vector provides a data structure with contiguous memory locations. This enables efficient, direct access to any element of a vector via the subscript operator [], exactly as with a C or C++ "raw" array. Class template vector is most commonly used when the data in the container must be easily accessible via a subscript or will be sorted. When a vector's memory is exhausted, the vector *allocates* a larger contiguous area of memory, *copies* the original elements into the new memory and *deallocates* the old memory.

Performance Tip 22.4
Choose the vector *container for the best random-access performance.*

Performance Tip 22.5
Objects of class template vector *provide rapid indexed access with the overloaded subscript operator* [] *because they're stored in* contiguous memory *like a C or C++ raw array.*

Performance Tip 22.6
It's faster to insert many elements into a container at once than one at a time.

An important part of every container is the type of iterator it supports. This determines which algorithms can be applied to the container. A vector supports random-access iterators—i.e., *all* iterator operations shown in Fig. 22.10 can be applied to a vector iterator. All STL algorithms can operate on a vector. The iterators for a vector are sometimes implemented as pointers to elements of the vector. Each STL algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality. If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that provides forward iterators, bidirectional iterators or random-access iterators. As long as the container supports the algorithm's minimum iterator functionality, the algorithm can operate on the container.

Using Vector and Iterators

Figure 22.14 illustrates several functions of the vector class template. Many of these functions are available in every first-class container. You must include header <vector> to use class template vector.

Line 14 defines an instance called integers of class template vector that stores int values. When this object is instantiated, an empty vector is created with size 0 (i.e., the number of elements stored in the vector) and capacity 0 (i.e., the number of elements that can be stored without allocating more memory to the vector).

Lines 16 and 17 demonstrate the size and capacity functions; each initially returns 0 for vector v in this example. Function size—available in *every* container—returns the number of elements currently stored in the container. Function **capacity** returns the number of elements that can be stored in the vector before the vector needs to *dynamically resize itself* to accommodate more elements.

```
// Fig. 22.14: Fig22_14.cpp
// Demonstrating Standard Library vector class template.
#include <iostream>
#include <vector> // vector class-template definition
using namespace std;

// prototype for function template printVector
template < typename T > void printVector( const vector< T > &integers2 );

int main()
{
   const int SIZE = 6; // define array size
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
   vector< int > integers; // create vector of ints

   cout << "The initial size of integers is: " << integers.size()
      << "\nThe initial capacity of integers is: " << integers.capacity();

   // function push_back is in every sequence container
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );

   cout << "\nThe size of integers is: " << integers.size()
      << "\nThe capacity of integers is: " << integers.capacity();
   cout << "\n\nOutput array using pointer notation: ";

   // display array using pointer notation
   for ( int *ptr = array; ptr != array + SIZE; ++ptr )
      cout << *ptr << ' ';

   cout << "\nOutput vector using iterator notation: ";
   printVector( integers );
   cout << "\nReversed contents of vector integers: ";

   // two const reverse iterators
   vector< int >::const_reverse_iterator reverseIterator;
   vector< int >::const_reverse_iterator tempIterator = integers.rend();

   // display vector in reverse order using reverse_iterator
   for ( reverseIterator = integers.rbegin();
      reverseIterator!= tempIterator; ++reverseIterator )
      cout << *reverseIterator << ' ';

   cout << endl;
} // end main

// function template for outputting vector elements
template < typename T > void printVector( const vector< T > &integers2 )
{
   typename vector< T >::const_iterator constIterator; // const_iterator

```

Fig. 22.14 | Standard Library `vector` class template. (Part 1 of 2.)

```
53      // display vector elements using const_iterator
54      for ( constIterator = integers2.begin();
55         constIterator != integers2.end(); ++constIterator )
56         cout << *constIterator << ' ';
57   } // end function printVector
```

```
The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2
```

Fig. 22.14 | Standard Library `vector` class template. (Part 2 of 2.)

Lines 20–22 use function **`push_back`**—available in all sequence containers—to add an element to the end of the `vector`. If an element is added to a full `vector`, the `vector` increases its size—some STL implementations have the `vector` *double* its capacity.

Performance Tip 22.7

It can be wasteful to double a `vector`'s size when more space is needed. For example, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added. This leaves 999,999 unused elements. You can use `resize` and `reserve` to control space usage better.

Lines 24 and 25 use `size` and `capacity` to illustrate the new size and capacity of the `vector` after the three `push_back` operations. Function `size` returns 3—the number of elements added to the `vector`. Function `capacity` returns 4, indicating that we can add one more element before the `vector` needs to add more memory. When we added the first element, the `vector` allocated space for one element, and the size became 1 to indicate that the `vector` contained only one element. When we added the second element, the capacity doubled to 2 and the size became 2 as well. When we added the third element, the capacity doubled again to 4. So we can actually add another element before the `vector` needs to allocate more space. When the `vector` eventually fills its allocated capacity and the program attempts to add one more element to the `vector`, the `vector` will double its capacity to 8 elements.

The manner in which a `vector` grows to accommodate more elements—a time consuming operation—is *not* specified by the C++ Standard Document. C++ library implementors use various clever schemes to minimize the overhead of resizing a `vector`. Hence, the output of this program may vary, depending on the version of `vector` that comes with your compiler. Some library implementors allocate a large initial capacity. If a `vector` stores a small number of elements, such capacity may be a waste of space. However, it can greatly improve performance if a program adds many elements to a `vector` and does not have to reallocate memory to accommodate those elements. This is a classic space–time trade-off. Library implementors must balance the amount of memory used against the amount of time required to perform various `vector` operations.

Lines 29–30 demonstrate how to output the contents of an array using pointers and pointer arithmetic. Line 33 calls function `printVector` (defined in lines 49–57) to output the contents of a `vector` using iterators. Function template `printVector` receives a `const` reference to a `vector` (`integers2`) as its argument. Line 51 defines a `const_iterator` called `constIterator` that iterates through the `vector` and outputs its contents. Notice that the declaration in line 51 is prefixed with the keyword `typename`. Because `printVector` is a function template and `vector<T>` will be specialized differently for each function-template specialization, the compiler cannot tell at compile time whether or not `vector<T>::const_iterator` is a type. In a particular specialization, `const_iterator` could be a `static` variable. The compiler needs this information to compile the program correctly. Therefore, you must tell the compiler that a qualified name, when the qualifier is a dependent type, is expected to be a type in every specialization.

A `const_iterator` enables the program to read the elements of the `vector`, but does *not* allow the program to *modify* the elements. The `for` statement in lines 54–56 initializes `constIterator` using `vector` member function `begin`, which returns a `const_iterator` to the first element in the `vector`—there's another version of `begin` that returns an `iterator` that can be used for non-`const` containers. A `const_iterator` is returned because the identifier `integers2` was declared `const` in the parameter list of function `printVector`. The loop continues as long as `constIterator` has not reached the end of the `vector`. This is determined by comparing `constIterator` to the result of `integers2.end()`, which returns an iterator indicating the location past the last element of the `vector`. If `constIterator` is equal to this value, the end of the `vector` has been reached. Functions `begin` and `end` are available for all first-class containers. The body of the loop dereferences iterator `constIterator` to get the value in the current element of the `vector`. Remember that the iterator acts like a pointer to the element and that operator `*` is overloaded to return a reference to the element. The expression `++constIterator` (line 55) positions the iterator to the next element of the `vector`.

Performance Tip 22.8

Use prefix increment when applied to STL iterators because the prefix increment operator does not have the overhead of returning a value that must be stored in a temporary object.

Error-Prevention Tip 22.4

Only random-access iterators support `<`. It's better to use `!=` and `end` to test for the end of a container.

Common Programming Error 22.2

Attempting to dereference an iterator positioned outside its container is a runtime logic error. In particular, the iterator returned by `end` cannot be dereferenced or incremented.

Line 37 declares a `const_reverse_iterator` that can be used to iterate through a `vector` *backward*. Line 38 declares a `const_reverse_iterator` variable `tempIterator` and initializes it to the iterator returned by function **`rend`** (i.e., the iterator for the ending point when iterating through the container in reverse). All first-class containers support this type of iterator. Lines 41–43 use a `for` statement similar to that in function `printVector` to iterate through the `vector`. In this loop, function **`rbegin`** (i.e., the iterator for the starting point when iterating through the container in reverse) and `tempIterator` delineate the range of elements to output. As with functions `begin` and `end`, `rbegin` and `rend` can return a `const_reverse_iterator` or a `reverse_iterator`, based on whether or not the container is constant.

Performance Tip 22.9

For performance reasons, capture the loop ending value before the loop and compare against that, rather than having a (potentially expensive) function call for each iteration.

Vector Element-Manipulation Functions

Figure 22.15 illustrates functions that enable retrieval and manipulation of the elements of a `vector`. Line 15 uses an overloaded `vector` constructor that takes two iterators as arguments to initialize `integers`. Remember that pointers into an array can be used as iterators. Line 15 initializes `integers` with the contents of `array` from location `array` up to—but not including—location `array + SIZE`.

```
1  // Fig. 22.15: Fig22_15.cpp
2  // Testing Standard Library vector class template
3  // element-manipulation functions.
4  #include <iostream>
5  #include <vector> // vector class-template definition
6  #include <algorithm> // copy algorithm
7  #include <iterator> // ostream_iterator iterator
8  #include <stdexcept> // out_of_range exception
9  using namespace std;
10
11 int main()
12 {
13    const int SIZE = 6;
14    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
15    vector< int > integers( array, array + SIZE );
16    ostream_iterator< int > output( cout, " " );
17
18    cout << "Vector integers contains: ";
19    copy( integers.begin(), integers.end(), output );
20
21    cout << "\nFirst element of integers: " << integers.front()
22       << "\nLast element of integers: " << integers.back();
23
24    integers[ 0 ] = 7; // set first element to 7
25    integers.at( 2 ) = 10; // set element at position 2 to 10
26
27    // insert 22 as 2nd element
28    integers.insert( integers.begin() + 1, 22 );
29
30    cout << "\n\nContents of vector integers after changes: ";
31    copy( integers.begin(), integers.end(), output );
32
33    // access out-of-range element
34    try
35    {
36       integers.at( 100 ) = 777;
37    } // end try
```

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 1 of 2.)

```
38      catch ( out_of_range &outOfRange ) // out_of_range exception
39      {
40         cout << "\n\nException: " << outOfRange.what();
41      } // end catch
42
43      // erase first element
44      integers.erase( integers.begin() );
45      cout << "\n\nVector integers after erasing first element: ";
46      copy( integers.begin(), integers.end(), output );
47
48      // erase remaining elements
49      integers.erase( integers.begin(), integers.end() );
50      cout << "\nAfter erasing all elements, vector integers "
51         << ( integers.empty() ? "is" : "is not" ) << " empty";
52
53      // insert elements from array
54      integers.insert( integers.begin(), array, array + SIZE );
55      cout << "\n\nContents of vector integers before clear: ";
56      copy( integers.begin(), integers.end(), output );
57
58      // empty integers; clear calls erase to empty a collection
59      integers.clear();
60      cout << "\nAfter clear, vector integers "
61         << ( integers.empty() ? "is" : "is not" ) << " empty" << endl;
62   } // end main
```

```
Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty
```

Fig. 22.15 | `vector` class template element-manipulation functions. (Part 2 of 2.)

Line 16 defines an `ostream_iterator` called `output` that can be used to output integers separated by single spaces via `cout`. An `ostream_iterator< int >` is a type-safe output mechanism that outputs only values of type `int` or a compatible type. The first argument to the constructor specifies the output stream, and the second argument is a string specifying the separator for the values output—in this case, the string contains a space character. We use the `ostream_iterator` (defined in header `<iterator>`) to output the contents of the `vector` in this example.

Line 19 uses algorithm **`copy`** from the Standard Library to output the entire contents of `vector integers` to the standard output. Algorithm `copy` copies each element in the container starting with the location specified by the iterator in its first argument and continuing up to—but *not* including—the location specified by the iterator in its second argument. The first and second arguments must satisfy input iterator requirements—they must be iterators through which values can be read from a container. Also, applying ++ to the first iterator must eventually cause it to reach the second iterator argument in the container. The elements are copied to the location specified by the output iterator (i.e., an iterator through which a value can be stored or output) specified as the last argument. In this case, the output iterator is an ostream_iterator (output) that's attached to cout, so the elements are copied to the standard output. To use the algorithms of the Standard Library, you must include the header **<algorithm>**.

Lines 21–22 use functions front and back (available for all sequence containers) to determine the vector's first and last elements, respectively. Notice the difference between functions front and begin. Function front returns a reference to the first element in the vector, while function begin returns a random access iterator pointing to the first element in the vector. Also notice the difference between functions back and end. Function back returns a reference to the last element in the vector, while function end returns a random access iterator pointing to the end of the vector (the location after the last element).

Common Programming Error 22.3

The vector must not be empty; otherwise, results of the front and back functions are undefined.

Lines 24–25 illustrate two ways to subscript through a vector (which also can be used with the deque containers). Line 26 uses the subscript operator that's overloaded to return either a reference to the value at the specified location or a constant reference to that value, depending on whether the container is constant. Function at (line 25) performs the same operation, but with *bounds checking*. Function at first checks the value supplied as an argument and determines whether it's in the bounds of the vector. If not, function at throws an out_of_range exception defined in header <stdexcept> (as demonstrated in lines 34–41). Figure 22.16 shows some of the STL exception types. (The Standard Library exception types are discussed in Chapter 16.)

STL exception types	Description
out_of_range	Indicates when subscript is out of range—e.g., when an invalid subscript is specified to vector member function at.
invalid_argument	Indicates an invalid argument was passed to a function.
length_error	Indicates an attempt to create too long a container, string, etc.
bad_alloc	Indicates that an attempt to allocate memory with new (or with an allocator) failed because not enough memory was available.

Fig. 22.16 | Some STL exception types.

Line 28 uses one of the three overloaded **insert** functions provided by each sequence container. Line 28 inserts the value 22 before the element at the location specified by the iterator in the first argument. In this example, the iterator is pointing to the second element of the vector, so 22 is inserted as the second element and the original second element becomes the third element of the vector. Other versions of insert allow inserting multiple copies of the same value starting at a particular position in the container, or inserting a range of values from another container (or array), starting at a particular position in the original container.

Lines 44 and 49 use the two **`erase`** functions that are available in all first-class containers. Line 44 indicates that the element at the location specified by the iterator argument should be removed from the container (in this example, the element at the beginning of the `vector`). Line 49 specifies that all elements in the range starting with the location of the first argument up to—but not including—the location of the second argument should be erased from the container. In this example, all the elements are erased from the `vector`. Line 51 uses function **`empty`** (available for all containers and adapters) to confirm that the `vector` is empty.

Common Programming Error 22.4

Erasing an element that contains a pointer to a dynamically allocated object does not `delete` *that object; this can lead to a memory leak.*

Line 54 demonstrates the version of function `insert` that uses the second and third arguments to specify the starting location and ending location in a sequence of values (possibly from another container; in this case, from array of integers `array`) that should be inserted into the `vector`. Remember that the ending location specifies the position in the sequence after the last element to be inserted; copying is performed up to—but *not* including—this location.

Finally, line 59 uses function **`clear`** (found in all first-class containers) to empty the `vector`. This function calls the version of `erase` used in line 51 to empty the `vector`.

[*Note:* Other functions that are common to all containers and common to all sequence containers have not yet been covered. We'll cover most of these in the next few sections. We'll also cover many functions that are specific to each container.]

22.5.2 `list` Sequence Container

The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container. If most of the insertions and deletions occur at the ends of the container, the `deque` data structure (Section 22.5.3) provides a more efficient implementation. Class template `list` is implemented as a *doubly linked list*—every node in the `list` contains a pointer to the previous node in the `list` and to the next node in the `list`. This enables class template `list` to support bidirectional iterators that allow the container to be traversed both forward and backward. Any algorithm that requires input, output, forward or bidirectional iterators can operate on a `list`. Many `list` member functions manipulate the elements of the container as an ordered set of elements.

In addition to the member functions of all STL containers in Fig. 22.2 and the common member functions of all sequence containers discussed in Section 22.5, class template `list` provides nine other member functions—`splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`. Several of these member functions are `list`-optimized implementations of the STL algorithms presented in Section 22.8. Figure 22.17 demonstrates several features of class `list`. Remember that many of the functions presented in Figs. 22.14–22.15 can be used with class `list`. Header **`<list>`** must be included to use class `list`.

```
// Fig. 22.17: Fig22_17.cpp
// Standard library list class template test program.
#include <iostream>
#include <list> // list class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator
using namespace std;

// prototype for function template printList
template < typename T > void printList( const list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values; // create list of ints
   list< int > otherValues; // create list of ints

   // insert items in values
   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

   cout << "values contains: ";
   printList( values );

   values.sort(); // sort values
   cout << "\nvalues after sorting contains: ";
   printList( values );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   cout << "\nAfter insert, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert at end of values
   values.splice( values.end(), otherValues );
   cout << "\nAfter splice, values contains: ";
   printList( values );

   values.sort(); // sort values
   cout << "\nAfter sort, values contains: ";
   printList( values );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   otherValues.sort();
   cout << "\nAfter insert and sort, otherValues contains: ";
   printList( otherValues );

   // remove otherValues elements and insert into values in sorted order
   values.merge( otherValues );
```

Fig. 22.17 | Standard Library `list` class template. (Part 1 of 3.)

```
   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front(); // remove element from front
   values.pop_back(); // remove element from back
   cout << "\nAfter pop_front and pop_back:\n   values contains: "
   printList( values );

   values.unique(); // remove duplicate elements
   cout << "\nAfter unique, values contains: ";
   printList( values );

   // swap elements of values and otherValues
   values.swap( otherValues );
   cout << "\nAfter swap:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   // replace contents of values with elements of otherValues
   values.assign( otherValues.begin(), otherValues.end() );
   cout << "\nAfter assign, values contains: ";
   printList( values );

   // remove otherValues elements and insert into values in sorted order
   values.merge( otherValues );
   cout << "\nAfter merge, values contains: ";
   printList( values );

   values.remove( 4 ); // remove all 4s
   cout << "\nAfter remove( 4 ), values contains: ";
   printList( values );
   cout << endl;
} // end main

// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template < typename T > void printList( const list< T > &listRef )
{
   if ( listRef.empty() ) // list is empty
      cout << "List is empty";
   else
   {
      ostream_iterator< T > output( cout, " " );
      copy( listRef.begin(), listRef.end(), output );
   } // end else
} // end function printList
```

```
values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
```

Fig. 22.17 | Standard Library `list` class template. (Part 2 of 3.)

```
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8r
After unique, values contains: 2 3 4 6 8
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove( 4 ), values contains: 2 2 3 3 6 6 8 8
```

Fig. 22.17 | Standard Library `list` class template. (Part 3 of 3.)

Lines 16–17 instantiate two `list` objects capable of storing integers. Lines 20–21 use function **`push_front`** to insert integers at the beginning of `values`. Function `push_front` is specific to classes `list` and `deque` (not to `vector`). Lines 22–23 use function `push_back` to insert integers at the end of `values`. Remember that *function `push_back` is common to all sequence containers*.

Line 28 uses `list` member function **`sort`** to arrange the elements in the `list` in ascending order. [*Note:* This is different from the `sort` in the STL algorithms.] A second version of function `sort` allows you to supply a binary predicate function that takes two arguments (values in the list), performs a comparison and returns a `bool` value indicating the result. This function determines the order in which the elements of the `list` are sorted. This version could be particularly useful for a `list` that stores pointers rather than values. [*Note:* We demonstrate a unary predicate function in Fig. 22.28. A unary predicate function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.]

Line 38 uses `list` function **`splice`** to remove the elements in `otherValues` and insert them into `values` before the iterator position specified as the first argument. There are two other versions of this function. Function `splice` with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument. Function `splice` with four arguments uses the last two arguments to specify a range of locations that should be removed from the container in the second argument and placed at the location specified in the first argument.

After inserting more elements in `otherValues` and sorting both `values` and `otherValues`, line 53 uses `list` member function **`merge`** to remove all elements of `otherValues` and insert them in sorted order into `values`. Both `list`s must be sorted in the same order before this operation is performed. A second version of `merge` enables you to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value. The predicate function specifies the sorting order used by `merge`.

Line 59 uses `list` function **`pop_front`** to remove the first element in the `list`. Line 60 uses function **`pop_back`** (available for all sequence containers) to remove the last element in the `list`.

Line 64 uses list function **unique** to *remove duplicate elements* in the list. The list should be in *sorted* order (so that all duplicates are side by side) before this operation is performed, to guarantee that all duplicates are eliminated. A second version of unique enables you to supply a predicate function that takes two arguments (values in the list) and returns a bool value specifying whether two elements are equal.

Line 69 uses function **swap** (available to all first-class containers) to exchange the contents of values with the contents of otherValues.

Line 76 uses list function **assign** (available to all sequence containers) to replace the contents of values with the contents of otherValues in the range specified by the two iterator arguments. A second version of assign replaces the original contents with copies of the value specified in the second argument. The first argument of the function specifies the number of copies. Line 85 uses list function **remove** to delete all copies of the value 4 from the list.

22.5.3 deque Sequence Container

Class deque provides many of the benefits of a vector and a list in one container. The term deque is short for "double-ended queue." Class deque is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a vector. Class deque is also implemented for *efficient insertion and deletion operations at its front and back*, much like a list (although a list is also capable of efficient insertions and deletions in the middle of the list). Class deque provides support for random-access iterators, so deques can be used with all STL algorithms. One of the most common uses of a deque is to maintain a first-in, first-out queue of elements. In fact, a deque is the default underlying implementation for the queue adaptor (Section 22.7.2).

Additional storage for a deque can be allocated at either end of the deque in blocks of memory that are typically maintained as an array of pointers to those blocks.[1] Due to the *noncontiguous memory layout* of a deque, a deque iterator must be more intelligent than the pointers that are used to iterate through vectors or pointer-based arrays.

Performance Tip 22.10

In general, deque *has higher overhead than* vector*.*

Performance Tip 22.11

Insertions and deletions in the middle of a deque *are optimized to minimize the number of elements copied, so it's more efficient than a* vector *but less efficient than a* list *for this kind of modification.*

Class deque provides the same basic operations as class vector, but like list adds member functions push_front and pop_front to allow insertion and deletion at the beginning of the deque, respectively.

Figure 22.18 demonstrates features of class deque. Remember that many of the functions presented in Fig. 22.14, Fig. 22.15 and Fig. 22.17 also can be used with class deque. Header **<deque>** must be included to use class deque.

Line 11 instantiates a deque that can store double values. Lines 15–17 use functions push_front and push_back to insert elements at the beginning and end of the deque.

1. This is an implementation-specific detail, not a requirement of the C++ standard.

```
1  // Fig. 22.18: Fig22_18.cpp
2  // Standard Library class deque test program.
3  #include <iostream>
4  #include <deque> // deque class-template definition
5  #include <algorithm> // copy algorithm
6  #include <iterator> // ostream_iterator
7  using namespace std;
8
9  int main()
10 {
11    deque< double > values; // create deque of doubles
12    ostream_iterator< double > output( cout, " " );
13
14    // insert elements in values
15    values.push_front( 2.2 );
16    values.push_front( 3.5 );
17    values.push_back( 1.1 );
18
19    cout << "values contains: ";
20
21    // use subscript operator to obtain elements of values
22    for ( unsigned int i = 0; i < values.size(); ++i )
23       cout << values[ i ] << ' ';
24
25    values.pop_front(); // remove first element
26    cout << "\nAfter pop_front, values contains: ";
27    copy( values.begin(), values.end(), output );
28
29    // use subscript operator to modify element at location 1
30    values[ 1 ] = 5.4;
31    cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
32    copy( values.begin(), values.end(), output );
33    cout << endl;
34 } // end main
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[ 1 ] = 5.4, values contains: 2.2 5.4
```

Fig. 22.18 | Standard Library `deque` class template.

Remember that `push_back` is available for all sequence containers, but `push_front` is available only for class `list` and class `deque`.

The `for` statement in lines 22–23 uses the subscript operator to retrieve the value in each element of the `deque` for output. The condition uses function `size` to ensure that we do not attempt to access an element outside the bounds of the `deque`.

Line 25 uses function `pop_front` to demonstrate removing the first element of the `deque`. Remember that `pop_front` is available only for class `list` and class `deque` (not for class `vector`).

Line 30 uses the subscript operator to create an *lvalue*. This enables values to be assigned directly to any element of the `deque`.

22.6 Associative Containers

The STL's associative containers provide *direct access* to store and retrieve elements via **keys** (often called **search keys**). The four associative containers are multiset, set, multimap and map. Each associative container maintains its keys in *sorted order*. Iterating through an associative container traverses it in the sort order for that container. Classes **multiset** and **set** provide operations for manipulating sets of values where the values are the keys—there is *not* a separate value associated with each key. The primary difference between a multiset and a set is that a multiset allows duplicate keys and a set does not. Classes **multimap** and **map** provide operations for manipulating values associated with keys (these values are sometimes referred to as **mapped values**). The primary difference between a multimap and a map is that a multimap allows duplicate keys with associated values to be stored and a map allows only unique keys with associated values. In addition to the common member functions of all containers presented in Fig. 22.2, all associative containers also support several other member functions, including find, lower_bound, upper_bound and count. Examples of each of the associative containers and the common associative container member functions are presented in the next several subsections.

22.6.1 multiset Associative Container

The multiset associative container provides fast storage and retrieval of keys and allows duplicate keys. The ordering of the elements is determined by a **comparator function object**. For example, in an integer multiset, elements can be sorted in ascending order by ordering the keys with **comparator function object less<int>**. We discuss function objects in detail in Section 22.10. The data type of the keys in all associative containers must support comparison properly based on the comparator function object specified—keys sorted with less<T> must support comparison with operator<. If the keys used in the associative containers are of user-defined data types, those types must supply the appropriate comparison operators. A multiset supports bidirectional iterators (but not random-access iterators).

Figure 22.19 demonstrates the multiset associative container for a multiset of integers sorted in ascending order. Header **<set>** must be included to use class multiset. Containers multiset and set provide the same basic functionality.

Line 10 uses a typedef to create a new type name (alias) for a multiset of integers ordered in ascending order, using the function object less<int>. Ascending order is the default for a multiset, so less<int> can be omitted in line 10. This new type (Ims) is then used to instantiate an integer multiset object, intMultiset (line 16).

Good Programming Practice 22.1

Use typedefs to make code with long type names (such as multisets) easier to read.

```
1 // Fig. 22.19: Fig22_19.cpp
2 // Testing Standard Library class multiset
3 #include <iostream>
4 #include <set> // multiset class-template definition
5 #include <algorithm> // copy algorithm
6 #include <iterator> // ostream_iterator
```

Fig. 22.19 | Standard Library multiset class template. (Part 1 of 3.)

```
using namespace std;

// define short name for multiset type used in this program
typedef multiset< int, less< int > > Ims;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   Ims intMultiset; // Ims is typedef for "integer multiset"
   ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n";

   intMultiset.insert( 15 ); // insert 15 in intMultiset
   intMultiset.insert( 15 ); // insert 15 in intMultiset
   cout << "After inserts, there are " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n\n";

   // iterator that cannot be used to change element values
   Ims::const_iterator result;

   // find 15 in intMultiset; find returns iterator
   result = intMultiset.find( 15 );

   if ( result != intMultiset.end() ) // if iterator not at end
      cout << "Found value 15\n"; // found search value 15

   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );

   if ( result == intMultiset.end() ) // will be true hence
      cout << "Did not find value 20\n"; // did not find 20

   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );
   cout << "\nAfter insert, intMultiset contains:\n";
   copy( intMultiset.begin(), intMultiset.end(), output );

   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: "
      << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );

   // p represents pair of const_iterators
   pair< Ims::const_iterator, Ims::const_iterator > p;

   // use equal_range to determine lower and upper bound
   // of 22 in intMultiset
   p = intMultiset.equal_range( 22 );

```

Fig. 22.19 | Standard Library `multiset` class template. (Part 2 of 3.)

```
59     cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
60        << *( p.first ) << "\n   Upper bound: " << *( p.second );
61     cout << endl;
62  } // end main
```

```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

Fig. 22.19 | Standard Library `multiset` class template. (Part 3 of 3.)

The output statement in line 19 uses function **`count`** (available to all associative containers) to count the number of occurrences of the value `15` currently in the `multiset`.

Lines 22–23 use one of the three versions of function `insert` to add the value `15` to the `multiset` twice. A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified. A third version of `insert` takes two iterators as arguments that specify a range of values to add to the `multiset` from another container.

Line 31 uses function **`find`** (available to all associative containers) to locate the value `15` in the `multiset`. Function `find` returns an `iterator` or a `const_iterator` pointing to the earliest location at which the value is found. If the value is not found, `find` returns an `iterator` or a `const_iterator` equal to the value returned by a call to `end`. Line 40 demonstrates this case.

Line 43 uses function **`insert`** to insert the elements of array `a` into the `multiset`. In line 45, the `copy` algorithm copies the elements of the `multiset` to the standard output in ascending order.

Lines 49 and 50 use functions **`lower_bound`** and **`upper_bound`** (available in all associative containers) to locate the earliest occurrence of the value `22` in the `multiset` and the element *after* the last occurrence of the value `22` in the `multiset`. Both functions return `iterator`s or `const_iterator`s pointing to the appropriate location or the iterator returned by `end` if the value is not in the `multiset`.

Line 53 creates a `pair` object called `p`. Such objects associate pairs of values. In this example, the contents of a `pair` are two `const_iterator`s for our integer-based `multiset`. The purpose of `p` is to store the return value of `multiset` function **`equal_range`** that returns a `pair` containing the results of both a `lower_bound` and an `upper_bound` operation. Type `pair` contains two `public` data members called **`first`** and **`second`**.

Line 57 uses function `equal_range` to determine the `lower_bound` and `upper_bound` of `22` in the `multiset`. Line 60 uses `p.first` and `p.second`, respectively, to access the lower_bound and upper_bound. We dereferenced the iterators to output the values at the locations returned from equal_range.

22.6.2 set Associative Container

The set associative container is used for fast storage and retrieval of unique keys. The implementation of a set is identical to that of a multiset, except that a set must have unique keys. Therefore, if an attempt is made to insert a duplicate key into a set, the duplicate is ignored; because this is the intended mathematical behavior of a set, we do not identify it as a common programming error. A set supports bidirectional iterators (but not random-access iterators). Figure 22.20 demonstrates a set of doubles. Header <set> must be included to use class set.

```
// Fig. 22.20: Fig22_20.cpp
// Standard Library class set test program.
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator> // ostream_iterator
using namespace std;

// define short name for set type used in this program
typedef set< double, less< double > > DoubleSet;

int main()
{
   const int SIZE = 5;
   double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
   DoubleSet doubleSet( a, a + SIZE );
   ostream_iterator< double > output( cout, " " );

   cout << "doubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   // p represents pair containing const_iterator and bool
   pair< DoubleSet::const_iterator, bool > p;

   // insert 13.8 in doubleSet; insert returns pair in which
   // p.first represents location of 13.8 in doubleSet and
   // p.second represents whether 13.8 was inserted
   p = doubleSet.insert( 13.8 ); // value not in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   copy( doubleSet.begin(), doubleSet.end(), output );

   // insert 9.5 in doubleSet
   p = doubleSet.insert( 9.5 ); // value already in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
```

Fig. 22.20 | Standard Library set class template. (Part 1 of 2.)

```
39      copy( doubleSet.begin(), doubleSet.end(), output );
40      cout << endl;
41   } // end main
```

```
doubleSet contains: 2.1 3.7 4.2 9.5

13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
```

Fig. 22.20 | Standard Library set class template. (Part 2 of 2.)

Line 10 uses typedef to create a new type name (DoubleSet) for a set of double values ordered in ascending order, using the function object less<double>.

Line 16 uses the new type DoubleSet to instantiate object doubleSet. The constructor call takes the elements in array a between a and a + SIZE (i.e., the entire array) and inserts them into the set. Line 20 uses algorithm copy to output the contents of the set. Notice that the value 2.1—which appeared twice in array a—appears only once in doubleSet. This is because container set does not allow duplicates.

Line 23 defines a pair consisting of a const_iterator for a DoubleSet and a bool value. This object stores the result of a call to set function insert.

Line 28 uses function insert to place the value 13.8 in the set. The returned pair, p, contains an iterator p.first pointing to the value 13.8 in the set and a bool value that's true if the value was inserted and false if the value was not inserted (because it was already in the set). In this case, 13.8 was not in the set, so it was inserted. Line 35 attempts to insert 9.5, which is already in the set. The output of lines 36–37 shows that 9.5 was not inserted.

22.6.3 multimap Associative Container

The multimap associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs). Many of the functions used with multisets and sets are also used with multimaps and maps. The elements of multimaps and maps are pairs of keys and values instead of individual values. When inserting into a multimap or map, a pair object that contains the key and the value is used. The ordering of the keys is determined by a comparator function object. For example, in a multimap that uses integers as the key type, keys can be sorted in ascending order by ordering them with comparator function object less<int>. Duplicate keys are allowed in a multimap, so multiple values can be associated with a single key. This is called a **one-to-many relationship**. For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many students; in the military, one rank (like "private") has many people. A multimap supports bidirectional iterators, but not random-access iterators. Figure 22.21 demonstrates the multimap associative container. Header **<map>** must be included to use class multimap.

Performance Tip 22.12

A `multimap` is implemented to efficiently locate all values paired with a given key.

Line 8 uses `typedef` to define alias `Mmid` for a `multimap` type in which the key type is `int`, the type of a key's associated value is `double` and the elements are ordered in ascending order. Line 12 uses the new type to instantiate a `multimap` called `pairs`. Line 14 uses function `count` to determine the number of key/value pairs with a key of `15`.

```
1  // Fig. 22.21: Fig22_21.cpp
2  // Standard Library class multimap test program.
3  #include <iostream>
4  #include <map> // multimap class-template definition
5  using namespace std;
6
7  // define short name for multimap type used in this program
8  typedef multimap< int, double, less< int > > Mmid;
9
10 int main()
11 {
12    Mmid pairs; // declare the multimap pairs
13
14    cout << "There are currently " << pairs.count( 15 )
15       << " pairs with key 15 in the multimap\n";
16
17    // insert two value_type objects in pairs
18    pairs.insert( Mmid::value_type( 15, 2.7 ) );
19    pairs.insert( Mmid::value_type( 15, 99.3 ) );
20
21    cout << "After inserts, there are " << pairs.count( 15 )
22       << " pairs with key 15\n\n";
23
24    // insert five value_type objects in pairs
25    pairs.insert( Mmid::value_type( 30, 111.11 ) );
26    pairs.insert( Mmid::value_type( 10, 22.22 ) );
27    pairs.insert( Mmid::value_type( 25, 33.333 ) );
28    pairs.insert( Mmid::value_type( 20, 9.345 ) );
29    pairs.insert( Mmid::value_type( 5, 77.54 ) );
30
31    cout << "Multimap pairs contains:\nKey\tValue\n";
32
33    // use const_iterator to walk through elements of pairs
34    for ( Mmid::const_iterator iter = pairs.begin();
35       iter != pairs.end(); ++iter )
36       cout << iter->first << '\t' << iter->second << '\n';
37
38    cout << endl;
39 } // end main
```

```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15
```

Fig. 22.21 | Standard Library `multimap` class template. (Part 1 of 2.)

```
Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

Fig. 22.21 | Standard Library `multimap` class template. (Part 2 of 2.)

Line 18 uses function `insert` to add a new key/value pair to the `multimap`. The expression `Mmid::value_type( 15, 2.7 )` creates a `pair` object in which `first` is the key (`15`) of type `int` and `second` is the value (`2.7`) of type `double`. The type `Mmid::value_type` is defined as part of the `typedef` for the `multimap`. Line 19 inserts another `pair` object with the key `15` and the value `99.3`. Then lines 21–22 output the number of pairs with key `15`.

Lines 25–29 insert five additional `pair`s into the `multimap`. The `for` statement in lines 34–36 outputs the contents of the `multimap`, including both keys and values. Line 36 uses the `const_iterator` called `iter` to access the members of the `pair` in each element of the `multimap`. Notice in the output that the keys appear in ascending order.

22.6.4 map Associative Container

The `map` associative container performs fast storage and retrieval of unique keys and associated values. Duplicate keys are not allowed—a single value can be associated with each key. This is called a **one-to-one mapping**. For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a `map` that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively. With a `map` you specify the key and get back the associated data quickly. A `map` is also known as an **associative array**. Providing the key in a `map`'s subscript operator `[]` locates the value associated with that key in the `map`. Insertions and deletions can be made anywhere in a `map`.

Figure 22.22 demonstrates a `map` and uses the same features as Fig. 22.21 to demonstrate the subscript operator. Header `<map>` must be included to use class `map`. Lines 31–32 use the subscript operator of class `map`. When the subscript is a key that's already in the `map` (line 31), the operator returns a reference to the associated value. When the subscript is a key that's not in the `map` (line 32), the operator inserts the key in the `map` and returns a reference that can be used to associate a value with that key. Line 31 replaces the value for the key `25` (previously `33.333` as specified in line 19) with a new value, `9999.99`. Line 32 inserts a new key/value `pair` in the `map` (called **creating an association**).

```
1  // Fig. 22.22: Fig22_22.cpp
2  // Standard Library class map test program.
3  #include <iostream>
4  #include <map> // map class-template definition
5  using namespace std;
6
```

Fig. 22.22 | Standard Library `map` class template. (Part 1 of 3.)

```
// define short name for map type used in this program
typedef map< int, double, less< int > > Mid;

int main()
{
   Mid pairs;

   // insert eight value_type objects in pairs
   pairs.insert( Mid::value_type( 15, 2.7 ) );
   pairs.insert( Mid::value_type( 30, 111.11 ) );
   pairs.insert( Mid::value_type( 5, 1010.1 ) );
   pairs.insert( Mid::value_type( 10, 22.22 ) );
   pairs.insert( Mid::value_type( 25, 33.333 ) );
   pairs.insert( Mid::value_type( 5, 77.54 ) ); // dup ignored
   pairs.insert( Mid::value_type( 20, 9.345 ) );
   pairs.insert( Mid::value_type( 15, 99.3 ) ); // dup ignored

   cout << "pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter = pairs.begin();
      iter != pairs.end(); ++iter )
      cout << iter->first << '\t' << iter->second << '\n';

   pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
   pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40

   cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter2 = pairs.begin();
      iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t' << iter2->second << '\n';

   cout << endl;
} // end main
```

```
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After subscript operations, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
```

Fig. 22.22 | Standard Library `map` class template. (Part 2 of 3.)

```
30      111.11
40      8765.43
```

Fig. 22.22 | Standard Library `map` class template. (Part 3 of 3.)

22.7 Container Adapters

The STL provides three **container adapters**—`stack`, `queue` and `priority_queue`. Adapters are not first-class containers, because they do not provide the actual data-structure implementation in which elements can be stored and because adapters do not support iterators. The benefit of an adapter class is that you can choose an appropriate underlying data structure. All three adapter classes provide member functions **`push`** and **`pop`** that properly insert an element into each adapter data structure and properly remove an element from each adapter data structure. The next several subsections provide examples of the adapter classes.

22.7.1 stack Adapter

Class **`stack`** enables insertions into and deletions from the underlying data structure at one end (commonly referred to as a *last-in, first-out* data structure). A `stack` can be implemented with any of the sequence containers: `vector`, `list` and `deque`. This example creates three integer stacks, using each of the sequence containers of the Standard Library as the underlying data structure to represent the `stack`. By default, a `stack` is implemented with a `deque`. The `stack` operations are `push` to insert an element at the top of the `stack` (implemented by calling function `push_back` of the underlying container), `pop` to remove the top element of the `stack` (implemented by calling function `pop_back` of the underlying container), **`top`** to get a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `stack` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `stack` (implemented by calling function `size` of the underlying container).

Performance Tip 22.13

Each of the common operations of a `stack` is implemented as an `inline` function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

Performance Tip 22.14

For the best performance, use class `vector` as the underlying container for a `stack`.

Figure 22.23 demonstrates the `stack` adapter class. Header **`<stack>`** must be included to use class `stack`. Lines 18, 21 and 24 instantiate three integer stacks. Line 18 specifies a `stack` of integers that uses the default `deque` container as its underlying data structure. Line 21 specifies a `stack` of integers that uses a `vector` of integers as its underlying data structure. Line 24 specifies a `stack` of integers that uses a `list` of integers as its underlying data structure.

Function `pushElements` (lines 46–53) pushes the elements onto each `stack`. Line 50 uses function `push` (available in each adapter class) to place an integer on top of the `stack`. Line 51 uses `stack` function `top` to retrieve the top element of the `stack` for output. Function `top` *does not remove the top element.*

Function `popElements` (lines 56–63) pops the elements off each `stack`. Line 60 uses `stack` function `top` to retrieve the top element of the `stack` for output. Line 61 uses function `pop` (available in each adapter class) to remove the top element of the `stack`. Function `pop` does not return a value.

```
1  // Fig. 22.23: Fig22_23.cpp
2  // Standard Library adapter stack test program.
3  #include <iostream>
4  #include <stack> // stack adapter definition
5  #include <vector> // vector class-template definition
6  #include <list> // list class-template definition
7  using namespace std;
8
9  // pushElements function-template prototype
10 template< typename T > void pushElements( T &stackRef );
11
12 // popElements function-template prototype
13 template< typename T > void popElements( T &stackRef );
14
15 int main()
16 {
17    // stack with default underlying deque
18    stack< int > intDequeStack;
19
20    // stack with underlying vector
21    stack< int, vector< int > > intVectorStack;
22
23    // stack with underlying list
24    stack< int, list< int > > intListStack;
25
26    // push the values 0-9 onto each stack
27    cout << "Pushing onto intDequeStack: ";
28    pushElements( intDequeStack );
29    cout << "\nPushing onto intVectorStack: ";
30    pushElements( intVectorStack );
31    cout << "\nPushing onto intListStack: ";
32    pushElements( intListStack );
33    cout << endl << endl;
34
35    // display and remove elements from each stack
36    cout << "Popping from intDequeStack: ";
37    popElements( intDequeStack );
38    cout << "\nPopping from intVectorStack: ";
39    popElements( intVectorStack );
40    cout << "\nPopping from intListStack: ";
41    popElements( intListStack );
42    cout << endl;
```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 1 of 2.)

```
} // end main

// push elements onto stack object to which stackRef refers
template< typename T > void pushElements( T &stackRef )
{
   for ( int i = 0; i < 10; ++i )
   {
      stackRef.push( i ); // push element onto stack
      cout << stackRef.top() << ' '; // view (and display) top element
   } // end for
} // end function pushElements

// pop elements from stack object to which stackRef refers
template< typename T > void popElements( T &stackRef )
{
   while ( !stackRef.empty() )
   {
      cout << stackRef.top() << ' '; // view (and display) top element
      stackRef.pop(); // remove top element
   } // end while
} // end function popElements
```

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

Fig. 22.23 | Standard Library `stack` adapter class. (Part 2 of 2.)

22.7.2 queue Adapter

Class **`queue`** enables insertions at the back of the underlying data structure and deletions from the front (commonly referred to as a *first-in, first-out* data structure). A `queue` can be implemented with STL data structure `list` or `deque`. By default, a `queue` is implemented with a `deque`. The common `queue` operations are `push` to insert an element at the back of the `queue` (implemented by calling function `push_back` of the underlying container), `pop` to remove the element at the front of the `queue` (implemented by calling function `pop_front` of the underlying container), **`front`** to get a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container), **`back`** to get a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `queue` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `queue` (implemented by calling function `size` of the underlying container).

Performance Tip 22.15

For the best performance, use class `deque` *as the underlying container for a* `queue`*.*

Performance Tip 22.16

Each of the common operations of a `queue` *is implemented as an* `inline` *function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.*

Figure 22.24 demonstrates the `queue` adapter class. Header **`<queue>`** must be included to use a `queue`. Line 9 instantiates a `queue` that stores `double` values. Lines 12–14 use function `push` to add elements to the `queue`. The `while` statement in lines 19–23 uses function `empty` (available in all containers) to determine whether the `queue` is empty (line 19). While there are more elements in the `queue`, line 21 uses `queue` function `front` to read (but not remove) the first element in the `queue` for output. Line 22 removes the first element in the `queue` with function `pop` (available in all adapter classes).

```
1  // Fig. 22.24: Fig22_24.cpp
2  // Standard Library adapter queue test program.
3  #include <iostream>
4  #include <queue> // queue adapter definition
5  using namespace std;
6
7  int main()
8  {
9     queue< double > values; // queue with doubles
10
11    // push elements onto queue values
12    values.push( 3.2 );
13    values.push( 9.8 );
14    values.push( 5.4 );
15
16    cout << "Popping from values: ";
17
18    // pop elements from queue
19    while ( !values.empty() )
20    {
21       cout << values.front() << ' '; // view front element
22       values.pop(); // remove element
23    } // end while
24
25    cout << endl;
26 } // end main
```

```
Popping from values: 3.2 9.8 5.4
```

Fig. 22.24 | Standard Library `queue` adapter class templates.

22.7.3 `priority_queue` Adapter

Class **`priority_queue`** provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. A `priority_queue` can be implemented with STL sequence containers `vector` or `deque`. By default, a `priority_queue` is implemented with a `vector` as the underlying container. When elements are added to a `priority_queue`, they're inserted in priority order, such that the highest-priority element (i.e., the largest value) will be the first element removed from the `priority_queue`. This is usually accomplished by arranging the elements in a binary tree structure called a **heap** that always maintains the largest value (i.e., highest-priority element) at the front of the data structure. We discuss the STL's heap algorithms in Section 22.8.12. The comparison of elements is performed with comparator function object `less<T>` by default, but you can supply a different comparator.

There are several common `priority_queue` operations. `push` inserts an element at the appropriate location based on priority order of the `priority_queue` (implemented by calling function `push_back` of the underlying container, then reordering the elements using heapsort). `pop` removes the highest-priority element of the `priority_queue` (implemented by calling function `pop_back` of the underlying container after removing the top element of the heap). **`top`** gets a reference to the top element of the `priority_queue` (implemented by calling function `front` of the underlying container). `empty` determines whether the `priority_queue` is empty (implemented by calling function `empty` of the underlying container). `size` gets the number of elements in the `priority_queue` (implemented by calling function `size` of the underlying container).

Performance Tip 22.17

Each of the common operations of a `priority_queue` is implemented as an `inline` function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

Performance Tip 22.18

For the best performance, use class `vector` as the underlying container for a `priority_queue`.

Figure 22.25 demonstrates the `priority_queue` adapter class. Header `<queue>` must be included to use class `priority_queue`. Line 9 instantiates a `priority_queue` that stores `double` values and uses a `vector` as the underlying data structure. Lines 12–14 use function `push` to add elements to the `priority_queue`. The `while` statement in lines 19–23 uses function `empty` (available in all containers) to determine whether the `priority_queue` is empty (line 19). While there are more elements, line 21 uses `priority_queue` function `top` to retrieve the highest-priority element in the `priority_queue` for output. Line 22 removes the highest-priority element in the `priority_queue` with function `pop` (available in all adapter classes).

```
 1  // Fig. 22.25: Fig22_25.cpp
 2  // Standard Library adapter priority_queue test program.
 3  #include <iostream>
 4  #include <queue> // priority_queue adapter definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     priority_queue< double > priorities; // create priority_queue
10
```

Fig. 22.25 | Standard Library `priority_queue` adapter class. (Part 1 of 2.)

```
    // push elements onto priorities
    priorities.push( 3.2 );
    priorities.push( 9.8 );
    priorities.push( 5.4 );

    cout << "Popping from priorities: ";

    // pop element from priority_queue
    while ( !priorities.empty() )
    {
       cout << priorities.top() << ' '; // view top element
       priorities.pop(); // remove top element
    } // end while

    cout << endl;
 } // end main
```

```
Popping from priorities: 9.8 5.4 3.2
```

Fig. 22.25 | Standard Library `priority_queue` adapter class. (Part 2 of 2.)

22.8 Algorithms

Until the STL, class libraries of containers and algorithms were essentially incompatible among vendors. Early container libraries generally used inheritance and polymorphism, with the associated *overhead of `virtual` function calls*. Early libraries built the algorithms into the container classes as class behaviors. *The STL separates the algorithms from the containers*. This makes it much easier to add new algorithms. With the STL, the elements of containers are accessed through iterators. The next several subsections demonstrate many of the STL algorithms.

Performance Tip 22.19

The STL is implemented for efficiency. It avoids the overhead of `virtual` function calls.

Software Engineering Observation 22.7

STL algorithms do not *depend on the implementation details of the containers on which they operate. As long as the container's (or array's) iterators satisfy the requirements of the algorithm, STL algorithms can work on C-style, pointer-based arrays, on STL containers and on user-defined data structures.*

Software Engineering Observation 22.8

Algorithms can be added easily to the STL without modifying the container classes.

22.8.1 `fill`, `fill_n`, `generate` and `generate_n`

Figure 22.26 demonstrates algorithms `fill`, `fill_n`, `generate` and `generate_n`. Functions **`fill`** and **`fill_n`** set every element in a range of container elements to a specific value. Functions **generate** and **generate_n** use a **generator function** to create values for every element in a range of container elements. The generator function takes no arguments and returns a value that can be placed in an element of the container.

```
// Fig. 22.26: Fig22_26.cpp
// Standard Library algorithms fill, fill_n, generate and generate_n.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

char nextLetter(); // prototype of generator function

int main()
{
   vector< char > chars( 10 );
   ostream_iterator< char > output( cout, " " );
   fill( chars.begin(), chars.end(), '5' ); // fill chars with 5s

   cout << "Vector chars after filling with 5s:\n";
   copy( chars.begin(), chars.end(), output );

   // fill first five elements of chars with As
   fill_n( chars.begin(), 5, 'A' );

   cout << "\n\nVector chars after filling five elements with As:\n";
   copy( chars.begin(), chars.end(), output );

   // generate values for all elements of chars with nextLetter
   generate( chars.begin(), chars.end(), nextLetter );

   cout << "\n\nVector chars after generating letters A-J:\n";
   copy( chars.begin(), chars.end(), output );

   // generate values for first five elements of chars with nextLetter
   generate_n( chars.begin(), 5, nextLetter );

   cout << "\n\nVector chars after generating K-O for the"
      << " first five elements:\n";
   copy( chars.begin(), chars.end(), output );
   cout << endl;
} // end main

// generator function returns next letter (starts with A)
char nextLetter()
{
   static char letter = 'A';
   return ++letter;
} // end function nextLetter
```

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 1 of 2.)

```
Vector chars after filling with 5s:
5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:
A B C D E F G H I J

Vector chars after generating K-O for the first five elements:
K L M N O F G H I J
```

Fig. 22.26 | Algorithms `fill`, `fill_n`, `generate` and `generate_n`. (Part 2 of 2.)

Line 13 defines a 10-element `vector` that stores `char` values. Line 15 uses function `fill` to place the character `'5'` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`. The iterators supplied as the first and second argument must be at least forward iterators (i.e., they can be used for both input from a container and output to a container in the forward direction).

Line 21 uses function `fill_n` to place the character `'A'` in the first five elements of `vector chars`. The iterator supplied as the first argument must be at least an output iterator (i.e., it can be used for output to a container in the forward direction). The second argument specifies the number of elements to fill. The third argument specifies the value to place in each element.

Line 27 uses function `generate` to place the result of a call to generator function `nextLetter` in every element of `vector chars` from `chars.begin()` up to, but not including, `chars.end()`. The iterators supplied as the first and second arguments must be at least forward iterators. Function `nextLetter` (lines 42–46) begins with the character `'A'` maintained in a `static` local variable. The statement in line 45 postincrements the value of `letter` and returns the old value of `letter` each time `nextLetter` is called.

Line 33 uses function `generate_n` to place the result of a call to generator function `nextLetter` in five elements of `vector chars`, starting from `chars.begin()`. The iterator supplied as the first argument must be at least an output iterator.

22.8.2 `equal`, `mismatch` and `lexicographical_compare`

Figure 22.27 demonstrates comparing sequences of values for equality using algorithms `equal`, `mismatch` and `lexicographical_compare`.

```cpp
// Fig. 22.27: Fig22_27.cpp
// Standard Library functions equal, mismatch and lexicographical_compare.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 1 of 3.)

```

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
   vector< int > v1( a1, a1 + SIZE ); // copy of a1
   vector< int > v2( a1, a1 + SIZE ); // copy of a1
   vector< int > v3( a2, a2 + SIZE ); // copy of a2
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output );
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );
   cout << "\nVector v3 contains: ";
   copy( v3.begin(), v3.end(), output );

   // compare vectors v1 and v2 for equality
   bool result = equal( v1.begin(), v1.end(), v2.begin() );
   cout << "\n\nVector v1 " << ( result ? "is" : "is not" )
      << " equal to vector v2.\n";

   // compare vectors v1 and v3 for equality
   result = equal( v1.begin(), v1.end(), v3.begin() );
   cout << "Vector v1 " << ( result ? "is" : "is not" )
      << " equal to vector v3.\n";

   // location represents pair of vector iterators
   pair< vector< int >::iterator, vector< int >::iterator > location;

   // check for mismatch between v1 and v3
   location = mismatch( v1.begin(), v1.end(), v3.begin() );
   cout << "\nThere is a mismatch between v1 and v3 at location "
      << ( location.first - v1.begin() ) << "\nwhere v1 contains "
      << *location.first << " and v3 contains " << *location.second
      << "\n\n";

   char c1[ SIZE ] = "HELLO";
   char c2[ SIZE ] = "BYE BYE";

   // perform lexicographical comparison of c1 and c2
   result = lexicographical_compare( c1, c1 + SIZE, c2, c2 + SIZE );
   cout << c1 << ( result ? " is less than " :
      " is greater than or equal to " ) << c2 << endl;
} // end main
```

```
Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 2 of 3.)

```
There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v3 contains 1000

HELLO is greater than or equal to BYE BYE
```

Fig. 22.27 | Algorithms `equal`, `mismatch` and `lexicographical_compare`. (Part 3 of 3.)

Line 27 uses function **`equal`** to compare two sequences of values for equality. Each sequence need not necessarily contain the same number of elements—`equal` returns `false` if the sequences are not of the same length. The `== operator` (whether built-in or overloaded) performs the comparison of the elements. In this example, the elements in `vector` `v1` from `v1.begin()` up to, but not including, `v1.end()` are compared to the elements in `vector` `v2` starting from `v2.begin()`. In this example, `v1` and `v2` are equal. The three iterator arguments must be at least input iterators (i.e., they can be used for input from a sequence in the forward direction). Line 32 uses function `equal` to compare `vector`s `v1` and `v3`, which are not equal.

There is another version of function `equal` that takes a binary predicate function as a fourth parameter. The binary predicate function receives the two elements being compared and returns a `bool` value indicating whether the elements are equal. This can be useful in sequences that store objects or pointers to values rather than actual values, because you can define one or more comparisons. For example, you can compare `Employee` objects for age, social security number, or location rather than comparing entire objects. You can compare what pointers refer to rather than comparing the pointer values (i.e., the addresses stored in the pointers).

Lines 37–40 begin by instantiating a `pair` of iterators called `location` for a `vector` of integers. This object stores the result of the call to `mismatch` (line 40). Function **`mismatch`** compares two sequences of values and returns a `pair` of iterators indicating the location in each sequence of the mismatched elements. If all the elements match, the two iterators in the `pair` are equal to the last iterator for each sequence. The three iterator arguments must be at least input iterators. Line 42 determines the actual location of the mismatch in the `vector`s with the expression `location.first - v1.begin()`. The result of this calculation is the number of elements between the iterators (this is analogous to pointer arithmetic, which we studied in Chapter 8). This corresponds to the element number in this example, because the comparison is performed from the beginning of each `vector`. As with function `equal`, there is another version of function `mismatch` that takes a binary predicate function as a fourth parameter.

Line 50 uses function **`lexicographical_compare`** to compare the contents of two character arrays. This function's four iterator arguments must be at least input iterators. As you know, pointers into arrays are random-access iterators. The first two iterator arguments specify the range of locations in the first sequence. The last two specify the range of locations in the second sequence. While iterating through the sequences, the `lexicographical_compare` checks if the element in the first sequence is less than the corresponding element in the second sequence. If so, the function returns `true`. If the element in the first sequence is greater than or equal to the element in the second sequence, the function returns `false`. This function can be used to arrange sequences lexicographically. Typically, such sequences contain strings.

22.8.3 remove, remove_if, remove_copy and remove_copy_if

Figure 22.28 demonstrates removing values from a sequence with algorithms remove, remove_if, remove_copy and remove_copy_if.

```
// Fig. 22.28: Fig22_28.cpp
// Standard Library functions remove, remove_if,
// remove_copy and remove_copy_if.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

bool greater9( int ); // prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );
   vector< int > v( a, a + SIZE ); // copy of a
   vector< int >::iterator newLastElement;

   cout << "Vector v before removing all 10s:\n   ";
   copy( v.begin(), v.end(), output );

   // remove all 10s from v
   newLastElement = remove( v.begin(), v.end(), 10 );
   cout << "\nVector v after removing all 10s:\n   ";
   copy( v.begin(), newLastElement, output );

   vector< int > v2( a, a + SIZE ); // copy of a
   vector< int > c( SIZE, 0 ); // instantiate vector c
   cout << "\n\nVector v2 before removing all 10s and copying:\n   ";
   copy( v2.begin(), v2.end(), output );

   // copy from v2 to c, removing 10s in the process
   remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
   cout << "\nVector c after removing all 10s from v2:\n   ";
   copy( c.begin(), c.end(), output );

   vector< int > v3( a, a + SIZE ); // copy of a
   cout << "\n\nVector v3 before removing all elements"
      << "\ngreater than 9:\n   ";
   copy( v3.begin(), v3.end(), output );

   // remove elements greater than 9 from v3
   newLastElement = remove_if( v3.begin(), v3.end(), greater9 );
   cout << "\nVector v3 after removing all elements"
      << "\ngreater than 9:\n   ";
   copy( v3.begin(), newLastElement, output );

```

Fig. 22.28 | Algorithms remove, remove_if, remove_copy and remove_copy_if. (Part 1 of 2.)

```
49     vector< int > v4( a, a + SIZE ); // copy of a
50     vector< int > c2( SIZE, 0 ); // instantiate vector c2
51     cout << "\n\nVector v4 before removing all elements"
52        << "\ngreater than 9 and copying:\n   ";
53     copy( v4.begin(), v4.end(), output );
54
55     // copy elements from v4 to c2, removing elements greater
56     // than 9 in the process
57     remove_copy_if( v4.begin(), v4.end(), c2.begin(), greater9 );
58     cout << "\nVector c2 after removing all elements"
59        << "\ngreater than 9 from v4:\n   ";
60     copy( c2.begin(), c2.end(), output );
61     cout << endl;
62  } // end main
63
64  // determine whether argument is greater than 9
65  bool greater9( int x )
66  {
67     return x > 9;
68  } // end function greater9
```

```
Vector v before removing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v after removing all 10s:
   2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c after removing all 10s from v2:
   2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements
greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after removing all elements
greater than 9:
   2 4 6 8

Vector v4 before removing all elements
greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after removing all elements
greater than 9 from v4:
   2 4 6 8 0 0 0 0 0 0
```

Fig. 22.28 | Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`. (Part 2 of 2.)

Line 24 uses function **`remove`** to eliminate all elements with the value 10 in the range from `v.begin()` up to, but not including, `v.end()` from `v`. The first two iterator arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the `vector` or destroy the eliminated elements, but it does move all elements that are not eliminated toward the beginning of the `vector`. The function returns an iterator positioned after the last `vector` element that was not deleted. Elements from the iterator position to the end of the `vector` have undefined values (in this example, each "undefined" position has value 0).

Line 34 uses function **remove_copy** to copy all elements that do not have the value 10 in the range from v2.begin() up to, but not including, v2.end() from v2. The elements are placed in c, starting at position c.begin(). The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into vector c. Note, in line 29, the use of the vector constructor that receives the number of elements in the vector and the initial values of those elements.

Line 44 uses function **remove_if** to delete all those elements in the range from v3.begin() up to, but not including, v3.end() from v3 for which our user-defined unary predicate function greater9 returns true. Function greater9 (defined in lines 65–68) returns true if the value passed to it is greater than 9; otherwise, it returns false. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the vector, but it does move to the beginning of the vector all elements that are not eliminated. This function returns an iterator positioned after the last element in the vector that was not deleted. All elements from the iterator position to the end of the vector have undefined values.

Line 57 uses function **remove_copy_if** to copy all those elements in the range from v4.begin() up to, but not including, v4.end() from v4 for which the unary predicate function greater9 returns true. The elements are placed in c2, starting at position c2.begin(). The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into c2.

22.8.4 replace, replace_if, replace_copy and replace_copy_if

Figure 22.29 demonstrates replacing values from a sequence using algorithms replace, replace_if, replace_copy and replace_copy_if.

```
// Fig. 22.29: Fig22_29.cpp
// Standard Library functions replace, replace_if,
// replace_copy and replace_copy_if.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator> // ostream_iterator
using namespace std;

bool greater9( int ); // predicate function prototype

int main()
{
   const int SIZE = 10;
```

Fig. 22.29 | Algorithms replace, replace_if, replace_copy and replace_copy_if. (Part 1 of 3.)

```
   int a[ SIZE ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
   ostream_iterator< int > output( cout, " " );

   vector< int > v1( a, a + SIZE ); // copy of a
   cout << "Vector v1 before replacing all 10s:\n   ";
   copy( v1.begin(), v1.end(), output );

   // replace all 10s in v1 with 100
   replace( v1.begin(), v1.end(), 10, 100 );
   cout << "\nVector v1 after replacing 10s with 100s:\n   ";
   copy( v1.begin(), v1.end(), output );

   vector< int > v2( a, a + SIZE ); // copy of a
   vector< int > c1( SIZE ); // instantiate vector c1
   cout << "\n\nVector v2 before replacing all 10s and copying:\n   ";
   copy( v2.begin(), v2.end(), output );

   // copy from v2 to c1, replacing 10s with 100s
   replace_copy( v2.begin(), v2.end(), c1.begin(), 10, 100 );
   cout << "\nVector c1 after replacing all 10s in v2:\n   ";
   copy( c1.begin(), c1.end(), output );

   vector< int > v3( a, a + SIZE ); // copy of a
   cout << "\n\nVector v3 before replacing values greater than 9:\n   ";
   copy( v3.begin(), v3.end(), output );

   // replace values greater than 9 in v3 with 100
   replace_if( v3.begin(), v3.end(), greater9, 100 );
   cout << "\nVector v3 after replacing all values greater"
      << "\nthan 9 with 100s:\n   ";
   copy( v3.begin(), v3.end(), output );

   vector< int > v4( a, a + SIZE ); // copy of a
   vector< int > c2( SIZE ); // instantiate vector c2‘
   cout << "\n\nVector v4 before replacing all values greater "
      << "than 9 and copying:\n   ";
   copy( v4.begin(), v4.end(), output );

   // copy v4 to c2, replacing elements greater than 9 with 100
   replace_copy_if( v4.begin(), v4.end(), c2.begin(), greater9, 100 );
   cout << "\nVector c2 after replacing all values greater "
      << "than 9 in v4:\n   ";
   copy( c2.begin(), c2.end(), output );
   cout << endl;
} // end main

// determine whether argument is greater than 9
bool greater9( int x )
{
   return x > 9;
} // end function greater9
```

Fig. 22.29 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`. (Part 2 of 3.)

```
Vector v1 before replacing all 10s:
   10 2 10 4 16 6 14 8 12 10
Vector v1 after replacing 10s with 100s:
   100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c1 after replacing all 10s in v2:
   100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:
   10 2 10 4 16 6 14 8 12 10
Vector v3 after replacing all values greater
than 9 with 100s:
   100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater than 9 and copying:
   10 2 10 4 16 6 14 8 12 10
Vector c2 after replacing all values greater than 9 in v4:
   100 2 100 4 100 6 100 8 100 100
```

Fig. 22.29 | Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`. (Part 3 of 3.)

Line 23 uses function **`replace`** to replace all elements with the value `10` in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1` with the new value `100`. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

Line 33 uses function **`replace_copy`** to copy all elements in the range from `v2.begin()` up to, but not including, `v2.end()` from `v2`, replacing all elements with the value `10` with the new value `100`. The elements are copied into `c1`, starting at position `c1.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c1`.

Line 42 uses function **`replace_if`** to replace all those elements in the range from `v3.begin()` up to, but not including, `v3.end()` in `v3` for which the unary predicate function `greater9` returns `true`. Function `greater9` (defined in lines 62–65) returns `true` if the value passed to it's greater than 9; otherwise, it returns `false`. The value `100` replaces each value greater than 9. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

Line 54 uses function **`replace_copy_if`** to copy all elements in the range from `v4.begin()` up to, but not including, `v4.end()` from `v4`. Elements for which the unary predicate function `greater9` returns `true` are replaced with the value `100`. The elements are placed in `c2`, starting at position `c2.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c2`.

22.8.5 Mathematical Algorithms

Figure 22.30 demonstrates several common mathematical algorithms from the STL, including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` and `transform`.

```
// Fig. 22.30: Fig22_30.cpp
// Mathematical algorithms of the Standard Library.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <numeric> // accumulate is defined here
#include <vector>
#include <iterator>
using namespace std;

bool greater9( int ); // predicate function prototype
void outputSquare( int ); // output square of a value
int calculateCube( int ); // calculate cube of a value

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   vector< int > v( a1, a1 + SIZE ); // copy of a1
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before random_shuffle: ";
   copy( v.begin(), v.end(), output );

   random_shuffle( v.begin(), v.end() ); // shuffle elements of v
   cout << "\nVector v after random_shuffle: ";
   copy( v.begin(), v.end(), output );

   int a2[ SIZE ] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
   vector< int > v2( a2, a2 + SIZE ); // copy of a2
   cout << "\n\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output );

   // count number of elements in v2 with value 8
   int result = count( v2.begin(), v2.end(), 8 );
   cout << "\nNumber of elements matching 8: " << result;

   // count number of elements in v2 that are greater than 9
   result = count_if( v2.begin(), v2.end(), greater9 );
   cout << "\nNumber of elements greater than 9: " << result;

   // locate minimum element in v2
   cout << "\n\nMinimum element in Vector v2 is: "
      << *( min_element( v2.begin(), v2.end() ) );

   // locate maximum element in v2
   cout << "\nMaximum element in Vector v2 is: "
      << *( max_element( v2.begin(), v2.end() ) );
```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 1 of 2.)

```

   // calculate sum of elements in v
   cout << "\n\nThe total of the elements in Vector v is: "
      << accumulate( v.begin(), v.end(), 0 );

   // output square of every element in v
   cout << "\n\nThe square of every integer in Vector v is:\n";
   for_each( v.begin(), v.end(), outputSquare );

   vector< int > cubes( SIZE ); // instantiate vector cubes

   // calculate cube of each element in v; place results in cubes
   transform( v.begin(), v.end(), cubes.begin(), calculateCube );
   cout << "\n\nThe cube of every integer in Vector v is:\n";
   copy( cubes.begin(), cubes.end(), output );
   cout << endl;
} // end main

// determine whether argument is greater than 9
bool greater9( int value )
{
   return value > 9;
} // end function greater9

// output square of argument
void outputSquare( int value )
{
   cout << value * value << ' ';
} // end function outputSquare

// return cube of argument
int calculateCube( int value )
{
   return value * value * value;
} // end function calculateCube
```

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8
```

Fig. 22.30 | Mathematical algorithms of the Standard Library. (Part 2 of 2.)

Line 24 uses function **`random_shuffle`** to reorder randomly the elements in the range from `v.begin()` up to, but not including, `v.end()` in `v`. This function takes two random-access iterator arguments.

Line 34 uses function **`count`** to count the elements with the value `8` in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`. This function requires its two iterator arguments to be at least input iterators.

Line 38 uses function **`count_if`** to count elements in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2` for which the predicate function `greater9` returns `true`. Function `count_if` requires its two iterator arguments to be at least input iterators.

Line 43 uses function **`min_element`** to locate the smallest element in the range from `v2.begin()` up to, but not including, `v2.end()`. The function returns a forward iterator located at the smallest element, or `v2.end()` if the range is empty. The function's two iterator arguments must be at least input iterators. A second version of this function takes as its third argument a binary function that compares two elements in the sequence. This function returns the `bool` value `true` if the first argument is less than the second.

Good Programming Practice 22.2

It's a good practice to check that the range specified in a call to `min_element` is not empty and that the return value is not the "past the end" iterator.

Line 47 uses function **`max_element`** to locate the largest element in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`. The function returns an input iterator located at the largest element. The function's two iterator arguments must be at least input iterators. A second version of this function takes as its third argument a binary predicate function that compares the elements in the sequence. The binary function takes two arguments and returns the `bool` value `true` if the first argument is less than the second.

Line 51 uses function **`accumulate`** (the template of which is in header <numeric>) to sum the values in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The function's two iterator arguments must be at least input iterators and its third argument represents the initial value of the total. A second version of this function takes as its fourth argument a general function that determines how elements are accumulated. The general function must take two arguments and return a result. The first argument to this function is the current value of the accumulation. The second argument is the value of the current element in the sequence being accumulated.

Line 55 uses function **`for_each`** to apply a general function to every element in the range from `v.begin()` up to, but not including, `v.end()`. The general function takes the current element as an argument and may modify that element (if it's received by reference). Function `for_each` requires its two iterator arguments to be at least input iterators.

Line 60 uses function **`transform`** to apply a general function to every element in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The general function (the fourth argument) should take the current element as an argument, should not modify the element and should return the `transformed` value. Function `transform` requires its first two iterator arguments to be at least input iterators and its third argument to be at least an output iterator. The third argument specifies where the `transformed` values should be placed. Note that the third argument can equal the first. Another version of `transform` accepts five arguments—the first two arguments are input iterators that specify a range of elements from one source container, the third argument is an input iterator that specifies the first element in another source container, the fourth argument is an output iterator that specifies where the transformed values should be placed and the last argument is a general function that takes two arguments. This version of `transform` takes one element from each of the two input sources and applies the general function to that pair of elements, then places the transformed value at the location specified by the fourth argument.

22.8.6 Basic Searching and Sorting Algorithms

Figure 22.31 demonstrates some basic searching and sorting capabilities of the Standard Library, including `find`, `find_if`, `sort` and `binary_search`.

```
// Fig. 22.31: Fig22_31.cpp
// Standard Library search and sort algorithms.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator>
using namespace std;

bool greater10( int value ); // predicate function prototype

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
   vector< int > v( a, a + SIZE ); // copy of a
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output ); // display output vector

   // locate first occurrence of 16 in v
   vector< int >::iterator location;
   location = find( v.begin(), v.end(), 16 );

   if ( location != v.end() ) // found 16
      cout << "\n\nFound 16 at location " << ( location - v.begin() );
   else // 16 not found
      cout << "\n\n16 not found";

   // locate first occurrence of 100 in v
   location = find( v.begin(), v.end(), 100 );

   if ( location != v.end() ) // found 100
      cout << "\nFound 100 at location " << ( location - v.begin() );
   else // 100 not found
      cout << "\n100 not found";

   // locate first occurrence of value greater than 10 in v
   location = find_if( v.begin(), v.end(), greater10 );

```

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 1 of 2.)

```
41      if ( location != v.end() ) // found value greater than 10
42         cout << "\n\nThe first value greater than 10 is " << *location
43            << "\nfound at location " << ( location - v.begin() );
44      else // value greater than 10 not found
45         cout << "\n\nNo values greater than 10 were found";
46
47      // sort elements of v
48      sort( v.begin(), v.end() );
49      cout << "\n\nVector v after sort: ";
50      copy( v.begin(), v.end(), output );
51
52      // use binary_search to locate 13 in v
53      if ( binary_search( v.begin(), v.end(), 13 ) )
54         cout << "\n\n13 was found in v";
55      else
56         cout << "\n\n13 was not found in v";
57
58      // use binary_search to locate 100 in v
59      if ( binary_search( v.begin(), v.end(), 100 ) )
60         cout << "\n100 was found in v";
61      else
62         cout << "\n100 was not found in v";
63
64      cout << endl;
65   } // end main
66
67   // determine whether argument is greater than 10
68   bool greater10( int value )
69   {
70      return value > 10;
71   } // end function greater10
```

```
Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v
100 was not found in v
```

Fig. 22.31 | Basic searching and sorting algorithms of the Standard Library. (Part 2 of 2.)

Line 23 uses function **`find`** to locate the value 16 in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The function requires its two iterator arguments to be at least input iterators and returns an input iterator that either is positioned at the first element containing the value or indicates the end of the sequence (as is the case in line 31).

Line 39 uses function **`find_if`** to locate the first value in the range from `v.begin()` up to, but not including, `v.end()` in `v` for which the unary predicate function `greater10` returns `true`. Function `greater10` (defined in lines 68–71) takes an integer and returns a bool value indicating whether the integer argument is greater than 10. Function find_if requires its two iterator arguments to be at least input iterators. The function returns an input iterator that either is positioned at the first element containing a value for which the predicate function returns true or indicates the end of the sequence.

Line 48 uses function **sort** to arrange the elements in the range from v.begin() up to, but not including, v.end() in v in ascending order. The function requires its two iterator arguments to be random-access iterators. A second version of this function takes a third argument that's a binary predicate function taking two arguments that are values in the sequence and returning a bool indicating the sorting order—if the return value is true, the two elements being compared are in sorted order.

Common Programming Error 22.5

Attempting to sort a container by using an iterator other than a random-access iterator is a compilation error. Function sort requires a random-access iterator.

Line 53 uses function **binary_search** to determine whether the value 13 is in the range from v.begin() up to, but not including, v.end() in v. The sequence of values must be sorted in ascending order first. Function binary_search requires its two iterator arguments to be at least forward iterators. The function returns a bool indicating whether the value was found in the sequence. Line 59 demonstrates a call to function binary_search in which the value is not found. A second version of this function takes a fourth argument that's a binary predicate function taking two arguments that are values in the sequence and returning a bool. The predicate function returns true if the two elements being compared are in sorted order. To obtain the location of the search key in the container, use the lower_bound or find algorithms.

22.8.7 swap, iter_swap and swap_ranges

Figure 22.32 demonstrates algorithms swap, iter_swap and swap_ranges for swapping elements. Line 18 uses function **swap** to exchange two values. In this example, the first and second elements of array a are exchanged. The function takes as arguments references to the two values being exchanged.

```
1  // Fig. 22.32: Fig22_32.cpp
2  // Standard Library algorithms iter_swap, swap and swap_ranges.
3  #include <iostream>
4  #include <algorithm> // algorithm definitions
5  #include <iterator>
6  using namespace std;
7
8  int main()
9  {
10    const int SIZE = 10;
11    int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
12    ostream_iterator< int > output( cout, " " );
13
14    cout << "Array a contains:\n   ";
15    copy( a, a + SIZE, output ); // display array a
```

Fig. 22.32 | Demonstrating swap, iter_swap and swap_ranges. (Part 1 of 2.)

```
16
17       // swap elements at locations 0 and 1 of array a
18       swap( a[ 0 ], a[ 1 ] );
19
20       cout << "\nArray a after swapping a[0] and a[1] using swap:\n   ";
21       copy( a, a + SIZE, output ); // display array a
22
23       // use iterators to swap elements at locations 0 and 1 of array a
24       iter_swap( &a[ 0 ], &a[ 1 ] ); // swap with iterators
25       cout << "\nArray a after swapping a[0] and a[1] using iter_swap:\n   ";
26       copy( a, a + SIZE, output );
27
28       // swap elements in first five elements of array a with
29       // elements in last five elements of array a
30       swap_ranges( a, a + 5, a + 5 );
31
32       cout << "\nArray a after swapping the first five elements\n"
33          << "with the last five elements:\n   ";
34       copy( a, a + SIZE, output );
35       cout << endl;
36    } // end main
```

```
Array a contains:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
   2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
   1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
   6 7 8 9 10 1 2 3 4 5
```

Fig. 22.32 | Demonstrating `swap`, `iter_swap` and `swap_ranges`. (Part 2 of 2.)

Line 24 uses function **`iter_swap`** to exchange the two elements. The function takes two forward iterator arguments (in this case, pointers to elements of an array) and exchanges the values in the elements to which the iterators refer.

Line 30 uses function **`swap_ranges`** to exchange the elements from `a` up to, but not including, `a + 5` with the elements beginning at position `a + 5`. The function requires three forward iterator arguments. The first two arguments specify the range of elements in the first sequence that will be exchanged with the elements in the second sequence starting from the iterator in the third argument. In this example, the two sequences of values are in the same array, but the sequences can be from different arrays or containers.

22.8.8 `copy_backward`, `merge`, `unique` and `reverse`

Figure 22.33 demonstrates STL algorithms `copy_backward`, `merge`, `unique` and `reverse`. Line 26 uses function **`copy_backward`** to copy elements in the range from `v1.begin()` up to, but not including, `v1.end()`, placing the elements in `results` by starting from the element before `results.end()` and working toward the beginning of the `vector`. The function returns an iterator positioned at the last element copied into the `results` (i.e., the beginning of `results`, because of the backward copy). The elements are placed in `results` in the same order as v1. This function requires three bidirectional iterator arguments (iterators that can be incremented and decremented to iterate forward and backward through a sequence, respectively). One difference between `copy_backward` and `copy` is that the iterator returned from `copy` is positioned *after* the last element copied and the one returned from `copy_backward` is positioned *at* the last element copied (i.e., the first element in the sequence). Also, `copy_backward` can manipulate overlapping ranges of elements in a container as long as the first element to copy is not in the destination range of elements.

```cpp
// Fig. 22.33: Fig22_33.cpp
// Standard Library functions copy_backward, merge, unique and reverse.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE = 5;
   int a1[ SIZE ] = { 1, 3, 5, 7, 9 };
   int a2[ SIZE ] = { 2, 4, 5, 7, 9 };
   vector< int > v1( a1, a1 + SIZE ); // copy of a1
   vector< int > v2( a2, a2 + SIZE ); // copy of a2
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v1 contains: ";
   copy( v1.begin(), v1.end(), output ); // display vector output
   cout << "\nVector v2 contains: ";
   copy( v2.begin(), v2.end(), output ); // display vector output

   vector< int > results( v1.size() );

   // place elements of v1 into results in reverse order
   copy_backward( v1.begin(), v1.end(), results.end() );
   cout << "\n\nAfter copy_backward, results contains: ";
   copy( results.begin(), results.end(), output );

   vector< int > results2( v1.size() + v2.size() );

   // merge elements of v1 and v2 into results2 in sorted order
   merge( v1.begin(), v1.end(), v2.begin(), v2.end(), results2.begin() );

   cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
   copy( results2.begin(), results2.end(), output );

   // eliminate duplicate values from results2
   vector< int >::iterator endLocation;
   endLocation = unique( results2.begin(), results2.end() );

   cout << "\n\nAfter unique results2 contains:\n";
   copy( results2.begin(), endLocation, output );
```

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 1 of 2.)

```
44
45     cout << "\n\nVector v1 after reverse: ";
46     reverse( v1.begin(), v1.end() ); // reverse elements of v1
47     copy( v1.begin(), v1.end(), output );
48     cout << endl;
49  } // end main
```

```
Vector v1 contains: 1 3 5 7 9
Vector v2 contains: 2 4 5 7 9

After copy_backward, results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1
```

Fig. 22.33 | Demonstrating `copy_backward`, `merge`, `unique` and `reverse`. (Part 2 of 2.)

Line 33 uses function **`merge`** to combine two sorted ascending sequences of values into a third sorted ascending sequence. The function requires five iterator arguments. The first four must be at least input iterators and the last must be at least an output iterator. The first two arguments specify the range of elements in the first sorted sequence (`v1`), the second two arguments specify the range of elements in the second sorted sequence (`v2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged. A second version of this function takes as its sixth argument a binary predicate function that specifies the sorting order.

Line 30 creates vector `results2` with the number of elements `v1.size() + v2.size()`. Using the `merge` function as shown here requires that the sequence where the results are stored be at least the size of the two sequences being merged. If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation, you can use the following statements:

```
vector< int > results2;
merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
   back_inserter( results2 ) );
```

The argument `back_inserter(results2)` uses function template **`back_inserter`** (header `<iterator>`) for the container `results2`. A `back_inserter` calls the container's default `push_back` function to insert an element at the end of the container. If an element is inserted into a container that has no more space available, *the container grows in size*. Thus, the number of elements in the container does not have to be known in advance. There are two other inserters—**`front_inserter`** (to insert an element at the beginning of a container specified as its argument) and **`inserter`** (to insert an element before the iterator supplied as its second argument in the container supplied as its first argument).

Line 40 uses function **`unique`** on the sorted sequence of elements in the range from `results2.begin()` up to, but not including, `results2.end()` in `results2`. After this function is applied to a sorted sequence with duplicate values, only a single copy of each value remains in the sequence. The function takes two arguments that must be at least forward iterators. The function returns an iterator positioned after the last element in the sequence of unique values. The values of all elements in the container after the last unique value are undefined. A second version of this function takes as a third argument a binary predicate function specifying how to compare two elements for equality.

Line 46 uses function **`reverse`** to reverse all the elements in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1`. The function takes two arguments that must be at least bidirectional iterators.

22.8.9 `inplace_merge`, `unique_copy` and `reverse_copy`

Figure 22.34 demonstrates algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. Line 22 uses function **`inplace_merge`** to merge two sorted sequences of elements in the same container. In this example, the elements from `v1.begin()` up to, but not including, `v1.begin() + 5` are merged with the elements from `v1.begin() + 5` up to, but not including, `v1.end()`. This function requires its three iterator arguments to be at least bidirectional iterators. A second version of this function takes as a fourth argument a binary predicate function for comparing elements in the two sequences.

```
1  // Fig. 22.34: Fig22_34.cpp
2  // Standard Library algorithms inplace_merge,
3  // reverse_copy and unique_copy.
4  #include <iostream>
5  #include <algorithm> // algorithm definitions
6  #include <vector> // vector class-template definition
7  #include <iterator> // back_inserter definition
8  using namespace std;
9
10 int main()
11 {
12    const int SIZE = 10;
13    int a1[ SIZE ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
14    vector< int > v1( a1, a1 + SIZE ); // copy of a
15    ostream_iterator< int > output( cout, " " );
16
17    cout << "Vector v1 contains: ";
18    copy( v1.begin(), v1.end(), output );
19
20    // merge first half of v1 with second half of v1 such that
21    // v1 contains sorted set of elements after merge
22    inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
23
24    cout << "\nAfter inplace_merge, v1 contains: ";
25    copy( v1.begin(), v1.end(), output );
26
27    vector< int > results1;
28
29    // copy only unique elements of v1 into results1
30    unique_copy( v1.begin(), v1.end(), back_inserter( results1 ) );
31    cout << "\nAfter unique_copy results1 contains: ";
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 1 of 2.)

```
32     copy( results1.begin(), results1.end(), output );
33
34     vector< int > results2;
35
36     // copy elements of v1 into results2 in reverse order
37     reverse_copy( v1.begin(), v1.end(), back_inserter( results2 ) );
38     cout << "\nAfter reverse_copy, results2 contains: ";
39     copy( results2.begin(), results2.end(), output );
40     cout << endl;
41  } // end main
```

```
Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

Fig. 22.34 | Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. (Part 2 of 2.)

Line 30 uses function **`unique_copy`** to make a copy of all the unique elements in the sorted sequence of values from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are placed into `vector` `results1`. The first two arguments must be at least input iterators and the last must be at least an output iterator. In this example, we did not preallocate enough elements in `results1` to store all the elements copied from `v1`. Instead, we use function `back_inserter` (defined in header `<iterator>`) to add elements to the end of `v1`. The `back_inserter` uses class `vector`'s capability to insert elements at the end of the `vector`. Because the `back_inserter` inserts an element rather than replacing an existing element's value, the `vector` is able to grow to accommodate additional elements. A second version of the `unique_copy` function takes as a fourth argument a binary predicate function for comparing elements for equality.

Line 37 uses function **`reverse_copy`** to make a reversed copy of the elements in the range from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are inserted into `results2` using a `back_inserter` object to ensure that the `vector` can grow to accommodate the appropriate number of elements copied. Function `reverse_copy` requires its first two iterator arguments to be at least bidirectional iterators and its third to be at least an output iterator.

22.8.10 Set Operations

Figure 22.35 demonstrates functions `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union` for manipulating sets of sorted values. To demonstrate that STL functions can be applied to arrays and containers, this example uses only arrays (remember, a pointer into an array is a random-access iterator).

Lines 25 and 31 call function **`includes`**. Function `includes` compares two sets of sorted values to determine whether every element of the second set is in the first set. If so, `includes` returns `true`; otherwise, it returns `false`. The first two iterator arguments must be at least input iterators and must describe the first set of values. In line 25, the first set consists of the elements from `a1` up to, but not including, `a1 + SIZE1`. The last two iterator arguments must be at least input iterators and must describe the second set of values. In this example, the second set consists of the elements from `a2` up to, but not including, `a2`

+ SIZE2. A second version of function includes takes a fifth argument that's a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

```
// Fig. 22.35: Fig22_35.cpp
// Standard Library algorithms includes, set_difference,
// set_intersection, set_symmetric_difference and set_union.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
   int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
   int a3[ SIZE2 ] = { 4, 5, 6, 11, 15 };
   ostream_iterator< int > output( cout, " " );

   cout << "a1 contains: ";
   copy( a1, a1 + SIZE1, output ); // display array a1
   cout << "\na2 contains: ";
   copy( a2, a2 + SIZE2, output ); // display array a2
   cout << "\na3 contains: ";
   copy( a3, a3 + SIZE2, output ); // display array a3

   // determine whether set a2 is completely contained in a1
        includes( a1, a1 + SIZE1, a2, a2 + SIZE2 )
      cout << "\n\na1 includes a2";
   else
      cout << "\n\na1 does not include a2";

   // determine whether set a3 is completely contained in a1
        includes( a1, a1 + SIZE1, a3, a3 + SIZE2 )
      cout << "\na1 includes a3";
   else
      cout << "\na1 does not include a3";

   int difference[ SIZE1 ];

   // determine elements of a1 not in a2
   int *ptr = set_difference( a1, a1 + SIZE1,
      a2, a2 + SIZE2, difference );
   cout << "\n\nset_difference of a1 and a2 is: ";
   copy( difference, ptr, output );

   int intersection[ SIZE1 ];

   // determine elements in both a1 and a2
   ptr = set_intersection( a1, a1 + SIZE1,
      a2, a2 + SIZE2, intersection );
```

Fig. 22.35 | set operations of the Standard Library. (Part 1 of 2.)

```
49      cout << "\n\nset_intersection of a1 and a2 is: ";
50      copy( intersection, ptr, output );
51
52      int symmetric_difference[ SIZE1 + SIZE2 ];
53
54      // determine elements of a1 that are not in a2 and
55      // elements of a2 that are not in a1
56      ptr = set_symmetric_difference( a1, a1 + SIZE1,
57         a3, a3 + SIZE2, symmetric_difference );
58      cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
59      copy( symmetric_difference, ptr, output );
60
61      int unionSet[ SIZE3 ];
62
63      // determine elements that are in either or both sets
64      ptr = set_union( a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet );
65      cout << "\n\nset_union of a1 and a3 is: ";
66      copy( unionSet, ptr, output );
67      cout << endl;
68   } // end main
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15

a1 includes a2
a1 does not include a3

set_difference of a1 and a2 is: 1 2 3 9 10

set_intersection of a1 and a2 is: 4 5 6 7 8

set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15

set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

Fig. 22.35 | `set` operations of the Standard Library. (Part 2 of 2.)

Lines 39–40 use function **`set_difference`** to find the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in ascending order). The elements that are different are copied into the fifth argument (in this case, the array `difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_difference` takes a sixth argument that's a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 47–48 use function **`set_intersection`** to determine the elements from the first set of sorted values that are in the second set of sorted values (both sets of values must be in ascending order). The elements common to both sets are copied into the fifth argument (in this case, array `intersection`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are the same. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_intersection` takes a sixth argument that's a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 56–57 use function **`set_symmetric_difference`** to determine the elements in the first set that are not in the second set and the elements in the second set that are not in the first set (both sets must be in ascending order). The elements that are different are copied from both sets into the fifth argument (the array `symmetric_difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_symmetric_difference` takes a sixth argument that's a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Line 64 uses function **`set_union`** to create a set of all the elements that are in either or both of the two sorted sets (both sets of values must be in ascending order). The elements are copied from both sets into the fifth argument (in this case the array `unionSet`). Elements that appear in both sets are only copied from the first set. The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store the copied elements. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of `set_union` takes a sixth argument that's a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

22.8.11 `lower_bound`, `upper_bound` and `equal_range`

Figure 22.36 demonstrates functions `lower_bound`, `upper_bound` and `equal_range`. Line 22 uses function **`lower_bound`** to find the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the lower bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of function `lower_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

Line 28 uses function **`upper_bound`** to find the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the upper bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of `upper_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

```
// Fig. 22.36: Fig22_36.cpp
// Standard Library functions lower_bound, upper_bound and
// equal_range for a sorted sequence of values.
#include <iostream>
#include <algorithm> // algorithm definitions
#include <vector> // vector class-template definition
#include <iterator> // ostream_iterator
using namespace std;

int main()
{
   const int SIZE = 10;
   int a1[ SIZE ] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
   vector< int > v( a1, a1 + SIZE ); // copy of a1
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v contains:\n";
   copy( v.begin(), v.end(), output );

   // determine lower-bound insertion point for 6 in v
   vector< int >::iterator lower;
   lower = lower_bound( v.begin(), v.end(), 6 );
   cout << "\n\nLower bound of 6 is element "
      << ( lower - v.begin() ) << " of vector v";

   // determine upper-bound insertion point for 6 in v
   vector< int >::iterator upper;
   upper = upper_bound( v.begin(), v.end(), 6 );
   cout << "\nUpper bound of 6 is element "
      << ( upper - v.begin() ) << " of vector v";

   // use equal_range to determine both the lower- and
   // upper-bound insertion points for 6
   pair< vector< int >::iterator, vector< int >::iterator > eq;
   eq = equal_range( v.begin(), v.end(), 6 );
   cout << "\nUsing equal_range:\n   Lower bound of 6 is element "
        << ( eq.first - v.begin() ) << " of vector v";
   cout << "\n   Upper bound of 6 is element "
      << ( eq.second - v.begin() ) << " of vector v";
   cout << "\n\nUse lower_bound to locate the first point\n"
      << "at which 5 can be inserted in order";

   // determine lower-bound insertion point for 5 in v
   lower = lower_bound( v.begin(), v.end(), 5 );
   cout << "\n   Lower bound of 5 is element "
      << ( lower - v.begin() ) << " of vector v";
   cout << "\n\nUse upper_bound to locate the last point\n"
      << "at which 7 can be inserted in order";

```

Fig. 22.36 | Algorithms `lower_bound`, `upper_bound` and `equal_range`. (Part 1 of 2.)

```
50      // determine upper-bound insertion point for 7 in v
51      upper = upper_bound( v.begin(), v.end(), 7 );
52      cout << "\n   Upper bound of 7 is element "
53         << ( upper - v.begin() ) << " of vector v";
54      cout << "\n\nUse equal_range to locate the first and\n"
55         << "last point at which 5 can be inserted in order";
56
57      // use equal_range to determine both the lower- and
58      // upper-bound insertion points for 5
59      eq = equal_range( v.begin(), v.end(), 5 );
60      cout << "\n   Lower bound of 5 is element "
61         << ( eq.first - v.begin() ) << " of vector v";
62      cout << "\n   Upper bound of 5 is element "
63         << ( eq.second - v.begin() ) << " of vector v" << endl;
64   } // end main
```

```
Vector v contains:
2 2 4 4 4 6 6 6 6 8

Lower bound of 6 is element 5 of vector v
Upper bound of 6 is element 9 of vector v
Using equal_range:
   Lower bound of 6 is element 5 of vector v
   Upper bound of 6 is element 9 of vector v

Use lower_bound to locate the first point
at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v

Use upper_bound to locate the last point
at which 7 can be inserted in order
   Upper bound of 7 is element 9 of vector v

Use equal_range to locate the first and
last point at which 5 can be inserted in order
   Lower bound of 5 is element 5 of vector v
   Upper bound of 5 is element 5 of vector v
```

Fig. 22.36 | Algorithms `lower_bound`, `upper_bound` and `equal_range`. (Part 2 of 2.)

Line 35 uses function **`equal_range`** to return a `pair` of forward iterators containing the results of performing both a `lower_bound` and an `upper_bound` operation. The first two arguments must be at least forward iterators. The third is the value for which to locate the equal range. The function returns a `pair` of forward iterators for the lower bound (`eq.first`) and upper bound (`eq.second`), respectively.

Functions `lower_bound`, `upper_bound` and `equal_range` are often used to locate insertion points in sorted sequences. Line 44 uses `lower_bound` to locate the first point at which `5` can be inserted in order in `v`. Line 51 uses `upper_bound` to locate the last point at which `7` can be inserted in order in `v`. Line 59 uses `equal_range` to locate the first and last points at which `5` can be inserted in order in `v`.

22.8.12 Heapsort

Figure 22.37 demonstrates the Standard Library functions for performing the **heapsort sorting algorithm**. Heapsort is a sorting algorithm in which an array of elements is arranged into a special binary tree called a *heap*. The key features of a heap are that the largest element is always at the top of the heap and the values of the children of any node in the binary tree are always less than or equal to that node's value. A heap arranged in this manner is often called a **maxheap**. Heapsort is discussed in detail in computer science courses called "Data Structures" and "Algorithms."

```
// Fig. 22.37: Fig22_37.cpp
// Standard Library algorithms push_heap, pop_heap,
// make_heap and sort_heap.
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
   vector< int > v( a, a + SIZE ); // copy of a
   vector< int > v2;
   ostream_iterator< int > output( cout, " " );

   cout << "Vector v before make_heap:\n";
   copy( v.begin(), v.end(), output );

   make_heap( v.begin(), v.end() ); // create heap from vector v
   cout << "\nVector v after make_heap:\n";
   copy( v.begin(), v.end(), output );

   sort_heap( v.begin(), v.end() ); // sort elements with sort_heap
   cout << "\nVector v after sort_heap:\n";
   copy( v.begin(), v.end(), output );

   // perform the heapsort with push_heap and pop_heap
   cout << "\n\nArray a contains: ";
   copy( a, a + SIZE, output ); // display array a
   cout << endl;

   // place elements of array a into v2 and
   // maintain elements of v2 in heap
   for ( int i = 0; i < SIZE; ++i )
   {
      v2.push_back( a[ i ] );
      push_heap( v2.begin(), v2.end() );
      cout << "\nv2 after push_heap(a[" << i << "]): ";
      copy( v2.begin(), v2.end(), output );
   } // end for

   cout << endl;

```

Fig. 22.37 | Using Standard Library functions to perform a heapsort. (Part 1 of 2.)

```
46     // remove elements from heap in sorted order
47     for ( unsigned int j = 0; j < v2.size(); ++j )
48     {
49        cout << "\nv2 after " << v2[ 0 ] << " popped from heap\n";
50        pop_heap( v2.begin(), v2.end() - j );
51        copy( v2.begin(), v2.end(), output );
52     } // end for
53
54     cout << endl;
55  } // end main
```

```
Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22

v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100
v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100
```

Fig. 22.37 | Using Standard Library functions to perform a heapsort. (Part 2 of 2.)

Line 21 uses function **`make_heap`** to take a sequence of values in the range from `v.begin()` up to, but not including, `v.end()` and create a heap that can be used to produce a sorted sequence. The two iterator arguments must be random-access iterators, so this function will work only with arrays, `vector`s and `deque`s. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 25 uses function **`sort_heap`** to sort a sequence of values in the range from `v.begin()` up to, but not including, `v.end()` that are already arranged in a heap. The two iterator arguments must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 39 uses function **`push_heap`** to add a new value into a heap. We take one element of array `a` at a time, appendit to the end of `vector v2` and perform the `push_heap` operation. If the appended element is the only element in the `vector`, the `vector` is already a heap. Otherwise, function `push_heap` rearranges the `vector` elements into a heap. Each time `push_heap` is called, it assumes that the last element currently in the `vector` (i.e., the one that's appended before the `push_heap` function call) is the element being added to the heap and that all other elements in the `vector` are already arranged as a heap. The two iterator arguments to `push_heap` must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 50 uses **`pop_heap`** to remove the top heap element. This function assumes that the elements in the range specified by its two random-access iterator arguments are already a heap. Repeatedly removing the top heap element results in a sorted sequence of values. Function `pop_heap` swaps the first heap element (`v2.begin()`) with the last heap element (the element before `v2.end() - i`), then ensures that the elements up to, but not including, the last element still form a heap. Notice in the output that, after the `pop_heap` operations, the `vector` is sorted in ascending order. A second version of this function takes as a third argument a binary predicate function for comparing values.

22.8.13 `min` and `max`

Algorithms **`min`** and **`max`** determine the minimum and the maximum of two elements, respectively. Figure 22.38 demonstrates `min` and `max` for `int` and `char` values.

```cpp
// Fig. 22.38: Fig22_38.cpp
// Standard Library algorithms min and max.
#include <iostream>
#include <algorithm>
using namespace std;

int main()
{
   cout << "The minimum of 12 and 7 is: " << min( 12, 7 );
   cout << "\nThe maximum of 12 and 7 is: " << max( 12, 7 );
   cout << "\nThe minimum of 'G' and 'Z' is: " << min( 'G', 'Z' );
   cout << "\nThe maximum of 'G' and 'Z' is: " << max( 'G', 'Z' );
   cout << endl;
} // end main
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

Fig. 22.38 | Algorithms `min` and `max`.

22.8.14 STL Algorithms Not Covered in This Chapter

Figure 22.39 summarizes STL algorithms that are not covered in this chapter.

Algorithm	Description
`inner_product`	Calculate the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
`adjacent_difference`	Beginning with the second element in a sequence, calculate the difference (using operator –) between the current and previous elements, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
`partial_sum`	Calculate a running total (using operator +) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.
`nth_element`	Use three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this algorithm executes, all elements before the partitioning element are less than that element and all elements after the partitioning element are greater than or equal to that element. A second version of this algorithm takes as a fourth argument a binary comparison function.
`partition`	Similar to `nth_element`, but requires less powerful bidirectional iterators, making it more flexible. It requires two bidirectional iterators indicating the range of elements to partition. The third argument is a unary predicate function that helps partition the elements so that all elements for which the predicate is `true` are to the left (toward the beginning of the sequence) of those for which the predicate is `false`. A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns `false`.
`stable_partition`	Similar to `partition` except that this algorithm guarantees that equivalent elements will be maintained in their original order.
`next_permutation`	Next lexicographical permutation of a sequence.
`prev_permutation`	Previous lexicographical permutation of a sequence.
`rotate`	Use three forward iterator arguments to rotate the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 1 of 2.)

Algorithm	Description
`rotate_copy`	Identical to `rotate` except that the results are stored in a separate sequence indicated by the fourth argument—an output iterator. The two sequences must have the same number of elements.
`adjacent_find`	Returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the end of the sequence.
`search`	Searches for a subsequence of elements within a sequence of elements and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`search_n`	Searches a sequence of elements looking for a subsequence in which the values of a specified number of elements have a particular value and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
`partial_sort`	Use three random-access iterators to sort part of a sequence. The first and last arguments indicate the sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The elements from the second argument to the end of the sequence are in an undefined order.
`partial_sort_copy`	Use two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
`stable_sort`	The algorithm is similar to `sort` except that all equivalent elements are maintained in their original order. This sort is $O(n \log n)$ if enough memory is available; otherwise, it's $O(n(\log n)^2)$.

Fig. 22.39 | Algorithms not covered in this chapter. (Part 2 of 2.)

22.9 Class `bitset`

Class **`bitset`** makes it easy to create and manipulate **bit sets**, which are useful for representing a set of bit flags. `bitset`s are fixed in size at compile time. Class `bitset` is an alternate tool for bit manipulation, discussed in Chapter 21. The declaration

```
bitset< size > b;
```

creates `bitset` `b`, in which every bit is initially `0`. The statement

```
b.set( bitNumber );
```

sets bit `bitNumber` of `bitset` `b` "on." The expression `b.set()` sets all bits in `b` "on."

The statement

```
b.reset( bitNumber );
```

sets bit `bitNumber` of `bitset` `b` "off." The expression `b.reset()` sets all bits in `b` "off." The statement

```
b.flip( bitNumber );
```

"flips" bit `bitNumber` of `bitset` `b` (e.g., if the bit is on, `flip` sets it off). The expression `b.flip()` flips all bits in `b`. The statement

```
b[ bitNumber ];
```

returns a reference to the bit `bitNumber` of `bitset` `b`. Similarly,

```
b.at( bitNumber );
```

performs range checking on `bitNumber` first. Then, if `bitNumber` is in range, `at` returns a reference to the bit. Otherwise, `at` throws an `out_of_range` exception. The statement

```
b.test( bitNumber );
```

performs range checking on `bitNumber` first. If `bitNumber` is in range, `test` returns `true` if the bit is on, `false` it's off. Otherwise, `test` throws an `out_of_range` exception. The expression

```
b.size()
```

returns the number of bits in `bitset` `b`. The expression

```
b.count()
```

returns the number of bits that are set in `bitset` `b`. The expression

```
b.any()
```

returns `true` if any bit is set in `bitset` `b`. The expression

```
b.none()
```

returns `true` if none of the bits is set in `bitset` `b`. The expressions

```
b == b1
b != b1
```

compare the two `bitsets` for equality and inequality, respectively.

Each of the bitwise assignment operators `&=`, `|=` and `^=` can be used to combine `bitsets`. For example,

```
b &= b1;
```

performs a bit-by-bit logical AND between `bitsets` `b` and `b1`. The result is stored in `b`. Bitwise logical OR and bitwise logical XOR are performed by

```
b |= b1;
b ^= b2;
```

The expression

```
b >>= n;
```

shifts the bits in `bitset` `b` right by `n` positions. The expression

```
b <<= n;
```

shifts the bits in `bitset` `b` left by `n` positions. The expressions

```
b.to_string()
b.to_ulong()
```

convert `bitset` `b` to a `string` and an `unsigned long`, respectively.

Sieve of Eratosthenes with `bitset`

Figure 22.40 revisits the Sieve of Eratosthenes for finding prime numbers that we discussed in Exercise 7.29. A `bitset` is used instead of an array to implement the algorithm. The program displays all the prime numbers from 2 to 1023, then allows the user to enter a number to determine whether that number is prime.

```
// Fig. 22.40: Fig22_40.cpp
// Using a bitset to demonstrate the Sieve of Eratosthenes.
#include <iostream>
#include <iomanip>
#include <cmath>
#include <bitset> // bitset class definition
using namespace std;

int main()
{
   const int SIZE = 1024;
   int value;
   bitset< SIZE > sieve; // create bitset of 1024 bits
   sieve.flip(); // flip all bits in bitset sieve
   sieve.reset( 0 ); // reset first bit (number 0)
   sieve.reset( 1 ); // reset second bit (number 1)

   // perform Sieve of Eratosthenes
   int finalBit = sqrt( static_cast< double >( sieve.size() ) ) + 1;

   // determine all prime numbers from 2 to 1024
   for ( int i = 2; i < finalBit; ++i )
   {
      if ( sieve.test( i ) ) // bit i is on
      {
         for ( int j = 2 * i; j < SIZE; j += i )
            sieve.reset( j ); // set bit j off
      } // end if
   } // end for

   cout << "The prime numbers in the range 2 to 1023 are:\n";

   // display prime numbers in range 2-1023
   for ( int k = 2, counter = 1; k < SIZE; ++k )
   {
```

Fig. 22.40 | Class `bitset` and the Sieve of Eratosthenes. (Part 1 of 2.)

```
      if ( sieve.test( k ) ) // bit k is on
      {
         cout << setw( 5 ) << k;

         if ( counter++ % 12 == 0 ) // counter is a multiple of 12
            cout << '\n';
      } // end if
   } // end for

   cout << endl;

   // get value from user to determine whether value is prime
   cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
   cin >> value;

   // determine whether user input is prime
   while ( value != -1 )
   {
      if ( sieve[ value ] ) // prime number
         cout << value << " is a prime number\n";
      else // not a prime number
         cout << value << " is not a prime number\n";

      cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
      cin >> value;
   } // end while
} // end main
```

```
The prime numbers in the range 2 to 1023 are:
    2    3    5    7   11   13   17   19   23   29   31   37
   41   43   47   53   59   61   67   71   73   79   83   89
   97  101  103  107  109  113  127  131  137  139  149  151
  157  163  167  173  179  181  191  193  197  199  211  223
  227  229  233  239  241  251  257  263  269  271  277  281
  283  293  307  311  313  317  331  337  347  349  353  359
  367  373  379  383  389  397  401  409  419  421  431  433
  439  443  449  457  461  463  467  479  487  491  499  503
  509  521  523  541  547  557  563  569  571  577  587  593
  599  601  607  613  617  619  631  641  643  647  653  659
  661  673  677  683  691  701  709  719  727  733  739  743
  751  757  761  769  773  787  797  809  811  821  823  827
  829  839  853  857  859  863  877  881  883  887  907  911
  919  929  937  941  947  953  967  971  977  983  991  997
 1009 1013 1019 1021


Enter a value from 2 to 1023 (-1 to end): 389
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1
```

Fig. 22.40 | Class bitset and the Sieve of Eratosthenes. (Part 2 of 2.)

Line 13 creates a `bitset` of `size` bits (`size` is 1024 in this example). By default, all the bits in the `bitset` are set "off." Line 14 calls function **`flip`** to set all bits "on." Numbers 0 and 1 are not prime numbers, so lines 15–16 call function **`reset`** to set bits 0 and 1 "off." Lines 22–29 determine all the prime numbers from 2 to 1023. The integer `finalBit` (line 19) is used to determine when the algorithm is complete. The basic algorithm is that a number is prime if it has no divisors other than 1 and itself. Starting with the number 2, we can eliminate all multiples of that number. The number 2 is divisible only by 1 and itself, so it's prime. Therefore, we can eliminate 4, 6, 8 and so on. The number 3 is divisible only by 1 and itself. Therefore, we can eliminate all multiples of 3 (keep in mind that all even numbers have already been eliminated).

22.10 Function Objects

Many STL algorithms allow you to pass a function pointer into the algorithm to help the algorithm perform its task. For example, the `binary_search` algorithm that we discussed in Section 22.8.6 is overloaded with a version that requires as its fourth parameter a pointer to a function that takes two arguments and returns a `bool` value. The `binary_search` algorithm uses this function to compare the search key to an element in the collection. The function returns `true` if the search key and element being compared are equal; otherwise, the function returns `false`. This enables `binary_search` to search a collection of elements for which the element type does not provide an overloaded equality `==` operator.

The STL's designers made the algorithms more flexible by allowing any algorithm that can receive a function pointer to receive an object of a class that overloads the parentheses operator with a function named `operator()`, provided that the overloaded operator meets the requirements of the algorithm—in the case of `binary_search`, it must receive two arguments and return a `bool`. An object of such a class is known as a **function object** and can be used syntactically and semantically like a function or function pointer—the overloaded parentheses operator is invoked by using a function object's name followed by parentheses containing the arguments to the function. Together, function objects and functions are known as **functors**. Most algorithms can use function objects and functions interchangeably.

Function objects provide several advantages over function pointers. Since function objects are commonly implemented as class templates that are included into each source code file that uses them, the compiler can inline an overloaded `operator()` to improve performance. Also, since they're objects of classes, function objects can have data members that `operator()` can use to perform its task.

Predefined Function Objects of the Standard Template Library

Many predefined function objects can be found in the header **<functional>**. Figure 22.41 lists several of the STL function objects, which are all implemented as class templates. We used the function object `less<T>` in the `set`, `multiset` and `priority_queue` examples, to specify the sorting order for elements in a container.

Using the STL `Accumulate` Algorithm

Figure 22.42 demonstrates the `accumulate` numeric algorithm (discussed in Fig. 22.30) to calculate the sum of the squares of the elements in a `vector`. The fourth argument to `accumulate` is a **binary function object** (that is, a function object for which `operator()`

STL function objects	Type	STL function objects	Type
`divides< T >`	arithmetic	`logical_or< T >`	logical
`equal_to< T >`	relational	`minus< T >`	arithmetic
`greater< T >`	relational	`modulus< T >`	arithmetic
`greater_equal< T >`	relational	`negate< T >`	arithmetic
`less< T >`	relational	`not_equal_to< T >`	relational
`less_equal< T >`	relational	`plus< T >`	arithmetic
`logical_and< T >`	logical	`multiplies< T >`	arithmetic
`logical_not< T >`	logical		

Fig. 22.41 | Function objects in the Standard Library.

takes two arguments) or a function pointer to a **binary function** (that is, a function that takes two arguments). Function `accumulate` is demonstrated twice—once with a function pointer and once with a function object.

```
// Fig. 22.42: Fig22_42.cpp
// Demonstrating function objects.
#include <iostream>
#include <vector> // vector class-template definition
#include <algorithm> // copy algorithm
#include <numeric> // accumulate algorithm
#include <functional> // binary_function definition
#include <iterator> // ostream_iterator
using namespace std;

// binary function adds square of its second argument and the
// running total in its first argument, then returns the sum
int sumSquares( int total, int value )
{
   return total + value * value;
} // end function sumSquares

// binary function class template defines overloaded operator()
// that adds the square of its second argument and running
// total in its first argument, then returns sum
template< typename T >
class SumSquaresClass : public binary_function< T, T, T >
{
public:
   // add square of value to total and return result
   T operator()( const T &total, const T &value )
   {
      return total + value * value;
   } // end function operator()
}; // end class SumSquaresClass
```

Fig. 22.42 | Binary function object. (Part 1 of 2.)

```
31
32 int main()
33 {
34    const int SIZE = 10;
35    int array[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
36    vector< int > integers( array, array + SIZE ); // copy of array
37    ostream_iterator< int > output( cout, " " );
38    int result;
39
40    cout << "vector integers contains:\n";
41    copy( integers.begin(), integers.end(), output );
42
43    // calculate sum of squares of elements of vector integers
44    // using binary function sumSquares
45    result = accumulate( integers.begin(), integers.end(),
46       0, sumSquares );
47
48    cout << "\n\nSum of squares of elements in integers using "
49       << "binary\nfunction sumSquares: " << result;
50
51    // calculate sum of squares of elements of vector integers
52    // using binary function object
53    result = accumulate( integers.begin(), integers.end(),
54       0, SumSquaresClass< int >() );
55
56    cout << "\n\nSum of squares of elements in integers using "
57       << "binary\nfunction object of type "
58       << "SumSquaresClass< int >: " << result << endl;
59 } // end main
```

```
vector integers contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary
function sumSquares: 385

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385
```

Fig. 22.42 | Binary function object. (Part 2 of 2.)

Lines 13–16 define a function `sumSquares` that squares its second argument `value`, adds that square and its first argument `total` and returns the sum. Function `accumulate` will pass each of the elements of the sequence over which it iterates as the second argument to `sumSquares` in the example. On the first call to `sumSquares`, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`; `0` in this program). All subsequent calls to `sumSquares` receive as the first argument the running sum returned by the previous call to `sumSquares`. When `accumulate` completes, it returns the sum of the squares of all the elements in the sequence.

Lines 21–30 define a class `SumSquaresClass` that inherits from the **`binary_function`** class template (in header `<functional>`)—an empty base class for creating function objects in which `operator()` has two parameters and returns a value. The `binary_function` class accepts three type parameters that represent the types of the first argument, second argument and return value of `operator`, respectively. In this example, the type of these parameters is `T` (line 22). On the first call to the function object, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`: `0` in this program) and the second argument will be the first element in `vector integers`. All subsequent calls to `operator` receive as the first argument the result returned by the previous call to the function object, and the second argument will be the next element in the `vector`. When `accumulate` completes, it returns the sum of the squares of all the elements in the `vector`.

Lines 45–46 call function `accumulate` with a pointer to function `sumSquares` as its last argument. The statement in lines 53–54 calls function `accumulate` with an object of class `SumSquaresClass` as the last argument. The expression `SumSquaresClass<int>()` creates an instance of class `SumSquaresClass` (a function object) that's passed to `accumulate`, which sends the object the message (invokes the function) `operator`. The statement could be written as two separate statements, as follows:

```
SumSquaresClass< int > sumSquaresObject;
result = accumulate( integers.begin(), integers.end(),
   0, sumSquaresObject );
```

The first line defines an object of class `SumSquaresClass`. That object is then passed to function `accumulate`.

22.11 Wrap-Up

In this chapter, we introduced the Standard Template Library and discussed its three key components—containers, iterators and algorithms. You learned the STL sequence containers, `vector`, `deque` and `list`, which represent linear data structures. We discussed associative containers, `set`, `multiset`, `map` and `multimap`, which represent nonlinear data structures. You also saw that the container adapters `stack`, `queue` and `priority_queue` can be used to restrict the operations of the sequence containers for the purpose of implementing the specialized data structures represented by the container adapters. We then demonstrated many of the STL algorithms, including mathematical algorithms, basic searching and sorting algorithms and set operations. You learned the types of iterators each algorithm requires and that each algorithm can be used with any container that supports the minimum iterator functionality the algorithm requires. You also learned class `bitset`, which makes it easy to create and manipulate bit sets as a container. Finally, we introduced function objects that work syntactically and semantically like ordinary functions, but offer advantages such as performance and the ability to store data.

The next chapter discusses the new version of the C++ standard, known as C++0x, which will be released in 2011 or 2012. You'll learn about the new libraries and core language features being added to C++.

Summary

Section 22.1 Introduction to the Standard Template Library (STL)

- The Standard Template Library (p. 851) defines powerful, template-based, reusable components for common data structures, and algorithms used to process those data structures.
- The STL has three key components (p. 851)—containers, iterators and algorithms.

- There are three container-class categories (p. 851)—first-class containers, container adapters and near containers.
- STL algorithms are functions that perform such common data manipulations as searching, sorting and comparing elements or entire containers.

Section 22.2 Introduction to Containers

- Containers are divided into sequence containers, associative containers and container adapters (p. 853).
- The sequence containers (p. 853) represent linear data structures, such as vectors and linked lists.
- Associative containers are nonlinear containers that quickly locate elements stored in them, such as sets of values or key/value pairs (p. 853).
- Sequence containers and associative containers are collectively referred to as first-class containers.

Section 22.3 Introduction to Iterators

- First-class container function `begin` (p. 856) returns an iterator pointing to the first element of a container. Function `end` (p. 856) returns an iterator pointer after the container's last element—typically used in a loop to indicate when to terminate processing of the container's elements.
- An `istream_iterator` (p. 857) is capable of extracting values in a type-safe manner from an input stream. An `ostream_iterator` (p. 857) is capable of inserting values in an output stream.
- Input and output iterators (p. 858) can move only in the forward direction one element at a time.
- A forward iterator (p. 858) combines the capabilities of input and output iterators.
- A bidirectional iterator (p. 858) has the capabilities of a forward iterator and can move backwards.
- A random-access iterator (p. 859) has the capabilities of a bidirectional iterator and the ability to directly access any element of the container.

Section 22.4 Introduction to Algorithms

- Containers that support random-access iterators can be used with all algorithms in the STL.

Section 22.5 Sequence Containers

- The STL provides sequence containers `vector`, `list` and `deque`. Class templates `vector` and `deque` both are based on arrays. Class template `list` implements a linked-list data structure.

Section 22.5.1 `vector` Sequence Container

- Function `capacity` (p. 864) returns the number of elements that can be stored in a vector before the vector dynamically resizes itself to accommodate more elements.
- Sequence container function `push_back` (p. 866) adds an element to the end of a container.
- To use the algorithms of the STL, you must include the header `<algorithm>` (p. 870).
- Algorithm `copy` (p. 869) copies each element in a container starting with the location specified by its first iterator argument up to, but not including, the one specified by its second iterator argument.
- Function `front` (p. 863) returns a reference to the first element in a sequence container. Function `begin` returns an iterator pointing to the beginning of a sequence container.
- Function `back` (p. 863) returns a reference to the last element in a sequence container. Function `end` returns an iterator pointing to the element one past the end of a sequence container.
- Sequence container function `insert` (p. 870) inserts value(s) before the element at a specific location.
- Function `erase` (p. 871; in all first-class containers) removes specific element(s) from the container.

- Function `empty` (p. 871; in all containers and adapters) returns `true` if the container is empty.
- Function `clear` (p. 871; in all first-class containers) empties the container.

Section 22.5.2 `list` Sequence Container

- The `list` sequence container (p. 871) provides an efficient implementation for inserting and deleting anywhere in the container. Header `<list>` must be included to use class template `list`.
- `list` member function `push_front` (p. 874) inserts values at the beginning of a list.
- `list` member function `sort` (p. 874) arranges the elements in the list in ascending order.
- `list` member function `splice` (p. 874) removes elements in one `list` and inserts them into another `list` at a specific position.
- `list` member function `unique` (p. 875) removes duplicate elements in a `list`.
- `list` member function `assign` (p. 875) replaces the contents of one `list` with those of another.
- `list` member function `remove` (p. 875) deletes all copies of a specified value from a `list`.

Section 22.5.3 `deque` Sequence Container

- Class template `deque` (p. 875) provides the same operations as `vector`, but adds member functions `push_front` and `pop_front` (p. 874) to allow insertion and deletion at the beginning of a `deque`, respectively. Header `<deque>` must be included to use class template `deque`.

Section 22.6 Associative Containers

- The STL's associative containers provide direct access to store and retrieve elements via keys (p. 877).
- The four associative containers (p. 877) are `multiset`, `set`, `multimap` and `map`.
- Class templates `multiset` and `set` provide operations for manipulating sets of values where the values are the keys—there is not a separate value associated with each key. Header `<set>` must be included to use class templates `set` and `multiset`.
- A `multiset` allows duplicate keys and a `set` does not.

Section 22.6.1 `multiset` Associative Container

- The `multiset` associative container (p. 877) provides fast storage and retrieval of keys and allows duplicate keys. The ordering of the elements is determined by a comparator function object.
- A `multiset`'s keys can be sorted in ascending order by ordering the keys with comparator function object `less<T>` (p. 877).
- The type of the keys in all associative containers must support comparison properly based on the comparator function object specified.
- A `multiset` supports bidirectional iterators.
- Header `<set>` (p. 877) must be included to use class `multiset`.
- Function `count` (p. 879; available to all associative containers) counts the number of occurrences of the specified value currently in a container.
- Function `find` (p. 879; available to all associative containers) locates a specified value in a container.
- Associative container functions `lower_bound` and `upper_bound` (p. 879) locate the earliest occurrence of the specified value in a container and the element after the value's last occurrence, respectively.
- Associative container function `equal_range` (p. 879) returns a `pair` containing the results a `lower_bound` and an `upper_bound` operation.

Section 22.6.2 `set` *Associative Container*

- The `set` associative container is used for fast storage and retrieval of unique keys.
- If an attempt is made to insert a duplicate key into a `set`, the duplicate is ignored.
- A `set` supports bidirectional iterators.
- Header `<set>` must be included to use class set.

Section 22.6.3 `multimap` *Associative Container*

- Containers `multimap` and `map` provide operations for manipulating values associated with keys.
- The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.
- The `multimap` associative container is used for fast storage and retrieval of key/value pairs.
- Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is called a one-to-many relationship.
- Header `<map>` (p. 881) must be included to use class templates `map` and `multimap`.

Section 22.6.4 `map` *Associative Container*

- Duplicate keys are not allowed in a `map`, so only a single value can be associated with each key. This is called a one-to-one mapping (p. 883).
- A `map` is commonly called an associative array (p. 883).

Section 22.7 Container Adapters

- The STL provides three container adapters—`stack`, `queue` and `priority_queue`.
- Adapters are not first-class containers, because they do not provide the actual data structure implementation in which elements can be stored and they do not support iterators.
- All three adapter class templates provide member functions `push` and `pop` (p. 885) that properly insert an element into and remove an element from each adapter data structure, respectively.

Section 22.7.1 `stack` *Adapter*

- Class template `stack` (p. 885) is a last-in, first-out data structure. Header `<stack>` must be included to use class template `stack`.
- The `stack` member function `top` (p. 885) returns a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container).
- The `stack` member function `empty` determines whether the `stack` is empty (implemented by calling function `empty` of the underlying container).
- The `stack` member function `size` returns the number of elements in the `stack` (implemented by calling function `size` of the underlying container).

Section 22.7.2 `queue` *Adapter*

- Class template `queue` (p. 887) implements a FIFO data structure. Header `<queue>` (p. 888) must be included to use a `queue` or a `priority_queue`.
- The `queue` member function `front` returns a reference to the first element in the `queue`.
- The `queue` member function `back` (p. 887) returns a reference to the last element in the `queue`.
- The `queue` member function `empty` determines whether the `queue` is empty.
- The `queue` member function `size` returns the number of elements in the `queue`.

Section 22.7.3 `priority_queue` *Adapter*

- Class template `priority_queue` provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure.
- The common `priority_queue` (p. 888) operations are `push`, `pop`, `top`, `empty` and `size`.

Section 22.8.1 `fill`*,* `fill_n`*,* `generate` *and* `generate_n`

- Algorithms `fill` and `fill_n` (p. 890) set every element in a range of container elements to a specific value.
- Algorithms `generate` and `generate_n` (p. 891) use a generator function or function object to create values for every element in a range of container elements.

Section 22.8.2 `equal`*,* `mismatch` *and* `lexicographical_compare`

- Algorithm `equal` (p. 894) compares two sequences of values for equality.
- Algorithm `mismatch` (p. 894) compares two sequences of values and returns a pair of iterators indicating the location in each sequence of the mismatched elements.
- Algorithm `lexicographical_compare` (p. 894) compares the contents of two sequences.

Section 22.8.3 `remove`*,* `remove_if`*,* `remove_copy` *and* `remove_copy_if`

- Algorithm `remove` (p. 896) eliminates all elements with a specific value in a certain range.
- Algorithm `remove_copy` (p. 897) copies all elements that do not have a specific value in a certain range.
- Algorithm `remove_if` (p. 897) deletes all elements that satisfy the `if` condition in a certain range.
- Algorithm `remove_copy_if` (p. 897) copies all elements that satisfy the `if` condition in a certain range.

Section 22.8.4 `replace`*,* `replace_if`*,* `replace_copy` *and* `replace_copy_if`

- Algorithm `replace` (p. 899) replaces all elements with a specific value in certain range.
- Algorithm `replace_copy` (p. 899) copies all elements with a specific value in a certain range.
- Algorithm `replace_if` (p. 899) replaces all elements that satisfy the `if` condition in a certain range.
- Algorithm `replace_copy_if` (p. 899) copies all elements that satisfy the `if` condition in a certain range.

Section 22.8.5 Mathematical Algorithms

- Algorithm `random_shuffle` (p. 902) reorders randomly the elements in a certain range.
- Algorithm `count` (p. 902) counts the elements with a specific value in a certain range.
- Algorithm `count_if` (p. 902) counts the elements that satisfy the `if` condition in a certain range.
- Algorithm `min_element` (p. 902) locates the smallest element in a certain range.
- Algorithm `max_element` (p. 902) locates the largest element in a certain range.
- Algorithm `accumulate` (p. 902) sums the values in a certain range.
- Algorithm `for_each` (p. 902) applies a general function or function object to every element in a range.
- Algorithm `transform` (p. 902) applies a general function or function object to every element in a range and replaces each element with the result of the function.

Section 22.8.6 Basic Searching and Sorting Algorithms

- Algorithm `find` (p. 904) locates a specific value in a certain range.
- Algorithm `find_if` (p. 904) locates the first value in a certain range that satisfies the `if` condition.
- Algorithm `sort` (p. 905) arranges the elements in a certain range in ascending order or an order specified by a predicate.
- Algorithm `binary_search` (p. 905) if whether a specific value is in a sorted range of elements.

Section 22.8.7 `swap`, `iter_swap` and `swap_ranges`

- Algorithm `swap` (p. 905) exchanges two values.
- Algorithm `iter_swap` (p. 906) exchanges the two elements.
- Algorithm `swap_ranges` (p. 906) exchanges the elements in a certain range.

Section 22.8.8 `copy_backward`, `merge`, `unique` and `reverse`

- Algorithm `copy_backward` (p. 906) copies elements in a range and places the elements into a container starting from the end and working toward the front.
- Algorithm `merge` (p. 908) combines two sorted ascending sequences of values into a third sorted ascending sequence.
- Algorithm `unique` (p. 908) removes duplicated elements in a certain range of a sorted sequence.
- Algorithm `reverse` (p. 909) reverses all the elements in a certain range.

Section 22.8.9 `inplace_merge`, `unique_copy` and `reverse_copy`

- Algorithm `inplace_merge` (p. 909) merges two sorted sequences of elements in the same container.
- Algorithm `unique_copy` (p. 910) makes a copy of all the unique elements in the sorted sequence of values in a certain range.
- Algorithm `reverse_copy` (p. 910) makes a reversed copy of the elements in a certain range.

Section 22.8.10 Set Operations

- The `set` function `includes` compares two `set`s of sorted values to determine whether every element of the second `set` is in the first `set`.
- The `set` function `set_difference` (p. 912) finds the elements from the first `set` of sorted values that are not in the second `set` of sorted values (both `set`s of values must be in ascending order).
- The `set` function `set_intersection` (p. 912) determines the elements from the first `set` of sorted values that are in the second `set` of sorted values (both `set`s of values must be in ascending order).
- The `set` function `set_symmetric_difference` (p. 913) determines the elements in the first `set` that are not in the second `set` and the elements in the second `set` that are not in the first `set` (both `set`s of values must be in ascending order).
- The `set` function `set_union` (p. 913) creates a `set` of all the elements that are in either or both of the two sorted `set`s (both `set`s of values must be in ascending order).

Section 22.8.11 `lower_bound`, `upper_bound` and `equal_range`

- Algorithm `lower_bound` (p. 913) finds the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.
- Algorithm `upper_bound` (p. 913) finds the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.

- Algorithm `equal_range` (p. 915) performs returns the lower bound and upper bound as a `pair`.

Section 22.8.12 Heapsort

- Algorithm `make_heap` (p. 917) takes a sequence of values in a certain range and creates a heap that can be used to produce a sorted sequence.
- Algorithm `sort_heap` (p. 918) sorts a sequence of values in a certain range of a heap.
- Algorithm `pop_heap` (p. 918) removes the top heap element.

Section 22.8.13 `min` and `max`

- Algorithms `min` and `max` (p. 918) determine the minimum of two elements and the maximum of two elements, respectively.

Section 22.9 Class `bitset`

- Class template `bitset` (p. 920) makes it easy to create and manipulate bit sets, which are useful for representing a set of bit flags.

Section 22.10 Function Objects

- A function object (p. 924) is an instance of a class that overloads `operator()`.
- The STL provides many predefined function objects, which can be found in header `<functional>` (p. 924).
- Binary function objects (p. 924) take two arguments and return a value. The `binary_function` class template (p. 926) is an empty base class for creating binary function objects that provides standard type names for the function's parameters and result.

Self-Review Exercises

State whether the following are *true* or *false* , or fill in the blanks. If the answer is *false*, explain why,.

22.1 (T/F) The STL makes abundant use of inheritance and `virtual` functions.

22.2 The two types of first-class STL containers are sequence containers and ________ containers.

22.3 The five main iterator types are ________, ________, ________, ________ and ________.

22.4 (T/F) An iterator acts like a pointer to an element.

22.5 (T/F) STL algorithms can operate on C-like pointer-based arrays.

22.6 (T/F) STL algorithms are encapsulated as member functions within each container class.

22.7 (T/F) When using the `remove` algorithm on a `vector`, the algorithm does not decrease the size of the `vector` from which elements are being removed.

22.8 The three STL container adapters are ________, ________ and ________.

22.9 (T/F) Container member function `end` yields the position of the container's last element.

22.10 STL algorithms operate on container elements indirectly, using ________.

22.11 The `sort` algorithm requires a(n) ________ iterator.

Answers to Self-Review Exercises

22.1 False. These were avoided for performance reasons.

22.2 Associative.

22.3 Input, output, forward, bidirectional, random access.

22.4 True.

22.5 True.

22.6 False. STL algorithms are not member functions. They operate indirectly on containers, through iterators.

22.7 True.

22.8 `stack`, `queue`, `priority_queue`.

22.9 False. It actually yields the position just after the end of the container.

22.10 Iterators.

22.11 Random-access.

Exercises

22.12 *(Palindromes)* Write a function template `palindrome` that takes a `vector` parameter and returns `true` or `false` according to whether the `vector` does or does not read the same forward as backward (e.g., a `vector` containing 1, 2, 3, 2, 1 is a palindrome, but a `vector` containing 1, 2, 3, 4 is not).

22.13 *(Sieve of Eratosthenes)* Modify Fig. 22.40, the Sieve of Eratosthenes, so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number. Remember that a prime number's factors are only 1 and the prime number itself. Every nonprime number has a unique prime factorization. For example, the factors of 54 are 2, 3, 3 and 3. When these values are multiplied together, the result is 54. For the number 54, the prime factors output should be 2 and 3.

22.14 *(Prime Numbers)* Modify Exercise 22.13 so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number and the number of times each prime factor appears in the unique prime factorization. For example, the output for the number 54 should be

```
The unique prime factorization of 54 is: 2 * 3 * 3 * 3
```

Recommended Reading

Ammeraal, L. *STL for C++ Programmers*. New York: John Wiley & Sons, 1997.

Austern, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Boston, MA: Addison-Wesley, 2000.

Glass, G., and B. Schuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

Josuttis, N. *The C++ Standard Library: A Tutorial and Handbook*. Boston: Addison-Wesley, 2000.

Koenig, A., and B. Moo. *Ruminations on C++*. Boston: Addison-Wesley, 1997.

Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Boston, MA: Addison-Wesley, 2001.

Musser, D. R., G. Derge and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition.* Boston: Addison-Wesley, 2010.

Musser, D. R., and A. A. Stepanov. "Algorithm-Oriented Generic Libraries," *Software Practice and Experience,* Vol. 24, No. 7, July 1994.

Nelson, M. *C++ Programmer's Guide to the Standard Template Library.* Foster City, CA: Programmer's Press, 1995.

Pohl, I. *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide.* Boston: Addison-Wesley, 1997.

Reese, G. *C++ Standard Library Practical Tips.* Hingham, MA: Charles River Media, 2005.

Robson, R. *Using the STL: The C++ Standard Template Library, Second Edition*. New York: Springer, 2000.

Schildt, H. *STL Programming from the Ground Up*, New York: Osborne McGraw-Hill, 1999.

Stepanov, A., and M. Lee. "The Standard Template Library," *Internet Distribution* 31 October 1995 `<www.cs.rpi.edu/~musser/doc.ps>`.

Stroustrup, B. "Making a `vector` Fit for a Standard," *The C++ Report,* October 1994.

Stroustrup, B. *The Design and Evolution of C++*. Boston: Addison-Wesley, 1994.

Stroustrup, B. *The C++ Programming Language, Third Edition.* Boston: Addison-Wesley, 2000.

Vandevoorde, D., and N. Josuttis. *C++ Templates: The Complete Guide.* Boston: Addison-Wesley, 2003.

Vilot, M. J. "An Introduction to the Standard Template Library," *The C++ Report,* Vol. 6, No. 8, October 1994.

Wilson, M. *Extended STL, Volume 1: Collections and Iterators.* Boston: Addison-Wesley, 2007

23 Boost Libraries, Technical Report 1 and C++0x

The danger from computers is not that they will eventually get as smart as men, but we will meanwhile agree to meet them halfway.
—**Bernard Avishai**

Objectives

In this chapter you'll learn:

- Future directions for C++.
- What the Boost Libraries are.
- A brief history of the Boost open source project, how new libraries are added to Boost, and how to install Boost.
- To use `Boost.Regex` to search for strings, validate data and replace parts of strings using regular expressions.
- To avoid memory leaks by using `Boost.Smart_ptr` to manage dynamic memory allocation and deallocation.
- What Boost (and other) libraries are included in Technical Report 1 (TR1)—a description of the additions to the C++ Standard Library.
- The changes to the core language and Standard Library coming in the new C++ Standard—C++0x.

Outline

23.1 Introduction

Throughout the book, we've discussed many of the key features of the impending new C++ Standard (C++0x). In this chapter, we introduce the Boost C++ Libraries and Technical Report 1 (TR1), and consider additional C++0x features. The **Boost C++ Libraries** are free, open source libraries created by members of the C++ community. Boost provides useful, well-designed libraries that work well with the existing C++ Standard Library. Boost can be used on many platforms with many different compilers. We overview some of the popular Boost libraries and provide code examples for the regular expression and smart pointer libraries. **Technical Report 1** describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. These libraries add useful functionality to C++. **C++0x** is the working name for the next version of the C++ Standard. It includes some additions to the core language, many of the library additions described in TR1 and other library enhancements.

23.2 Deitel Online C++ and Related Resource Centers

We regularly post online Resource Centers on key programming, software, Web 2.0 and Internet business topics at `www.deitel.com/ResourceCenters.html`. We've created several online Resource Centers that provide links to key information on Boost and C++0x. Visit the C++ Boost Libraries Resource Center at `www.deitel.com/CPlusPlusBoostLibraries/` to find current information on the available libraries and new releases. You can find current information on TR1 and C++0x in the C++0x section of the C++ Resource Center at `www.deitel.com/cplusplus/` (click **C++0x** in the **Categories** list). We used GNU C++ 4.5 and Visual C++ 2010 Express Edition to compile the examples in this chapter.

23.3 Boost Libraries

The idea for an online repository of free, peer-reviewed, open source C++ libraries was first proposed in a paper by Beman Dawes in 1998.[1] He and Robert Klarer got the idea while

1. "Proposal for a C++ Library Repository Web Site," Beman G. Dawes, May 6, 1998, `www.boost.org/users/proposal.pdf`.

attending a C++ Standards Committee meeting. The paper suggested a website where C++ programmers could find and share libraries and foster further C++ development. That idea eventually developed into the Boost Libraries at `www.boost.org`. Boost has grown to over 100 libraries, with more being added frequently. Today there are thousands of programmers in the Boost community.

Adding a New Library to Boost

Boost accepts useful, well-designed, portable libraries from anyone willing to contribute. Potential Boost libraries should conform to the C++ Standard and use the C++ Standard Library—or other appropriate Boost libraries. There is a formal acceptance process to ensure that libraries meet Boost's high quality and portability standards.

The community's interest in a library is determined by posting to mailing lists and reading the responses. If there is interest in a library, a preliminary submission of the library is posted in the **Boost Sandbox** (`svn.boost.org/svn/boost/sandbox/`)—a code repository for libraries that are under development. The Sandbox allows other users to experiment with the library and provide feedback.

When the library is ready for a formal review, the code submission is posted to the Sandbox Vault and a review manager is selected from a list of approved volunteers. The review manager makes sure the code is ready for formal review, sets up the review schedule, reads all user reviews, and makes the final decision whether or not to accept the library. The review manager may accept the library with certain corrections or improvements that must be implemented before the library is officially added to Boost. Once a library has been accepted, the author is responsible for its maintenance.

The Boost Software License

The Boost Software License (`www.boost.org/users/license.html`) grants the rights to copy, modify, use and distribute the Boost source code and binaries for any commercial or noncommercial use. The only requirement is that the copyright and license information be distributed with any source code that is made public, though it isn't required that the source code be released. These conditions allow the Boost libraries to be used in any application. Every Boost library must conform to these conditions.

Installing the Boost Libraries

The Boost libraries can be used with minimal setup on many platforms and compilers. BoostPro Computing offers a free installer for using Boost with Visual Studio at `www.boostpro.com/download`. Most Linux distributions offer packages for Boost, though it is sometimes split up into separate packages for the headers and libraries. An installation guide available at `www.boost.org/more/getting_started/index.html` provides setup instructions for many compilers and platforms.

23.4 Boost Libraries Overview

There are many Boost libraries—too many to cover in this book. In this section, we overview some of the most useful and popular libraries. The ones listed here are part of the next C++ standard—C++0x. In the following sections, we demonstrate two of these libraries as implemented by using their implementations from the C++0x standard library.

Array [2]

Boost.Array is a wrapper for fixed-size arrays that enhances built-in arrays by supporting most of the STL container interface described in Section 22.1. Class `array` allows you to use fixed-size arrays in STL applications rather than `vector`s (dynamically sized arrays), which are not as efficient when there is no need for dynamic resizing. To use class `array` with compilers that support this C++0x feature, include the `<array>` header.

Bind [3]

Boost.Bind extends the functionality of the standard functions `std::bind1st` and `std::bind2nd`. The `bind1st` and `bind2nd` functions are used to adapt binary functions (i.e., functions that take two arguments) to be used with the standard algorithms which take unary functions (i.e., functions that take one argument). Class `bind` enhances that functionality by allowing you to adapt functions that take up to nine arguments. Class `bind` also makes it easy to reorder the arguments passed to the function using placeholders. To use class `bind` with compilers that support this C++0x feature, include the `<functional>` header.

Function [4]

Boost.Function allows you to store function pointers, member-function pointers and function objects in a function wrapper. A `function` can hold any function whose arguments and return type can be converted to match the signature of the function wrapper. For example, if the function wrapper was created to hold a function that takes a `string` and returns a `string`, it can also hold a function that takes a `char*` and returns a `char*`, because a `char*` can be converted to a `string`, using a conversion constructor. To use class `function` with compilers that support this C++0x feature, include the `<functional>` header.

Random [5]

Boost.Random allows you to create various random number generators and random number distributions. The `std::rand` and `std::srand` functions in the C++ Standard Library generate pseudo-random numbers. A **pseudo-random number generator** uses an initial state to produce seemingly random numbers—using the same initial state produces the same sequence of numbers. The `rand` function always uses the same initial state, therefore it produces the same sequence of numbers every time. The function `srand` allows you to set the initial state to vary the sequence. Pseudo-random numbers are often used in testing—the predictability enables you to confirm the results. `Boost.Random` provides pseudo-random number generators as well as generators that can produce **nondeterministic random numbers**—a set of random numbers that can't be predicted. Such random number generators are used in simulations and security scenarios where predictability is undesirable.

`Boost.Random` also allows you to specify the distribution of the numbers generated. A common distribution is the **uniform distribution**, which assigns the same probability to each number within a given range. This is similar to rolling a die or flipping a coin—each possible outcome is equally as likely. You can set this range at compile time. `Boost.Random`

2. Documentation for Boost.Array: `www.boost.org/doc/libs/1_45_0/doc/html/array.html`.
3. Documentation for Boost.Bind: `www.boost.org/doc/libs/1_45_0/libs/bind/bind.html`.
4. Documentation for Boost.Function: `www.boost.org/doc/libs/1_45_0/doc/html/function.html`.
5. Jens Maurer, "A Proposal to Add an Extensible Random Number Facility to the Standard Library," Document Number N1452, April 10, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html`.

allows you to use a distribution in combination with any random number generator and even create your own distributions. To use these new random-number capabilities with compilers that support these C++0x features, include the `<random>` header.

Regex [6]

`Boost.Regex` provides support for processing **regular expressions** in C++. Regular expressions are used to match specific character patterns in text. Many modern programming languages have built-in support for regular expressions, but C++ does not. With `Boost.Regex`, you can search for a particular expression in a `string`, replace parts of a `string` that match a regular expression, and split a `string` into tokens using regular expressions to define the delimiters. These techniques are commonly used for text processing, parsing and input validation. To use regular expressions with compilers that support this C++0x feature, include the `<regex>` header. We discuss some regular expression capabilities in more detail in Section 23.5.

Smart_ptr [7]

`Boost.Smart_ptr` defines smart pointers that help you manage dynamically allocated resources (e.g., memory, files and database connections). Programmers often get confused about when to deallocate memory or simply forget to do it, especially when the memory is referenced by more than one pointer. Smart pointers take care of these tasks automatically. TR1 includes several smart pointers from the `Boost.Smart_ptr` library. We discussed the `unique_ptr` class in Chapter 16. **`shared_ptrs`** handle lifetime management of dynamically allocated objects. The memory is released when there are no `shared_ptrs` referencing it. **`weak_ptrs`** allow you to observe the value held by a `shared_ptr` without assuming any management responsibilities. We discuss the `shared_ptr` and `weak_ptr` in more detail in Section 23.6. To use the smart pointer classes with compilers that support these C++0x features, thisinclude the `<regex>` header.

Tuple [8]

A **tuple** is a set of objects. **`Boost.Tuple`** allows you to create sets of objects in a generic way and allows generic functions to act on those sets. The library allows you to create tuples of up to 10 objects; that limit can be extended. Class `tuple` is basically an extension to the STL's `std::pair` class template. Tuples are often used to return multiple values from a function. They can also be used to store sets of elements in an STL container where each set of elements is an element of the container. Another useful feature is the ability to set the values of variables using the elements of a tuple. To use class `tuple` with compilers that support this C++0x feature, include the `<tuple>` header.

Type_traits [9]

The **`Boost.Type_traits`** library helps abstract the differences between types to allow generic programming implementations to be optimized. The `type_traits` classes allow you

---

6. Documentation for `Boost.Regex`: `www.boost.org/doc/libs/1_45_0/libs/regex/doc/html/`.
7. Documentation for `Boost.Smart_ptr`: `www.boost.org/doc/libs/1_45_0/libs/smart_ptr/smart_ptr.htm`.
8. Documentation for `Boost.Tuple`: `www.boost.org/doc/libs/1_45_0/libs/tuple/doc/tuple_users_guide.html`.
9. Documentation for `Boost.Type_traits`, Steve Cleary, Beman Dawes, Howard Hinnant and John Maddock, `www.boost.org/doc/libs/1_45_0/libs/type_traits/doc/html/index.html`.

to determine specific traits of a type (e.g., is it a pointer or a reference type, or does the type have a `const` qualifier?) and perform type transformations to allow the object to be used in generic code. Such information can be used to optimize generic code. For example, sometimes it is more efficient to copy a collection of objects using the C function `memcpy` rather than by iterating through all the elements of the collection, as the STL `copy` algorithm does. With the `Boost.Type_traits` library, generic algorithms can be optimized by first checking the traits of the types being processed, then performing the algorithm accordingly. C++0x compilers that support these features include them in the `<type_traits>` header.

23.5 Regular Expressions with the regex Library

[*Note:* The C++0x library features used in this section's examples were not fully implemented in GNU C++ at the time of this writing. For now, if you wish to use these features in GNU C++, you can install the Boost version of the regular expressions library as discussed in Section 23.3]

Regular expressions are specially formatted `string`s that are used to find patterns in text. They can be used to validate data to ensure that it is in a particular format. For example, a zip code must consist of five digits, and a last name must start with a capital letter.

The `std::tr1::regex` library (from header `<regex>`) provides several classes and algorithms (in namespce `std::tr1`) for recognizing and manipulating regular expressions. Class template **`basic_regex`** represents a regular expression. The algorithm **`regex_match`** returns true if a `string` matches the regular expression. With `regex_match`, the entire string must match the regular expression. The `regex` library also provides the algorithm **`regex_search`**, which returns `true` if any part of an arbitrary `string` matches the regular expression.

Regular Expression Character Classes

The table in Fig. 23.1 specifies some **character classes** that can be used with regular expressions. A character class is not a C++ class—rather it's simply an escape sequence that represents a group of characters that might appear in a `string`.

Character class	Matches	Character class	Matches
`\d`	any decimal digit	`\D`	any non-digit
`\w`	any word character	`\W`	any non-word character
`\s`	any whitespace character	`\S`	any non-whitespace character

Fig. 23.1 | Character classes.

A **word character** is any alphanumeric character or underscore. A **whitespace** character is a space, tab, carriage return, newline or form feed. A **digit** is any numeric character. Regular expressions are not limited to the character classes in Fig. 23.1. In Fig. 23.2, you'll see that regular expressions can use other notations to search for complex patterns in `string`s.

23.5.1 Regular Expression Example

The program in Fig. 23.2 tries to match birthdays to a regular expression. For demonstration purposes, the expression in line 11 matches only birthdays that do not occur in April and that belong to people whose names begin with `"J"`.

```
1  // Fig. 23.2: fig23_02.cpp
2  // Demonstrating regular expressions.
3  #include <iostream>
4  #include <string>
5  #include <regex>
6  using namespace std; // allows use of features in both std and std::tr1
7
8  int main()
9  {
10    // create a regular expression
11    regex expression( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
12
13    // create a string to be tested
14    string string1 = "Jane's Birthday is 05-12-75\n"
15       "Dave's Birthday is 11-04-68\n"
16       "John's Birthday is 04-28-73\n"
17       "Joe's Birthday is 12-17-77";
18
19    // create an smatch object to hold the search results
20    smatch match;
21
22    // match regular expression to string and print out all matches
23    while ( regex_search( string1, match, expression,
24       regex_constants::match_not_dot_newline ) )
25    {
26       cout << match.str() << endl; // print the matching string
27
28       // remove the matched substring from the string
29       string1 = match.suffix();
30    } // end while
31 } // end function main
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 23.2 | Regular expressions checking birthdays.

Creating the Regular Expression

Line 11 creates a **`regex`** object by passing a regular expression to the `regex` constructor. The name `regex` is a `typedef` of the `basic_regex` class template that uses `char`s. We precede each backslash character in the initializer string with an additional backslash. Recall that C++ treats a backslash in a string literal as the beginning of an escape sequence. To insert a literal backslash in a string, you must escape the backslash character with another backslash. For example, the character class `\d` must be represented as `\\d` in a C++ string literal.

The first character in the regular expression, `"J"`, is a literal character. Any `string` matching this regular expression is required to start with `"J"`. In a regular expression, the dot character "." matches any single character. When the dot character is followed by an asterisk, as in ".*", the regular expression matches any number of unspecified characters. In general, when the operator "*" is applied to a pattern, the pattern will match *zero or more* occurrences. By contrast, applying the operator "+" to a pattern causes the pattern to match *one or more* occurrences. For example, both "A*" and "A+" will match "A", but only "A*" will match an empty string.

As indicated in Fig. 23.1, "\d" matches any decimal digit. To specify sets of characters other than those that belong to a predefined character class, characters can be listed in square brackets, []. For example, the pattern "[aeiou]" matches any vowel. Ranges of characters are represented by placing a dash (-) between two characters. In the example, "[0-35-9]" matches only digits in the ranges specified by the pattern—i.e., any digit between 0 and 3 or between 5 and 9; therefore, the pattern matches any digit except 4. You can also specify that a pattern should match anything other than the characters in the brackets. To do so, place ^ as the first character in the brackets. It is important to note that "[^4]" is not the same as "[0-35-9]"; "[^4]" matches any non-digit and digits other than 4.

Although the "-" character indicates a range when it is enclosed in square brackets, instances of the "-" character outside grouping expressions are treated as literal characters. Thus, the regular expression in line 11 searches for a string that starts with the letter “J”, followed by any number of characters, followed by a two-digit number (of which the second digit cannot be 4), followed by a dash, another two-digit number, a dash and another two-digit number.

Using the Regular Expression to Search for Matches

Line 20 creates an smatch (pronounced “ess-match”; a typedef for match_results) object. A **match_results** object, when passed as an argument to one of the regex algorithms, stores the regular expression’s match. An **smatch** stores an object of type string::const_iterator that you can use to access the matching string. There are typedefs to support other string representations such as const char* (cmatch).

The while statement (lines 23–30) searches string1 for matches to the regular expression until none can be found. We use the call to regex_search as the while statement condition (lines 23–24). regex_search returns true if the string (string1) contains a match to the regular expression (expression). We also pass an smatch object to regex_search so we can access the matching string. The last argument, **match_not_eol**, prevents the "." character from matching a newline character. The body of the while statement prints the substring that matched the regular expression by callig the match object’s str function (line 26) and removes it from the string being searched by calling the match object’s suffix function and assigning its result back to string1 (line 29). The call to the match_results member function **suffix** returns a string from the end of the match to the end of the string being searched. The output in Fig. 23.2 displays the two matches that were found in string1. Notice that both matches conform to the pattern specified by the regular expression.

Quantifiers

The asterisk (*) in line 11 of Fig. 23.2 is more formally called a **quantifier**. Figure 23.3 lists various quantifiers that you can place after a pattern in a regular expression and the purpose of each quantifier.

Quantifier	Matches
`*`	Matches zero or more occurrences of the preceding pattern.
`+`	Matches one or more occurrences of the preceding pattern.
`?`	Matches zero or one occurrences of the preceding pattern.
`{n}`	Matches exactly *n* occurrences of the preceding pattern.
`{n,}`	Matches at least *n* occurrences of the preceding pattern.
`{n,m}`	Matches between *n* and *m* (inclusive) occurrences of the preceding pattern.

Fig. 23.3 | Quantifiers used in regular expressions.

We've already discussed how the asterisk (`*`) and plus (`+`) quantifiers work. The question mark (`?`) quantifier matches zero or one occurrences of the pattern that it quantifies. A set of braces containing one number, `{n}`, matches exactly *n* occurrences of the pattern it quantifies. We demonstrate this quantifier in the next example. Including a comma after the number enclosed in braces matches at least *n* occurrences of the quantified pattern. The set of braces containing two numbers, `{n,m}`, matches between *n* and *m* occurrences (inclusively) of the pattern that it quantifies. All of the quantifiers are **greedy**—they'll match as many occurrences of the pattern as possible until the pattern fails to make a match. If a quantifier is followed by a question mark (`?`), the quantifier becomes **lazy** and will match as few occurrences as possible as long as there is a successful match.

23.5.2 Validating User Input with Regular Expressions

The program in Fig. 23.4 presents a more involved example that uses regular expressions to validate name, address and telephone number information input by a user.

```
// Fig. 23.4: fig23_04.cpp
// Validating user input with regular expressions.
#include <iostream>
#include <string>
#include <regex>
using namespace std;

bool validate( const string&, const string& ); // validate prototype
string inputData( const string&, const string& ); // inputData prototype

int main()
{
   // enter the last name
   string lastName = inputData( "last name", "[A-Z][a-zA-Z]*" );

   // enter the first name
   string firstName = inputData( "first name", "[A-Z][a-zA-Z]*" );

```

Fig. 23.4 | Validating user input with regular expressions. (Part 1 of 3.)

```
   // enter the address
   string address = inputData( "address",
      "[0-9]+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the city
   string city =
      inputData( "city", "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the state
   string state = inputData( "state",
      "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );

   // enter the zip code
   string zipCode = inputData( "zip code", "\\d{5}" );

   // enter the phone number
   string phoneNumber = inputData( "phone number",
      "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );

   // display the validated data
   cout << "\nValidated Data\n\n"
      << "Last name: " << lastName << endl
      << "First name: " << firstName << endl
      << "Address: " << address << endl
      << "City: " << city << endl
      << "State: " << state << endl
      << "Zip code: " << zipCode << endl
      << "Phone number: " << phoneNumber << endl;
} // end of function main

// validate the data format using a regular expression
bool validate( const string &data, const string &expression )
{
   // create a regex to validate the data
   regex validationExpression = regex( expression );
   return regex_match( data, validationExpression );
} // end of function validate

// collect input from the user
string inputData( const string &fieldName, const string &expression )
{
   string data; // store the data collected

   // request the data from the user
   cout << "Enter " << fieldName << ": ";
   getline( cin, data );

   // validate the data
   while ( !( validate( data, expression ) ) )
   {
      cout << "Invalid " << fieldName << ".\n";
      cout << "Enter " << fieldName << ": ";
```

Fig. 23.4 | Validating user input with regular expressions. (Part 2 of 3.)

```
71         getline( cin, data );
72      } // end while
73
74      return data;
75   } // end of function inputData
```

```
Enter last name: 12345
Invalid last name.
Enter last name: Blue
Enter first name: Betty
Enter address: 123
Invalid address.
Enter address: 123 Main Street
Enter city: SomeCity
Enter state: SomeState
Enter zip code: 1
Invalid zip code.
Enter zip code: 55555
Enter phone number: 555-555-123
Invalid phone number.
Enter phone number: 555-555-1234

Validated Data

Last name: Blue
First name: Betty
Address: 123 Main Street
City: SomeCity
State: SomeState
Zip code: 55555
Phone number: 555-555-1234
```

Fig. 23.4 | Validating user input with regular expressions. (Part 3 of 3.)

The program first asks the user to input a last name (line 14) by calling the `inputData` function. The `inputData` function (lines 58–75) takes two arguments, the name of the data being input and a regular expression that it must match. The function prompts the user (line 63) to input the specified data. Then `inputData` checks whether the input is in the correct format by calling the `validate` function (lines 50–55). That function takes two arguments—the `string` to validate and the regular expression it must match. The function first uses the expression to create a `regex` object (line 53). Then it calls `regex_match` to determine whether the `string` matches the expression. If the input isn't valid, `inputData` prompts the user to enter the information again. Once the user enters a valid input, the data is returned as a `string`. The program repeats that process until all the data fields have been validated (lines 14–36). Then we display all the information (lines 39–46).

In the previous example, we searched a `string` for substrings that matched a regular expression. In this example, we want to ensure that the entire `string` for each input conforms to a particular regular expression. For example, we want to accept `"Smith"` as a last name, but not `"9@Smith#"`. We use `regex_match` here instead of `regex_search`—`regex_match` returns `true` only if the entire `string` matches the regular expression. Alternatively, you can use a regular expression that begins with a `"^"` character and ends with a `"$"` character. The characters `"^"` and `"$"` represent the beginning and end of a `string`, respectively. Together, these characters force a regular expression to return a match only if the entire string being processed matches the regular expression.

The regular expression in line 14 uses the square bracket and range notation to match an uppercase first letter followed by letters of any case—a-z matches any lowercase letter, and A-Z matches any uppercase letter. The * quantifier signifies that the second range of characters may occur zero or more times in the string. Thus, this expression matches any string consisting of one uppercase letter, followed by zero or more additional letters.

The notation \s matches a single white-space character (lines 21, 25 and 29). The expression \d{5}, used for the zipCode string (line 32), matches any five digits. The character "|" (lines 21, 25 and 29) matches the expression to its left *or* the expression to its right. For example, Hi (John|Jane) matches both Hi John and Hi Jane. In line 21, we use the character "|" to indicate that the address can contain a word of one or more characters *or* a word of one or more characters followed by a space and another word of one or more characters. Note the use of parentheses to group parts of the regular expression. Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.

The lastName and firstName variables (lines 14 and 17) both accept strings of any length that begin with an uppercase letter. The regular expression for the address string (line 21) matches a number of at least one digit, followed by a space, then either one or more letters or else one or more letters followed by a space and another series of one or more letters. Therefore, "10 Broadway" and "10 Main Street" are both valid addresses. As currently formed, the regular expression in line 21 doesn't match an address that does not start with a number, or that has more than two words. The regular expressions for the city (line 25) and state (line 29) strings match any word of at least one character or, alternatively, any two words of at least one character if the words are separated by a single space. This means both Waltham and West Newton would match. Again, these regular expressions would not accept names that have more than two words. The regular expression for the zipCode string (line 32) ensures that the zip code is a five-digit number. The regular expression for the phoneNumber string (line 36) indicates that the phone number must be of the form xxx-yyy-yyyy, where the xs represent the area code and the ys the number. The first x and the first y cannot be zero, as specified by the range [1-9] in each case.

23.5.3 Replacing and Splitting Strings

Sometimes it's useful to replace parts of one string with another or to split a string according to a regular expression. For this purpose, the regex library provides the algorithm regex_replace and the regex_token_iterator class, which we demonstrate in Fig. 23.5.

```
// Fig. 23.5: fig23_05.cpp
// Using regex_replace algorithm.
#include <iostream>
#include <string>
#include <regex>
using namespace std;

int main()
{
```

Fig. 23.5 | Using regex_replace algorithm. (Part 1 of 3.)

```
   // create the test strings
   string testString1 = "This sentence ends in 5 stars *****";
   string testString2 = "1, 2, 3, 4, 5, 6, 7, 8";
   string output;

   cout << "Original string: " << testString1 << endl;

   // replace every * with a ^
   testString1 =
      regex_replace( testString1, regex( "\\*" ), string( "^" ) );
   cout << "^ substituted for *: " << testString1 << endl;

   // replace "stars" with "carets"
   testString1 =
      regex_replace( testString1, regex( "stars" ), string( "carets" ) );
   cout << "\"carets\" substituted for \"stars\": "
      << testString1 << endl;

   // replace every word with "word"
   testString1 =
      regex_replace( testString1, regex( "\\w+" ), string( "word" ) );
   cout << "Every word replaced by \"word\": " << testString1 << endl;

   // replace the first three digits with "digit"
   cout << "\nOriginal string: " << testString2 << endl;
   string testString2Copy = testString2;

   for ( int i = 0; i < 3; ++i ) // loop three times
   {
      testString2Copy = regex_replace( testString2Copy,
         regex( "\\d" ), "digit", regex_constants::format_first_only );
   } // end for

   cout << "Replace first 3 digits by \"digit\": "
      << testString2Copy << endl;

   // split the string at the commas
   cout << "string split at commas [";

   regex splitter( ",\\s" ); // regex to split a string at commas
   sregex_token_iterator tokenIterator( testString2.begin(),
      testString2.end(), splitter, -1 ); // token iterator
   sregex_token_iterator end;  // empty iterator

   while ( tokenIterator != end ) // tokenIterator isn't empty
   {
      output += "\"" + (*tokenIterator).str() + "\", ";
      ++tokenIterator; // advance the iterator
   } // end while

   // delete the ", " at the end of output string
   cout << output.substr( 0, output.length() - 2 ) << "]" << endl;
} // end of function main
```

Fig. 23.5 | Using `regex_replace` algorithm. (Part 2 of 3.)

```
Original string: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original string: 1, 2, 3, 4, 5, 6, 7, 8
Replace first 3 digits by "digit": digit, digit, digit, 4, 5, 6, 7, 8
string split at commas ["1", "2", "3", "4", "5", "6", "7", "8"]
```

Fig. 23.5 | Using `regex_replace` algorithm. (Part 3 of 3.)

Replacing Substrings with `regex_replace`

Algorithm **`regex_replace`** replaces text in a `string` with new text wherever the original `string` matches a regular expression. In line 19, `regex_replace` replaces every instance of `"*"` in `testString1` with `"^"`. The regular expression (`"\*"`) precedes character `"*"` with a backslash, \. Typically, `"*"` is a quantifier indicating that a regular expression should match any number of occurrences of a preceding pattern. However, in this case we want to find all occurrences of the literal character `"*"`; to do this, we must escape character `"*"` with character "\". By escaping a special regular expression character with a \, we tell the regular expression matching engine to find the actual character `"*"` rather than use it as a quantifier. Also, the first and last arguments to this version of function regex_replace must be `string`s. Lines 23–24 use `regex_replace` to replace the string `"stars"` in `testString1` with the string `"carets"`. Lines 29–30 use `regex_replace` to replace every word in `testString1` with the string `"word"`.

Lines 37–41 replace the first three instances of a digit (`"\d"`) in `testString2` with the text `"digit"`. We pass **`regex_constants::format_first_only`** as an additional argument to `regex_replace` (lines 39–40). This argument tells `regex_replace` to replace only the first substring that matches the regular expression. Normally `regex_replace` would replace all occurrences of the pattern. We put this call inside a `for` loop that runs three times; each time replacing the first instance of a digit with the text `"digit"`. We use a copy of `testString2` (line 35) so we can use the original `textString2` for the next part of the example.

Obtaining Substrings with a `regex_token_iterator`

Next we use a `regex_token_iterator` to divide a `string` into several substrings. A **`regex_token_iterator`** iterates through the parts of a `string` that match a regular expression. Lines 49 and 51 use `sregex_token_iterator`, which is a `typedef` that indicates the results are to be manipulated with a `string::const_iterator`. We create the iterator (lines 49–50) by passing the constructor two iterators (`testString2.begin()` and `testString2.end()`), which represent the beginning and end of the `string` to iterate over and the regular expression to look for. In our case we want to iterate over the parts of the `string` that *don't* match the regular expression. To do that we pass `-1` to the constructor. This indicates that it should iterate over each substring that doesn't match the regular expression. The original `string` is broken at delimiters that match the specified regular expression. We use a `while` statement (lines 53–57) to add each substring to the `string` `output`. The `regex_token_iterator` `end` (line 51) is an empty iterator. We've iterated over the entire `string` when `tokenIterator` equals `end` (line 53).

23.6 Smart Pointers

[*Note:* The C++0x library features used in this section's examples work in both Microsoft Visual C++ 2010 Express and GNU C++ 4.5. GNU C++ considers these features experimental and requires you to use the command line option `-std:c++0x` to compile the examples correctly.]

Many common bugs in C and C++ code are related to pointers. **Smart pointers** help you avoid errors by providing additional functionality to standard pointers. This functionality typically strengthens the process of memory allocation and deallocation. Smart pointers also help you write exception safe code. If a program throws an exception before `delete` has been called on a pointer, it creates a memory leak. After an exception is thrown, a smart pointer's destructor will still be called, which calls `delete` on the pointer for you.

Section 16.11 showed one of the smart pointer classes—`unique_ptr`—which is responsible for managing dynamically allocated memory. A `unique_ptr` automatically calls `delete` to free its associated dynamic memory when the `unique_ptr` is destroyed or goes out of scope. A `unique_ptr` is a basic smart pointer. C++0x provides other smart pointer options with additional functionality.

23.6.1 Reference Counted `shared_ptr`

`shared_ptrs` (from header `<memory>`) hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared with other objects in the program. You can have any number of `shared_ptrs` to the same resource. `shared_ptrs` really do share the resource—if you change the resource with one `shared_ptr`, the changes also will be "seen" by the other `shared_ptrs`. The internal pointer is deleted once the last `shared_ptr` to the resource is destroyed. `shared_ptrs` use **reference counting** to determine how many `shared_ptrs` point to the resource. Each time a new `shared_ptr` to the resource is created, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

`shared_ptrs` are useful in situations where multiple pointers to the same resource are needed, such as in STL containers. `shared_ptrs` can safely be copied and used in STL containers.

`shared_ptrs` also allow you to determine how the resource will be destroyed. For most dynamically allocated objects, `delete` is used. However, some resources require more complex cleanup. In that case, you can supply a custom **deleter** function, or function object, to the `shared_ptr` constructor. The deleter determines how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the `shared_ptr` calls the custom deleter function. This functionality enables a `shared_ptr` to manage almost any kind of resource.

Example Using `shared_ptr`

Figures 23.6–23.7 define a simple class to represent a `Book` with a `string` to represent the title of the `Book`. The destructor for class `Book` (Fig. 23.7, lines 12–15) displays a message on the screen indicating that an instance is being destroyed. We use this class to demonstrate the common functionality of `shared_ptr`.

```
// Fig. 23.6: Book.h
// Declaration of class Book.
#ifndef BOOK_H
#define BOOK_H
#include <string>
using namespace std;

class Book
{
public:
   Book( const string &bookTitle ); // constructor
   ~Book(); // destructor
   string title; // title of the Book
};
#endif // BOOK_H
```

Fig. 23.6 | Book header.

```
// Fig. 23.7: Book.cpp
// Member-function definitions for class Book.
#include <iostream>
#include <string>
#include "Book.h"
using namespace std;

Book::Book( const string &bookTitle ) : title( bookTitle )
{
}

Book::~Book()
{
   cout << "Destroying Book: " << title << endl;
} // end of destructor
```

Fig. 23.7 | Book member-function definitions.

Creating shared_ptrs

The program in Fig. 23.8 uses shared_ptrs (from the header <memory>) to manage several instances of class Book. We also create a typedef, BookPtr, as an alias for the type shared_ptr<Book> (line 10). Line 28 creates a shared_ptr to a Book titled "C++ How to Program" (using the BookPtr typedef). The shared_ptr constructor takes as its argument a pointer to an object. We pass it the pointer returned from the new operator. This creates a shared_ptr that manages the Book object and sets the reference count to one. The constructor can also take another shared_ptr, in which case it shares ownership of the resource with the other shared_ptr and the reference count is increased by one. The first shared_ptr to a resource should always be created using the new operator. A shared_ptr created with a regular pointer assumes it's the first shared_ptr assigned to that resource and starts the reference count at one. If you make multiple shared_ptrs with the same pointer, the shared_ptrs won't acknowledge each other and the reference count will be wrong. When the shared_ptrs are destroyed, they both call delete on the resource.

```
// Fig. 23.8: fig23_08.cpp
// Demonstrate shared_ptrs.
#include <algorithm>
#include <iostream>
#include <memory>
#include <vector>
#include "Book.h"
using namespace std;

typedef shared_ptr< Book > BookPtr; // shared_ptr to a Book

// a custom delete function for a pointer to a Book
void deleteBook( Book* book )
{
   cout << "Custom deleter for a Book, ";
   delete book; // delete the Book pointer
} // end of deleteBook

// compare the titles of two Books for sorting
bool compareTitles( BookPtr bookPtr1, BookPtr bookPtr2 )
{
   return ( bookPtr1->title < bookPtr2->title );
} // end of compareTitles

int main()
{
   // create a shared_ptr to a Book and display the reference count
   BookPtr bookPtr( new Book( "C++ How to Program" ) );
   cout << "Reference count for Book " << bookPtr->title << " is: "
      << bookPtr.use_count() << endl;

   // create another shared_ptr to the Book and display reference count
   BookPtr bookPtr2( bookPtr );
   cout << "Reference count for Book " << bookPtr->title << " is: "
      << bookPtr.use_count() << endl;

   // change the Book's title and access it from both pointers
   bookPtr2->title = "Java How to Program";
   cout << "The Book's title changed for both pointers: "
      << "\nbookPtr: " << bookPtr->title
      << "\nbookPtr2: " << bookPtr2->title << endl;

   // create a std::vector of shared_ptrs to Books (BookPtrs)
   vector< BookPtr > books;
   books.push_back( BookPtr( new Book( "C How to Program" ) ) );
   books.push_back( BookPtr( new Book( "VB How to Program" ) ) );
   books.push_back( BookPtr( new Book( "C# How to Program" ) ) );
   books.push_back( BookPtr( new Book( "C++ How to Program" ) ) );

   // print the Books in the vector
   cout << "\nBooks before sorting: " << endl;
   for ( int i = 0; i < books.size(); ++i )
      cout << ( books[ i ] )->title << "\n";
```

Fig. 23.8 | shared_ptr example program. (Part 1 of 2.)

```
54
55     // sort the vector by Book title and print the sorted vector
56     sort( books.begin(), books.end(), compareTitles );
57     cout << "\nBooks after sorting: " << endl;
58     for ( int i = 0; i < books.size(); ++i )
59        cout << ( books[ i ] )->title << "\n";
60
61     // create a shared_ptr with a custom deleter
62     cout << "\nshared_ptr with a custom deleter." << endl;
63     BookPtr bookPtr3( new Book( "Small C++ How to Program" ), deleteBook);
64     bookPtr3.reset(); // release the Book this shared_ptr manages
65
66     // shared_ptrs are going out of scope
67     cout << "\nAll shared_ptr objects are going out of scope." << endl;
68  } // end of main
```

```
Reference count for Book C++ How to Program is: 1
Reference count for Book C++ How to Program is: 2

The Book's title changed for both pointers:
bookPtr: Java How to Program
bookPtr2: Java How to Program

Books before sorting:
C How to Program
VB How to Program
C# How to Program
C++ How to Program

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
VB How to Program

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Small C++ How to Program

All shared_ptr objects are going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: VB How to Program
Destroying Book: Java How to Program
```

Fig. 23.8 | `shared_ptr` example program. (Part 2 of 2.)

Manipulating `shared_ptrs`

Lines 29–30 display the `Book`'s `title` and the number of `shared_ptrs` referencing that instance. Notice that we use the `->` operator to access the `Book`'s data member `title`, as we would with a regular pointer. `shared_ptrs` provide the pointer operators `*` and `->`. We get the reference count using the `shared_ptr` member function **`use_count`**, which returns the number of `shared_ptrs` to the resource. Then we create another `shared_ptr` to the instance of class `Book` (line 33). Here we use use the `shared_ptr` constructor with the original shared_ptr as its argument. You can also use the assignment operator (=) to create a shared_ptr to the same resource. Lines 34–35 print the reference count of the original shared_ptr to show that the count increased by one when we created the second shared_ptr. As mentioned earlier, changes made to the resource of a shared_ptr are "seen" by all shared_ptrs to that resource. When we change the title of the Book using bookPtr2 (line 38), we can see the change when using bookPtr (lines 39–41).

Manipulating **shared_ptr***s in an STL Container*

Next we demonstrate using shared_ptrs in an STL container. We create a vector of BookPtrs (line 44) and add four elements (recall that BookPtr is a typedef for a shared_ptr<Book>, line 10). Lines 51–53 print the contents of the vector. Then we sort the Books in the vector by title (line 56). We use the function compareTitles (lines 20–23) in the sort algorithm to compare the title data members of each Book alphabetically.

shared_ptr *Custom Deleter*

Line 63 creates a shared_ptr with a custom deleter. We define the custom deleter function deleteBook (lines 13–17) and pass it to the shared_ptr constructor along with a pointer to a new instance of class Book. When the shared_ptr destroys the instance of class Book, it calls deleteBook with the internal Book * as the argument. Notice that deleteBook takes a Book *, not a shared_ptr. A custom deleter function must take one argument of the shared_ptr's internal pointer type. deleteBook displays a message to show that the custom deleter was called, then deletes the pointer. A primary use for custom deleters is when using third-party C libraries. Rather than providing a class with a constructor and destructor as a C++ library would, C libraries frequently provide one function that returns a pointer to a struct representing a resource and another that does the necessary cleanup when the resource is no longer needed. Using a custom deleter allows you to use a shared_ptr to keep track of the resource and still ensure it is freed correctly.

Resetting a **shared_ptr**

We call the shared_ptr member function reset (line 64) to show the custom deleter at work. The **reset** function releases the current resource and sets the shared_ptr to NULL. If there are no other shared_ptrs to the resource, it's destroyed. You can also pass a pointer or shared_ptr representing a new resource to the reset function, in which case the shared_ptr will manage the new resource. But, as with the constructor, you should only use a regular pointer returned by the new operator.

shared_ptr*s Are Destroyed When They Go Out of Scope*

All the shared_ptrs and the vector go out of scope at the end of the main function and are destroyed. When the vector is destroyed, so are the shared_ptrs in it. The program output shows that each instance of class Book is destroyed automatically by the shared_ptrs. There is no need to delete each pointer placed in the vector.

23.6.2 weak_ptr: shared_ptr Observer

A weak_ptr points to the resource managed by a shared_ptr without assuming any responsibility for it. The reference count for a shared_ptr doesn't increase when a weak_ptr references it. That means that the resource of a shared_ptr can be deleted while there are still weak_ptrs pointing to it. When the last shared_ptr is destroyed, the resource is deleted and any remaining weak_ptrs are set to NULL. One use for weak_ptrs, as we'll demonstrate later in this section, is to avoid memory leaks caused by circular references.

A weak_ptr can't directly access the resource it points to—you must create a shared_ptr from the weak_ptr to access the resource. There are two ways to do this. You can pass the weak_ptr to the shared_ptr constructor. That creates a shared_ptr to the resource being pointed to by the weak_ptr and properly increases the reference count. If the resource has already been deleted, the shared_ptr constructor will throw a **bad_weak_ptr** exception. You can also call the weak_ptr member function **lock**, which returns a shared_ptr to the weak_ptr's resource. If the weak_ptr points to a deleted resource (i.e., NULL), lock will return an empty shared_ptr (i.e., a shared_ptr to NULL). lock should be used when an empty shared_ptr isn't considered an error. You can access the resource once you have a shared_ptr to it. weak_ptrs should be used in any situation where you need to observe the resource but don't want to assume any management responsibilities for it. The following example demonstrates the use of weak_ptrs in **circularly referential data**, a situation in which two objects refer to each other internally.

Example Using **weak_ptr**

Figures 23.9–23.12 define classes Author and Book. Each class has a pointer to an instance of the other class. This creates a circular reference between the two classes. Note that we use both weak_ptrs and shared_ptrs to hold the cross reference to each class (Fig. 23.9 and 23.10, lines 20–21 in each figure). If we set the shared_ptrs, it creates a memory leak—we'll explain why soon and show how we can use the weak_ptrs to fix this problem.

```
1  // Fig. 23.9: Author.h
2  // Definition of class Author.
3  #ifndef AUTHOR_H
4  #define AUTHOR_H
5  #include <string>
6  #include <memory>
7
8  using namespace std;
9
10 class Book; // forward declaration of class Book
11
12 // Author class definition
13 class Author
14 {
15 public:
16    Author( const string &authorName ); // constructor
17    ~Author(); // destructor
18    void printBookTitle(); // print the title of the Book
19    string name; // name of the Author
20    weak_ptr< Book > weakBookPtr; // Book the Author wrote
21    shared_ptr< Book > sharedBookPtr; // Book the Author wrote
22 };
23 #endif // AUTHOR_H
```

Fig. 23.9 | Author class definition.

```cpp
// Fig. 23.10: Book.h
// Definition of class Book.
#ifndef BOOK_H
#define BOOK_H
#include <string>
#include <memory>

using namespace std;

class Author; // forward declaration of class Author

// Book class definition
class Book
{
public:
   Book( const string &bookTitle ); // constructor
   ~Book(); // destructor
   void printAuthorName(); // print the name of the Author
   string title; // title of the Book
   weak_ptr< Author > weakAuthorPtr; // Author of the Book
   shared_ptr< Author > sharedAuthorPtr; // Author of the Book
};
#endif // BOOK_H
```

Fig. 23.10 | Book class definition.

Classes `Author` and `Book` define destructors that each display a message to indicate when an instance of either class is destroyed (Figs. 23.11 and 23.12, lines 15–18). Each class also defines a member function to print the `title` of the `Book` and `Author`'s `name` (lines 21–34 in each figure). Recall that you can't access the resource directly through a `weak_ptr`, so first we create a `shared_ptr` from the `weak_ptr` data member (line 24 in each figure). If the resource the `weak_ptr` is referencing doesn't exist, the call to the `lock` function returns a `shared_ptr` which points to `NULL` and the condition fails. Otherwise, the new `shared_ptr` contains a valid pointer to the `weak_ptr`'s resource, and we can access the resource. If the condition in line 24 is true (i.e., `bookPtr` and `authorPtr` aren't `NULL`), we print the reference count to show that it increased with the creation of the new `shared_ptr`, then we print the `title` of the `Book` and `Author`'s `name`. The `shared_ptr` is destroyed when the function exits so the reference count decreases by one.

```cpp
1  // Fig. 23.11: Author.cpp
2  // Member-function definitions for class Author.
3  #include <iostream>
4  #include <string>
5  #include <memory>
6  #include "Author.h"
7  #include "Book.h"
8
9  using namespace std;
10
```

Fig. 23.11 | `Author` member-function definitions. (Part 1 of 2.)

```
Author::Author( const string &authorName ) : name( authorName )
{
}

Author::~Author()
{
   cout << "Destroying Author: " << name << endl;
} // end of destructor

// print the title of the Book this Author wrote
void Author::printBookTitle()
{
   // if weakBookPtr.lock() returns a non-empty shared_ptr
   if ( shared_ptr< Book > bookPtr = weakBookPtr.lock() )
   {
      // show the reference count increase and print the Book's title
      cout << "Reference count for Book " << bookPtr->title
         << " is " << bookPtr.use_count() << "." << endl;
      cout << "Author " << name << " wrote the book " << bookPtr->title
         << "\n" << endl;
   } // end if
   else // weakBookPtr points to NULL
      cout << "This Author has no Book." << endl;
} // end of printBookTitle
```

Fig. 23.11 | Author member-function definitions. (Part 2 of 2.)

```
// Fig. 23.12: Book.cpp
// Member-function definitions for class Book.
#include <iostream>
#include <string>
#include <memory>
#include "Author.h"
#include "Book.h"

using namespace std;

Book::Book( const string &bookTitle ) : title( bookTitle )
{
}

Book::~Book()
{
   cout << "Destroying Book: " << title << endl;
} // end of destructor

// print the name of this Book's Author
void Book::printAuthorName()
{
   // if weakAuthorPtr.lock() returns a non-empty shared_ptr
        shared_ptr< Author > authorPtr = weakAuthorPtr.lock()
   {
```

Fig. 23.12 | Book member-function definitions. (Part 1 of 2.)

```
      // show the reference count increase and print the Author's name
      cout << "Reference count for Author " << authorPtr->name
         << " is " << authorPtr.use_count() << "." << endl;
      cout << "The book " << title << " was written by "
         << authorPtr->name << "\n" << endl;
   } // end if
   else // weakAuthorPtr points to NULL
      cout << "This Book has no Author." << endl;
} // end of printAuthorName
```

Fig. 23.12 | Book member-function definitions. (Part 2 of 2.)

Figure 23.13 defines a `main` function that demonstrates the memory leak caused by the circular reference between classes `Author` and `Book`. Lines 12–13 create `shared_ptrs` to an instance of each class. The `weak_ptr` data members are set in lines 16–17. Lines 20–21 set the `shared_ptr` data members for each class. The instances of classes `Author` and `Book` now reference each other. We then print the reference count for the `shared_ptrs` to show that each instance is referenced by two `shared_ptrs` (lines 24–27), the ones we create in the `main` function and the data member of each instance. Remember that `weak_ptrs` don't affect the reference count. Then we call each class's member function to print the information stored in the `weak_ptr` data member (lines 32–33). The functions also display the fact that another `shared_ptr` was created during the function call. Finally, we print the reference counts again to show that the additional `shared_ptrs` created in the `printAuthorName` and `printBookTitle` member functions are destroyed when the functions finish.

```
1  // Fig. 23.13: fig23_13.cpp
2  // Demonstrate use of weak_ptr.
3  #include <iostream>
4  #include <memory>
5  #include "Author.h"
6  #include "Book.h"
7  using namespace std;
8
9  int main()
10 {
11    // create a Book and an Author
12    shared_ptr< Book > bookPtr( new Book( "C++ How to Program" ) );
13    shared_ptr< Author > authorPtr( new Author( "Deitel & Deitel" ) );
14
15    // reference the Book and Author to each other
16    bookPtr->weakAuthorPtr = authorPtr;
17    authorPtr->weakBookPtr = bookPtr;
18
19    // set the shared_ptr data members to create the memory leak
20    bookPtr->sharedAuthorPtr = authorPtr;
21    authorPtr->sharedBookPtr = bookPtr;
22
```

Fig. 23.13 | `shared_ptrs` cause a memory leak in circularly referential data. (Part 1 of 2.)

```
23      // reference count for bookPtr and authorPtr is one
24      cout << "Reference count for Book " << bookPtr->title << " is "
25         << bookPtr.use_count() << endl;
26      cout << "Reference count for Author " << authorPtr->name << " is "
27         << authorPtr.use_count() << "\n" << endl;
28
29      // access the cross references to print the data they point to
30      cout << "\nAccess the Author's name and the Book's title through "
31         << "weak_ptrs." << endl;
32      bookPtr->printAuthorName();
33      authorPtr->printBookTitle();
34
35      // reference count for each shared_ptr is back to one
36      cout << "Reference count for Book " << bookPtr->title << " is "
37         << bookPtr.use_count() << endl;
38      cout << "Reference count for Author " << authorPtr->name << " is "
39         << authorPtr.use_count() << "\n" << endl;
40
41      // the shared_ptrs go out of scope, the Book and Author are destroyed
42      cout << "The shared_ptrs are going out of scope." << endl;
43   } // end of main
```

```
Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 3.
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 3.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

The shared_ptrs are going out of scope.
```

Fig. 23.13 | `shared_ptr`s cause a memory leak in circularly referential data. (Part 2 of 2.)

Memory Leak

At the end of `main`, the `shared_ptr`s to the instances of `Author` and `Book` we created go out of scope and are destroyed. Notice that the output doesn't show the destructors for classes `Author` and `Book`. The program has a memory leak—the instances of `Author` and `Book` aren't destroyed because of the `shared_ptr` data members. When `bookPtr` is destroyed at the end of the `main` function, the reference count for the instance of class `Book` becomes one—the instance of `Author` still has a `shared_ptr` to the instance of `Book`, so it isn't deleted. When `authorPtr` goes out of scope and is destroyed, the reference count for the instance of class `Author` also becomes one—the instance of `Book` still has a `shared_ptr` to the instance of `Author`. Neither instance is deleted because the reference count for each is still one.

Fixing the Memory Leak

Now, comment out lines 20–21 by placing `//` at the beginning of each line. This prevents the code from setting the `shared_ptr` data members for classes `Author` and `Book`. Recompile the code and run the program again. Figure 23.14 shows the output. Notice that the initial reference count for each instance is now one instead of two because we don't set the `shared_ptr` data members. The last two lines of the output show that the instances of classes `Author` and `Book` were destroyed at the end of the `main` function. We eliminated the memory leak by using the `weak_ptr` data members rather than the `shared_ptr` data members. The `weak_ptrs` don't affect the reference count but still allow us to access the resource when we need it by creating a temporary `shared_ptr` to the resource. When the `shared_ptrs` we created in `main` are destroyed, the reference counts become zero and the instances of classes `Author` and `Book` are deleted properly.

```
Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 2.
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 2.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

The shared_ptrs are going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program
```

Fig. 23.14 | `weak_ptrs` used to prevent a memory leak in circularly referential data.

23.7 Technical Report 1

Technical Report 1 (TR1) describes proposed additions to the C++ Standard Library. Many of the libraries in TR1 will be accepted by the C++ Standards Committee but they are not considered part of the C++ standard until C++0x is finalized. The library additions provide solutions for many common programming problems. Most of the additions are based on 11 Boost libraries—the ones discussed in Section 23.4 and several other minor ones. Descriptions of the three additional TR1 libraries follow.

Visual Studio 2010 and recent versions of GNU C++ support most of TR1 already. Boost provides a compatibility layer that automatically falls back to the Boost implementation of each library if it was not supplied with the compiler.[10]

Many libraries didn't make it into TR1 due to time constraints. **Technical Report 2 (TR2)**, which will be released after C++0x, contains additional library proposals that weren't included in TR1. The release of TR2 will bring even more functionality to the standard library without having to wait for another new standard.

Unordered Associative Containers[11]

The Unordered Associative Containers library defines four new containers—`unordered_set`, `unordered_map`, `unordered_multiset` and `unordered_multimap`. These

10. Documentation for Boost.TR1: www.boost.org/doc/libs/1_45_0/doc/html/boost_tr1.html.
11. Matthew Austern, "A Proposal to Add Hash Tables to the Standard Library," Document Number N1456=03-0039, April 9, 2003, www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html.

associative containers are implemented as hash tables. A **hash table** is split into sections sometimes called "**buckets**." A key is used to determine where to store an element in the container. The key is passed to a **hash function** which returns a `size_t`. The `size_t` returned by the hash function determines the "bucket" that the value is placed in. If two values are equal, so are the `size_t`s returned by the hash function. Multiple values can be placed in the same "bucket." You retrieve an element from the container using the key much as you do with a `set` or `map`. The key determines which "bucket" the value was placed in, then the "bucket" is searched for the value.

With **`unordered_set`** and **`unordered_multiset`**, the element itself is used as the key. **`unordered_map`** and **`unordered_multimap`** use a separate key to determine where to place the element—the arguments are passed as a `pair<const Key, Value>`. `unordered_set` and `unordered_map` require that all the keys used are unique; `unordered_multiset` and `unordered_multimap` don't enforce that restriction. The containers are defined in the `<unordered_set>` and `<undordered_map>` headers.

Mathematical Special Functions[12]

This library incorporates mathematical functions added to **C99**—the C standard published in 1999—that are missing in the C++ Standard. C99 supplies trigonometric, hyperbolic, exponential, logarithmic, power and special functions. This library adds those functions, among others, to C++ in the `<cmath>` header.

Increased Compatibility with C99[13]

C++ evolved from the C programming language. Most C++ compilers can also compile C programs, but there are some incompatibilities between the languages. The goal of this library is to increase compatibility between C++ and C99. Most of this library involves adding items to C++ headers to support C99 features—this is often accomplished by including the corresponding C99 headers.

23.8 C++0x

The C++ Standards Committee is currently revising the C++ Standard. The last standard was published in 1998. Work on the new standard, currently referred to as C++0x, began in 2003. The new standard, likely to be released in late 2011 or early 2012, includes the TR1 libraries and additions to the core language. Browse the C++0x section of the Deitel C++ Resource Center at `www.deitel.com/cplusplus/` and click **C++0x** in the **Categories** list to find current information on C++0x.

Standardization Process

The **International Organization for Standardization (ISO)** oversees the creation of international programming language standards, including those for C and C++. Every addition or change to the current C++ standard must be approved by the ISO/IEC JTC 1/SC 22 Working Group 21 (WG21), the committee that maintains the C++ standard. This com-

12. Walter E. Brown, "A Proposal to Add Mathematical Special Functions to the C++ Standard Library," Document Number N1422=03-0004, February 24, 2003, `std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1422.html`.
13. P. J. Plauger, "Proposed Additions to TR-1 to Improve Compatibility With C99," Document Number N1568=04-0008, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1568.htm`.

mittee of volunteers from the C++ programming community meets twice a year to discuss issues pertaining to the standard. Smaller, unofficial meetings are held more frequently to consider proposals between official committee meetings. ISO requires at least 5 years between new drafts of a standard.

Goals for C++0x[14]
Bjarne Stroustrup, creator of the C++ programming language, has expressed his vision for the future of C++—the main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language. He also provides an overview of C++0x's new features in his C++0x FAQ at `www2.research.att.com/~bs/C++0xFAQ.html`.

23.9 Core Language Changes

A listing of proposed changes to the core language can be found at `www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2869.html`. There are also links to the papers associated with each proposal. We briefly discuss some of the important core language changes that have been accepted into the working draft of the new standard. The number of proposals that make it into the working draft is likely to increase before the standard is finalized. The GNU C++ compiler has an optional C++0x mode which allows you to experiment with a number of the core language changes (`gcc.gnu.org/projects/cxx0x.html`). Visual Studio 2010 also supports some C++0x features—the Visual C++ Team Blog (`blogs.msdn.com/vcblog/`) contains updates on the status of C++0x in Visual Studio.

Rvalue *Reference*[15]

The ***rvalue* reference** type in C++0x allows you to bind an *rvalue* (temporary object) to a non-`const` reference. An *rvalue* reference is declared as `T&&` (where `T` is the type of the object being referenced) to distinguish it from a normal reference `T&` (now called an *lvalue* reference). An *rvalue* reference can be used to effectively implement move semantics—instead of being copied, the state of an object is moved, leaving the original with an empty value. For example, currently the following code creates a temporary `string` object and passes it to `push_back`, which then copies it into the vector.

```
vector< string > myVector;
myVector.push_back( "message" );
```

If `push_back` were overloaded to take an *rvalue* reference, the storage allocated by the temporary `string` can be reused directly by the one in the `vector`. The temporary `string` will be destroyed anyway when the function returns, so there's no need for it to keep its value.

Rvalue references can also be used in "forwarding functions"—function objects that adapt a function to take fewer arguments (e.g., `std::bind1st` or function objects created using `Boost.Bind`). Normally, each reference parameter would need a `const` and non-`const` version to account for *lvalue*s, `const` *lvalue*s and *rvalue*s. With *rvalue* references you need only one forwarding function.

14. Bjarne Stroustrup, "The Design of C++0x," May 2005, `www.research.att.com/~bs/rules.pdf`.
15. Howard E. Hinnant, "A Proposal to Add an *rvalue* Reference to the C++ Language," October 19, 2006, Document Number N2118=06-0188, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2118.html`.

static_assert[16]

The **`static_assert` declaration** allows you to test certain aspects of the program at compile time. A `static_assert` declaration takes a constant integral expression and a `string` literal. If the expression evaluates to 0 (false), the compiler reports the error. The error message includes the `string` literal provided in the declaration. The `static_assert` declaration can be used at namespace, class or local scope.

The addition of `static_assert` makes learning C++ easier. The assertions can be used to provide more informative error messages when novices make common mistakes such as using the wrong type of argument in a function call or template instantiation. They're also useful in library development—incorrect usage of the library can be reported much more effectively.

Compatibility with New C99 Features

C++0x will incorporate many changes added in the 1999 C standard. These include changes to the preprocessor,[17] the addition of the `long long` integer type,[18] and imposing rules on extensions that add additional integer types[19] (for example, a 128-bit integer type). These changes allow modern C code to compile correctly as C++.

Delegating Constructors[20]

This feature allows a constructor to delegate to another of the class's constructors (i.e., call another of the class's constructors). This makes it easier to write overloaded constructors. Currently, an overloaded constructor must duplicate the code that is common to the other constructor. This leads to repetitive and error-prone code. A mistake in one constructor could cause inconsistency in object initialization. By calling another version of the constructor, the common code doesn't need to be repeated and the chance of error decreases.

Right Angle Brackets[21]

Currently, it's necessary to put a space between trailing right angle brackets (>) when using nested template types. Without the space, the compiler assumes that the two brackets are the right shift operator (>>). This means that writing `vector<vector<int>>` causes a compiler error. The statement would have to be written as `vector<vector<int> >`. Many novices stumble on this quirk of C++. In C++0x, the C++ compiler will recognize when >> is part of a template rather than the right-shift operator.

---

16. Robert Klarer, Dr. John Maddock, Beman Dawes and Howard Hinnant, "Proposal to Add Static Assertions to the Core Language," Document Number N1720, October 20, 2004, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html`.
17. Clark Nelson, "Working Draft Changes for C99 Preprocessor Synchronization," Document Number N1653, July 16, 2004, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm`.
18. J. Stephen Adamczyk, "Adding the `long long` Type to C++," Document Number N1811, April 29, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf`.
19. J. Stephen Adamczyk, "Adding Extended Integer Types to C++," Document Number N1988, April 19, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1988.pdf`.
20. Herb Sutter and Francis Glassborow, "Delegating Constructors," Document Number N1986=06-0056, April 6, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf`.
21. Daveed Vandevoorde, "Right Angle Brackets," Document Number N1857=05-0017, January 14, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html`.

Deducing the Type of Variable from Its Initializer[22]

This proposal defines new functionality for the keyword `auto`—it automatically determines variable types based on the initializer expression. `auto` can be used in place of long, complicated types that are unmanageable to type by hand. `auto` can also be used with `const` and `volatile` qualifiers. You can create pointers and references with `auto` as you would with the full type name. `auto` supports the declaration of multiple variables in one statement (e.g., `auto x = 1, y = 2`). The `auto` keyword is meant to save time, ease the learning process and improve generic programming. The following code creates a `vector` of instances of a hypothetical `Class< T >`.

```
vector< Class< T > > myVector;
vector< Class< T > >::const_iterator iterator = myVector.begin();
```

Using `auto`, the declaration of `iterator` can be written as

```
auto iterator = myVector.begin();
```

The type of `iterator` is `vector<`*Class*`<T>>::const_iterator`. You can also create two variables of the same type in one declaration. Both variables in

```
auto iteratorBegin = myVector.begin(), iteratorEnd = myVec-
tor.end();
```

are created with the type `vector<`*Class*`<T>>::const_iterator`. You can also use `auto` with `const` or `volatile` qualifiers and create pointers or references. The statement

```
const auto &iteratorRef = myVector.begin();
```

creates a `const` reference to a `vector<`*Class*`<T>>::const_iterator`. `auto` can save you a lot of time by automatically determining the type of the variable you're declaring—especially with complex template types, like those used in Section 22.6.

Variadic Templates[23]

Currently, each class or function template has a fixed number of template parameters. If you need a class or function template with different numbers of template parameters, you must define a template for each case. A **variadic template** accepts any number of arguments, which can greatly simplify template programming. For example, you can provide one variadic function template rather than many overloaded ones with different parameters. Many template libraries, such as `Boost.Bind`, `Boost.Tuple` and `Boost.Function`, include large amounts of duplicate code or make use of complex preprocessor macros to generate all the necessary template definitions. Variadic templates will make it easier to implement such libraries.

Template Aliases[24]

Libraries often use templates with many parameters to implement generic programming. There may be situations where it would be useful to be able to specify certain arguments

22. Jaakko Järvi, Bjarne Stroustrup and Gabriel Dos Reis, "Deducing the Type of Variable From Its Initializer Expression," Document Number N1984=06-0054, April 6, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf`.
23. Douglas Gregor, Jaakko Järvi and Gary Powell, "Variadic Templates," Document Number N2080=06-0150, September 9, 2006, `www.osl.iu.edu/~dgregor/cpp/variadic-templates.pdf`.
24. Gabriel Dos Reis and Mat Marcus, "Proposal to Add Template Aliases to C++," Document # N1449-03-0032, April 7, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1449.pdf`.

for the template that remain consistent but still be able to vary the rest. This can be done with a template alias. A **template alias** is similar to a `typedef`—it introduces a name used to refer to a template. In a `typedef`, all the template parameters are specified. Using a template alias, certain parameters are specified and others may still vary. You can use a general-purpose template in a more specific role where many of the arguments are always the same by using a template alias to set the consistent parameters while still being able to vary those that change in each instantiation. For example, the following declares a template `MyStack` that uses `list<T>` as its underlying implementation instead of the `deque` used by default in `std::stack`.

```
template< typename T > using MyStack< T > =
   stack< T, list< T > >;
```

Initializer Lists for User-Defined Types[25]

Currently, initializer lists can be used only with arrays and `struct`s. In C++0x, a class can define a constructor taking a parameter of type **`std::initializer_list<T>`**. An initializer list can then be used to initialize an object of the class, as in:

```
vector< int > second = { 4, 5, 6 }; // legal in C++0x
```

The initializer values are stored in an `initializer_list` object, which is passed to the class's constructor. All standard library container classes will be updated to have constructors taking an `initializer_list`.

Range-Based `for` *Statement*[26]

A common use of the `for` statement is to iterate over a container of elements. Currently, the syntax for built-in arrays and library containers is different—built-in arrays use an index or raw pointers, and container classes use iterators returned by the `begin` and `end` member functions. In addition to providing simpler syntax, the new **range-based `for` statement** allows you to use the same syntax for iterating over both arrays and containers. The following code iterates through a collection of `int` values.

```
for ( int &item : items ) // items can be an array or container
   item *= 2;
```

Lambda Expressions[27]

Many library functions receive function pointers or function objects as parameters. Currently, the functions or function objects must be defined before they can be passed to these library functions as arguments. **Lambda expressions** (or **lambda functions**) enable you to define function objects as they are being passed to a function. They are defined locally inside functions and can "capture" (by value or by reference) the local variables of the enclosing function then manipulate these variables in the lambda's body. Lambda

25. J. Stephen Adamczyk, Gabriel Dos Reis and Bjarne Stroustrup, "Initializer list WP wording (Revision 2)," Document Number N2531=08-0041, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2531.pdf`
26. Thorsten Ottosen, "Wording for range-based for-loop (revision 3)," Document Number N2934=07-0254, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2394.html`
27. Jaakko Järvi, John Freeman and Lawrence Crowl, "Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)," Document Number N2550=08-0060, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf`

expressions are implemented in Visual Studio 2010 and GNU C++ 4.5. Figure 23.15 provides a simple lambda expression example that doubles the value of each element in an `int` array.

```
1  // Fig. 23.15: fig23_15.cpp
2  // Example of lambda expressions in C++0x.
3  #include <iostream>
4  #include <algorithm>
5  using namespace std;
6
7  int main()
8  {
9     const int size = 4; // size of array values
10    int values[ size ] = { 1, 2, 3, 4 }; // initialize values
11
12    // output each element multiplied by two
13    for_each( values, values + size,
14       []( int i ) { cout << i * 2 << endl; } );
15
16    int sum = 0; // initialize sum to zero
17
18    // add each element to sum
19    for_each( values, values + size,
20       [ &sum ]( int i ) { sum += i; } );
21
22    cout << "sum is " << sum << endl; // output sum
23 } // end main
```

```
2
4
6
8
sum is 10
```

Fig. 23.15 | Example of lambda expressions in C++0x.

Lines 9 and 10 declare and initialize a small array of integers. Lines 13–14 call the `for_each` algorithm on the elements of the `values` array. The third argument to `for_each` is a lambda expression. Lambdas begin with “lambda introducer” (`[]`), followed by a parameter list and function body. Return types can be inferred automatically if the body is a single statement of the form `return` *expression*`;`—otherwise, the return type is `void` by default. The lambda expression in line 14 receives an `int`, multiplies it by 2 and displays the result. The `for_each` algorithm passes each element of the array to the lambda.

The second call to the `for_each` algorithm (lines 19–20) calculates the sum of the array elements. The lambda introducer `[&sum]` indicates that this lambda expression is capturing the local variable `sum` by reference (note the use of the ampersand), so that the lambda can modify `sum`’s value. Without the ampersand, `sum` would be captured by value and the local variable would not be updated. The `for_each` algorithm passes each element of the array to the lambda, which adds the value to the `sum`. Line 22 then displays the value of `sum`.

23.10 Wrap-Up

In this chapter we discussed various aspects of the future of C++. We introduced the Boost C++ Libraries and described some of the most popular libraries.

We discussed the `regex` library and the symbols that are used to form regular expressions. We provided examples of how to use regular-expression classes, including `regex`, `match_results` and `regex_token_iterator`. You learned how to find patterns in a string and match entire strings to patterns with algorithms `regex_search` and `regex_match`. We demonstrated how to replace characters in a string with `regex_replace` and how to split strings into tokens with a `regex_token_iterator`.

We showed how to use the `Boost.Smart_ptr` library. You learned how to use the `shared_ptr` and `weak_ptr` classes to avoid memory leaks when using dynamically allocated memory. We demonstrated how to use custom deleter functions to allow `shared_ptrs` to manage resources that require special destruction procedures. We also explained how `weak_ptrs` can be used to prevent memory leaks in circularly referential data.

We overviewed the upcoming revised standard, C++0x, discussing TR1 and the changes to the core language. We introduced the libraries accepted into TR1. We described the new core language features including the `auto` keyword, *rvalue* reference, improvements in compatibility with C99, initializer lists and lambda expressions. Remember that Boost, TR1 and C++0x are constantly changing—visit our Resource Centers to stay up to date with all three.

Summary

Section 23.2 Deitel Online C++ and Related Resource Centers

- Visit the C++ Boost Libraries Resource Center at `www.deitel.com/CPlusPlusBoostLibraries/` to find current information on the available libraries and new releases.
- Find current information on TR1 and C++0x (p. 937) in the **`C++0x`** category of the C++ Resource Center at `www.deitel.com/cplusplus/`.
- For more information on Visual C++, visit our Visual C++ Resource Center at `www.deitel.com/VisualCPlusPlus/`.

Section 23.3 Boost Libraries

- The Boost Libraries (p. 937) at `www.boost.org` provide free peer-reviewed C++ libraries.
- Boost libraries must conform to the C++ standard and use the C++ Standard Library—or other appropriate Boost libraries.
- A preliminary submission of each Boost library is posted in the Boost Sandbox Vault (p. 938).
- The review manager makes sure the code is ready for formal review, sets up the review schedule, reads all user reviews, and makes the final decision whether or not to accept the library.

Section 23.4 Boost Libraries Overview

- `Boost.Array` (p. 939) provides fixed-size arrays that support the STL container interface.
- `Boost.Bind` (p. 939) extends the functionality provided by the standard functions `bind1st` and `bind2nd`. It allows you to adapt functions that take up to nine arguments. It also makes it easy to reorder the arguments passed to the function.

- Boost.Function (p. 939) allows you to store function pointers, member-function pointers and function objects in a function wrapper. A function can hold any function whose arguments and return type can be converted to match the function wrapper's signature.
- Boost.Random (p. 939) allows you to create a variety of random number generators and distributions.
- A pseudo-random number generator uses an initial state to produce seemingly random numbers—using the same initial state produces the same sequence of numbers.
- Regular expressions (p. 940) are used to match character patterns in text.
- With Boost.Regex (p. 940), you can search for a particular expression in a string, replace parts of a string that match a regular expression and split a string into tokens using regular expressions.
- Boost.Smart_ptr (p. 940) defines smart pointers that help you manage dynamically allocated resources.
- shared_ptrs (p. 940) handle lifetime management of dynamically allocated objects. Memory is released automatically when there are no shared_ptrs referencing it.
- A weak_ptr (p. 940) allowsyou to observe a shared_ptr's value without any management responsibilities.
- Boost.Tuple (p. 940) allows you to create sets of objects that can be used by generic functions.
- The type_traits (p. 940) classes allow you to determine specific traits of a type and perform type transformations to allow the object to be used in generic code.

Section 23.5 Regular Expressions with the **regex** *Library*

- Regular expressions are specially formatted strings used to find patterns in text.
- basic_regex (p. 941) represents a regular expression.
- Algorithm regex_match (p. 941) returns true only if an entire string matches the regular expression.
- Algorithm regex_search (p. 941) returns true if any part of a string matches the regular expression.
- To use the regex library, include the header <regex>.
- A character class (p. 941) represents a group of characters.
- A word character (\w; p. 941) is any alphanumeric character or underscore. A whitespace character (\s; p. 941) is a space, tab, carriage return, newline or form feed. A digit (\d; p. 941) is any numeric character.

Section 23.5.1 Regular Expression Example

- You must precede each character class's backslash character with an additional backslash in strings.
- To specify sets of characters other than those that belong to a predefined character class, list the characters in square brackets, []. Ranges of characters are represented by placing a "-" between two characters. Instances of the "-" character outside [] characters are treated as literals.
- Place ^ as the first character in the brackets to specify that a pattern should match anything other than the characters in the brackets.
- A match_results (p. 943) is an object that holds a match to a regular expression. The typedef smatch (p. 943) represents a match_results that provides access to the match result via a string::const_iterator.
- regex_constants::match_not_eol (p. 943) prevents the "." character from matching a newline character.

- `match_results` member function `suffix` (p. 943) returns a `string` from the end of the match to the end of the `string` being searched.
- The `"*"` quantifier (p. 944) will match *zero or more* occurrences.
- The `"+"` quantifier (p. 944) will match *one or more* occurrences.
- The `"?"` quantifier (p. 944) will match *zero or one* occurrences.
- A set of braces containing one number, `{n}` (p. 944), matches *exactly n* occurrences.
- Including a comma after the number enclosed in braces (p. 944) matches *at least n* occurrences.
- The notation `{n,m}` (p. 944) matches *between n* and *m* occurrences (inclusively).
- Quantifiers are greedy (p. 944)—they'll match as many occurrences of the pattern as possible until the pattern fails to make a match.
- A quantifier followed by a question mark (`?`) becomes lazy and will match as few occurrences as possible as long as there is a successful match.

Section 23.5.2 Validating User Input with Regular Expressions
- The characters `"^"` and `"$"` represent the beginning and end of a `string`, respectively.
- The character `"|"` matches the expression to its left *or* the expression to its right.
- You can apply quantifiers to patterns in parentheses to create more complex regular expressions.

Section 23.5.3 Replacing and Splitting Strings
- Algorithm `regex_replace` (p. 949) replaces text in a `string` with new text wherever the original `string` matches a regular expression.
- Escaping a `"*"` character with a `\` tells the regular expression matching engine to find the actual character `"*"` rather than use it as a quantifier.
- `regex_constants::format_first_only` (p. 949) tells `regex_replace` to replace only the first substring that matches the regular expression. Normally `regex_replace` would replace all occurrences of the pattern.
- A `regex_token_iterator` (p. 949) iterates through the parts of a `string` that match the regular expression.
- Create a `regex_token_iterator` by passing the constructor two iterators which represent the beginning and end of the `string` to iterate over, and the regular expression to match.
- Pass `-1` to the `regex_token_iterator` constructor to indicate that it should iterate over each substring that *doesn't* match the regular expression.

Section 23.6 Smart Pointers
- Smart pointers (p. 950) avoid errors by strengthening the process of memory allocation and deallocation.
- After an exception is thrown, a smart pointer's destructor will call `delete` on the pointer for you.

Section 23.6.1 Reference Counted `shared_ptr`
- `shared_ptrs` (p. 950) hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared with other objects in the program.
- Changes to the resource of a `shared_ptr` will be "seen" by the other `shared_ptrs` to that resource.
- `shared_ptrs` use reference counting (p. 950) to determine how many `shared_ptrs` point to the resource. When the reference count reaches zero, the internal pointer is deleted.
- `shared_ptrs` can safely be copied and can be used in STL containers.

- You can create a `shared_ptr` with a custom deleter function which specifies how to destroy the resource. A custom deleter function (p. 950) must take one argument of the internal pointer's type.
- Include the `<memory>` header to use `shared_ptr`s.
- The `shared_ptr` constructor takes a pointer to an object. The constructor can also take another `shared_ptr`, in which case it shares ownership of the resource with the other `shared_ptr` and the reference count is increased by one.
- The first `shared_ptr` to a resource should always be created using the `new` operator.
- `shared_ptr`s provide the pointer operators `*` and `->`.
- The `shared_ptr` member function `use_count` (p. 953) returns the number of `shared_ptr`s to the resource.
- Function `reset` (p. 954) releases the current resource and sets the `shared_ptr` to `NULL`. You can also pass a pointer or `shared_ptr` to the `reset` function; the `shared_ptr` will manage the new resource.

Section 23.6.2 `weak_ptr`: `shared_ptr` Observer

- A `weak_ptr` is used to point to the resource managed by a `shared_ptr` without assuming any responsibility for it—the reference count for the `shared_ptr` doesn't increase.
- When the last `shared_ptr` is destroyed, the resource is deleted and any remaining `weak_ptr`s are set to `NULL`.
- A `weak_ptr` can't access the resource it points to—you must create a `shared_ptr` from the `weak_ptr` to access the resource. You can pass the `weak_ptr` to the `shared_ptr` constructor. You can also call `weak_ptr` function `lock` (p. 955), which returns a `shared_ptr` to the `weak_ptr`'s resource.
- Include the `<memory>` header to use `weak_ptr`s.

Section 23.7 Technical Report 1

- Technical Report 1 (TR1) describes additions to the C++ Standard Library. Most of the additions are based on 11 Boost libraries.
- Visual Studio 2008 SP1 and recent versions of GNU C++ support most of TR1 already.
- Boost provides a compatibility layer that automatically falls back to the Boost implementation of each library if it was not supplied with the compiler.
- Technical Report 2 (TR2; p. 960) contains additional library proposals that were not in TR1.
- The Unordered Associative Containers library (p. 961) defines four new containers—`unordered_set`, `unordered_map`, `unordered_multiset` and `unordered_multimap`. These associative containers are implemented as hash tables and are defined in `<unordered_set>` and `<unordered_map>`.
- `unordered_set` and `unordered_multiset` use the element as the key. `unordered_map` and `unordered_multimap` store key–value pairs.
- `unordered_set` and `unordered_map` require unique keys; `unordered_multiset` and `unordered_multimap` don't enforce that restriction.
- TR1 includes trigonometric, hyperbolic, exponential, logarithmic, power and special functions from C99 (p. 961).
- TR1 includes C99 headers to increase compatibility between C++ and C99.

Section 23.8 C++0x

- The new standard, C++0x, includes the TR1 libraries and changes to the core language.
- The International Organization for Standardization (ISO; p. 961) oversees the creation of international programming language standards. ISO Working Group 21 maintains the C++ standard.

- The main goals for the new standard are to make C++ easier to learn, improve library building capabilities and increase compatibility with the C programming language.

Section 23.9 Core Language Changes

- The GNU C++ compiler has an optional C++0x mode which allows you to experiment with a number of the core language changes. Visual Studio 2010 also supports some C++0x features.
- The *rvalue* reference type (p. 962) in C++0x allows you to bind an *rvalue* (temporary object) to a non-`const` reference.
- An *rvalue* reference is declared as `T&&` (where `T` is the type of the object being referenced).
- An *rvalue* reference can be used to implement move semantics.
- A `static_assert` declaration (p. 963) allows you to test certain aspects of the program at compile time.
- A `static_assert` declaration takes a constant integral expression and a `string`. If the expression evaluates to 0 (false), the compiler reports the error using the `string` provided in the declaration.
- C++0x will incorporate many changes added in the 1999 C standard, including changes to the preprocessor, the addition of the `long long` integer type, and imposing rules on extensions that add additional integer types. These changes allow modern C code to compile correctly as C++.
- A constructor can call another of the class's constructors directly (p. 963).
- The C++ compiler will recognize when `>>` is part of a template.
- Keyword `auto` (p. 964) automatically determines a variable's type based on its initializer expression. `auto` takes the place of the full type name.
- A variadic template (p. 964) accepts any number of arguments.
- Variadic templates make it easier to implement other template libraries such as `Boost.Bind`, `Boost.Tuple` and `Boost.Function`.
- Unlike a `typedef`, when using a template alias (p. 965) certain parameters are specified and others may still vary.
- In C++0x, a class can define a constructor that receives a `std::initializer_list<T>` (p. 965). An initializer list can then be used to initialize objects of that class.
- The new range-based `for` statement (p. 965) will allow you to use the same syntax for iterating over arrays and containers.
- Lambda expressions (or lambda functions; p. 965) provide a simplified syntax for defining function objects directly where they are used.
- A lambda function can capture local variables (by value or by reference) and manipulate them inside the lambda's body.
- Lambdas begin with the lambda introducer `[]`, followed by a parameter and function body. Return types can be inferred automatically if the body is a single statement of the form `return` *expression*`;`—otherwise, the return type is `void` by default.
- To capture a local variable, specify it in the lambda introducer. To capture by reference, use an ampersand.

Self-Review Exercises

23.1 Fill in the blanks in each of the following statements:

a) The ________ describes proposed changes to the C++ Standard Library.

b) The ________ library helps manage the release of dynamically allocated memory to prevent memory leaks.

c) `Boost.Bind` enhances the ________ and ________ standard library functions.
d) `shared_ptrs` use a(n) ________ to determine when to delete the resource.
e) Class ________ represents a regular expression in `Boost.Regex`.
f) Class `regex_token_iterator` is located in namespace ________.
g) The `Boost.Regex` algorithm ________ changes all occurrences of a pattern in a `string` to a specified `string`.
h) Regular expression quantifier ________ matches zero or more occurrences of an expression.
i) Regular expression operator ________ inside square brackets will not match any of the characters in that set of brackets.
j) The ________ keyword in C++0x automatically determines the type of a variable when it's initialized.
k) Move semantics and forwarding functions in C++0x can be written using ________.

23.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Creating a `weak_ptr` to a resource increases the reference count.
b) A regular expression matches a `string` to a pattern.
c) The expression `\d` in a regular expression denotes all letters.

23.3 Write statements to accomplish each of the following tasks:
a) Create a regular expression to match either a five-letter word or five-digit number.
b) Create a regular expression to match a phone number in the form of `(123) 456-7890`.
c) Create a `shared_ptr` to the `int` 5 called `intPtr`.
d) Create a `weak_ptr` to `intPtr` called `weakIntPtr`.
e) Access the `int`'s value using `weakIntPtr`.

Answers to Self-Review Exercises

23.1 a) TR1. b) `Boost.Smart_ptr`. c) `bind1st`, `bind2nd`. d) reference count. e) `regex` or `basic_regex`. f) `boost`. g) `regex_replace`. h) `*`. i) `^`. j) `auto`. k) *rvalue* references.

23.2 a) False. A `weak_ptr` assumes no ownership of its resource and doesn't affect the reference count. b) True. c) False. The expression `\d` in a regular expression denotes all decimal digits.

23.3 a) `regex( "\\w{5}|\\d{5}" );`
b) `regex( "\\(\\d{3}\\)\\s\\d{3}-\\d{4}" );`
c) `shared_ptr< int > intPtr( new int( 5 ) );`
d) `weak_ptr< int > weakIntPtr( intPtr );`
e)
```
shared_ptr< int > sharedIntPtr = weakIntPtr.lock();
*sharedIntPtr;
```

Exercises

23.4 (***Pig Latin***) Write an application that encodes English language phrases into Pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form Pig Latin phrases. For simplicity, use the following algorithm:

To translate each English word into a Pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus, the word "`jump`" becomes "`umpjay`," the word "`the`" becomes "`hetay`" and the word "`computer`" becomes "`omputercay`." Blanks between words remain blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Enable the user to input a sentence. Use a `regex_token_iterator` to divide the sentence into separate words. Function `getPigLatin` should translate a single word into Pig Latin.

23.5 *(Using Regular Expressions to Convert to Uppercase)* Write a program that uses regular expressions to convert the first letter of all words to uppercase. Have it do this for an arbitrary string input by the user.

23.6 *(Counting Character Types with Regular Expressions)* Use a regular expression to count the number of digits, characters and white-space characters in a string.

23.7 *(Searching for Numbers)* Write a regular expression that will search a string and match a valid number. A number can have any number of digits, but it can have only digits and a decimal point. The decimal point is optional, but if it appears in the number, there must be only one, and it must have digits on its left and its right. There should be white space or a beginning- or end-of-line character on either side of a valid number. Negative numbers are preceded by a minus sign.

23.8 *(Counting HTML Tags)* Write a program that will take HTML as input and will output the number of HTML tags in the string. The program should use regular expressions to count the number of elements nested at each level. For example, the HTML:

```
<p><strong>hi</strong></p>
```

has a `p` element (nesting level 0—i.e., not nested in another tag) and a `strong` element (nesting level 1). For simplicity, use HTML in which none of the elements contain nested elements of the same type—for example, a `table` element should not contain another `table` element.

This solution requires a regular expression concept called a back reference to determine the start and end tags of an HTML element. To find these tags, the same word must appear in the start and end tags. A back reference allows you to use a previous match in the expression in another part of the regular expression. When you enclose a portion of a regular expression in parentheses, the match for that subexpression is stored for you. You can then access the result of that expression using the syntax *digit*, where *digit* is a number in the range 1–9. For example, the regular expression

```
^(7*).*\1$
```

matches an entire string that starts and ends with one or more 7s. The strings `"777abcd777"` and `"7abcdef7"` both match this regular expression. The `\1` in the preceding regular expression is a back reference indicating that whatever matched the subexpression `(7*)` should also appear at the end of the string. The first parenthesized subexpression is back referenced with `\1`, the second is back referenced with `\2`, etc.

You'll need a recursive function so that you can process the nested HTML elements. In each recursive call, you'll need to pass the contents of an element as the string to be processed in that call—for example, the contents of the `p` element in this example's HTML would be

```
<strong>hi</strong>
```

Use parentheses to store the content that appears between the start and end tags of a string that matches your regular expression. This value is stored in the `match_results` object and can be accessed using the `[]` operator on that object. As with back references, the subexpression matches are indexed from 1 to 9.

23.9 *(Removing Extra Spaces)* Write a program that asks the user to enter a sentence and uses a regular expression to check whether the sentence contains more than one space between words. If so, the program should remove the extra spaces. For example, the string `"Hello    World"` should be `"Hello World"`.

23.10 Answer the following questions about smart pointers:

a) Describe a situation in which a custom deleter function would be used.
b) Describe a situation in which you'd use a `weak_ptr` that is not responsible for lifetime management of its resource.

24 Other Topics

What's in a name? that which we call a rose
By any other name would smell as sweet.
—William Shakespeare

O Diamond! Diamond! thou little knowest the mischief done!
—Sir Isaac Newton

Objectives

In this chapter you'll learn:

- To use `const_cast` to temporarily treat a `const` object as a non-`const` object.
- To use `namespace`s.
- To use operator keywords.
- To use `mutable` members in `const` objects.
- To use class-member pointer operators `.*` and `->*`.
- To use multiple inheritance.
- The role of `virtual` base classes in multiple inheritance.

Outline

24.1 Introduction

We now consider additional C++ features. First, we discuss the `const_cast` operator, which allows you to add or remove the `const` qualification of a variable. Next, we discuss `namespaces`, which can be used to ensure that every identifier in a program has a unique name and can help resolve naming conflicts caused by using libraries that have the same variable, function or class names. We then present several operator keywords that are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. We continue our discussion with the `mutable` storage-class specifier, which enables you to indicate that a data member should always be modifiable, even when it appears in an object that's currently being treated as a `const` object by the program. Next we introduce two special operators that we can use with pointers to class members to access a data member or member function without knowing its name in advance. Finally, we introduce multiple inheritance, which enables a derived class to inherit the members of several base classes. As part of this introduction, we discuss potential problems with multiple inheritance and how `virtual` inheritance can be used to solve those problems.

24.2 `const_cast` Operator

C++ provides the **`const_cast`** operator for casting away `const` or `volatile` qualification. You declare a variable with the **`volatile`** qualifier when you expect the variable to be modified by hardware or other programs not known to the compiler. Declaring a variable `volatile` indicates that the compiler should not optimize the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.

In general, it's dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`. There are cases in which it's desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions that have non-`const` parameters and that do not modify their parameters—if you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.

Similarly, you could pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant. In such cases, you might need to cast away the `const`-ness of the returned data, as we demonstrate in Fig. 24.1.

```
1  // Fig. 24.1: fig24_01.cpp
2  // Demonstrating const_cast.
3  #include <iostream>
4  #include <cstring> // contains prototypes for functions strcmp and strlen
5  #include <cctype> // contains prototype for function toupper
6  using namespace std;
7
8  // returns the larger of two C-style strings
9  const char *maximum( const char *first, const char *second )
10 {
11    return ( strcmp( first, second ) >= 0 ? first : second );
12 } // end function maximum
13
14 int main()
15 {
16    char s1[] = "hello"; // modifiable array of characters
17    char s2[] = "goodbye"; // modifiable array of characters
18
19    // const_cast required to allow the const char * returned by maximum
20    // to be assigned to the char * variable maxPtr
21    char *maxPtr = const_cast< char * >( maximum( s1, s2 ) );
22
23    cout << "The larger string is: " << maxPtr << endl;
24
25    for ( size_t i = 0; i < strlen( maxPtr ); ++i )
26       maxPtr[ i ] = toupper( maxPtr[ i ] );
27
28    cout << "The larger string capitalized is: " << maxPtr << endl;
29 } // end main
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

Fig. 24.1 | Demonstrating operator `const_cast`.

In this program, function `maximum` (lines 9–12) receives two C-style strings as `const char *` parameters and returns a `const char *` that points to the larger of the two strings. Function `main` declares the two C-style strings as non-`const char` arrays (lines 16–17); thus, these arrays are modifiable. In `main`, we wish to output the larger of the two C-style strings, then modify that C-style string by converting it to uppercase letters.

Function `maximum`'s two parameters are of type `const char *`, so the function's return type also must be declared as `const char *`. If the return type is specified as only `char *`, the compiler issues an error message indicating that the value being returned cannot be converted from `const char *` to `char *`—a dangerous conversion, because it attempts to treat data that the function believes to be `const` as if it were non-`const` data.

Even though function `maximum` *believes* the data to be constant, we know that the original arrays in `main` do *not* contain constant data. Therefore, `main` should be able to modify the contents of those arrays as necessary. Since we know these arrays are modifiable, we use `const_cast` (line 21) to *cast away the* `const`*-ness* of the pointer returned by `maximum`, so we can then modify the data in the array representing the larger of the two C-style strings. We can then use the pointer as the name of a character array in the `for` statement (lines 25–26) to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 21, this program will not compile, because you are not allowed to assign a pointer of type `const char *` to a pointer of type `char *`.

Error-Prevention Tip 24.1

In general, a `const_cast` should be used only when it is known in advance that the original data is not constant. Otherwise, unexpected results may occur.

24.3 `mutable` Class Members

In Section 24.2, we introduced the `const_cast` operator, which allowed us to remove the "`const`-ness" of a type. A `const_cast` operation can also be applied to a data member of a `const` object from the body of a `const` member function of that object's class. This enables the `const` member function to modify the data member, even though the object is considered to be `const` in the body of that function. Such an operation might be performed when most of an object's data members should be considered `const`, but a particular data member still needs to be modified.

As an example, consider a linked list that maintains its contents in sorted order. Searching through the linked list does not require modifications to the data of the linked list, so the search function could be a `const` member function of the linked-list class. However, it's conceivable that a linked-list object, in an effort to make future searches more efficient, might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match, rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.

If a data member such as the one described above should *always* be modifiable, C++ provides the storage-class specifier **`mutable`** as an alternative to `const_cast`. A `mutable` data member is always modifiable, even in a `const` member function or `const` object.

Portability Tip 24.1

The effect of attempting to modify an object that was defined as constant, regardless of whether that modification was made possible by a `const_cast` or C-style cast, varies among compilers.

`mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` *must* be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable. Operations involving `const_cast` are typically *hidden* in a member function's implementation. The user of a class might not be aware that a member is being modified.

Software Engineering Observation 24.1

`mutable` members are useful in classes that have "secret" implementation details that do not contribute to a client's use of an object of the class.

Mechanical Demonstration of a **`mutable`** *Data Member*

Figure 24.2 demonstrates using a `mutable` member. The program defines class `TestMutable` (lines 7–21), which contains a constructor, function `getValue` and a `private` data member `value` that's declared `mutable`. Lines 15–18 define function `getValue` as a `const` member function that returns a copy of `value`. Notice that the function increments `mutable` data member `value` in the `return` statement. Normally, a `const` member function cannot modify data members unless the object on which the function operates—i.e., the one to which `this` points—is cast (using `const_cast`) to a non-`const` type. Because `value` is `mutable`, this `const` function can modify the data.

```
1  // Fig. 24.2: fig24_02.cpp
2  // Demonstrating storage-class specifier mutable.
3  #include <iostream>
4  using namespace std;
5
6  // class TestMutable definition
7  class TestMutable
8  {
9  public:
10    TestMutable( int v = 0 )
11    {
12       value = v;
13    } // end TestMutable constructor
14
15    int getValue() const
16    {
17       return ++value; // increments value
18    } // end function getValue
19 private:
20    mutable int value; // mutable member
21 }; // end class TestMutable
22
23 int main()
24 {
25    const TestMutable test( 99 );
26
27    cout << "Initial value: " << test.getValue();
28    cout << "\nModified value: " << test.getValue() << endl;
29 } // end main
```

```
Initial value: 99
Modified value: 100
```

Fig. 24.2 | Demonstrating a `mutable` data member.

Line 25 declares `const TestMutable` object `test` and initializes it to `99`. Line 27 calls the `const` member function `getValue`, which adds one to `value` and returns its previous contents. Notice that the compiler *allows* the call to member function `getValue` on the object `test` because it's a `const` object and `getValue` is a `const` member function. However, `getValue` modifies variable `value`. Thus, when line 28 invokes `getValue` again, the new `value` (100) is output to prove that the `mutable` data member was indeed modified.

24.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" (i.e., collide) with a variable of the *same* name in a *different* scope, possibly creating a naming conflict. Such overlapping can occur at many levels. Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions). This can cause compiler errors.

The C++ standard solves this problem with **namespaces**. Each `namespace` defines a scope in which identifiers and variables are placed. To use a **namespace member**, either the member's name must be qualified with the `namespace` name and the scope resolution operator (`::`), as in

```
MyNameSpace::member
```

or a `using` directive must appear *before* the name is used in the program. Typically, such `using` statements are placed at the beginning of the file in which members of the `namespace` are used. For example, placing the following `using` directive at the beginning of a source-code file

```
using namespace MyNameSpace;
```

specifies that members of `namespace` *MyNameSpace* can be used in the file without preceding each member with *MyNameSpace* and the scope resolution operator (`::`).

A `using` directive of the form

```
using std::cout;
```

brings *one* name into the scope where the directive appears. A `using` directive f the form

```
using namespace std;
```

brings *all* the names from the specified namespace (`std`) into the scope where the directive appears.

Error-Prevention Tip 24.2

Precede a member with its `namespace` name and the scope resolution operator (`::`) if the possibility exists of a naming conflict.

Not all `namespace`s are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their `namespace` names. Figure 24.3 demonstrates the use of `namespace`s.

```
// Fig. 24.3: fig24_03.cpp
// Demonstrating namespaces.
#include <iostream>
using namespace std;

int integer1 = 98; // global variable

// create namespace Example
namespace Example
{
```

Fig. 24.3 | Demonstrating the use of `namespace`s. (Part 1 of 3.)

```
   // declare two constants and one variable
   const double PI = 3.14159;
   const double E = 2.71828;
   int integer1 = 8;

   void printValues(); // prototype

   // nested namespace
   namespace Inner
   {
      // define enumeration
      enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
   } // end Inner namespace
} // end Example namespace

// create unnamed namespace
namespace
{
   double doubleInUnnamed = 88.22; // declare variable
} // end unnamed namespace

int main()
{
   // output value doubleInUnnamed of unnamed namespace
   cout << "doubleInUnnamed = " << doubleInUnnamed;

   // output global variable
   cout << "\n(global) integer1 = " << integer1;

   // output values of Example namespace
   cout << "\nPI = " << Example::PI << "\nE = " << Example::E
      << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
      << Example::Inner::FISCAL3 << endl;

   Example::printValues(); // invoke printValues function
} // end main

// display variable and constant values
void Example::printValues()
{
   cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
      << PI << "\nE = " << E << "\ndoubleInUnnamed = "
      << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
      << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
} // end printValues
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992
```

Fig. 24.3 | Demonstrating the use of `namespace`s. (Part 2 of 3.)

```
In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

Fig. 24.3 | Demonstrating the use of `namespace`s. (Part 3 of 3.)

Defining Namespaces

Lines 9–24 use the keyword `namespace` to define namespace `Example`. The body of a namespace is delimited by braces (`{}`). Namespace `Example`'s members consist of two constants (`PI` and `E` in lines 12–13), an `int` (`integer1` in line 14), a function (`printValues` in line 16) and a **nested namespace** (`Inner` in lines 19–23). Notice that member `integer1` has the same name as global variable `integer1` (line 6). *Variables that have the same name must have different scopes*—otherwise compilation errors occur. A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the *global scope* or be *nested* within other namespaces. Unlike classes, different namespace members can be defined in separate `namespace` blocks—each standard library header has a `namespace` block placing its contents in `namespace std`.

Lines 27–30 create an **unnamed namespace** containing the member `doubleInUnnamed`. Variables, classes and functions in an unnamed namespace are accessible only in the current **translation unit** (a `.cpp` file and the files it `include`s). However, unlike variables, classes or functions with `static` linkage, those in the unnamed namespace may be used as template arguments. The unnamed namespace has an implicit `using` directive, so its members appear to occupy the **global namespace**, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file.

Software Engineering Observation 24.2

Each separate compilation unit has its own unique unnamed namespace; i.e., the unnamed namespace replaces the `static` linkage specifier.

Accessing Namespace Members with Qualified Names

Line 35 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the unnamed namespace. Line 38 outputs the value of global variable `integer1`. For both of these variables, the compiler first attempts to locate a local declaration of the variables in `main`. Since there are no local declarations, the compiler assumes those variables are in the global namespace.

Lines 41–43 output the values of `PI`, `E`, `integer1` and `FISCAL3` from namespace `Example`. Notice that each must be qualified with `Example::` because the program does not provide any `using` directive or declarations indicating that it will use members of namespace `Example`. In addition, member `integer1` must be qualified, because a global variable has the same name. Otherwise, the global variable's value is output. `FISCAL3` is a member of nested namespace `Inner`, so it must be qualified with `Example::Inner::`.

Function `printValues` (defined in lines 49–55) is a member of `Example`, so it can access other members of the `Example` namespace directly without using a namespace qualifier. The output statement in lines 51–54 outputs `integer1`, `PI`, `E`, `doubleInUnnamed`, global variable `integer1` and `FISCAL3`. Notice that `PI` and `E` are not qualified with `Example`. Variable `doubleInUnnamed` is still accessible, because it's in the unnamed namespace and the variable name does not conflict with any other members of namespace `Example`. The global version of `integer1` must be qualified with the scope resolution operator (`::`), because its name conflicts with a member of namespace `Example`. Also, `FISCAL3` must be qualified with `Inner::`. When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).

Common Programming Error 24.1

Placing `main` in a namespace is a compilation error.

`using` Directives Should Not *Be Placed in Headers*

Namespaces are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. When working on such projects, there should *never* be a `using` directive in a header. Having one brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors. Instead, use only fully qualified names in headers (for example, `std::cout` or `std::string`).

Aliases for Namespace Names

Namespaces can be *aliased*. For example the statement

```
namespace CPPHTP = CPlusPlusHowToProgram;
```

creates the **namespace alias** `CPPHTP` for `CPlusPlusHowToProgram`.

24.5 Operator Keywords

The C++ standard provides **operator keywords** (Fig. 24.4) that can be used in place of several C++ operators. You can use operator keywords if you have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Operator	Operator keyword	Description
Logical operator keywords		
`&&`	`and`	logical AND
`||`	`or`	logical OR
`!`	`not`	logical NOT
Inequality operator keyword		
`!=`	`not_eq`	inequality

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 1 of 2.)

Operator	Operator keyword	Description
Bitwise operator keywords		
&	`bitand`	bitwise AND
|	`bitor`	bitwise inclusive OR
^	`xor`	bitwise exclusive OR
~	`compl`	bitwise complement
Bitwise assignment operator keywords		
&=	`and_eq`	bitwise AND assignment
|=	`or_eq`	bitwise inclusive OR assignment
^=	`xor_eq`	bitwise exclusive OR assignment

Fig. 24.4 | Operator keyword alternatives to operator symbols. (Part 2 of 2.)

Figure 24.5 demonstrates the operator keywords. Microsoft Visual C++ 2010 requires the header `<ciso646>` (line 4) to use the operator keywords. In GNU C++, this header is empty because the operator keywords are always defined.

```
1  // Fig. 24.5: fig24_05.cpp
2  // Demonstrating operator keywords.
3  #include <iostream>
4  #include <ciso646> // enables operator keywords in Microsoft Visual C++
5  using namespace std;
6
7  int main()
8  {
9     bool a = true;
10    bool b = false;
11    int c = 2;
12    int d = 3;
13
14    // sticky setting that causes bool values to display as true or false
15    cout << boolalpha;
16
17    cout << "a = " << a << "; b = " << b
18       << "; c = " << c << "; d = " << d;
19
20    cout << "\n\nLogical operator keywords:";
21    cout << "\n   a and a: " << ( a and a );
22    cout << "\n   a and b: " << ( a and b );
23    cout << "\n    a or a: " << ( a or a );
24    cout << "\n    a or b: " << ( a or b );
25    cout << "\n     not a: " << ( not a );
26    cout << "\n     not b: " << ( not b );
27    cout << "\na not_eq b: " << ( a not_eq b );
28
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 1 of 2.)

```
29     cout << "\n\nBitwise operator keywords:";
30     cout << "\nc bitand d: " << ( c bitand d );
31     cout << "\nc bit_or d: " << ( c bitor d );
32     cout << "\n   c xor d: " << ( c xor d );
33     cout << "\n   compl c: " << ( compl c );
34     cout << "\nc and_eq d: " << ( c and_eq d );
35     cout << "\n c or_eq d: " << ( c or_eq d );
36     cout << "\nc xor_eq d: " << ( c xor_eq d ) << endl;
37  } // end main
```

```
a = true; b = false; c = 2; d = 3

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true

Bitwise operator keywords:
c bitand d: 2
c bit_or d: 3
   c xor d: 1
   compl c: -3
c and_eq d: 2
 c or_eq d: 3
c xor_eq d: 0
```

Fig. 24.5 | Demonstrating the operator keywords. (Part 2 of 2.)

The program declares and initializes two `bool` variables and two integer variables (lines 9–12). Logical operations (lines 21–27) are performed with `bool` variables `a` and `b` using the various logical operator keywords. Bitwise operations (lines 30–36) are performed with the `int` variables `c` and `d` using the various bitwise operator keywords. The result of each operation is output.

24.6 Pointers to Class Members (`.*` and `->*`)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This is a rarely used capability that's used primarily by advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 24.6 demonstrates the pointer-to-class-member operators.

```
// Fig. 24.6: fig24_06.cpp
// Demonstrating operators .* and ->*.
#include <iostream>
using namespace std;

```

Fig. 24.6 | Demonstrating the `.*` and `->*` operators. (Part 1 of 2.)

```
6  // class Test definition
7  class Test
8  {
9  public:
10    void func()
11    {
12       cout << "In func\n";
13    } // end function func
14
15    int value; // public data member
16 }; // end class Test
17
18 void arrowStar( Test * ); // prototype
19 void dotStar( Test * ); // prototype
20
21 int main()
22 {
23    Test test;
24    test.value = 8; // assign value 8
25    arrowStar( &test ); // pass address to arrowStar
26    dotStar( &test ); // pass address to dotStar
27 } // end main
28
29 // access member function of Test object using ->*
30 void arrowStar( Test *testPtr )
31 {
32    void ( Test::*memberPtr )() = &Test::func; // declare function pointer
33    ( testPtr->*memberPtr )(); // invoke function indirectly
34 } // end arrowStar
35
36 // access members of Test object data member using .*
37 void dotStar( Test *testPtr2 )
38 {
39    int Test::*vPtr = &Test::value; // declare pointer
40    cout << ( *testPtr2 ).*vPtr << endl; // access value
41 } // end dotStar
```

```
In test function
8
```

Fig. 24.6 | Demonstrating the .* and ->* operators. (Part 2 of 2.)

The program declares class `Test` (lines 7–16), which provides `public` member function `test` and `public` data member `value`. Lines 18–19 provide prototypes for the functions `arrowStar` (defined in lines 30–34) and `dotStar` (defined in lines 37–41), which demonstrate the `->*` and `.*` operators, respectively. Lines 23 creates object `test`, and line 24 assigns `8` to its data member `value`. Lines 25–26 call functions `arrowStar` and `dotStar` with the address of the object `test`.

Line 32 in function `arrowStar` declares and initializes variable `memPtr` as a pointer to a member function. In this declaration, `Test::*` indicates that the variable `memPtr` is a pointer to a member of class `Test`. To declare a pointer to a function, enclose the pointer name preceded by `*` in parentheses, as in `(Test::*memPtr)`. A pointer to a function must specify, as part of its type, both the return type of the function it points to and the parameter list of that function. The function's return type appears to the left of the left parenthesis and the parameter list appears in a separate set of parentheses to the right of the pointer declaration. In this case, the function has a `void` return type and no parameters. The pointer `memPtr` is initialized with the address of class `Test`'s member function named `test`. The header of the function must match the function pointer's declaration—i.e., function `test` must have a `void` return type and no parameters. Notice that the right side of the assignment uses the address operator (`&`) to get the address of the member function `test`. Also, notice that *neither the left side nor the right side of the assignment in line 32 refers to a specific object of class* `Test`. Only the class name is used with the scope resolution operator (`::`). Line 33 invokes the member function stored in `memPtr` (i.e., `test`), using the `->*` operator. Because `memPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

Line 39 declares and initializes `vPtr` as a pointer to an `int` data member of class `Test`. The right side of the assignment specifies the address of the data member `value`. Line 40 dereferences the pointer `testPtr2`, then uses the `.*` operator to access the member to which `vPtr` points. *The client code can create pointers to class members for only those class members that are accessible to the client code.* In this example, both member function `test` and data member `value` are publicly accessible.

Common Programming Error 24.2

Declaring a member-function pointer without enclosing the pointer name in parentheses is a syntax error.

Common Programming Error 24.3

Declaring a member-function pointer without preceding the pointer name with a class name followed by the scope resolution operator (`::`) is a syntax error.

Common Programming Error 24.4

Attempting to use the `->` or `*` operator with a pointer to a class member generates syntax errors.

24.7 Multiple Inheritance

In Chapters 12 and 13, we discussed *single inheritance, in which each class is derived from exactly one base class*. In C++, a class may be derived from *more than one* base class—a technique known as **multiple inheritance** in which a derived class inherits the members of two or more base classes. This powerful capability encourages interesting forms of software reuse but can cause a variety of ambiguity problems. *Multiple inheritance is a difficult concept that should be used only by experienced programmers.* In fact, some of the problems associated with multiple inheritance are so subtle that newer programming languages, such as Java and C#, do not enable a class to derive from more than one base class.

Software Engineering Observation 24.3

Great care is required in the design of a system to use multiple inheritance properly; it should not be used when single inheritance and/or composition will do the job.

A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name. This can lead to ambiguity problems when you attempt to compile. Consider the multiple-inheritance example (Fig. 24.7, Fig. 24.8, Fig. 24.9, Fig. 24.10, Fig. 24.11). Class `Base1` (Fig. 24.7) contains one `protected` `int` data member—`value` (line 20), a constructor (lines 10–13) that sets `value` and `public` member function `getData` (lines 15–18) that returns `value`.

```
1  // Fig. 24.7: Base1.h
2  // Definition of class Base1
3  #ifndef BASE1_H
4  #define BASE1_H
5
6  // class Base1 definition
7  class Base1
8  {
9  public:
10    Base1( int parameterValue )
11    {
12       value = parameterValue;
13    } // end Base1 constructor
14
15    int getData() const
16    {
17       return value;
18    } // end function getData
19 protected: // accessible to derived classes
20    int value; // inherited by derived class
21 }; // end class Base1
22
23 #endif // BASE1_H
```

Fig. 24.7 | Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 24.8) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 20). Like class `Base1`, `Base2` has a `public` member function `getData`, but this function returns the value of `char` data member `letter`.

```
1  // Fig. 24.8: Base2.h
2  // Definition of class Base2
3  #ifndef BASE2_H
4  #define BASE2_H
5
6  // class Base2 definition
7  class Base2
8  {
9  public:
10    Base2( char characterData )
11    {
12       letter = characterData;
13    } // end Base2 constructor
```

Fig. 24.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 1 of 2.)

```

   char getData() const
   {
      return letter;
   } // end function getData
protected: // accessible to derived classes
   char letter; // inherited by derived class
}; // end class Base2

#endif // BASE2_H
```

Fig. 24.8 | Demonstrating multiple inheritance—`Base2.h`. (Part 2 of 2.)

Class `Derived` (Figs. 24.9–24.10) inherits from both class `Base1` and class `Base2` through multiple inheritance. Class `Derived` has a `private` data member of type `double` named `real` (line 20), a constructor to initialize all the data of class `Derived` and a `public` member function `getReal` that returns the value of `double` variable `real`.

```
1  // Fig. 24.9: Derived.h
2  // Definition of class Derived which inherits
3  // multiple base classes (Base1 and Base2).
4  #ifndef DERIVED_H
5  #define DERIVED_H
6
7  #include <iostream>
8  #include "Base1.h"
9  #include "Base2.h"
10 using namespace std;
11
12 // class Derived definition
13 class Derived : public Base1, public Base2
14 {
15    friend ostream &operator<<( ostream &, const Derived & );
16 public:
17    Derived( int, char, double );
18    double getReal() const;
19 private:
20    double real; // derived class's private data
21 }; // end class Derived
22
23 #endif // DERIVED_H
```

Fig. 24.9 | Demonstrating multiple inheritance—`Derived.h`.

```
// Fig. 24.10: Derived.cpp
// Member-function definitions for class Derived
#include "Derived.h"

// constructor for Derived calls constructors for
// class Base1 and class Base2.
```

Fig. 24.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 1 of 2.)

```
7  // use member initializers to call base-class constructors
8  Derived::Derived( int integer, char character, double double1 )
9     : Base1( integer ), Base2( character ), real( double1 ) { }
10
11 // return real
12 double Derived::getReal() const
13 {
14    return real;
15 } // end function getReal
16
17 // display all data members of Derived
18 ostream &operator<<( ostream &output, const Derived &derived )
19 {
20    output << "    Integer: " << derived.value << "\n  Character: "
21         << derived.letter << "\nReal number: " << derived.real;
22    return output; // enables cascaded calls
23 } // end operator<<
```

Fig. 24.10 | Demonstrating multiple inheritance—`Derived.cpp`. (Part 2 of 2.)

To indicate multiple inheritance we follow the colon (`:`) after `class Derived` with a comma-separated list of base classes (line 13). In Fig. 24.10, notice that constructor `Derived` explicitly calls base-class constructors for each of its base classes—`Base1` and `Base2`—using the member-initializer syntax (line 9). The *base-class constructors are called in the order that the inheritance is specified, not in the order in which their constructors are mentioned*; also, *if the base-class constructors are not explicitly called in the member-initializer list, their default constructors will be called implicitly*.

The overloaded stream insertion operator (Fig. 24.10, lines 18–23) uses its second parameter—a reference to a `Derived` object—to display a `Derived` object's data. This operator function is a `friend` of `Derived`, so `operator<<` can directly access *all* of class `Derived`'s `protected` and `private` members, including the `protected` data member `value` (inherited from class `Base1`), `protected` data member `letter` (inherited from class `Base2`) and `private` data member `real` (declared in class `Derived`).

Now let's examine the `main` function (Fig. 24.11) that tests the classes in Figs. 24.7–24.10. Line 11 creates `Base1` object `base1` and initializes it to the `int` value `10`, then creates the pointer `base1Ptr` and initializes it to the null pointer (i.e., 0). Line 12 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`, then creates the pointer `base2Ptr` and initializes it to the null pointer. Line 13 creates `Derived` object `derived` and initializes it to contain the `int` value `7`, the `char` value `'A'` and the `double` value `3.5`.

```
1 // Fig. 24.11: fig24_11.cpp
2 // Driver for multiple-inheritance example.
3 #include <iostream>
4 #include "Base1.h"
5 #include "Base2.h"
6 #include "Derived.h"
7 using namespace std;
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 1 of 2.)

```
 8
 9 int main()
10 {
11    Base1 base1( 10 ), *base1Ptr = 0; // create Base1 object
12    Base2 base2( 'Z' ), *base2Ptr = 0; // create Base2 object
13    Derived derived( 7, 'A', 3.5 ); // create Derived object
14
15    // print data members of base-class objects
16    cout << "Object base1 contains integer " << base1.getData()
17       << "\nObject base2 contains character " << base2.getData()
18       << "\nObject derived contains:\n" << derived << "\n\n";
19
20    // print data members of derived-class object
21    // scope resolution operator resolves getData ambiguity
22    cout << "Data members of Derived can be accessed individually:"
23       << "\n    Integer: " << derived.Base1::getData()
24       << "\n  Character: " << derived.Base2::getData()
25       << "\nReal number: " << derived.getReal() << "\n\n";
26    cout << "Derived can be treated as an object of either base class:\n";
27
28    // treat Derived as a Base1 object
29    base1Ptr = &derived;
30    cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
31
32    // treat Derived as a Base2 object
33    base2Ptr = &derived;
34    cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
35 } // end main
```

```
Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
    Integer: 7
  Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
    Integer: 7
  Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 24.11 | Demonstrating multiple inheritance. (Part 2 of 2.)

Lines 16–18 display each object's data values. For objects `base1` and `base2`, we invoke each object's `getData` member function. Even though there are two `getData` functions in this example, the calls are not ambiguous. In line 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 18 displays the contents of object `derived` using the overloaded stream insertion operator.

Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes

Lines 22–25 output the contents of object `derived` again by using the *get* member functions of class `Derived`. However, there is an *ambiguity* problem, because this object contains two `getData` functions, one inherited from class `Base1` and one inherited from class `Base2`. This problem is easy to solve by using the scope resolution operator. The expression `derived.Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `value`) and `derived.Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `letter`). The `double` value in `real` is printed without ambiguity with the call `derived.getReal()`—there are no other member functions with that name in the hierarchy.

Demonstrating the Is-A *Relationships in Multiple Inheritance*

The *is-a* relationships of single inheritance also apply in multiple-inheritance relationships. To demonstrate this, line 29 assigns the address of object `derived` to the `Base1` pointer `base1Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base1`. Line 30 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 33 assigns the address of object `derived` to the `Base2` pointer `base2Ptr`. This is allowed because an object of class `Derived` *is an* object of class `Base2`. Line 34 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

24.8 Multiple Inheritance and `virtual` Base Classes

In Section 24.7, we discussed multiple inheritance, the process by which one class inherits from two or more classes. Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream` (Fig. 24.12).

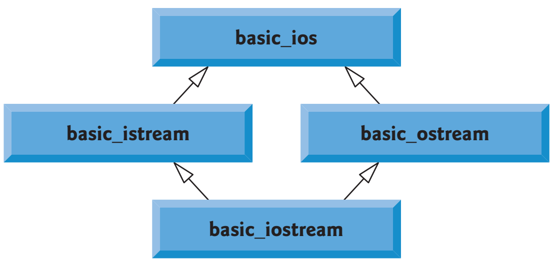


Fig. 24.12 | Multiple inheritance to form class `basic_iostream`.

Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`, each of which is formed with single inheritance. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables class `basic_iostream` objects to provide the functionality of `basic_istreams` and `basic_ostreams`. In multiple-inheritance hierarchies, the situation described in Fig. 24.12 is referred to as **diamond inheritance**

Because classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, a potential problem exists for `basic_iostream`. Class `basic_iostream` could contain *two* copies of the members of class `basic_ios`—one inherited via class `basic_istream` and one inherited via class `basic_ostream`). Such a situation would be ambiguous and would result in a compilation error, because the compiler would not know which version of the members from class `basic_ios` to use. Of course, `basic_iostream` does not really suffer from the problem we mentioned. In this section, you'll see how using `virtual` base classes solves the problem of inheriting duplicate copies of an indirect base class.

Compilation Errors Produced When Ambiguity Arises in Diamond Inheritance

Figure 24.13 demonstrates the ambiguity that can occur in diamond inheritance. Class `Base` (lines 8–12) contains pure `virtual` function `print` (line 11). Classes `DerivedOne` (lines 15–23) and `DerivedTwo` (lines 26–34) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain what the C++ standard refers to as a **base-class subobject**—i.e., the members of class `Base` in this example.

```
1  // Fig. 24.13: fig24_13.cpp
2  // Attempting to polymorphically call a function that is
3  // multiply inherited from two base classes.
4  #include <iostream>
5  using namespace std;
6
7  // class Base definition
8  class Base
9  {
10 public:
11    virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : public Base
16 {
17 public:
18    // override print function
19    void print() const
20    {
21       cout << "DerivedOne\n";
22    } // end function print
23 }; // end class DerivedOne
24
25 // class DerivedTwo definition
26 class DerivedTwo : public Base
27 {
28 public:
29    // override print function
30    void print() const
31    {
32       cout << "DerivedTwo\n";
33    } // end function print
34 }; // end class DerivedTwo
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 1 of 2.)

```
39 public:
40    // qualify which version of function print
41    void print() const
42    {
43       DerivedTwo::print();
44    } // end function print
45 }; // end class Multiple
46
47 int main()
48 {
49    Multiple both; // instantiate Multiple object
50    DerivedOne one; // instantiate DerivedOne object
51    DerivedTwo two; // instantiate DerivedTwo object
52    Base *array[ 3 ]; // create array of base-class pointers
53
54    array[ 0 ] = &both; // ERROR--ambiguous
55    array[ 1 ] = &one;
56    array[ 2 ] = &two;
57
58    // polymorphically invoke print
59    for ( int i = 0; i < 3; ++i )
60       array[ i ] -> print();
61 } // end main
```

Microsoft Visual C++ compiler error message:

```
c:\cpphtp8_examples\ch25\Fig24_13\fig24_13.cpp(54) : error C2594: '=' :
   ambiguous conversions from 'Multiple *' to 'Base *'
```

GNU C++ compiler error message:

```
fig24_13.cpp: In function ‘int main()’:
fig24_13.cpp:54: error: ‘Base’ is an ambiguous base of ‘Multiple’
```

Fig. 24.13 | Attempting to call a multiply inherited function polymorphically. (Part 2 of 2.)

Class `Multiple` (lines 37–45) inherits from both classes `DerivedOne` and `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 43). Notice that we must qualify the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 47–61) declares objects of classes `Multiple` (line 49), `DerivedOne` (line 50) and `DerivedTwo` (line 51). Line 52 declares an array of `Base *` pointers. Each array element is initialized with the address of an object (lines 54–56). An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[ 0 ]`. The object `both` actually contains two subobjects of type `Base`, so the compiler does not know which subobject the pointer `array[ 0 ]` should point to, and it generates a compilation error indicating an ambiguous conversion.

Eliminating Duplicate Subobjects with `virtual` *Base-Class Inheritance*

The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only one subobject will appear in the derived class—a process called **`virtual` base-class inheritance**. Figure 24.14 revises the program of Fig. 24.13 to use a `virtual` base class.

```
// Fig. 24.14: fig24_14.cpp
// Using virtual base classes.
#include <iostream>
using namespace std;

// class Base definition
class Base
{
public:
   virtual void print() const = 0; // pure virtual
}; // end class Base

// class DerivedOne definition
class DerivedOne : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedOne\n";
   } // end function print
}; // end DerivedOne class

// class DerivedTwo definition
class DerivedTwo : virtual public Base
{
public:
   // override print function
   void print() const
   {
      cout << "DerivedTwo\n";
   } // end function print
}; // end DerivedTwo class

// class Multiple definition
class Multiple : public DerivedOne, public DerivedTwo
{
public:
   // qualify which version of function print
   void print() const
   {
      DerivedTwo::print();
   } // end function print
}; // end Multiple class

int main()
{
   Multiple both; // instantiate Multiple object
   DerivedOne one; // instantiate DerivedOne object
```

Fig. 24.14 | Using `virtual` base classes. (Part 1 of 2.)

```
50     DerivedTwo two; // instantiate DerivedTwo object
51
52     // declare array of base-class pointers and initialize
53     // each element to a derived-class type
54     Base *array[ 3 ];
55     array[ 0 ] = &both;
56     array[ 1 ] = &one;
57     array[ 2 ] = &two;
58
59     // polymorphically invoke function print
60     for ( int i = 0; i < 3; ++i )
61        array[ i ]->print();
62  } // end main
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 24.14 | Using `virtual` base classes. (Part 2 of 2.)

The key change is that classes `DerivedOne` (line 14) and `DerivedTwo` (line 25) each inherit from `Base` by specifying `virtual public Base`. Since both classes inherit from `Base`, they each contain a `Base` subobject. The benefit of `virtual` inheritance is not clear until class `Multiple` inherits from `DerivedOne` and `DerivedTwo` (line 36). Since each of the base classes used `virtual` inheritance to inherit class `Base`'s members, the compiler ensures that only one `Base` subobject is inherited into class `Multiple`. This eliminates the ambiguity error generated by the compiler in Fig. 24.13. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[ 0 ]` in line 55 in `main`. The `for` statement in lines 60–61 polymorphically calls `print` for each object.

Constructors in Multiple-Inheritance Hierarchies with `virtual` Base Classes

Implementing hierarchies with `virtual` base classes is simpler if default constructors are used for the base classes. Figures 24.13 and 24.14 use compiler-generated ones. If a `virtual` base class provides a constructor that requires arguments, the derived-class implementations become more complicated, because the **most derived class** must explicitly invoke the `virtual` base class's constructor.

Software Engineering Observation 24.4

Providing a default constructor for `virtual` base classes simplifies hierarchy design.

Additional Information on Multiple Inheritance

Multiple inheritance is a complex topic typically covered in more advanced C++ texts. For more information on multiple inheritance, please visit our C++ Resource Center at

```
www.deitel.com/cplusplus/
```

In the *C++ Multiple Inheritance* category, you'll find links to several articles and resources, including a multiple inheritance FAQ and tips for using multiple inheritance.

24.9 Wrap-Up

In this chapter, you learned how to use the `const_cast` operator to remove the `const` qualification of a variable. We then showed how to use `namespace`s to ensure that every identifier in a program has a unique name and explained how they can help resolve naming conflicts. You saw several operator keywords to use if your keyboards do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. Next, we showed how the `mutable` storage-class specifier enables you to indicate that a data member should always be modifiable, even when it appears in an object that's currently being treated as a `const`. We also showed the mechanics of using pointers to class members and the `->*` and `.*` operators. Finally, we introduced multiple inheritance and discussed problems associated with allowing a derived class to inherit the members of several base classes. As part of this discussion, we demonstrated how `virtual` inheritance can be used to solve those problems.

Summary

*Section 24.2 **`const_cast`** Operator*

- C++ provides the `const_cast` operator for casting away `const` or `volatile` qualification.
- A program declares a variable with the `volatile` qualifier (p. 975) when that program expects the variable to be modified by other programs. Declaring a variable `volatile` indicates that the compiler should not optimize the use of that variable because doing so could affect the ability of those other programs to access and modify the `volatile` variable.
- In general, it is dangerous to use the `const_cast` operator, because it allows a program to modify a variable that was declared `const`, and thus was not supposed to be modifiable.
- There are cases in which it is desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions with non-`const` parameters and that do not modify their parameters. If you wish to pass `const` data to such a function, you'd need to cast away the data's `const`-ness; otherwise, the compiler would report error messages.
- If you pass non-`const` data to a function that treats the data as if it were constant, then returns that data as a constant, you might need to cast away the `const`-ness of the returned data to access and modify that data.

*Section 24.3 **`mutable`** Class Members*

- If a data member should always be modifiable, C++ provides the storage-class specifier `mutable` as an alternative to `const_cast`. A `mutable` data member (p. 977) is always modifiable, even in a `const` member function or `const` object. This reduces the need to cast away "`const`-ness."
- `mutable` and `const_cast` are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` must be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable.
- Operations involving `const_cast` are typically hidden in a member function's implementation. The user of a class might not be aware that a member is being modified.

*Section 24.4 **`namespace`**s*

- A program includes many identifiers defined in different scopes. Sometimes a variable of one scope will "overlap" with a variable of the same name in a different scope, possibly creating a naming conflict. The C++ standard solves this problem with `namespace`s (p. 979).

- Each `namespace` defines a scope in which identifiers are placed. To use a `namespace` member (p. 979), either the member's name must be qualified with the `namespace` name and the scope resolution operator (`::`) or a `using` directive or declaration must appear before the name is used in the program.
- Typically, `using` statements are placed at the beginning of the file in which members of the `namespace` are used.
- Not all `namespaces` are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their `namespace` names.
- A namespace can contain constants, data, classes, nested namespaces (p. 981), functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces.
- An unnamed namespace (p. 981) has an implicit `using` directive, so its members appear to occupy the global namespace, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace.
- When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).
- Namespaces can be aliased (p. 982).

Section 24.5 Operator Keywords
- The C++ standard provides operator keywords (p. 982) that can be used in place of several C++ operators. Operator keywords are useful for programmers who have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Section 24.6 Pointers to Class Members (`.*` and `->*`)
- C++ provides the `.*` and `->*` operators (p. 984) for accessing class members via pointers. This is a rarely used capability that's used primarily by advanced C++ programmers.
- Declaring a pointer to a function requires that you enclose the pointer name preceded by an `*` in parentheses. A pointer to a function must specify, as part of its type, both the return type of the function it points to and the parameter list of that function.

Section 24.7 Multiple Inheritance
- In C++, a class may be derived from more than one base class—a technique known as multiple inheritance (p. 986), in which a derived class inherits the members of two or more base classes.
- A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name. This can lead to ambiguity problems when you attempt to compile.
- The *is-a* relationships of single inheritance also apply in multiple-inheritance relationships.
- Multiple inheritance is used in the C++ Standard Library to form class `basic_iostream`. Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. In multiple-inheritance hierarchies, the situation described here is referred to as diamond inheritance.

Section 24.8 Multiple Inheritance and `virtual` Base Classes
- The ambiguity in diamond inheritance (p. 991) occurs when a derived-class object inherits two or more base-class subobjects (p. 992). The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual` (p. 994), only one subobject will appear in the derived class—a process called `virtual` base-class inheritance.
- Implementing hierarchies with `virtual` base classes is simpler if default constructors are used for the base classes. If a `virtual` base class provides a constructor that requires arguments, the im-

plementation of the derived classes becomes more complicated, because the most derived class (p. 995) must explicitly invoke the virtual base class's constructor to initialize the members inherited from the `virtual` base class.

Self-Review Exercises

24.1 Fill in the blanks for each of the following:

a) The ________ operator qualifies a member with its namespace.
b) The ________ operator allows an object's "`const`-ness" to be cast away.
c) Because an unnamed `namespace` has an implicit `using` directive, its members appear to occupy the ________, are accessible directly and do not have to be qualified with a `namespace` name.
d) Operator ________ is the operator keyword for inequality.
e) ________ allows a class to be derived from more than one base class.
f) When a base class is inherited as ________, only one subobject of the base class will appear in the derived class.

24.2 State which of the following are *true* and which are *false*. If a statement is *false*, explain why.

a) When passing a non-`const` argument to a `const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the function.
b) A `mutable` data member cannot be modified in a `const` member function.
c) `namespace`s are guaranteed to be unique.
d) Like class bodies, `namespace` bodies also end in semicolons.
e) `namespace`s cannot have `namespace`s as members.

Answers to Self-Review Exercises

24.1 a) binary scope resolution (`::`). b) `const_cast`. c) global `namespace`. d) `not_eq`. e) multiple inheritance. f) `virtual`.

24.2 a) False. It is legal to pass a non-`const` argument to a `const` function. However, when passing a `const` reference or pointer to a non-`const` function, the `const_cast` operator should be used to cast away the "`const`-ness" of the reference or pointer
b) False. A `mutable` data member is always modifiable, even in a `const` member function.
c) False. Programmers might inadvertently choose the `namespace` already in use.
d) False. `namespace` bodies do not end in semicolons.
e) False. `namespace`s can be nested.

Exercises

24.3 ***(Fill in the Blanks)*** Fill in the blanks for each of the following:

a) Keyword ________ specifies that a `namespace` or `namespace` member is being used.
b) Operator ________ is the operator keyword for logical OR.
c) Storage specifier ________ allows a member of a `const` object to be modified.
d) The ________ qualifier specifies that an object can be modified by other programs.
e) Precede a member with its ________ name and the scope resolution operator if the possibility exists of a scoping conflict.
f) The body of a namespace is delimited by ________.
g) For a `const` object with no ________ data members, operator ________ must be used every time a member is to be modified.

24.4 ***(`Currency namespace`)*** Write a `namespace`, `Currency`, that defines constant members `ONE`, `TWO`, `FIVE`, `TEN`, `TWENTY`, `FIFTY` and `HUNDRED`. Write two short programs that use `Currency`. One program should make all constants available and the other should make only `FIVE` available.

24.5 Given the namespaces in Fig. 24.15, determine whether each statement is *true* or *false*. Explain any *false* answers.

a) Variable `kilometers` is visible within namespace `Data`.
b) Object `string1` is visible within namespace `Data`.
c) Constant `POLAND` is not visible within namespace `Data`.
d) Constant `GERMANY` is visible within namespace `Data`.
e) Function `function` is visible to namespace `Data`.
f) Namespace `Data` is visible to namespace `CountryInformation`.
g) Object `map` is visible to namespace `CountryInformation`.
h) Object `string1` is visible within namespace `RegionalInformation`.

```
namespace CountryInformation
{
   using namespace std;
   enum Countries { POLAND, SWITZERLAND, GERMANY,
                    AUSTRIA, CZECH_REPUBLIC };
   int kilometers;
   string string1;

   namespace RegionalInformation
   {
      short getPopulation(); // assume definition exists
      MapData map; // assume definition exists
   } // end RegionalInformation
} // end CountryInformation

namespace Data
{
   using namespace CountryInformation::RegionalInformation;
   void *function( void *, int );
} // end Data
```

Fig. 24.15 | namespaces for Exercise 24.5.

24.6 Compare and contrast `mutable` and `const_cast`. Give at least one example of when one might be preferred over the other. [*Note:* This exercise does not require any code to be written.]

24.7 *(Modifying a `const` Variable)* Write a program that uses `const_cast` to modify a `const` variable. [*Hint:* Use a pointer in your solution to point to the `const` identifier.]

24.8 *(`virtual` Base Classes)* What problem do `virtual` base classes solve?

24.9 *(`virtual` Base Classes)* Write a program that uses `virtual` base classes. The class at the top of the hierarchy should provide a constructor that takes at least one argument (i.e., do not provide a default constructor). What challenges does this present for the inheritance hierarchy?

24.10 *(Find the Code Errors)* Find the error(s) in each of the following. When possible, explain how to correct each error.

a)
```
namespace Name {
   int x;
   int y;
   mutable int z;
};
```
b)
```
int integer = const_cast< int >( double );
```
c)
```
namespace PCM( 111, "hello" ); // construct namespace
```

This page intentionally left blank

Chapters on the Web

The following chapters are available as PDF documents from this book's Companion Website, which is accessible from `www.pearsonhighered.com/deitel/`:

- Chapter 25, ATM Case Study, Part 1: Object-Oriented Design with the UML
- Chapter 26, ATM Case Study, Part 2: Implementing an Object-Oriented Design
- Game Programming with OGRE (from *C++ How to Program, Seventh Edition*)

These files can be viewed in Adobe® Reader® (`get.adobe.com/reader`). The index entries for these chapters include the chapter number and an underscore, as in 25_1.

New copies of this book come with a Companion Website access code that is located on the card inside the book's front cover. If the access code is already visible or there is no card, you purchased a used book or an edition that does not come with an access code. In this case, you can purchase access directly from the Companion Website.

A

Operator Precedence and Associativity

Operators are shown in decreasing order of precedence from top to bottom (Fig. A.1).

Operator	Type	Associativity
`::`	binary scope resolution	left to right
`::`	unary scope resolution	
`()`	grouping parentheses *[See caution in Fig. 2.10]*	
`()`	function call	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postfix increment	
`--`	unary postfix decrement	
`typeid`	runtime type information	
`dynamic_cast` < *type* >	runtime type-checked cast	
`static_cast`< *type* >	compile-time type-checked cast	
`reinterpret_cast`< *type* >	cast for nonstandard conversions	
`const_cast`< *type* >	cast away `const`-ness	
`++`	unary prefix increment	right to left
`--`	unary prefix decrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`~`	unary bitwise complement	
`sizeof`	determine size in bytes	
`&`	address	
`*`	dereference	
`new`	dynamic memory allocation	
`new[]`	dynamic array allocation	
`delete`	dynamic memory deallocation	
`delete[]`	dynamic array deallocation	
(*type*)	C-style unary cast	right to left

Fig. A.1 | Operator precedence and associativity chart. (Part 1 of 2.)

Operator	Type	Associativity
.*	pointer to member via object	left to right
->*	pointer to member via pointer	
*	multiplication	left to right
/	division	
%	modulus	
+	addition	left to right
-	subtraction	
<<	bitwise left shift	left to right
>>	bitwise right shift	
<	relational less than	left to right
<=	relational less than or equal to	
>	relational greater than	
>=	relational greater than or equal to	
==	relational is equal to	left to right
!=	relational is not equal to	
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
?:	ternary conditional	right to left
=	assignment	right to left
+=	addition assignment	
-=	subtraction assignment	
*=	multiplication assignment	
/=	division assignment	
%=	modulus assignment	
&=	bitwise AND assignment	
^=	bitwise exclusive OR assignment	
|=	bitwise inclusive OR assignment	
<<=	bitwise left-shift assignment	
>>=	bitwise right-shift assignment	
,	comma	left to right

Fig. A.1 | Operator precedence and associativity chart. (Part 2 of 2.)

B

ASCII Character Set

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | ASCII character set.

The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

Most users of this book are interested in the ASCII character set used to represent English characters on many computers. The ASCII character set is a subset of the Unicode character set that represents characters from most of the world’s languages.

C

Fundamental Types

Figure C.1 lists C++'s fundamental types. The C++ Standard Document does not provide the exact number of bytes required to store variables of these types in memory. However, the C++ Standard Document does indicate how the memory requirements for fundamental types relate to one another. By order of increasing memory requirements, the signed integer types are `signed char`, `short int`, `int` and `long int`. This means that a `short int` must provide at least as much storage as a `signed char`; an `int` must provide at least as much storage as a `short int`; and a `long int` must provide at least as much storage as an `int`. Each signed integer type has a corresponding unsigned integer type that has the same memory requirements. Unsigned types cannot represent negative values, but can represent twice as many positive values as their associated signed types. By order of increasing memory requirements, the floating-point types are `float`, `double` and `long double`. Like integer types, a `double` must provide at least as much storage as a `float` and a `long double` must provide at least as much storage as a `double`.

Integral types	Floating-point types
`bool`	`float`
`char`	`double`
`signed char`	`long double`
`unsigned char`	
`short int`	
`unsigned short int`	
`int`	
`unsigned int`	
`long int`	
`unsigned long int`	
`wchar_t`	

Fig. C.1 | C++ fundamental types.

The exact sizes and ranges of values for the fundamental types are implementation dependent. The header files `<climits>` (for the integral types) and `<cfloat>` (for the floating-point types) specify the ranges of values supported on your system.

The range of values a type supports depends on the number of bytes that are used to represent that type. For example, consider a system with 4 byte (32 bit) `int`s. For the signed `int` type, the nonnegative values are in the range 0 to 2,147,483,647 ($2^{31} - 1$). The negative values are in the range –1 to –2,147,483,648 (-2^{31}). This is a total of 2^{32} possible values. An `unsigned int` on the same system would use the same number of bits to represent data, but would not represent any negative values. This results in values in the range 0 to 4,294,967,295 ($2^{32} - 1$). On the same system, a `short int` could not use more than 32 bits to represent its data and a `long int` must use at least 32 bits.

C++ provides the data type `bool` for variables that can hold only the values `true` and `false`. The new C++ standard introduces the types `long long` and `unsigned long long`—typically for 64-bit integer values (though this is not required by the standard).

Number Systems

D

Here are only numbers ratified.
—William Shakespeare

Objectives

In this appendix you'll learn:

- To understand basic number systems concepts, such as base, positional value and symbol value.
- To understand how to work with numbers in the binary, octal and hexadecimal number systems.
- To abbreviate binary numbers as octal numbers or hexadecimal numbers.
- To convert octal numbers and hexadecimal numbers to binary numbers.
- To convert back and forth between decimal numbers and their binary, octal and hexadecimal equivalents.
- To understand binary arithmetic and how negative binary numbers are represented using two's complement notation.

Outline

D.1 Introduction

In this appendix, we introduce the key number systems that C++ programmers use, especially when they are working on software projects that require close interaction with machine-level hardware. Projects like this include operating systems, computer networking software, compilers, database systems and applications requiring high performance.

When we write an integer such as 227 or –63 in a C++ program, the number is assumed to be in the **decimal (base 10) number system**. The **digits** in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0 and the highest is 9—one less than the base of 10. Internally, computers use the **binary (base 2) number system**. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest is 1—one less than the base of 2.

As we'll see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages, and in high-level languages like C++ that enable them to reach down to the machine level, find it cumbersome to work with binary numbers. So two other number systems—the **octal number system (base 8)** and the **hexadecimal number system (base 16)**—are popular, primarily because they make it convenient to abbreviate binary numbers.

In the octal number system, the digits range from 0 to 7. Because both the binary and the octal number systems have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system poses a problem because it requires 16 digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers. The digits of the binary, octal, decimal and hexadecimal number systems are summarized in Figs. D.1–D.2.

Each of these number systems uses **positional notation**—each position in which a digit is written has a different **positional value**. For example, in the decimal number 937 (the 9, the 3 and the 7 are referred to as **symbol values**), we say that the 7 is written in the ones position, the 3 is written in the tens position and the 9 is written in the hundreds position. Note that each of these positions is a power of the base (base 10) and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. D.3).

Binary digit	Octal digit	Decimal digit	Hexadecimal digit
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (decimal value of 10)
			B (decimal value of 11)
			C (decimal value of 12)
			D (decimal value of 13)
			E (decimal value of 14)
			F (decimal value of 15)

Fig. D.1 | Digits of the binary, octal, decimal and hexadecimal number systems.

Attribute	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Lowest digit	0	0	0	0
Highest digit	1	7	9	F

Fig. D.2 | Comparing the binary, octal, decimal and hexadecimal number systems.

Positional values in the decimal number system			
Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional value as a power of the base (10)	10^2	10^1	10^0

Fig. D.3 | Positional values in the decimal number system.

For longer decimal numbers, the next positions to the left would be the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power) and so on.

In the binary number 101, the rightmost 1 is written in the ones position, the 0 is written in the twos position and the leftmost 1 is written in the fours position. Note that each position is a power of the base (base 2) and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. D.4). So, $101 = 2^2 + 2^0 = 4 + 1 = 5$.

Positional values in the binary number system			
Binary digit	1	0	1
Position name	Fours	Twos	Ones
Positional value	4	2	1
Positional value as a power of the base (2)	2^2	2^1	2^0

Fig. D.4 | Positional values in the binary number system.

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fours position (2 to the 6th power) and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position and the 4 is written in the sixty-fours position. Note that each of these positions is a power of the base (base 8) and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. D.5).

Positional values in the octal number system			
Decimal digit	4	2	5
Position name	Sixty-fours	Eights	Ones
Positional value	64	8	1
Positional value as a power of the base (8)	8^2	8^1	8^0

Fig. D.5 | Positional values in the octal number system.

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty-eights position (8 to the 5th power) and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position and the 3 is written in the two-hundred-and-fifty-sixes position. Note that each of these positions is a power of the base (base 16) and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. D.6).

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-sixes position (16 to the 4th power) and so on.

Positional values in the hexadecimal number system			
Decimal digit	3	D	A
Position name	Two-hundred-and-fifty-sixes	Sixteens	Ones
Positional value	256	16	1
Positional value as a power of the base (16)	16^2	16^1	16^0

Fig. D.6 | Positional values in the hexadecimal number system.

D.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

The main use for octal and hexadecimal numbers in computing is for abbreviating lengthy binary representations. Figure D.7 highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system.

Decimal number	Binary representation	Octal representation	Hexadecimal representation
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. D.7 | Decimal, binary, octal and hexadecimal equivalents.

A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2).

Consider the following 12-digit binary number and its octal and hexadecimal equivalents. See if you can determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal. The answers follow the numbers.

Binary number	Octal equivalent	Hexadecimal equivalent
`100011010001`	`4321`	`8D1`

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, starting from the right, and write those groups over the corresponding digits of the octal number as follows:

```
100   011   010   001
4     3     2     1
```

Note that the octal digit you've written under each group of three bits corresponds precisely to the octal equivalent of that 3-digit binary number, as shown in Fig. D.7.

The same kind of relationship can be observed in converting from binary to hexadecimal. Break the 12-digit binary number into groups of four consecutive bits each, starting from the right, and write those groups over the corresponding digits of the hexadecimal number as follows:

```
1000  1101  0001
8     D     1
```

Notice that the hexadecimal digit you wrote under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number as shown in Fig. D.7.

D.3 Converting Octal and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting them as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101 and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101 and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

D.4 Converting from Binary, Octal or Hexadecimal to Decimal

We are accustomed to working in decimal, and therefore it is often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is "really" worth. Our diagrams in Section D.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum these products. For example, the binary number 110101 is converted to decimal 53 as shown in Fig. D.8.

Converting a binary number to decimal						
Positional values:	32	16	8	4	2	1
Symbol values:	1	1	0	1	0	1
Products:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Sum:	= 32 + 16 + 0 + 4 + 0s + 1 = 53					

Fig. D.8 | Converting a binary number to decimal.

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values, as shown in Fig. D.9.

Converting an octal number to decimal				
Positional values:	512	64	8	1
Symbol values:	7	6	1	4
Products	7*512=3584	6*64=384	1*8=8	4*1=4
Sum:	= 3584 + 384 + 8 + 4 = 3980			

Fig. D.9 | Converting an octal number to decimal.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values, as shown in Fig. D.10.

Converting a hexadecimal number to decimal				
Positional values:	4096	256	16	1
Symbol values:	A	D	3	B
Products	A*4096=40960	D*256=3328	3*16=48	B*1=11
Sum:	= 40960 + 3328 + 48 + 11 = 44347			

Fig. D.10 | Converting a hexadecimal number to decimal.

D.5 Converting from Decimal to Binary, Octal or Hexadecimal

The conversions in Section D.4 follow naturally from the positional notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	64	32	16	8	4	2	1

Then we discard the column with positional value 64, leaving:

Positional values:	32	16	8	4	2	1

Next we work from the leftmost column to the right. We divide 32 into 57 and observe that there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe that there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe that there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of 0 when their positional values are divided into 1, so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1, so we write 1 in the 1 column. This yields:

Positional values:	32	16	8	4	2	1
Symbol values:	1	1	1	0	0	1

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	512	64	8	1

Then we discard the column with positional value 512, yielding:

Positional values:	64	8	1

Next we work from the leftmost column to the right. We divide 64 into 103 and observe that there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values:	64	8	1
Symbol values:	1	4	7

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	4096	256	16	1

Then we discard the column with positional value 4096, yielding:

Positional values:	256	16	1

Next we work from the leftmost column to the right. We divide 256 into 375 and observe that there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values:	256	16	1
Symbol values:	1	7	7

and thus decimal 375 is equivalent to hexadecimal 177.

D.6 Negative Binary Numbers: Two's Complement Notation

The discussion so far in this appendix has focused on positive numbers. In this section, we explain how computers represent negative numbers using **two's complement notation**. First we explain how the two's complement of a binary number is formed, then we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose

```
int value = 13;
```

The 32-bit representation of `value` is

```
00000000 00000000 00000000 00001101
```

To form the negative of `value` we first form its **one's complement** by applying C++'s **bitwise complement operator (~)**:

```
onesComplementOfValue = ~value;
```

Internally, `~value` is now `value` with each of its bits reversed—ones become zeros and zeros become ones, as follows:

```
value:
00000000 00000000 00000000 00001101

~value (i.e., value's one's complement):
11111111 11111111 11111111 11110010
```

To form the two's complement of `value`, we simply add 1 to `value`'s one's complement. Thus

```
Two's complement of value:
11111111 11111111 11111111 11110011
```

Now if this is in fact equal to –13, we should be able to add it to binary 13 and obtain a result of 0. Let's try this:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
------------------------------------
 00000000 00000000 00000000 00000000
```

The carry bit coming out of the leftmost column is discarded and we indeed get 0 as a result. If we add the one's complement of a number to the number, the result will be all 1s. The key to getting a result of all zeros is that the two's complement is one more than the one's complement. The addition of 1 causes each column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit, and thus the resulting number is all zeros.

Computers actually perform a subtraction, such as

```
x = a - value;
```

by adding the two's complement of `value` to `a`, as follows:

```
x = a + (~value + 1);
```

Suppose a is 27 and value is 13 as before. If the two's complement of value is actually the negative of value, then adding the two's complement of value to a should produce the result 14. Let's try this:

```
a (i.e., 27)        00000000 00000000 00000000 00011011
+(~value + 1)      +11111111 11111111 11111111 11110011
                   ------------------------------------
                    00000000 00000000 00000000 00001110
```

which is indeed equal to 14.

Summary

- An integer such as 19 or 227 or –63 in a C++ program is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0 and the highest is 9—one less than the base of 10.
- Computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest is 1—one less than the base of 2.
- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.
- The digits of the octal number system range from 0 to 7.
- The hexadecimal number system poses a problem because it requires 16 digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Each number system uses positional notation—each position in which a digit is written has a different positional value.
- A particularly important relationship of both the octal and the hexadecimal number systems to the binary system is that their bases (8 and 16 respectively) are powers of the base of the binary number system (base 2).
- To convert from octal to binary, replace each octal digit with its three-digit binary equivalent.
- To convert a hexadecimal to a binary number, simply replace each hexadecimal digit with its four-digit binary equivalent.
- Because we are accustomed to working in decimal, it is convenient to convert a binary, octal or hexadecimal number to decimal to get a sense of the number's "real" worth.
- To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum the products.
- Computers represent negative numbers using two's complement notation.
- To form the negative of a value in binary, first form its one's complement by applying C++'s bitwise complement operator (~). This reverses the bits of the value. To form the two's complement of a value, simply add one to the value's one's complement.

Self-Review Exercises

D.1 The bases of the decimal, binary, octal and hexadecimal number systems are ________, ________, ________ and ________ respectively.

D.2 In general, the decimal, octal and hexadecimal representations of a given binary number contain (more/fewer) digits than the binary number contains.

D.3 (*True/False*) A popular reason for using the decimal number system is that it forms a convenient notation for abbreviating binary numbers simply by substituting one decimal digit per group of four binary bits.

D.4 The [octal/hexadecimal/decimal] representation of a large binary value is the most concise (of the given alternatives).

D.5 (*True/False*) The highest digit in any base is one more than the base.

D.6 (*True/False*) The lowest digit in any base is one less than the base.

D.7 The positional value of the rightmost digit of any number in either binary, octal, decimal or hexadecimal is always ________.

D.8 The positional value of the digit to the left of the rightmost digit of any number in binary, octal, decimal or hexadecimal is always equal to ________.

D.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binary	...	...	...	...
octal	512	...	8	...

D.10 Convert binary 110101011000 to octal and to hexadecimal.

D.11 Convert hexadecimal FACE to binary.

D.12 Convert octal 7316 to binary.

D.13 Convert hexadecimal 4FEC to octal. [*Hint:* First convert 4FEC to binary, then convert that binary number to octal.]

D.14 Convert binary 1101110 to decimal.

D.15 Convert octal 317 to decimal.

D.16 Convert hexadecimal EFD4 to decimal.

D.17 Convert decimal 177 to binary, to octal and to hexadecimal.

D.18 Show the binary representation of decimal 417. Then show the one's complement of 417 and the two's complement of 417.

D.19 What's the result when a number and its two's complement are added to each other?

Answers to Self-Review Exercises

D.1 10, 2, 8, 16.

D.2 Fewer.

D.3 False. Hexadecimal does this.

D.4 Hexadecimal.

D.5 False. The highest digit in any base is one less than the base.

D.6 False. The lowest digit in any base is zero.

D.7 1 (the base raised to the zero power).

D.8 The base of the number system.

D.9 Filled in chart shown below:

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binary	8	4	2	1
octal	512	64	8	1

D.10 Octal 6530; Hexadecimal D58.

D.11 Binary 1111 1010 1100 1110.

D.12 Binary 111 011 001 110.

D.13 Binary 0 100 111 111 101 100; Octal 47754.

D.14 Decimal 2 + 4 + 8 + 32 + 64 = 110.

D.15 Decimal 7 + 1 * 8 + 3 * 64 = 7 + 8 + 192 = 207.

D.16 Decimal 4 + 13 * 16 + 15 * 256 + 14 * 4096 = 61396.

D.17 Decimal 177
to binary:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

to octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

to hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

D.18 Binary:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001
```

One's complement: 001011110
Two's complement: 001011111
Check: Original binary number + its two's complement

```
110100001
001011111
---------
000000000
```

D.19 Zero.

Exercises

D.20 Some people argue that many of our calculations would be easier in the base 12 than in the base 10 (decimal) number system because 12 is divisible by so many more numbers than 10. What's the lowest digit in base 12? What would be the highest symbol for the digit in base 12? What are the positional values of the rightmost four positions of any number in the base 12 number system?

D.21 Complete the following chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

D.22 Convert binary 100101111010 to octal and to hexadecimal.

D.23 Convert hexadecimal 3A7D to binary.

D.24 Convert hexadecimal 765F to octal. [*Hint:* First convert 765F to binary, then convert that binary number to octal.]

D.25 Convert binary 1011110 to decimal.

D.26 Convert octal 426 to decimal.

D.27 Convert hexadecimal FFFF to decimal.

D.28 Convert decimal 299 to binary, to octal and to hexadecimal.

D.29 Show the binary representation of decimal 779. Then show the one's complement of 779 and the two's complement of 779.

D.30 Show the two's complement of integer value –1 on a machine with 32-bit integers.

E Preprocessor

Hold thou the good; define it well.
—Alfred, Lord Tennyson

I have found you an argument; but I am not obliged to find you an understanding.
—Samuel Johnson

A good symbol is the best argument, and is a missionary to persuade thousands.
—Ralph Waldo Emerson

Conditions are fundamentally sound.
—Herbert Hoover [December 1929]

Objectives

In this appendix you'll learn:

- To use `#include` for developing large programs.
- To use `#define` to create macros and macros with arguments.
- To understand conditional compilation.
- To display error messages during conditional compilation.
- To use assertions to test if the values of expressions are correct.

Outline

E.1 Introduction

This chapter introduces the **preprocessor**. Preprocessing occurs before a program is compiled. Some possible actions are inclusion of other files in the file being compiled, definition of **symbolic constants** and **macros**, **conditional compilation** of program code and **conditional execution of preprocessor directives.** All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). Preprocessor directives are processed fully before compilation begins.

Common Programming Error E.1

Placing a semicolon at the end of a preprocessor directive can lead to a variety of errors, depending on the type of preprocessor directive.

Software Engineering Observation E.1

Many preprocessor features (especially macros) are more appropriate for C programmers than for C++ programmers. C++ programmers should familiarize themselves with the preprocessor, because they might need to work with C legacy code.

E.2 `#include` Preprocessor Directive

The **`#include` preprocessor directive** has been used throughout this text. The `#include` directive causes a copy of a specified file to be included in place of the directive. The two forms of the `#include` directive are

```
#include <filename>
#include "filename"
```

The difference between these is the location the preprocessor searches for the file to be included. If the filename is enclosed in angle brackets (< and >)—used for standard library header files—the preprocessor searches for the specified file in an implementation-dependent manner, normally through predesignated directories. If the file name is enclosed in quotes, the preprocessor searches first in the same directory as the file being compiled, then in the same implementation-dependent manner as for a file name enclosed in angle brackets. This method is normally used to include programmer-defined header files.

The `#include` directive is used to include standard header files such as `<iostream>` and `<iomanip>`. The `#include` directive is also used with programs consisting of several source files that are to be compiled together. A header file containing declarations and definitions common to the separate program files is often created and included in the file. Examples of such declarations and definitions are classes, structures, unions, enumerations, function prototypes, constants and stream objects (e.g., `cin`).

E.3 `#define` Preprocessor Directive: Symbolic Constants

The **`#define` preprocessor directive** creates **symbolic constants**—constants represented as symbols—and macros—operations defined as symbols. The `#define` preprocessor directive format is

```
#define identifier replacement-text
```

When this line appears in a file, all subsequent occurrences (except those inside a string) of *identifier* in that file will be replaced by *replacement-text* before the program is compiled. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`. Symbolic constants enable you to create a name for a constant and use the name throughout the program. Later, if the constant needs to be modified throughout the program, it can be modified once in the `#define` preprocessor directive—and when the program is recompiled, all occurrences of the constant in the program will be modified. [*Note:* Everything to the right of the symbolic constant name replaces the symbolic constant. For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of `PI` with `= 3.14159`. Such replacement is the cause of many subtle logic and syntax errors.] Redefining a symbolic constant with a new value without first undefining it is also an error. Note that `const` variables in C++ are preferred over symbolic constants. Constant variables have a specific data type and are visible by name to a debugger. Once a symbolic constant is replaced with its replacement text, only the replacement text is visible to a debugger. A disadvantage of `const` variables is that they might require a memory location of their data type size—symbolic constants do not require any additional memory.

Common Programming Error E.2

Using symbolic constants in a file other than the file in which the symbolic constants are defined is a compilation error (unless they are `#included` from a header file).

Good Programming Practice E.1

Using meaningful names for symbolic constants makes programs more self-documenting.

E.4 `#define` Preprocessor Directive: Macros

[*Note:* This section is included for the benefit of C++ programmers who will need to work with C legacy code. In C++, macros can often be replaced by templates and inline functions.] A macro is an operation defined in a `#define` preprocessor directive. As with symbolic constants, the ***macro-identifier*** is replaced with the ***replacement-text*** before the program is compiled. Macros may be defined with or without ***arguments***. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the arguments are substituted in the *replacement-text*, then the macro is expanded—i.e., the *replacement-text* replaces the macro-identifier and argument list in the program. There is no data type checking for macro arguments. A macro is used simply for text substitution.

Consider the following macro definition with one argument for the area of a circle:

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

Wherever `CIRCLE_AREA( y )` appears in the file, the value of `y` is substituted for `x` in the replacement text, the symbolic constant `PI` is replaced by its value (defined previously) and the macro is expanded in the program. For example, the statement

```
area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Because the expression consists only of constants, at compile time the value of the expression can be evaluated, and the result is assigned to `area` at runtime. The parentheses around each `x` in the replacement text and around the entire expression force the proper order of evaluation when the macro argument is an expression. For example, the statement

```
area = CIRCLE_AREA( c + 2 );
```

is expanded to

```
area = ( 3.14159 * ( c + 2 ) * ( c + 2 ) );
```

which evaluates correctly, because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expansion is

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly as

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

because of the rules of operator precedence.

Common Programming Error E.3

Forgetting to enclose macro arguments in parentheses in the replacement text is an error.

Macro `CIRCLE_AREA` could be defined as a function. Function `circleArea`, as in

```
double circleArea( double x ) { return 3.14159 * x * x; }
```

performs the same calculation as `CIRCLE_AREA`, but the overhead of a function call is associated with function `circleArea`. The advantages of `CIRCLE_AREA` are that macros insert code directly in the program—avoiding function overhead—and the program remains readable because `CIRCLE_AREA` is defined separately and named meaningfully. A disadvantage is that its argument is evaluated twice. Also, every time a macro appears in a program, the macro is expanded. If the macro is large, this produces an increase in program size. Thus, there is a trade-off between execution speed and program size (if disk space is low).

Note that `inline` functions (see Chapter 6) are preferred to obtain the performance of macros and the software engineering benefits of functions.

Performance Tip E.1

Macros can sometimes be used to replace a function call with `inline` code prior to execution time. This eliminates the overhead of a function call. Inline functions are preferable to macros because they offer the type-checking services of functions.

The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
```

Wherever `RECTANGLE_AREA( a, b )` appears in the program, the values of `a` and `b` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

is expanded to

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

The value of the expression is evaluated and assigned to variable `rectArea`.

The replacement text for a macro or symbolic constant is normally any text on the line after the identifier in the `#define` directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be placed at the end of each line of the macro (except the last line), indicating that the replacement text continues on the next line.

Symbolic constants and macros can be discarded using the **`#undef` preprocessor directive**. Directive `#undef` "undefines" a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached. Once undefined, a name can be redefined with `#define`.

Note that expressions with side effects (e.g., variable values are modified) should not be passed to a macro, because macro arguments may be evaluated more than once.

Common Programming Error E.4

Macros often replace a name that wasn't intended to be a use of the macro but just happened to be spelled the same. This can lead to exceptionally mysterious compilation and syntax errors.

E.5 Conditional Compilation

Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code. Each of the conditional preprocessor directives evaluates a constant integer expression that will determine whether the code will be compiled. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives because these are all determined by the compiler and preprocessing happens before compilation.

The conditional preprocessor construct is much like the `if` selection structure. Consider the following preprocessor code:

```
#ifndef NULL
   #define NULL 0
#endif
```

which determines whether the symbolic constant `NULL` is already defined. The expression `#ifndef NULL` includes the code up to `#endif` if `NULL` is not defined, and skips the code if `NULL` is defined. Every **`#if`** construct ends with **`#endif`**. Directives **`#ifdef`** and **`#ifndef`** are shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`. A multiple-part conditional preprocessor construct may be tested using the `#elif` (the equivalent of `else if` in an `if` structure) and the `#else` (the equivalent of `else` in an `if` structure) directives.

During program development, programmers often find it helpful to "comment out" large portions of code to prevent it from being compiled. If the code contains C-style comments, `/*` and `*/` cannot be used to accomplish this task, because the first `*/` encountered would terminate the comment. Instead, you can use the following preprocessor construct:

```
#if 0
   code prevented from compiling
#endif
```

To enable the code to be compiled, simply replace the value `0` in the preceding construct with the value `1`.

Conditional compilation is commonly used as a debugging aid. Output statements are often used to print variable values and to confirm the flow of control. These output statements can be enclosed in conditional preprocessor directives so that the statements are compiled only until the debugging process is completed. For example,

```
#ifdef DEBUG
   cerr << "Variable x = " << x << endl;
#endif
```

causes the `cerr` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined before directive `#ifdef DEBUG`. This symbolic constant is normally set by a command-line compiler or by settings in the IDE (e.g., Visual Studio) and not by an explicit `#define` definition. When debugging is completed, the `#define` directive is removed from the source file, and the output statements inserted for debugging purposes are ignored during compilation. In larger programs, it might be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

Common Programming Error E.5

Inserting conditionally compiled output statements for debugging purposes in locations where C++ currently expects a single statement can lead to syntax errors and logic errors. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

E.6 #error and #pragma Preprocessor Directives

The **`#error` directive**

```
#error tokens
```

prints an implementation-dependent message including the *tokens* specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
#error 1 - Out of range error
```

contains six tokens. In one popular C++ compiler, for example, when a `#error` directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

The **`#pragma` directive**

```
#pragma tokens
```

causes an implementation-defined action. A pragma not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize pragmas that enable you to take advantage of that compiler's specific capabilities. For more information on `#error` and `#pragma`, see the documentation for your C++ implementation.

E.7 Operators # and

The **#** and **##** preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO( x ) cout << "Hello, " #x << endl;
```

When `HELLO(John)` appears in a program file, it is expanded to

```
cout << "Hello, " "John" << endl;
```

The string `"John"` replaces `#x` in the replacement text. Strings separated by white space are concatenated during preprocessing, so the above statement is equivalent to

```
cout << "Hello, John" << endl;
```

Note that the # operator must be used in a macro with arguments, because the operand of # refers to an argument of the macro.

The ## operator concatenates two tokens. Consider the following macro definition:

```
cout << "Hello, John" << endl;
#define TOKENCONCAT( x, y )  x ## y
```

When `TOKENCONCAT` appears in the program, its arguments are concatenated and used to replace the macro. For example, `TOKENCONCAT(O, K)` is replaced by `OK` in the program. The ## operator must have two operands.

E.8 Predefined Symbolic Constants

There are six **predefined symbolic constants** (Fig. E.1). The identifiers for each of these begin and (except for `__cplusplus`) end with *two* underscores. These identifiers and preprocessor operator `defined` (Section E.5) cannot be used in `#define` or `#undef` directives.

Symbolic constant	Description
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form `"Mmm dd yyyy"` such as `"Aug 19 2002"`).
`__STDC__`	Indicates whether the program conforms to the ANSI/ISO C standard. Contains value 1 if there is full conformance and is undefined otherwise.
`__TIME__`	The time the source file is compiled (a string literal of the form `"hh:mm:ss"`).
`__cplusplus`	Contains the value `199711L` (the date the ISO C++ standard was approved) if the file is being compiled by a C++ compiler, undefined otherwise. Allows a file to be set up to be compiled as either C or C++.

Fig. E.1 | The predefined symbolic constants.

E.9 Assertions

The **`assert` macro**—defined in the **`<cassert>`** header file—tests the value of an expression. If the value of the expression is `0` (false), then `assert` prints an error message and calls function **`abort`** (of the general utilities library—`<cstdlib>`) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable `x` should never be larger than `10` in a program. An assertion may be used to test the value of `x` and print an error message if the value of `x` is incorrect. The statement would be

```
assert( x <= 10 );
```

If `x` is greater than `10` when the preceding statement is encountered in a program, an error message containing the line number and file name is printed, and the program terminates. You may then concentrate on this area of the code to find the error. If the symbolic constant `NDEBUG` is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed (i.e., when debugging is complete), we insert the line

```
#define NDEBUG
```

in the program file rather than deleting each assertion manually. As with the `DEBUG` symbolic constant, `NDEBUG` is often set by compiler command-line options or through a setting in the IDE.

Most C++ compilers now include exception handling. C++ programmers prefer using exceptions rather than assertions. But assertions are still valuable for C++ programmers who work with C legacy code.

E.10 Wrap-Up

This appendix discussed the `#include` directive, which is used to develop larger programs. You also learned about the `#define` directive, which is used to create macros. We introduced conditional compilation, displaying error messages and using assertions.

Summary

Section E.2 `#include` Preprocessor Directive

- All preprocessor directives begin with `#` and are processed before the program is compiled.
- Only white-space characters may appear before a preprocessor directive on a line.
- The `#include` directive includes a copy of the specified file. If the filename is enclosed in quotes, the preprocessor begins searching in the same directory as the file being compiled for the file to be included. If the filename is enclosed in angle brackets (`<` and `>`), the search is performed in an implementation-defined manner.

Section E.3 `#define` Preprocessor Directive: Symbolic Constants

- The `#define` preprocessor directive is used to create symbolic constants and macros.
- A symbolic constant is a name for a constant.

Section E.4 `#define` Preprocessor Directive: Macros

- A macro is an operation defined in a `#define` preprocessor directive. Macros may be defined with or without arguments.
- The replacement text for a macro or symbolic constant is any text remaining on the line after the identifier (and, if any, the macro argument list) in the `#define` directive. If the replacement text for a macro or symbolic constant is too long to fit on one line, a backslash (\) is placed at the end of the line, indicating that the replacement text continues on the next line.
- Symbolic constants and macros can be discarded using the `#undef` preprocessor directive. Directive `#undef` "undefines" the symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached.

Section E.5 Conditional Compilation

- Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code.
- The conditional preprocessor directives evaluate constant integer expressions. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives.
- Every `#if` construct ends with `#endif`.
- Directives `#ifdef` and `#ifndef` are provided as shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`.
- A multiple-part conditional preprocessor construct is tested with directives `#elif` and `#else`.

Section E.6 `#error` and `#pragma` Preprocessor Directives

- The `#error` directive prints an implementation-dependent message that includes the tokens specified in the directive and terminates preprocessing and compiling.
- The `#pragma` directive causes an implementation-defined action. If the pragma is not recognized by the implementation, the pragma is ignored.

Section E.7 Operators # and ##

- The `#` operator causes the following replacement text token to be converted to a string surrounded by quotes. The `#` operator must be used in a macro with arguments, because the operand of `#` must be an argument of the macro.
- The `##` operator concatenates two tokens. The `##` operator must have two operands.

Section E.8 Predefined Symbolic Constants

- There are six predefined symbolic constants. Constant `__LINE__` is the line number of the current source-code line (an integer). Constant `__FILE__` is the presumed name of the file (a string). Constant `__DATE__` is the date the source file is compiled (a string). Constant `__TIME__` is the time the source file is compiled (a string). Note that each of the predefined symbolic constants begins (and, with the exception of `__cplusplus`, ends) with two underscores.

Section E.9 Assertions

- The `assert` macro—defined in the `<cassert>` header file—tests the value of an expression. If the value of the expression is `0` (false), then `assert` prints an error message and calls function `abort` to terminate program execution.

Self-Review Exercises

E.1 Fill in the blanks in each of the following:

a) Every preprocessor directive must begin with ________.
b) The conditional compilation construct may be extended to test for multiple cases by using the ________ and the ________ directives.
c) The ________ directive creates macros and symbolic constants.
d) Only ________ characters may appear before a preprocessor directive on a line.
e) The ________ directive discards symbolic constant and macro names.
f) The ________ and ________ directives are provided as shorthand notation for `#if defined(`*name*`)` and `#if !defined(`*name*`)`.
g) ________ enables you to control the execution of preprocessor directives and the compilation of program code.
h) The ________ macro prints a message and terminates program execution if the value of the expression the macro evaluates is `0`.
i) The ________ directive inserts a file in another file.
j) The ________ operator concatenates its two arguments.
k) The ________ operator converts its operand to a string.
l) The character ________ indicates that the replacement text for a symbolic constant or macro continues on the next line.

E.2 Write a program to print the values of the predefined symbolic constants `__LINE__`, `__FILE__`, `__DATE__` and `__TIME__` listed in Fig. E.1.

E.3 Write a preprocessor directive to accomplish each of the following:

a) Define symbolic constant `YES` to have the value `1`.
b) Define symbolic constant `NO` to have the value `0`.
c) Include the header file `common.h`. The header is found in the same directory as the file being compiled.
d) If symbolic constant `TRUE` is defined, undefine it, and redefine it as `1`. Do not use `#ifdef`.
e) If symbolic constant `TRUE` is defined, undefine it, and redefine it as `1`. Use the `#ifdef` preprocessor directive.
f) If symbolic constant `ACTIVE` is not equal to `0`, define symbolic constant `INACTIVE` as `0`. Otherwise, define `INACTIVE` as `1`.
g) Define macro `CUBE_VOLUME` that computes the volume of a cube (takes one argument).

Answers to Self-Review Exercises

E.1 a) `#`. b) `#elif`, `#else`. c) `#define`. d) white-space. e) `#undef`. f) `#ifdef`, `#ifndef`. g) Conditional compilation. h) `assert`. i) `#include`. j) `##`. k) `#`. l) `\`.

E.2 (See below.)

```
// exF_02.cpp
// Self-Review Exercise E.2 solution.
#include <iostream>
using namespace std;

int main()
{
   cout << "__LINE__ = " << __LINE__ << endl
        << "__FILE__ = " << __FILE__ << endl
        << "__DATE__ = " << __DATE__ << endl
        << "__TIME__ = " << __TIME__ << endl
        << "__cplusplus = " << __cplusplus << endl;
}  // end main
```

```
__LINE__ = 9
__FILE__ = c:\cpp4e\ch19\ex19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

E.3

a)
```
#define YES 1
```
b)
```
#define NO 0
```
c)
```
#include "common.h"
```
d)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```
e)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```
f)
```
#if ACTIVE
   #define INACTIVE 0
#else
   #define INACTIVE 1
#endif
```
g)
```
#define CUBE_VOLUME( x )  ( ( x ) * ( x ) * ( x ) )
```

Exercises

E.4 Write a program that defines a macro with one argument to compute the volume of a sphere. The program should compute the volume for spheres of radii from 1 to 10 and print the results in tabular format. The formula for the volume of a sphere is

$$(4.0 / 3) * \pi * r^3$$

where π is `3.14159`.

E.5 Write a program that produces the following output:

```
The sum of x and y is 13
```

The program should define macro `SUM` with two arguments, `x` and `y`, and use `SUM` to produce the output.

E.6 Write a program that uses macro `MINIMUM2` to determine the smaller of two numeric values. Input the values from the keyboard.

E.7 Write a program that uses macro `MINIMUM3` to determine the smallest of three numeric values. Macro `MINIMUM3` should use macro `MINIMUM2` defined in Exercise E.6 to determine the smallest number. Input the values from the keyboard.

E.8 Write a program that uses macro `PRINT` to print a string value.

E.9 Write a program that uses macro `PRINTARRAY` to print an array of integers. The macro should receive the array and the number of elements in the array as arguments.

E.10 Write a program that uses macro `SUMARRAY` to sum the values in a numeric array. The macro should receive the array and the number of elements in the array as arguments.

E.11 Rewrite the solutions to Exercises E.4–E.10 as `inline` functions.

E.12 For each of the following macros, identify the possible problems (if any) when the preprocessor expands the macros:

a) `#define SQR( x ) x * x`
b) `#define SQR( x ) ( x * x )`
c) `#define SQR( x ) ( x ) * ( x )`
d) `#define SQR( x ) ( ( x ) * ( x ) )`

This page intentionally left blank

Appendices on the Web

The following appendices are available as PDF documents from this book's Companion Website, which is accessible from `www.pearsonhighered.com/deitel/`:

- Appendix F, C Legacy Code Topics
- Appendix G, UML 2: Additional Diagram Types
- Appendix H, Using the Visual Studio Debugger
- Appendix I, Using the GNU C++ Debugger

These files can be viewed in Adobe® Reader® (`get.adobe.com/reader`). The index entries for these appendices include the appendix letter and an underscore, as in F_1.

New copies of this book come with a Companion Website access code that is located on the card inside the book's front cover. If the access code is already visible or there is no card, you purchased a used book or an edition that does not come with an access code. In this case, you can purchase access directly from the Companion Website.

This page intentionally left blank

Index

[*Note:* Page references for defining occurrences of terms appear in **bold blue**. The page references for online chapters and appendices include the chapter number or appendix letter and an underscore, as in 25_1 or F_1. You can find the online chapters and appendices on the book's Companion Website, which is accessible from `www.pearsonhighered.com/deitel/`.]

Symbols

D

E

F

G

J

K

L

M

P

Q

T

U

X

Y

Z

This page intentionally left blank

ATM Case Study, Part 1: Object-Oriented Design with the UML

25

Action speaks louder than words but not nearly as often.
—Mark Twain

Always design a thing by considering it in its next larger context.
—Eliel Saarinen

Oh, life is a glorious cycle of song.
—Dorothy Parker

The Wright brothers' design … allowed them to survive long enough to learn how to fly.
—Michael Potts

Objectives

In this chapter you'll learn:

- A simple object-oriented design methodology.
- What a requirements document is.
- To identify classes and class attributes from a requirements document.
- To identify objects' states, activities and operations from a requirements document.
- To determine the collaborations among objects in a system.
- To work with the UML's use case, class, state, activity, communication and sequence diagrams to graphically model an object-oriented system.

Outline

25.1 Introduction

Now we begin the optional portion of our object-oriented design and implementation case study. In this chapter and Chapter 26, you'll design and implement an object-oriented automated teller machine (ATM) software system. The case study provides you with a concise, carefully paced, complete design and implementation experience. You'll perform the steps of an object-oriented design (OOD) process using the UML while relating them to the object-oriented concepts discussed in Chapters 2–13. In this chapter, you'll work with six popular types of UML diagrams to graphically represent the design. In Chapter 26, you'll tune the design with inheritance and polymorphism, then fully implement the ATM in an 850-line C++ application (Section 26.4).

This is *not* an exercise; rather, it's an end-to-end learning experience that concludes with a detailed walkthrough of the *complete* C++ code that implements our design. It will acquaint you with the kinds of substantial problems encountered in industry.

These chapters can be studied as a continuous unit after you've completed the introduction to object-oriented programming in Chapters 2–13. Or, you can pace the sections after Chapters 3–7, 9 and 13. Each section of the case study begins with a note telling you the chapter after which it can be covered.

25.2 Introduction to Object-Oriented Analysis and Design

What if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1000 software developers building the next U.S. air traffic control system. For projects so large and complex, you cannot simply sit down and start writing programs.

To create the best solutions, you should follow a process for **analyzing** your project's **requirements** (i.e., determining *what* the system should do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (or have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Analysis and design can save many hours by helping you to avoid an ill-planned system-development approach that has to be abandoned part of the way through its implementation, possibly wasting considerable time, money and effort. Small problems do not require an exhaustive OOAD process. It may be sufficient to write pseudocode before you begin writing C++ code.

As problems and the groups of people solving them increase in size, the methods of OOAD become more appropriate than pseudocode. Ideally, members of a group should agree on a strictly defined process for solving their problem and a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s under the initial direction of three software methodologists—Grady Booch, James Rumbaugh and Ivar Jacobson.

25.3 Examining the ATM Requirements Document

[*Note:* **This section can be studied after Chapter 3.**]

We begin our design process by presenting a **requirements document** that specifies the ATM system's overall purpose and *what* it must do. Throughout the case study, we refer to the requirements document to determine what functionality the system must include.

Requirements Document

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 25.1). Each user can have only one account at the bank. ATM users should be able to *view their account balance*, *withdraw cash* (i.e., take money out of an account) and *deposit funds* (i.e., place money into an account).

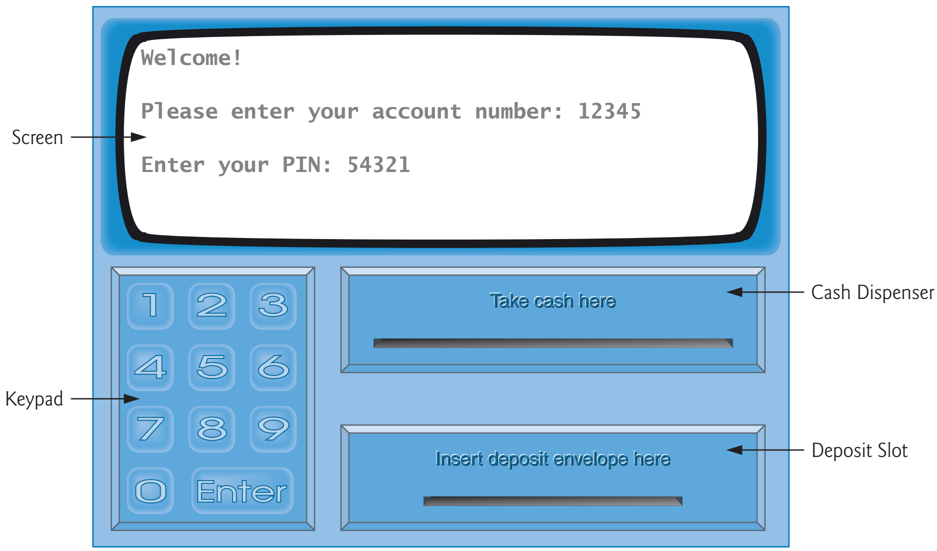


Fig. 25.1 | Automated teller machine user interface.

The user interface of the automated teller machine contains the following hardware components:

- a screen that displays messages to the user

- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.

The cash dispenser begins each day loaded with 500 $20 bills. [*Note:* Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user's account number from an ATM card, whereas this ATM asks the user to type an account number using the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM's hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version of the software to run on a personal computer. This version should use the computer's monitor to simulate the ATM's screen, and the computer's keyboard to simulate the ATM's keypad.

An ATM session consists of authenticating a user (i.e., proving the user's identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank's account information database. [*Note:* A database is an organized collection of data stored on a computer.] For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [*Note:* For simplicity, we assume that *the bank plans to build only one ATM, so we do not need to worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM.* Also, any business system like an ATM faces reasonably complicated security issues that go well beyond the scope of a first- or second-semester computer science course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM, the user should experience the following sequence of events (shown in Fig. 25.1):

1. The screen displays a welcome message and prompts the user to enter an account number.
2. The user enters a five-digit account number, using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN, using the keypad.
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 25.2). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to *Step 1* to restart the authentication process.

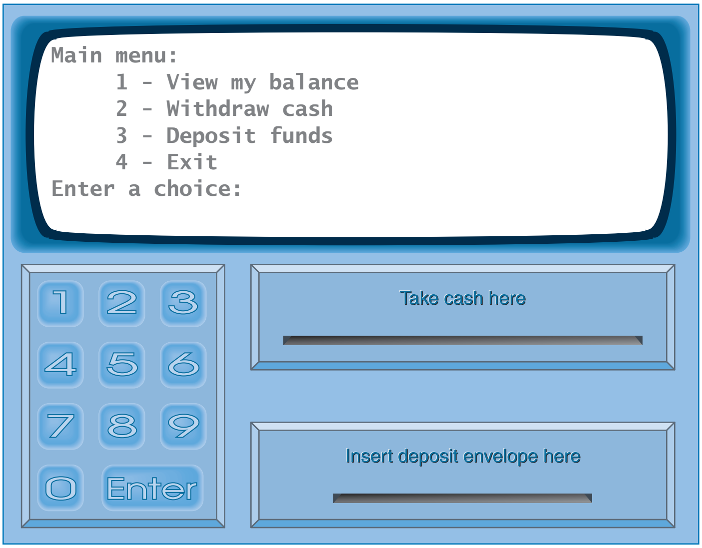


Fig. 25.2 | ATM main menu.

After the ATM authenticates the user, the main menu (Fig. 25.2) displays a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). The main menu also displays an option that allows the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4). If the user enters an invalid option, the screen displays an error message, then redisplays to the main menu.

If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database.

The following actions occur when the user enters 2 to make a withdrawal:

1. The screen displays a menu (shown in Fig. 25.3) containing standard withdrawal amounts: $20 (option 1), $40 (option 2), $60 (option 3), $100 (option 4) and $200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).
2. The user enters a menu selection (1–6) using the keypad.
3. If the withdrawal amount chosen is greater than the user's account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to *Step 1*. If the withdrawal amount chosen is less than or equal to the user's account balance (i.e., an acceptable withdrawal amount), the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (option 6), the ATM displays the main menu (Fig. 25.2) and waits for user input.
4. If the cash dispenser contains enough cash to satisfy the request, the ATM proceeds to *Step 5*. Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then returns to *Step 1*.

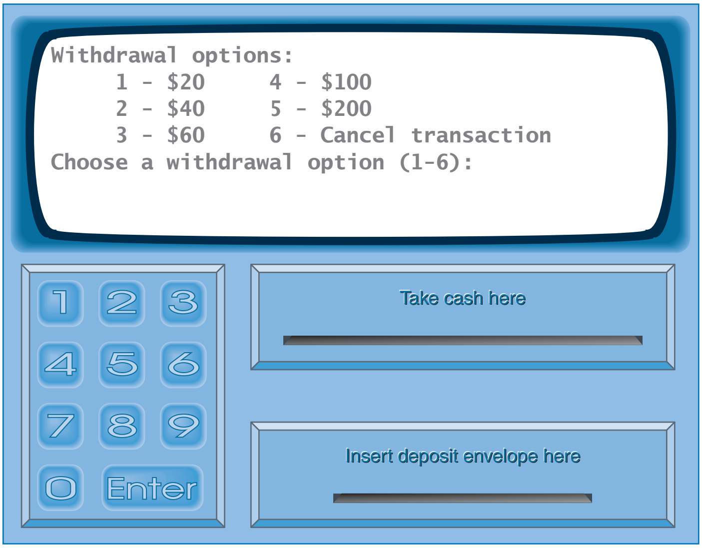


Fig. 25.3 | ATM withdrawal menu.

5. The ATM debits (i.e., subtracts) the withdrawal amount from the user's account balance in the bank's database.
6. The cash dispenser dispenses the desired amount of money to the user.
7. The screen displays a message reminding the user to take the money.

The following actions occur when the user enters 3 (while the main menu is displayed) to make a deposit:

1. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.
2. The user enters a deposit amount or 0, using the keypad. [*Note:* The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., $1.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 125). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 125 ÷ 100 = 1.25).]
3. If the user specifies a deposit amount, the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu (Fig. 25.2) and waits for user input.
4. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.
5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits (i.e., adds) the deposit amount to the user's account balance in the bank's database. *This money is not immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the en-*

> *velope must clear (i.e., money must be transferred from the check writer's account to the check recipient's account). When either of these events occurs, the bank appropriately updates the user's balance stored in its database. This occurs independently of the ATM system.* If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, the system should redisplay the main menu (Fig. 25.2) so that the user can perform additional transactions. If the user chooses to exit the system (option 4), the screen should display a thank you message, then display the welcome message for the next user.

Analyzing the ATM System

The preceding statement is a simplified example of a requirements document. Typically, such a document is the result of a detailed **requirements gathering** process that might include interviews with potential users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements document for banking software (e.g., the ATM system described here) might interview financial experts to gain a better understanding of *what* the software must do. The analyst would use the information gained to compile a list of **system requirements** to guide systems designers.

The process of requirements gathering is a key task of the first stage of the software life cycle. The **software life cycle** specifies the stages through which software evolves from the time it's first conceived to the time it's retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages. **Waterfall models** perform each stage once in succession, whereas **iterative models** may repeat one or more stages several times throughout a product's life cycle.

The analysis stage of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly *solve the problem right*, but of equal importance, one must *solve the right problem*. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements document describes our ATM system in sufficient detail that you do not need to go through an extensive analysis stage—it has been done for you.

To capture what a proposed system should do, developers often employ a technique known as **use case modeling**. This process identifies the **use cases** of the system, each of which represents a different capability that the system provides to its clients. For example, ATMs typically have several use cases, such as "View Account Balance," "Withdraw Cash," "Deposit Funds," "Transfer Funds Between Accounts" and "Buy Postage Stamps." The simplified ATM system we build in this case study allows only the first three of these use cases (Fig. 25.4).

Each use case describes a typical scenario in which the user uses the system. You've already read descriptions of the ATM system's use cases in the requirements document; the lists of steps required to perform each type of transaction (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—"View Account Balance," "Withdraw Cash" and "Deposit Funds."

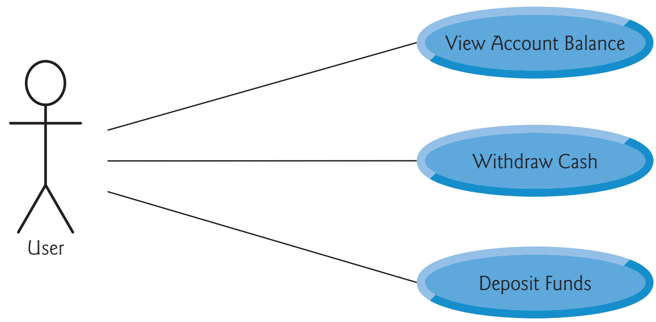


Fig. 25.4 | Use case diagram for the ATM system from the User's perspective.

Use Case Diagrams

We now introduce the first of several UML diagrams in our ATM case study. We create a **use case diagram** to model the interactions between a system's clients (in this case study, bank customers) and the system. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout the case study). Use case diagrams are often accompanied by informal text that describes the use cases in more detail—like the text that appears in the requirements document. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are simple but indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 25.4 shows the use case diagram for our ATM system. The stick figure represents an **actor**, which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. Note that a use case diagram can include multiple actors. For example, the use case diagram for a real bank's ATM system might also include an actor named Administrator who refills the cash dispenser each day.

We identify the actor in our system by examining the requirements document, which states, "ATM users should be able to view their account balance, withdraw cash and deposit funds." So, the actor in each of the three use cases is the User who interacts with the ATM. An external entity—a real person—plays the part of the User to perform financial transactions. Figure 25.4 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

Software engineers (more precisely, systems analysts) must analyze the requirements document or a set of use cases and design the system before programmers implement it. During the analysis stage, systems analysts focus on understanding the requirements document to produce a high-level specification that describes *what* the system is supposed to do. The output of the design stage—a **design specification**—should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text. Recall that the UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation (now a division of IBM). RUP is a rich process intended for designing "industrial strength" applications. For this case study, we present our own simplified design process.

Designing the ATM System

We now begin the ATM system's design. A **system** is a set of components that interact to solve a problem. To perform the ATM system's designated tasks, our ATM system has a user interface (Fig. 25.1), contains software that executes financial transactions and interacts with a database of bank account information. **System structure** describes the system's objects and their interrelationships. **System behavior** describes how the system changes as its objects interact with one another. Every system has both structure and behavior—designers must specify both. There are several distinct types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 specifies 13 diagram types for documenting the models of systems. Each models a distinct characteristic of a system's structure or behavior—six diagrams relate to system structure; the remaining seven relate to system behavior. We list here only the six types of diagrams used in our case study—one of these (class diagrams) models system structure—the remaining five model system behavior. We overview the remaining seven UML diagram types in Appendix G, UML 2: Additional Diagram Types.

1. **Use case diagrams**, such as the one in Fig. 25.4, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as "View Account Balance," "Withdraw Cash" and "Deposit Funds").
2. **Class diagrams**, which you'll study in Section 25.4, model the classes, or "building blocks," used in a system. Each noun or "thing" described in the requirements document is a candidate to be a class in the system (e.g., "account," "keypad"). Class diagrams help us specify the structural relationships between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically composed of a screen, a keypad, a cash dispenser and a deposit slot.
3. **State machine diagrams**, which you'll study in Section 25.6, model the ways in which an object changes state. An object's **state** is indicated by the values of all the object's attributes at a given time. When an object changes state, that object may behave differently in the system. For example, after validating a user's PIN, the ATM transitions from the "user not authenticated" state to the "user authenticated" state, at which point the ATM allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).
4. **Activity diagrams**, which you'll also study in Section 25.6, model an object's **activity**—the object's workflow (sequence of events) during program execution. An activity diagram models the actions the object performs and specifies the order in which it performs these actions. For example, an activity diagram shows that the ATM must obtain the balance of the user's account (from the bank's account information database) before the screen can display the balance to the user.

5. **Communication diagrams** (called **collaboration diagrams** in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You'll learn in Section 25.8 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank's account information database to retrieve an account balance.
6. **Sequence diagrams** also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You'll learn in Section 25.8 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 25.4, we continue designing our ATM system by identifying the classes from the requirements document. We accomplish this by extracting key nouns and noun phrases from the requirements document. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

Web Resources

We've created an extensive UML Resource Center (`www.deitel.com/UML/`) that contains many links to additional information, including introductions, tutorials, blogs, books, certification, conferences, developer tools, documentation, e-books, FAQs, forums, groups, UML in C++, podcasts, security, tools, downloads, training courses, videos and more.

Self-Review Exercises for Section 25.3

25.1 Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 25.4 to reflect this change.

25.2 ________ model the interactions among objects in a system with an emphasis on *when* these interactions occur.

a) Class diagrams
b) Sequence diagrams
c) Communication diagrams
d) Activity diagrams

25.3 Which of the following choices lists stages of a typical software life cycle in sequential order?

a) design, analysis, implementation, testing
b) design, analysis, testing, implementation
c) analysis, design, testing, implementation
d) analysis, design, implementation, testing

25.4 Identifying the Classes in the ATM Requirements Document

[*Note:* **This section can be studied after Chapter 3.**]

Now we begin designing the ATM system that we introduced in Section 25.3. In this section, we identify the classes that are needed to build the ATM system by analyzing the nouns and noun phrases that appear in the requirements document. We introduce UML class diagrams to model the relationships between these classes. This is an important first step in defining the structure of our system.

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We'll eventually describe these classes using UML class diagrams and implement these classes in C++. First, we review the requirements document of Section 25.3 and find key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do *not* correspond to parts of the system and thus should *not* be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 25.5 lists the nouns and noun phrases in the requirements document. We list them from left to right in the order in which they appear in the requirements document. We list only the singular form of each noun or noun phrase.

Nouns and noun phrases in the requirements document			
bank	money / fund	account number	ATM
screen	PIN	user	keypad
bank database	customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal	account
deposit slot	deposit	balance	deposit envelope

Fig. 25.5 | Nouns and noun phrases in the requirements document.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We don't need to model "bank" as a class, because it'is not a part of the ATM system—the bank simply wants us to build the ATM. "Customer" and "user" also represent outside entities—they are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 25.4.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are *not* part of what's being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 25.5.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements document doesn't say anything about what the system should do with deposit envelopes after it receives them. We can assume that acknowledging the receipt of an envelope—an operation performed by the class that models the deposit slot—is sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 25.7.)

In our simplified ATM system, representing various amounts of "money," including an account's "balance," as attributes of other classes seems most appropriate. Likewise, the nouns "account number" and "PIN" represent significant information in the ATM system. They are important attributes of a bank account. They do *not*, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements document frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. *In Section 26.3, we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concepts of abstract classes and inheritance.*

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 25.5. Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry
- withdrawal
- deposit

The elements of this list are likely to be classes we'll need to implement our system.

We can now model the classes in our system based on the list we've created. We capitalize class names in the design process—a UML convention—as we'll do when we write the actual C++ code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize the first letter of each word (e.g., `MultipleWordName`). Using this convention, we create classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` and `Deposit`. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Modeling Classes

The UML enables us to model, via **class diagrams**, the ATM system's classes and their interrelationships. Figure 25.6 represents class `ATM`. Each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class, centered horizontally and in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 25.5 and Section 25.6.) The bottom compartment contains the class's operations (discussed in Section 25.7). In Fig. 25.6 the middle and bottom compartments are empty, because we've not yet determined this class's attributes and operations.

Class diagrams also show the relationships among the classes of the system. Figure 25.7 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity; we present a more

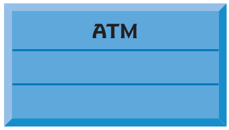


Fig. 25.6 | Representing a class in the UML using a class diagram.

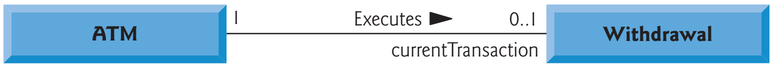


Fig. 25.7 | Class diagram showing an association among classes.

complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner, when appropriate, to create more readable diagrams. Such a diagram is said to be an **elided diagram**—one in which some information, such as the contents of the second and third compartments, is not modeled. We'll place information in these compartments in Section 25.5 and Section 25.7.

In Fig. 25.7, the solid line that connects the two classes represents an **association**—a relationship between classes. The numbers near each end of the line are **multiplicity** values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one `ATM` object participates in an association with either zero or one `Withdrawal` objects—zero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 25.8 lists and explains the multiplicity types.

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Fig. 25.8 | Multiplicity types.

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 25.7 indicates the name of that association. This part of the diagram reads "one object of class `ATM` executes zero or one objects of class

Withdrawal." Association names are directional, as indicated by the filled arrowhead—so it would be improper, for example, to read the preceding association from right to left as "zero or one objects of class Withdrawal execute one object of class ATM."

The word currentTransaction at the Withdrawal end of the association line in Fig. 25.7 is a **role name**, which identifies the role the Withdrawal object plays in its relationship with the ATM. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 25.7, the role name currentTransaction indicates that the Withdrawal object participating in the Executes association with an object of class ATM represents the transaction currently being processed by the ATM. In other contexts, a Withdrawal object may take on other roles (e.g., the previous transaction). Notice that we do *not* specify a role name for the ATM end of the Executes association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being composed of objects of other classes. Consider a real-world automated teller machine. What "pieces" does a manufacturer put together to build a working ATM? Our requirements document tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

In Fig. 25.9, the **solid diamonds** attached to the association lines of class ATM indicate that class ATM has a **composition** relationship with classes Screen, Keypad, CashDispenser and DepositSlot. Composition implies a whole/part relationship. The class that has the *composition symbol* (the solid diamond) on its end of the association line is the whole (in this case, ATM), and the classes on the other end of the association lines are the parts—in this case, classes Screen, Keypad, CashDispenser and DepositSlot. The compositions in Fig. 25.9 indicate that an object of class ATM is formed from one object of class Screen, one object of class CashDispenser, one object of class Keypad and one object of class DepositSlot. The ATM *has a* screen, a keypad, a cash dispenser and a deposit slot. The ***has-a* relationship** defines composition. (We'll see in Section 26.3 that the *is-a* relationship defines inheritance.)

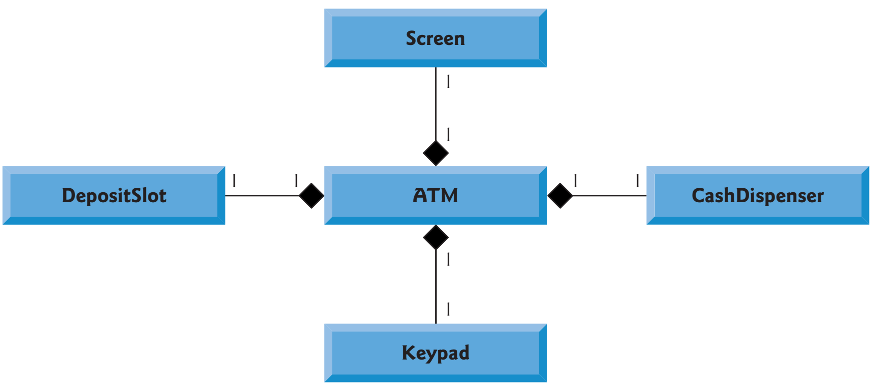


Fig. 25.9 | Class diagram showing composition relationships.

According to the UML specification, composition relationships have the following properties:

1. Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.
2. The parts in a composition relationship exist only as long as the whole, and the whole is responsible for creating and destroying its parts. For example, the act of constructing an ATM includes manufacturing its parts. Furthermore, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.
3. A part may belong to only one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these three properties. If a *has-a* relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate **aggregation**—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer *has a* monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating the second and third properties of composition.

Figure 25.10 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements document. [*Note:* Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we've chosen to omit them from this diagram to keep it simple. In Section 26.3, we expand our class diagram to include all the classes in the ATM system.]

Figure 25.10 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account` and several associations that were not present in either Fig. 25.7 or Fig. 25.9. The class diagram shows that class `ATM` has a **one-to-one relationship** with class `BankDatabase`—one `ATM` object authenticates users against one `BankDatabase` object. In Fig. 25.10, we also model the fact that the bank's database contains information about many accounts—one object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`. Recall from Fig. 25.8 that the multiplicity value 0..* at the `Account` end of the association between class `BankDatabase` and class `Account` indicates that zero or more objects of class `Account` take part in the association. Class `BankDatabase` has a **one-to-many relationship** with class `Account`—the `BankDatabase` contains many `Account`s. Similarly, class `Account` has a **many-to-one relationship** with class `BankDatabase`—there can be many `Account`s contained in the `BankDatabase`. [*Note:* Recall from Fig. 25.8 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

Figure 25.10 also indicates that if the user is performing a withdrawal, "one object of class `Withdrawal` accesses/modifies an account balance through one object of class `BankDatabase`." We could have created an association directly between class `Withdrawal` and class `Account`. The requirements document, however, states that the "ATM must interact with the bank's account information database" to perform transactions. A bank account

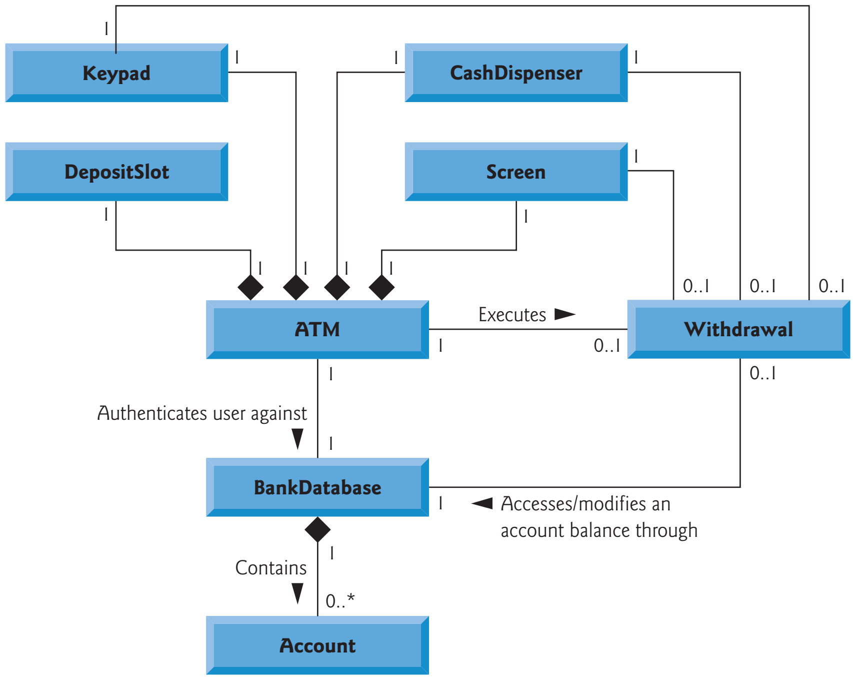


Fig. 25.10 | Class diagram for the ATM system model.

contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the `BankDatabase` can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 25.10 also models associations between class `Withdrawal` and classes `Screen`, `CashDispenser` and `Keypad`. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 25.10, take part in several associations with the other classes of the ATM system. Like class `Withdrawal`, each of these classes associates with classes `ATM` and `BankDatabase`. An object of class `BalanceInquiry` also associates with an object of class `Screen` to display the balance of an account to the user. Class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class `Deposit` accesses the deposit slot.

We've now identified the classes in our ATM system (although we may discover others as we proceed with the design and implementation). In Section 25.5, we determine the attributes for each of these classes, and in Section 25.6, we use these attributes to examine how the system changes over time. In Section 25.7, we determine the operations of the classes in our system.

Self-Review Exercises for Section 25.4

25.4 Suppose we have a class `Car` that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 25.9) that models some of the composition relationships of class `Car`.

25.5 Suppose we have a class `File` that represents an electronic document in a stand-alone, non-networked computer represented by class `Computer`. What sort of association exists between class `Computer` and class `File`?

a) Class `Computer` has a *one-to-one* relationship with class `File`.
b) Class `Computer` has a *many-to-one* relationship with class `File`.
c) Class `Computer` has a *one-to-many* relationship with class `File`.
d) Class `Computer` has a *many-to-many* relationship with class `File`.

25.6 State whether the following statement is *true* or *false*, and if *false*, explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

25.7 Modify the class diagram of Fig. 25.10 to include class `Deposit` instead of class `Withdrawal`.

25.5 Identifying Class Attributes

[*Note:* **This section can be studied after Chapter 4.**]

In Section 25.4, we began the first stage of an object-oriented design (OOD) for our ATM system—analyzing the requirements document and identifying the classes needed to implement the system. We listed the *nouns* and *noun phrases* in the requirements document and identified a separate class for each one that plays a significant role in the ATM system. We then modeled the classes and their relationships in a UML class diagram (Fig. 25.10).

Classes have attributes (data) and operations (behaviors). Class attributes are implemented in C++ programs as data members, and class operations are implemented as member functions. In this section, we determine many of the attributes needed in the ATM system. In Section 25.6, we examine how these attributes represent an object's state. In Section 25.7, we determine class operations.

Identifying Attributes

Consider the attributes of some real-world objects: A person's attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio's attributes include its station setting, its volume setting and its AM or FM setting. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it's in. A personal computer's attributes include its manufacturer (e.g., Dell, HP, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements document. For each one we find that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 25.4. We also create attributes to represent any additional data that a class may need, as such needs become apparent throughout the design process.

Figure 25.11 lists the words or phrases from the requirements document that describe each class. We formed this list by reading the requirements document and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements document describes the steps taken to obtain a "withdrawal amount," so we list "amount" next to class `Withdrawal`.

Figure 25.11 leads us to create one attribute of class `ATM`. Class `ATM` maintains information about the state of the ATM. The phrase "user is authenticated" describes a state of the ATM (we introduce states in Section 25.6), so we include `userAuthenticated` as a **`Boolean` attribute** (i.e., an attribute that has a value of either `true` or `false`). The UML `Boolean` type is equivalent to the `bool` type in C++. This attribute indicates whether the ATM has successfully authenticated the current user—`userAuthenticated` must be `true` for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Class	Descriptive words and phrases
`ATM`	user is authenticated
`BalanceInquiry`	account number
`Withdrawal`	account number amount
`Deposit`	account number amount
`BankDatabase`	[no descriptive words or phrases]
`Account`	account number PIN balance
`Screen`	[no descriptive words or phrases]
`Keypad`	[no descriptive words or phrases]
`CashDispenser`	begins each day loaded with 500 $20 bills
`DepositSlot`	[no descriptive words or phrases]

Fig. 25.11 | Descriptive words and phrases from the ATM requirements.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share one attribute. Each transaction involves an "account number" that corresponds to the account of the user making the transaction. We assign an integer attribute `accountNumber` to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements document also suggest some differences in the attributes required by each transaction class. The requirements document indicates that to withdraw cash or deposit funds, users must enter a specific "amount" of money to be withdrawn or deposited, respectively. Thus, we assign to classes `Withdrawal` and `Deposit` an attribute `amount` to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for them to take place. Class `BalanceInquiry`, however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class `Account` has several attributes. The requirements document states that each bank account has an "account number" and "PIN," which the system uses for identifying accounts and authenticating users. We assign to class `Account` two integer attributes: `accountNumber` and `pin`. The requirements document also specifies that an account maintains a "balance" of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes of UML type `Double`: `availableBalance` and `totalBalance`. Attribute `availableBalance` tracks the amount of money that a user can withdraw from the account. Attribute `totalBalance` refers to the total amount of money that the user has "on deposit" (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits $50.00 into an empty account. The `totalBalance` attribute would increase to $50.00 to record the deposit, but the `availableBalance` would remain at $0. [*Note:* We assume that the bank updates the `availableBalance` attribute of an `Account` soon after the ATM transaction occurs, in response to confirming that $50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

Class `CashDispenser` has one attribute. The requirements document states that the cash dispenser "begins each day loaded with 500 $20 bills." The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class `CashDispenser` an integer attribute `count`, which is initially set to `500`.

For real problems in industry, there is no guarantee that requirements specifications will be rich enough and precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional (or fewer) classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we too will continue to add, modify and delete information about the classes in our system.

Modeling Attributes

The class diagram in Fig. 25.12 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 25.11 helped us identify these attributes. For simplicity, Fig. 25.12 does not show the associations among classes—we showed these in Fig. 25.10. This is a common practice of systems designers when designs are being developed. Recall from Section 25.4 that in the UML, a class's attributes are placed in the middle compartment of the class's rectangle. We list each attribute's name and type separated by a colon (`:`), followed in some cases by an equal sign (=) and an initial value.

Consider the `userAuthenticated` attribute of class `ATM`:

```
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The **attribute name** is `userAuthenticated`. The **attribute type** is `Boolean`. In C++, an attribute can be represented by a fundamental type, such as `bool`, `int` or `double`, or a class type. We've chosen to model only primitive-type attributes in Fig. 25.12—we discuss the reasoning behind this decision shortly. [*Note:* Figure 25.12 lists UML data types for the attributes. When we implement the system, we'll associate the UML types `Boolean`, `Integer` and `Double` with the C++ fundamental types `bool`, `int` and `double`, respectively.]

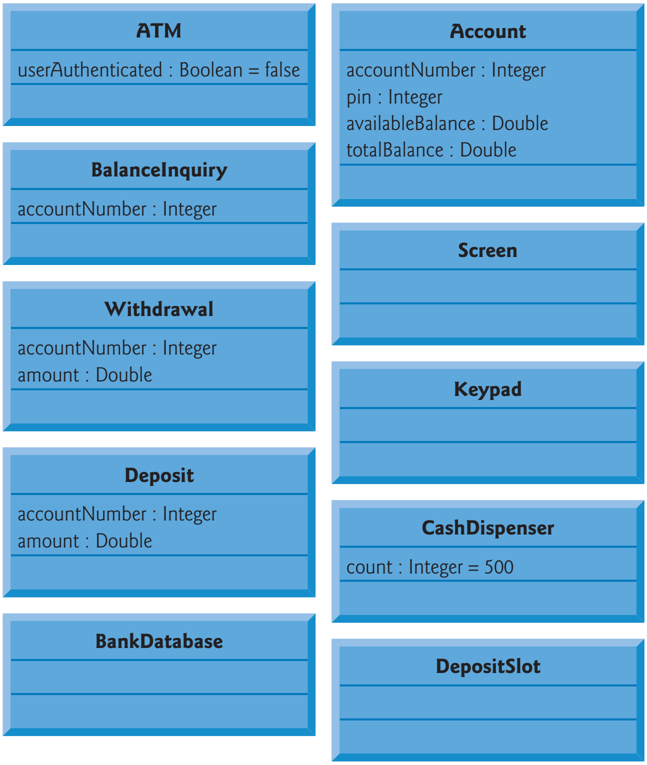


Fig. 25.12 | Classes with attributes.

We can also indicate an *initial value* for an attribute. The `userAuthenticated` attribute in class `ATM` has an initial value of `false`. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the `accountNumber` attribute of class `BalanceInquiry` is an `Integer`. Here we show no initial value, because the value of this attribute is a number that we do not yet know—it will be determined at execution time based on the account number entered by the current ATM user.

Figure 25.12 does not include any attributes for classes `Screen`, `Keypad` and `DepositSlot`. These are important components of our system, for which our design process simply has not yet revealed any attributes. We may still discover some, however, in the remaining design phases or when we implement these classes in C++. This is perfectly normal for the iterative process of software engineering.

Software Engineering Observation 25.1

At the early stages in the design process, classes often lack attributes (and operations). Such classes should not *be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.*

Figure 25.12 also does not include attributes for class `BankDatabase`. Recall that attributes can be represented by either fundamental types or class types. We've chosen to include only fundamental-type attributes in the class diagram in Fig. 25.12 (and in similar class diagrams throughout the case study). A class-type attribute is modeled more clearly as an association (in particular, a composition) between the class with the attribute and the class of the object of which the attribute is an instance. For example, the class diagram in Fig. 25.10 indicates that class `BankDatabase` participates in a composition relationship with zero or more `Account` objects. From this composition, we can determine that when we implement the ATM system in C++, we'll be required to create an attribute of class `BankDatabase` to hold zero or more `Account` objects. Similarly, we'll assign attributes to class `ATM` that correspond to its composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. These composition-based attributes would be redundant if modeled in Fig. 25.12, because the compositions modeled in Fig. 25.10 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 25.12 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 25.6, we identify the states and activities of the objects in the model, and in Section 25.7 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we'll continue to strengthen the structure of our model.

Self-Review Exercises for Section 25.5

25.8 We typically identify the attributes of the classes in our system by analyzing the ________ in the requirements document.

a) nouns and noun phrases
b) descriptive words and phrases
c) verbs and verb phrases
d) All of the above.

25.9 Which of the following is *not* an attribute of an airplane?

a) length
b) wingspan
c) fly
d) number of seats

25.10 Describe the meaning of the following attribute declaration of class `CashDispenser` in the class diagram in Fig. 25.12:

```
count : Integer = 500
```

25.6 Identifying Objects' States and Activities

[*Note:* **This section can be studied after Chapter 5.**]

In Section 25.5, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 25.12. In this section, we show how these attributes represent an object's *state*. We identify some key states that our objects may occupy and discuss how objects change state in response to various events occurring in the system. We also discuss the workflow, or **activities**, that objects perform in the ATM system. We present the activities of `BalanceInquiry` and `Withdrawal` transaction objects in this section, as they represent two of the key activities in the ATM system.

State Machine Diagrams

Each object in a system goes through a series of discrete states. An object's current state is indicated by the values of the object's attributes at a given time. **State machine diagrams** (commonly called **state diagrams**) model key states of an object and show under what circumstances the object changes state. Unlike the class diagrams presented in earlier case study sections, which focused primarily on the *structure* of the system, state diagrams model some of the *behavior* of the system.

Figure 25.13 is a simple state diagram that models some of the states of an object of class `ATM`. The UML represents each state in a state diagram as a **rounded rectangle** with the name of the state placed inside it. A **solid circle** with an attached stick arrowhead designates the **initial state**. Recall that we modeled this state information as the `Boolean` attribute `userAuthenticated` in the class diagram of Fig. 25.12. This attribute is initialized to `false`, or the "User not authenticated" state, according to the state diagram.

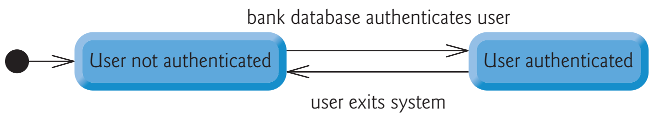


Fig. 25.13 | State diagram for the `ATM` object.

The arrows with stick arrowheads indicate **transitions** between states. An object can transition from one state to another in response to various *events* that occur in the system. The name or description of the event that causes a transition is written near the line that corresponds to the transition. For example, the `ATM` object changes from the "User not authenticated" state to the "User authenticated" state *after* the database authenticates the user. Recall from the requirements document that the database authenticates a user by comparing the account number and PIN entered by the user with those of the corresponding account in the database. If the database indicates that the user has entered a valid account number and the correct PIN, the `ATM` object transitions to the "User authenticated" state and changes its `userAuthenticated` attribute to a value of `true`. When the user exits the system by choosing the "exit" option from the main menu, the `ATM` object returns to the "User not authenticated" state in preparation for the next ATM user.

Software Engineering Observation 25.2

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.

Activity Diagrams

Like a state diagram, an activity diagram models aspects of system *behavior*. Unlike a state diagram, an activity diagram models an object's workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. Recall that we used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapters 4 and 5.

Figure 25.14 models the actions involved in executing a `BalanceInquiry` transaction. We assume that a `BalanceInquiry` object has been initialized and assigned a valid account number (that of the current user), so the object knows which balance to retrieve. The diagram includes the actions that occur after the user selects a balance inquiry from the main menu and before the ATM returns the user to the main menu—a `BalanceInquiry` object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the available balance of the user's account from the database. Next, the `BalanceInquiry` retrieves the total balance of the account. Finally, the transaction displays the balances on the screen. This action completes the execution of the transaction.

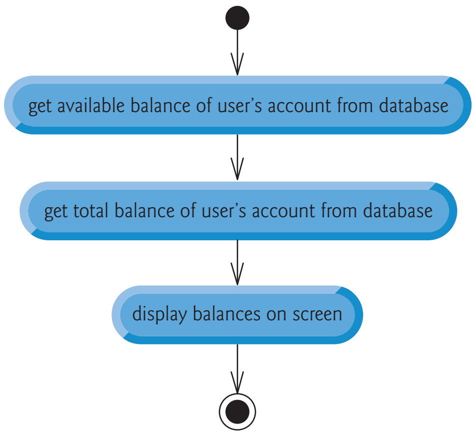


Fig. 25.14 | Activity diagram for a `BalanceInquiry` transaction.

The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expression—for example, "get available balance of user's account from database"—that specifies an action to be performed. An arrow with a stick arrowhead connects two action states, indicating the order in which the actions represented by the action states occur. The solid circle (at the top of Fig. 25.14) represents the activity's initial state—the beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the "get available balance of user's account from database" action expression. Second, the transaction retrieves the total balance. Finally, the transaction displays both balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 25.14) represents the final state—the end of the workflow after the object performs the modeled actions.

Figure 25.15 shows an activity diagram for a `Withdrawal` transaction. We assume that a `Withdrawal` object has been assigned a valid account number. We do not model the user selecting a withdrawal from the main menu or the ATM returning the user to the main menu because these are not actions performed by a `Withdrawal` object. The transaction first displays a menu of standard withdrawal amounts (Fig. 25.3) and an option to cancel the transaction. The transaction then inputs a menu selection from the user. The activity flow now arrives at a decision symbol. This point determines the next action based on the associated guard conditions. If the user cancels the transaction, the system displays an appropriate message. Next, the cancellation flow reaches a merge symbol, where this activity flow joins the transaction's other possible activity flows (which we discuss shortly).

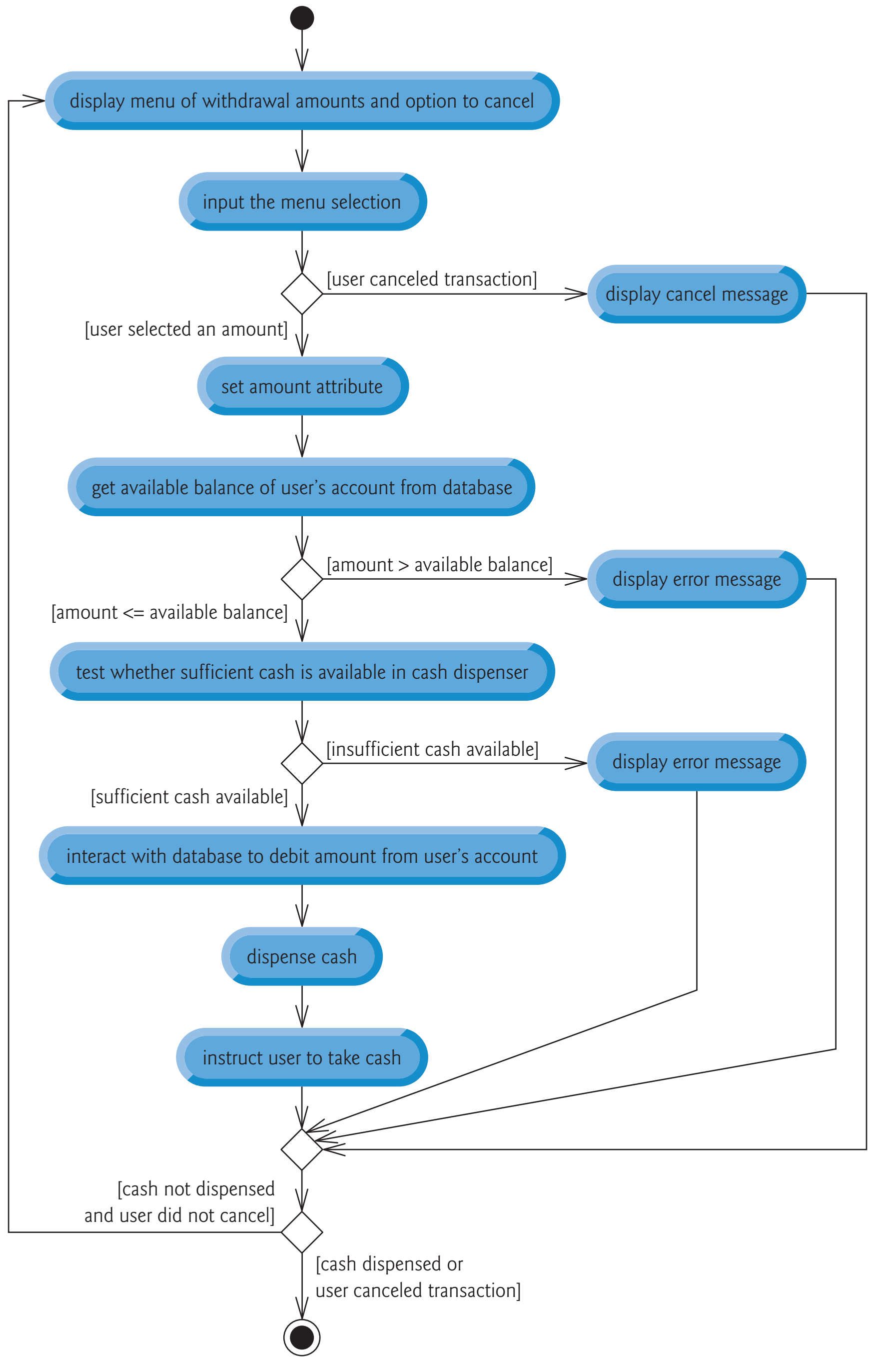


Fig. 25.15 | Activity diagram for a `Withdrawal` transaction.

A merge can have *any* number of incoming transition arrows, but only *one* outgoing transition arrow. The decision at the bottom of the diagram determines whether the transaction should repeat from the beginning. When the user has canceled the transaction, the guard condition "cash dispensed or user canceled transaction" is true, so control transitions to the activity's final state.

If the user selects a withdrawal amount from the menu, `the` transaction sets `amount` (an attribute of class `Withdrawal` originally modeled in Fig. 25.12) to the value chosen by the user. The transaction next gets the available balance of the user's account (i.e., the `availableBalance` attribute of the user's `Account` object) from the database. The activity flow then arrives at another decision. If the requested withdrawal amount exceeds the user's available balance, the system displays an appropriate error message informing the user of the problem. Control then merges with the other activity flows before reaching the decision at the bottom of the diagram. The guard decision "cash not dispensed and user did not cancel" is true, so the activity flow returns to the top of the diagram, and the transaction prompts the user to input a new amount.

If the requested withdrawal amount is less than or equal to the user's available balance, the transaction tests whether the cash dispenser has enough cash to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message and passes through the merge before reaching the final decision. Cash was not dispensed, so the activity flow returns to the beginning of the activity diagram, and the transaction prompts the user to choose a new amount. If sufficient cash is available, the transaction interacts with the database to debit the withdrawal amount from the user's account (i.e., subtract the amount from both the `availableBalance` and `totalBalance` attributes of the user's `Account` object). The transaction then dispenses the desired amount of cash and instructs the user to take the cash that is dispensed. The main flow of activity next merges with the two error flows and the cancellation flow. In this case, cash was dispensed, so the activity flow reaches the final state.

We've taken the first steps in modeling the ATM system's behavior and have shown how an object's attributes participate in the object's activities. In Section 25.7, we investigate the operations of our classes to create a more complete model of the system's behavior.

Self-Review Exercises for Section 25.6

25.11 State whether the following statement is *true* or *false*, and if *false*, explain why: State diagrams model structural aspects of a system.

25.12 An activity diagram models the __________ that an object performs and the order in which it performs them.

a) actions
b) attributes
c) states
d) state transitions

25.13 Based on the requirements document, create an activity diagram for a deposit transaction.

25.7 Identifying Class Operations

[*Note:* **This section can be studied after Chapter 6.**]

In Sections 25.4–25.6, we performed the first few steps in the object-oriented design of our ATM system. In Section 25.4, we identified the classes that we'll need to implement and we created our first class diagram. In Section 25.5, we described some attributes of our classes. In Section 25.6, we examined object states and modeled object state transitions and activities. Now, we determine some of the class operations (or behaviors) needed to implement the ATM system.

Identifying Operations

An *operation* is a service that objects of a class provide to clients of the class. Consider the operations of some real-world objects. A radio's operations include setting its station and volume (typically invoked by a person adjusting the radio's controls). A car's operations include accelerating (invoked by the driver pressing the accelerator pedal), decelerating (invoked by the driver pressing the brake pedal or releasing the gas pedal), turning and shifting gears. Software objects can offer operations as well—for example, a software graphics object might offer operations for drawing a circle, drawing a line, drawing a square and the like. A spreadsheet software object might offer operations like printing the spreadsheet, totaling the elements in a row or column and graphing information in the spreadsheet as a bar chart or pie chart.

We can derive many of the operations of each class by examining the key verbs and verb phrases in the requirements document. We then relate each of these to particular classes in our system (Fig. 25.16). The verb phrases in Fig. 25.16 help us determine the operations of each class.

Class	Verbs and verb phrases
`ATM`	executes financial transactions
`BalanceInquiry`	[none in the requirements document]
`Withdrawal`	[none in the requirements document]
`Deposit`	[none in the requirements document]
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Account`	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Screen`	displays a message to the user
`Keypad`	receives numeric input from the user
`CashDispenser`	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
`DepositSlot`	receives a deposit envelope

Fig. 25.16 | Verbs and verb phrases for each class in the ATM system.

Modeling Operations

To identify operations, we examine the verb phrases listed for each class in Fig. 25.16. The "executes financial transactions" phrase associated with class `ATM` implies that class `ATM` instructs transactions to execute. Therefore, classes `BalanceInquiry`, `Withdrawal` and `Deposit` each need an operation to provide this service to the ATM. We place this operation (which we've named `execute`) in the third compartment of the three transaction classes in the updated class diagram of Fig. 25.17. During an ATM session, the `ATM` object will invoke the `execute` operation of each transaction object to tell it to execute.

Fig. 25.17 | Classes in the ATM system with attributes and operations.

The UML represents operations (which are implemented as member functions in C++) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

operationName(*parameter1*, *parameter2*, …, *parameterN*) : *return type*

Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:

parameterName : *parameterType*

For the moment, we do not list the operations' parameters—we'll identify and model the parameters of some of the operations shortly. For some, we do not yet know the return types, so we also omit them from the diagram. These omissions are perfectly normal at this point. As our design and implementation proceed, we'll add the remaining return types.

Operations of Class `BankDatabase` *and Class* `Account`

Figure 25.16 lists the phrase "authenticates a user" next to class `BankDatabase`—the database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class `BankDatabase` needs an operation that provides an authentication service to the ATM. We place the operation `authenticateUser` in the third compartment of class `BankDatabase` (Fig. 25.17). However, an object of class `Account`, not class `BankDatabase`, stores the account number and PIN that must be accessed to authenticate a user, so class `Account` must provide a service to validate a PIN obtained through user input against a PIN stored in an `Account` object. Therefore, we add a `validatePIN` operation to class `Account`. We specify return type `Boolean` for the `authenticateUser` and `validatePIN` operations. Each operation returns a value indicating either that the operation was successful in performing its task (i.e., a return value of `true`) or that it was not (i.e., a return value of `false`).

Figure 25.16 lists several additional verb phrases for class `BankDatabase`: "retrieves an account balance," "credits a deposit amount to an account" and "debits a withdrawal amount from an account." Like "authenticates a user," these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class `Account` actually perform the operations to which these phrases refer. Thus, we assign an operation to both class `BankDatabase` and class `Account` to correspond to each of these phrases. Recall from Section 25.4 that, because a bank account contains sensitive information, we do *not* allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we'll see in Section 25.8, class `ATM` invokes the operations of class `BankDatabase`, each of which in turn invokes the operation with the same name in class `Account`.

The phrase "retrieves an account balance" suggests that classes `BankDatabase` and `Account` each need a `getBalance` operation. However, recall that we created two attributes in class `Account` to represent a balance—`availableBalance` and `totalBalance`. A balance inquiry requires access to both balance attributes so that it can display them to the user, but a withdrawal needs to check only the value of `availableBalance`. To allow objects in the system to obtain each balance attribute individually, we add operations `getAvailableBalance` and `getTotalBalance` to the third compartment of classes `BankDatabase` and `Account` (Fig. 25.17). We specify a return type of `Double` for each of these operations, because the balance attributes which they retrieve are of type `Double`.

The phrases "credits a deposit amount to an account" and "debits a withdrawal amount from an account" indicate that classes `BankDatabase` and `Account` must perform operations to update an account during a deposit and withdrawal, respectively. We therefore assign `credit` and `debit` operations to classes `BankDatabase` and `Account`. You may recall that crediting an account (as in a deposit) adds an amount only to the `totalBalance` attribute. Debiting an account (as in a withdrawal), on the other hand, subtracts the amount from both balance attributes. We hide these implementation details inside class `Account`. This is a good example of encapsulation and information hiding.

If this were a real ATM system, classes `BankDatabase` and `Account` would also provide a set of operations to allow another banking system to update a user's account balance after either confirming or rejecting all or part of a deposit. Operation `confirmDepositAmount`, for example, would add an amount to the `availableBalance` attribute, thus making deposited funds available for withdrawal. Operation `rejectDepositAmount` would subtract an amount from the `totalBalance` attribute to indicate that a specified amount, which had recently been deposited through the ATM and added to the `totalBalance`, was not found in the deposit envelope. The bank would invoke this operation after determining either that the user failed to include the correct amount of cash or that any checks did not clear (i.e, they "bounced"). While adding these operations would make our system more complete, we do not include them in our class diagrams or our implementation because they are beyond the scope of the case study.

Operations of Class `Screen`

Class `Screen` "displays a message to the user" at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class `Screen` an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (`displayMessage`) in the third compartment of class `Screen` in our class diagram (Fig. 25.17). We do not worry about the parameter of this operation at this time—we model the parameter later in this section.

Operations of Class `Keypad`

From the phrase "receives numeric input from the user" listed by class `Keypad` in Fig. 25.16, we conclude that class `Keypad` should perform a `getInput` operation. Because the ATM's keypad, unlike a computer keyboard, contains only the numbers 0–9, we specify that this operation returns an integer value. Recall from the requirements document that in different situations the user may be required to enter a different type of number (e.g., an account number, a PIN, the number of a menu option, a deposit amount as a number of cents). Class `Keypad` simply obtains a numeric value for a client of the class—it does *not* determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user enters appropriate numbers, and if not, display error messages via class `Screen`). [*Note:* When we implement the system, we simulate the ATM's keypad with a computer keyboard, and for simplicity we assume that the user does not enter nonnumeric input using keys on the computer keyboard that do not appear on the ATM's keypad.]

Operations of Class `CashDispenser` *and Class* `DepositSlot`

Figure 25.16 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 25.17. Class `CashDispenser` also "indicates whether it contains enough cash to satisfy a withdrawal request." Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`. Figure 25.16 also lists "receives a deposit envelope" for class `DepositSlot`. The deposit slot must indicate whether it received an envelope, so we place an operation `isEnvelopeReceived`, which returns a `Boolean` value, in the third compartment of class `DepositSlot`. [*Note:* A real hardware deposit slot would most likely send the ATM a signal to indicate that an envelope was received. We simulate this behavior, however, with an operation in class `DepositSlot` that class `ATM` can invoke to find out whether the deposit slot received an envelope.]

Operations of Class `ATM`

We do not list any operations for class `ATM` at this time. We are not yet aware of any services that class `ATM` provides to other classes in the system. When we implement the system with C++ code, however, operations of this class, and additional operations of the other classes in the system, may emerge.

Identifying and Modeling Operation Parameters

So far, we've not been concerned with the parameters of our operations—we've attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires to perform its assigned task.

Consider the `authenticateUser` operation of class `BankDatabase`. To authenticate a user, this operation must know the account number and PIN supplied by the user. Thus we specify that operation `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to the account number and PIN of an `Account` object in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and the attribute names that belong to class `Account`. We list these parameters in the class diagram in Fig. 25.18 that models only class `BankDatabase`. [*Note:* It's perfectly normal to model only one class in a class diagram. In this case, we are most concerned with examining the parameters of this one class in particular, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit the parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). Figure 25.18 thus specifies that operation `authenticateUser` takes two parameters—`userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in C++, we'll represent these parameters with `int` values.

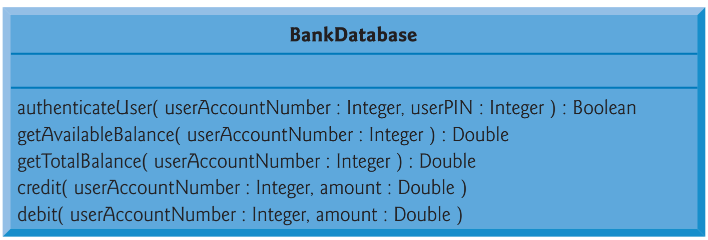


Fig. 25.18 | Class `BankDatabase` with operation parameters.

Class `BankDatabase` operations `getAvailableBalance`, `getTotalBalance`, `credit` and `debit` also each require a `userAccountNumber` parameter to identify the account to which the database must apply the operations, so we include these parameters in the class diagram of Fig. 25.18. In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

The class diagram in Fig. 25.19 models the parameters of class `Account`'s operations. Operation `validatePIN` requires only a `userPIN` parameter, which contains the user-specified PIN to be compared with the PIN associated with the account. Like their counterparts in class `BankDatabase`, operations `credit` and `debit` in class `Account` each require a `Double` parameter `amount` that indicates the amount of money involved in the operation. Operations `getAvailableBalance` and `getTotalBalance` in class `Account` require no additional data to perform their tasks. Class `Account`'s operations do not require an account number parameter—each of these operations can be invoked only on a specific `Account` object, so including a parameter to specify an `Account` is unnecessary.

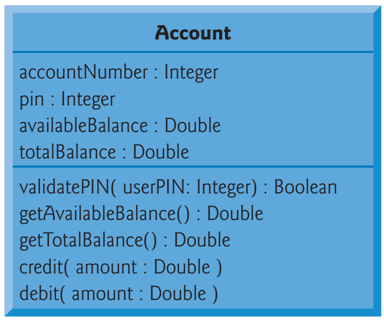


Fig. 25.19 | Class `Account` with operation parameters.

Figure 25.20 models class `Screen` with a parameter specified for operation `displayMessage`. This operation requires only a `String` parameter `message` that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the `String` type listed in Fig. 25.20 refers to the UML type. When we implement the system in C++, we'll in fact use a C++ `string` object to represent this parameter.

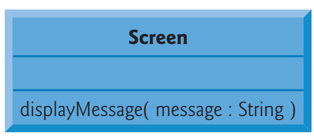


Fig. 25.20 | Class `Screen` with operation parameters.

The class diagram in Fig. 25.21 specifies that operation `dispenseCash` of class `CashDispenser` takes a `Double` parameter `amount` to indicate the amount of cash (in dollars) to be dispensed. Operation `isSufficientCashAvailable` also takes a `Double` parameter `amount` to indicate the amount of cash in question.

We do not discuss parameters for operation `execute` of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, operation `getInput` of class `Keypad` and operation `isEnvelopeReceived` of class `DepositSlot`. At this point in our design process, we cannot determine

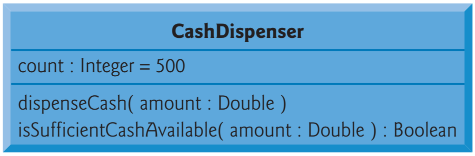


Fig. 25.21 | Class `CashDispenser` with operation parameters.

whether these operations require additional data to perform their tasks, so we leave their parameter lists empty. As we progress through the case study, we may decide to add parameters to these operations.

In this section, we've determined many of the operations performed by the classes in the ATM system. We've identified the parameters and return types of some of the operations. As we continue our design process, the number of operations belonging to each class may vary—we might find that new operations are needed or that some current operations are unnecessary—and we might determine that some of our class operations need additional parameters and different return types.

Self-Review Exercises for Section 25.7

25.14 Which of the following is not a behavior?
a) reading data from a file
b) printing output
c) text output
d) obtaining input from the user

25.15 If you were to add to the ATM system an operation that returns the `amount` attribute of class `Withdrawal`, how and where would you specify this operation in the class diagram of Fig. 25.17?

25.16 Describe the meaning of the following operation listing that might appear in a class diagram for an object-oriented design of a calculator:

```
add( x : Integer, y : Integer ) : Integer
```

25.8 Indicating Collaboration Among Objects

[*Note:* **This section can be studied after Chapter 7.**]
In this section, we concentrate on the collaborations (interactions) among objects in our ATM system. When two objects communicate with each other to accomplish a task, they are said to **collaborate**—they do this by invoking one another's operations. A **collaboration** consists of an object of one class sending a **message** to an object of another class. Messages are sent in C++ via member-function calls.

In Section 25.7, we determined many of the *operations* of the system's classes. Next, we concentrate on the *messages* that invoke these operations. To identify the collaborations, we return to the requirements document in Section 25.3. Recall that this document specifies the range of activities that occur during an ATM session (e.g., authenticating a user, performing transactions). The steps used to describe how the system must perform each of these tasks are our first indication of the collaborations in our system. As we proceed through this and the remaining sections, we may discover additional collaborations.

Identifying the Collaborations in a System

We identify the collaborations in the system by carefully reading the requirements document sections that specify what the ATM should do to authenticate a user and to perform each transaction type. For each action or step described, we decide which objects in our system must interact to achieve the desired result. We identify one object as the *sending object* (i.e., the object that sends the message) and another as the *receiving object* (i.e., the object that offers that operation to clients of the class). We then select one of the receiving object's operations (identified in Section 25.7) that must be invoked by the sending object to produce the proper behavior. For example, the ATM displays a welcome message when idle. We know that an object of class `Screen` displays a message to the user via its `displayMessage` operation. Thus, we decide that the system can display a welcome message by employing a collaboration between the `ATM` and the `Screen` in which the `ATM` sends a `displayMessage` message to the `Screen` by invoking the `displayMessage` operation of class `Screen`. [*Note:* To avoid repeating the phrase "an object of class…," we refer to each object simply by using its class name preceded by an article ("a," "an" or "the")—for example, "the `ATM`" refers to an object of class `ATM`.]

Figure 25.22 lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they are discussed in the requirements document. We list each collaboration involving a unique sender, message and recipient only once, even though the collaboration may occur several times during an ATM session. For example, the first row in Fig. 25.22 indicates that the `ATM` collaborates with the `Screen` whenever the `ATM` needs to display a message to the user.

An object of class...	sends the message...	to an object of class...
`ATM`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`authenticateUser`	`BankDatabase`
	`execute`	`BalanceInquiry`
	`execute`	`Withdrawal`
	`execute`	`Deposit`
`BalanceInquiry`	`getAvailableBalance`	`BankDatabase`
	`getTotalBalance`	`BankDatabase`
	`displayMessage`	`Screen`
`Withdrawal`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`getAvailableBalance`	`BankDatabase`
	`isSufficientCashAvailable`	`CashDispenser`
	`debit`	`BankDatabase`
	`dispenseCash`	`CashDispenser`
`Deposit`	`displayMessage`	`Screen`
	`getInput`	`Keypad`
	`isEnvelopeReceived`	`DepositSlot`
	`credit`	`BankDatabase`

Fig. 25.22 | Collaborations in the ATM system. (Part 1 of 2.)

An object of class...	sends the message...	to an object of class...
`BankDatabase`	`validatePIN`	`Account`
	`getAvailableBalance`	`Account`
	`getTotalBalance`	`Account`
	`debit`	`Account`
	`credit`	`Account`

Fig. 25.22 | Collaborations in the ATM system. (Part 2 of 2.)

Let's consider the collaborations in Fig. 25.22. Before allowing a user to perform any transactions, the ATM must prompt the user to enter an account number, then to enter a PIN. It accomplishes each of these tasks by sending a `displayMessage` message to the `Screen`. Both of these actions refer to the same collaboration between the `ATM` and the `Screen`, which is already listed in Fig. 25.22. The `ATM` obtains input in response to a prompt by sending a `getInput` message to the `Keypad`. Next, the ATM must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an `authenticateUser` message to the `BankDatabase`. Recall that the `BankDatabase` cannot authenticate a user directly—only the user's `Account` (i.e., the `Account` that contains the account number specified by the user) can access the user's PIN to authenticate the user. Figure 25.22 therefore lists a collaboration in which the `BankDatabase` sends a `validatePIN` message to an `Account`.

After the user is authenticated, the `ATM` displays the main menu by sending a series of `displayMessage` messages to the `Screen` and obtains input containing a menu selection by sending a `getInput` message to the `Keypad`. We've already accounted for these collaborations. After the user chooses a type of transaction to perform, the `ATM` executes the transaction by sending an `execute` message to an object of the appropriate transaction class (i.e., a `BalanceInquiry`, a `Withdrawal` or a `Deposit`). For example, if the user chooses to perform a balance inquiry, the `ATM` sends an `execute` message to a `BalanceInquiry`.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A `BalanceInquiry` retrieves the amount of money available in the user's account by sending a `getAvailableBalance` message to the `BankDatabase`, which responds by sending a `getAvailableBalance` message to the user's `Account`. Similarly, the `BalanceInquiry` retrieves the amount of money on deposit by sending a `getTotalBalance` message to the `BankDatabase`, which sends the same message to the user's `Account`. To display both measures of the user's balance at the same time, the `BalanceInquiry` sends a `displayMessage` message to the `Screen`.

A `Withdrawal` sends the `Screen` several `displayMessage` messages to display a menu of standard withdrawal amounts (i.e., \$20, \$40, \$60, \$100, \$200). The `Withdrawal` sends the `Keypad` a `getInput` message to obtain the user's menu selection, then determines whether the requested withdrawal amount is less than or equal to the user's account balance. The `Withdrawal` can obtain the amount of money available in the account by sending the `BankDatabase` a `getAvailableBalance` message. The `Withdrawal` then tests whether the cash dispenser contains enough cash by sending the `CashDispenser` an `isSufficientCashAvailable` message. A `Withdrawal` sends the `BankDatabase` a `debit` message to decrease the user's account balance. The `BankDatabase` sends the same message to the appropriate `Account`. Recall that debiting funds from an `Account` decreases both the `totalBalance` and the `availableBalance`. To dispense the requested amount of cash, the `Withdrawal` sends the `CashDispenser` a `dispenseCash` message. Finally, the `Withdrawal` sends a `displayMessage` message to the `Screen`, instructing the user to take the cash.

A `Deposit` responds to an `execute` message first by sending a `displayMessage` message to the `Screen` to prompt the user for a deposit amount. The `Deposit` sends a `getInput` message to the `Keypad` to obtain the user's input. The `Deposit` then sends a `displayMessage` message to the `Screen` to tell the user to insert a deposit envelope. To determine whether the deposit slot received an incoming deposit envelope, the `Deposit` sends an `isEnvelopeReceived` message to the `DepositSlot`. The `Deposit` updates the user's account by sending a `credit` message to the `BankDatabase`, which subsequently sends a `credit` message to the user's `Account`. Recall that crediting funds to an `Account` increases the `totalBalance` but not the `availableBalance`.

Interaction Diagrams

Now that we've identified possible collaborations between the objects in our ATM system, let's graphically model these interactions using the UML. Several types of **interaction diagrams** model the behavior of a system by showing how objects interact with one another. The **communication diagram** emphasizes which objects participate in collaborations. [*Note:* Communication diagrams were called **collaboration diagrams** in earlier versions of the UML.] Like the communication diagram, the **sequence diagram** shows collaborations among objects, but it emphasizes *when* messages are sent between objects *over time*.

Communication Diagrams

Figure 25.23 shows a communication diagram that models the `ATM` executing a `BalanceInquiry`. Objects are modeled in the UML as rectangles containing names in the form `objectName : ClassName`. In this example, which involves only one object of each type, we disregard the object name and list only a colon followed by the class name. [*Note:* Specifying the name of each object in a communication diagram is recommended when modeling multiple objects of the same type.] Communicating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of an operation (i.e., a member function) belonging to the receiving object—think of the name as a service that the receiving object provides to sending objects (its "clients").

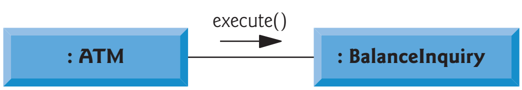


Fig. 25.23 | Communication diagram of the ATM executing a balance inquiry.

The solid filled arrow in Fig. 25.23 represents a message—or **synchronous call**—in the UML and a function call in C++. This arrow indicates that the flow of control is from the sending object (the `ATM`) to the receiving object (a `BalanceInquiry`). *Since this is a synchronous call, the sending object may not send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object—the sender just* *waits*. For example, in Fig. 25.23, the `ATM` calls member function `execute` of a `BalanceInquiry` and may not send another message until `execute` has finished and returns control to the `ATM`. [*Note:* If this were an **asynchronous call**, represented by a stick arrowhead, the sending object would not have to wait for the receiving object to return control—it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls often can be implemented in C++ using platform-specific libraries provided with your compiler. Such techniques are beyond the scope of this book.]

Sequence of Messages in a Communication Diagram

Figure 25.24 shows a communication diagram that models the interactions among objects in the system when an object of class `BalanceInquiry` executes. We assume that the object's `accountNumber` attribute contains the account number of the current user. The collaborations in Fig. 25.24 begin after the `ATM` sends an `execute` message to a `BalanceInquiry` (i.e., the interaction modeled in Fig. 25.23). The number to the left of a message name indicates the order in which the message is passed. The **sequence of messages** in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message `1` and ends with message `3`. The `BalanceInquiry` first sends a `getAvailableBalance` message to the `BankDatabase` (message `1`), then sends a `getTotalBalance` message to the `BankDatabase` (message `2`). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a C++ function call)—the `BalanceInquiry` passes attribute `accountNumber` with its messages to the `BankDatabase` to indicate which `Account`'s balance information to retrieve. Recall from Fig. 25.18 that operations `getAvailableBalance` and `getTotalBalance` of class `BankDatabase` each require a parameter to identify an account. The `BalanceInquiry` next displays the `availableBalance` and the `totalBalance` to the user by passing a `displayMessage` message to the `Screen` (message `3`) that includes a parameter indicating the `message` to be displayed.

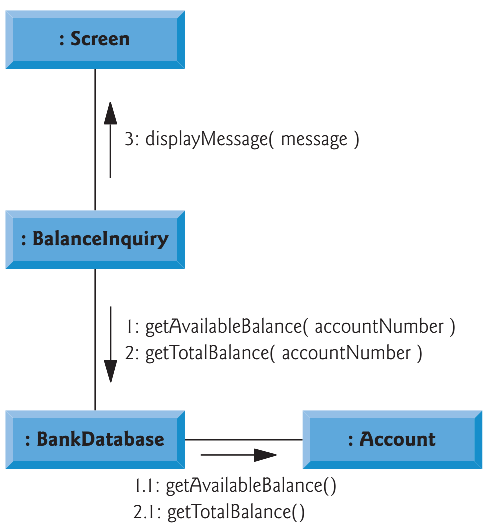


Fig. 25.24 | Communication diagram for executing a balance inquiry.

Figure 25.24 models two additional messages passing from the `BankDatabase` to an `Account` (message `1.1` and message `2.1`). To provide the `ATM` with the two balances of the user's `Account` (as requested by messages `1` and `2`), the `BankDatabase` must pass a `getAvailableBalance` and a `getTotalBalance` message to the user's `Account`. Messages passed within the handling of another message are called **nested messages**. The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message `1.1` is the first message nested in message `1`—the `BankDatabase` passes a `getAvailableBalance` message while processing `BankDatabase`'s message of the same name. [*Note:* If the `BankDatabase` needed to pass a second nested message while processing message `1`, the second message would be numbered `1.2`.] A message may be passed only when all the nested messages from the previous message have been passed—e.g., the `BalanceInquiry` passes message `3` only after messages `2` and `2.1` have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in Fig. 25.24 using a flat numbering scheme (i.e., `1`, `2`, `3`, `4`, `5`), someone looking at the diagram might not be able to determine that `BankDatabase` passes the `getAvailableBalance` message (message `1.1`) to an `Account` *during* the `BankDatabase`'s processing of message `1`, as opposed to *after* completing the processing of message `1`. The nested decimal numbers make it clear that the second `getAvailableBalance` message (message `1.1`) is passed to an `Account` within the handling of the first `getAvailableBalance` message (message `1`) by the `BankDatabase`.

Sequence Diagrams

Communication diagrams emphasize the participants in collaborations but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. Figure 25.25 shows a sequence diagram modeling the sequence of interactions that occur when a `Withdrawal` executes. The dotted line extending down from an object's rectangle is that object's **lifeline**, which represents the progression of time. Actions typically occur along an object's lifeline in *chronological order* from top to bottom—an action near the top typically happens before one near the bottom.

Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object's lifeline. An **activation**, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return message, represented as a dashed line with a stick arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 25.25 begins when a `Withdrawal` prompts the user to choose a withdrawal amount by sending a `displayMessage` message to the `Screen`. The `Withdrawal` then sends a `getInput` message to the `Keypad`, which obtains input from the user. We've already modeled the control logic involved in a `Withdrawal` in the activity diagram of Fig. 25.15, so we do not show this logic in the sequence diagram of Fig. 25.25. Instead, we model the best-case scenario in which the balance of the user's account is greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a

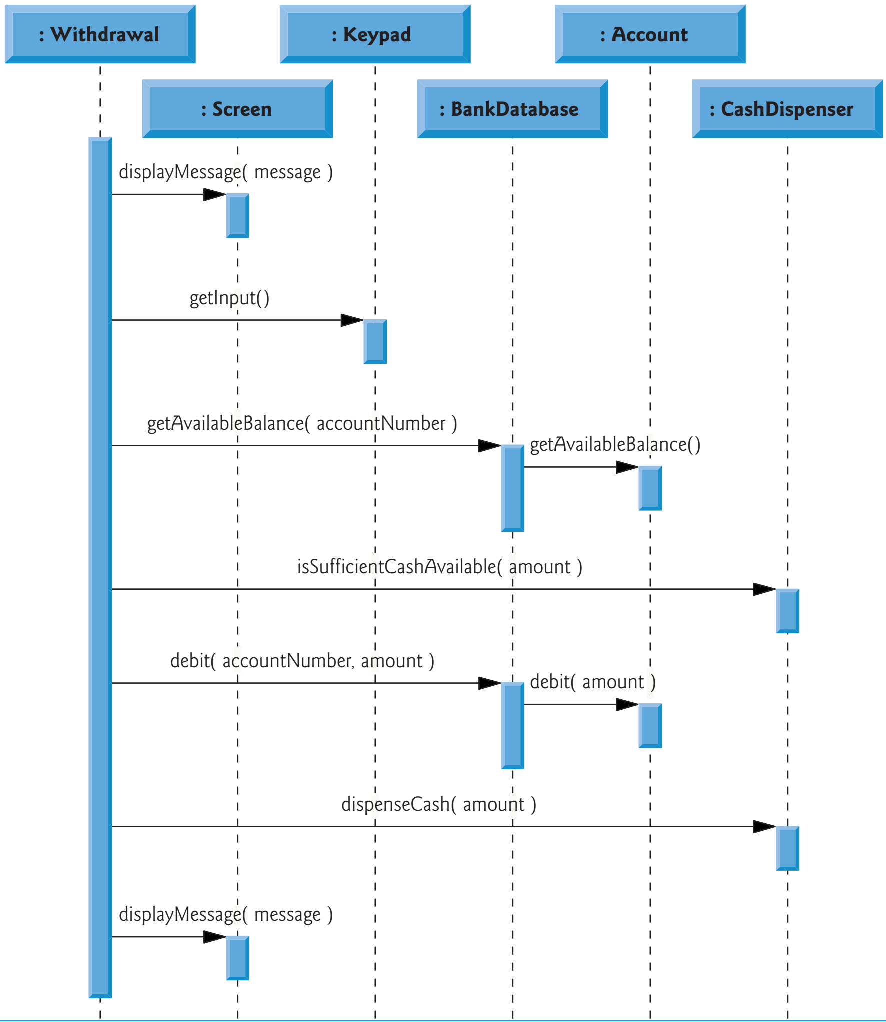


Fig. 25.25 | Sequence diagram that models a `Withdrawal` executing.

sufficient amount of cash to satisfy the request. For information on how to model control logic in a sequence diagram, please refer to the web resources at the end of Section 25.3.

After obtaining a withdrawal amount, the `Withdrawal` sends a `getAvailableBalance` message to the `BankDatabase`, which in turn sends a `getAvailableBalance` message to the user's `Account`. Assuming that the user's account has enough money available to permit the transaction, the `Withdrawal` next sends an `isSufficientCashAvailable` message to the `CashDispenser`. Assuming that there is enough cash available, the `Withdrawal` decreases the balance of the user's account (i.e., both the `totalBalance` and the `availableBalance`) by sending a `debit` message to the `BankDatabase`. The `BankDatabase` responds by sending a `debit` message to the user's `Account`. Finally, the `Withdrawal` sends a `dispenseCash` message to the `CashDispenser` and a `displayMessage` message to the `Screen`, telling the user to remove the cash from the machine.

We've identified the collaborations among the ATM system's objects and modeled some of them using UML interaction diagrams—both communication diagrams and sequence diagrams. In Section 26.2, we enhance the structure of our model to complete a preliminary object-oriented design, then we implement the ATM system in C++.

Self-Review Exercises for Section 25.8

25.17 A(n) ________ consists of an object of one class sending a message to an object of another class.
a) association
b) aggregation
c) collaboration
d) composition

25.18 Which form of interaction diagram emphasizes *what* collaborations occur? Which form emphasizes *when* collaborations occur?

25.19 Create a sequence diagram that models the interactions among objects in the ATM system that occur when a `Deposit` executes successfully, and explain the sequence of messages modeled by the diagram.

25.9 Wrap-Up

In this chapter, you learned how to work from a detailed requirements document to develop an object-oriented design. You worked with six popular types of UML diagrams to graphically model an object-oriented automated teller machine software system. In Section 26.3, we tune the design using inheritance, then completely implement the design in an 850-line C++ application.

Answers to Self-Review Exercises

25.1 Figure 25.26 shows a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

25.2 b.

25.3 d.

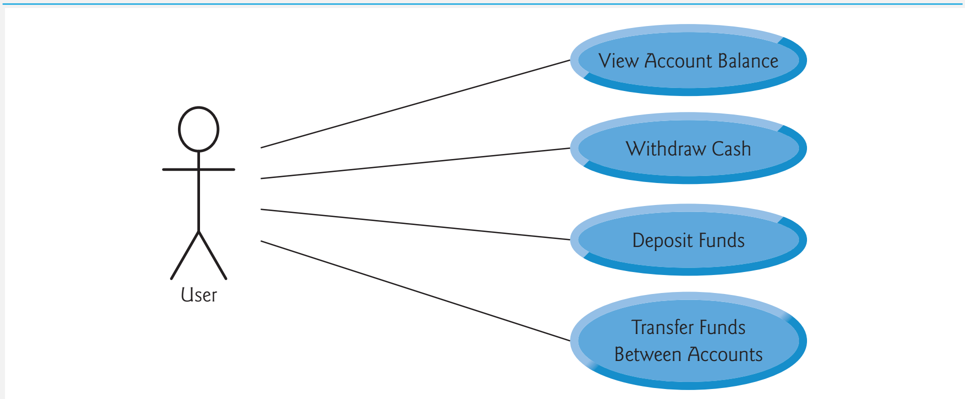


Fig. 25.26 | Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

25.4 [*Note:* Answers may vary.] Figure 25.27 presents a class diagram that shows some of the composition relationships of a class `Car`.

25.5 c. [*Note:* In a computer network, this relationship could be many-to-many.]

25.6 True.

25.7 Figure 25.28 presents an ATM class diagram including class `Deposit` instead of class `Withdrawal`. Note that `Deposit` does not access `CashDispenser`, but does access `DepositSlot`.

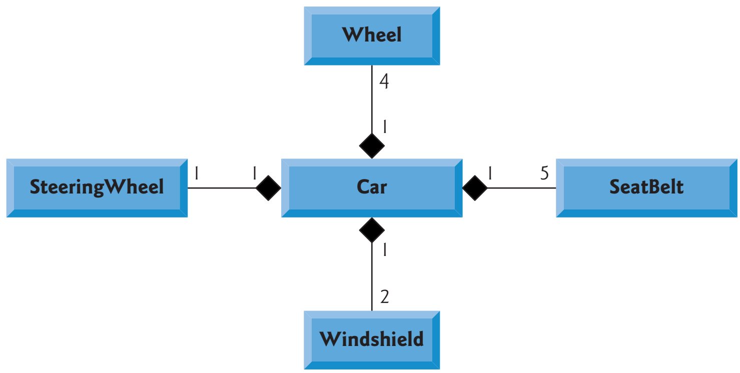


Fig. 25.27 | Class diagram showing composition relationships of a class `Car`.

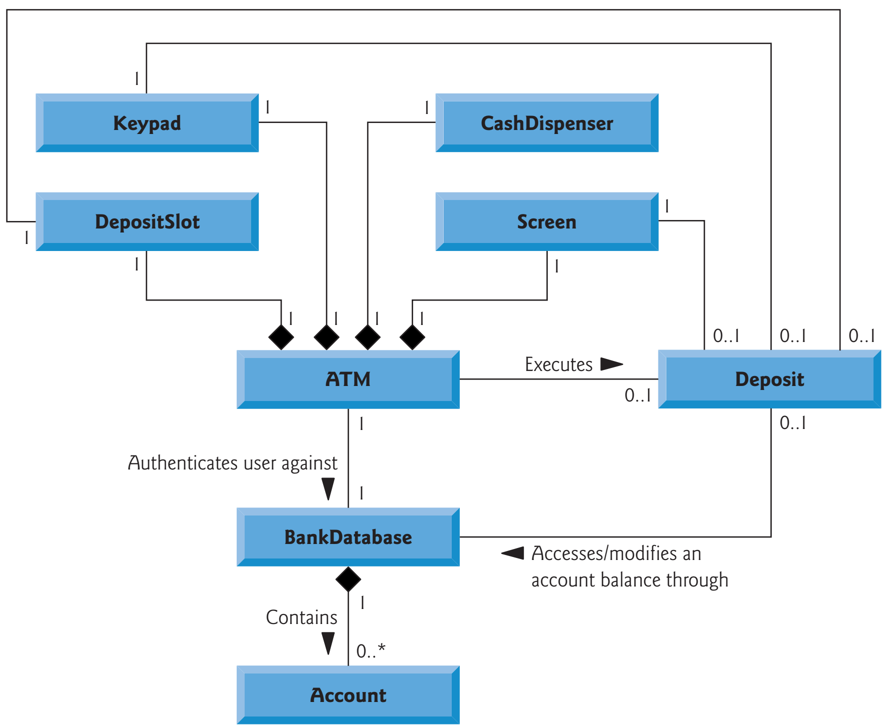


Fig. 25.28 | Class diagram for the ATM system model including class `Deposit`.

25.8 b.

25.9 c. Fly is an operation or behavior of an airplane, not an attribute.

25.10 This indicates that `count` is an `Integer` with an initial value of `500`. This attribute keeps track of the number of bills available in the `CashDispenser` at any given time.

25.11 False. State diagrams model some of the behavior of a system.

25.12 a.

25.13 Figure 25.29's activity diagram models the actions that occur after the user chooses the deposit option from the main menu and before the ATM returns the user to the main menu. Recall that part of receiving a deposit amount from the user involves converting an integer number of cents to a dollar amount. Also recall that crediting a deposit amount to an account involves increasing only the `totalBalance` attribute of the user's `Account` object. The bank updates the `availableBal-`

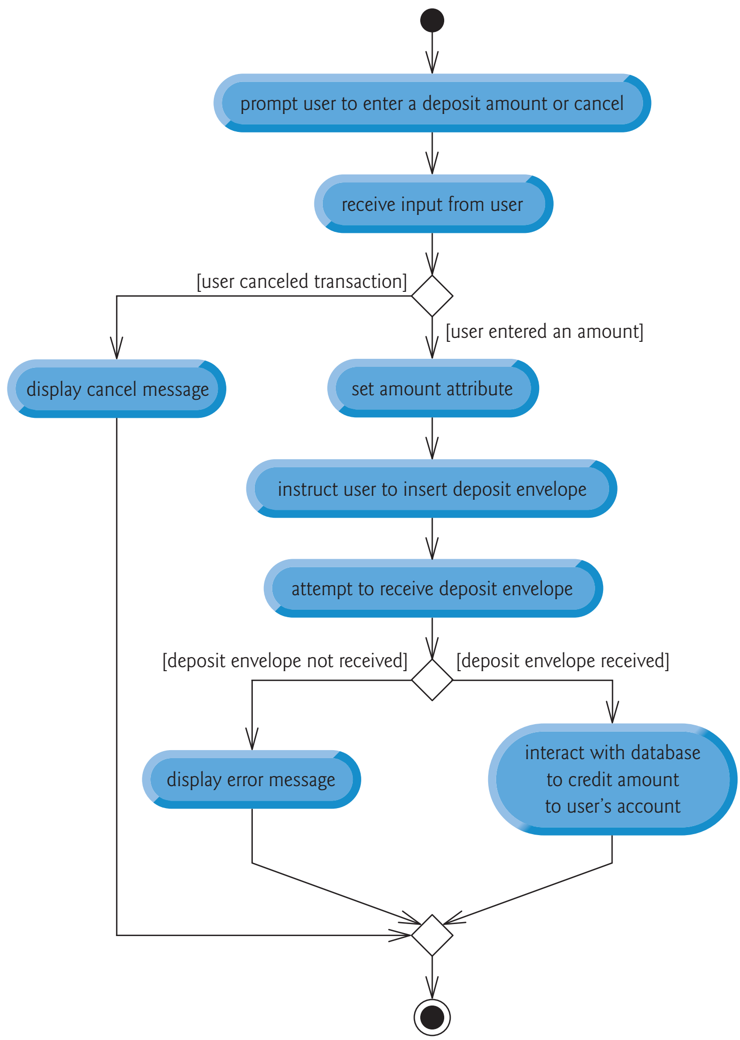


Fig. 25.29 | Activity diagram for a `Deposit` transaction.

ance attribute of the user's `Account` object only after confirming the amount of cash in the deposit envelope and after the enclosed checks clear—this occurs independently of the ATM system.

25.14 c.

25.15 To specify an operation that retrieves the `amount` attribute of class `Withdrawal`, the following operation would be placed in the operation (i.e., third) compartment of class `Withdrawal`:

```
getAmount( ) : Double
```

25.16 This is an operation named `add` that takes integers `x` and `y` as parameters and returns an integer value.

25.17 c.

25.18 Communication diagrams emphasize *what* collaborations occur. Sequence diagrams emphasize *when* collaborations occur.

25.19 Figure 25.30 presents a sequence diagram that models the interactions between objects that occur when a `Deposit` executes successfully. A `Deposit` first sends a `displayMessage` message to the `Screen` to ask the user to enter a deposit amount. Next, it sends a `getInput` message to the `Keypad` to receive input from the user. Then, it instructs the user to insert a deposit envelope by sending a `displayMessage` message to the `Screen`. It then sends an `isEnvelopeReceived` message to the `DepositSlot` to confirm that the deposit envelope has been received. Finally, it increases the `totalBalance` attribute (but not the `availableBalance` attribute) of the user's `Account` by sending a `credit` message to the `BankDatabase`. The `BankDatabase` responds by sending the same message to the user's `Account`.

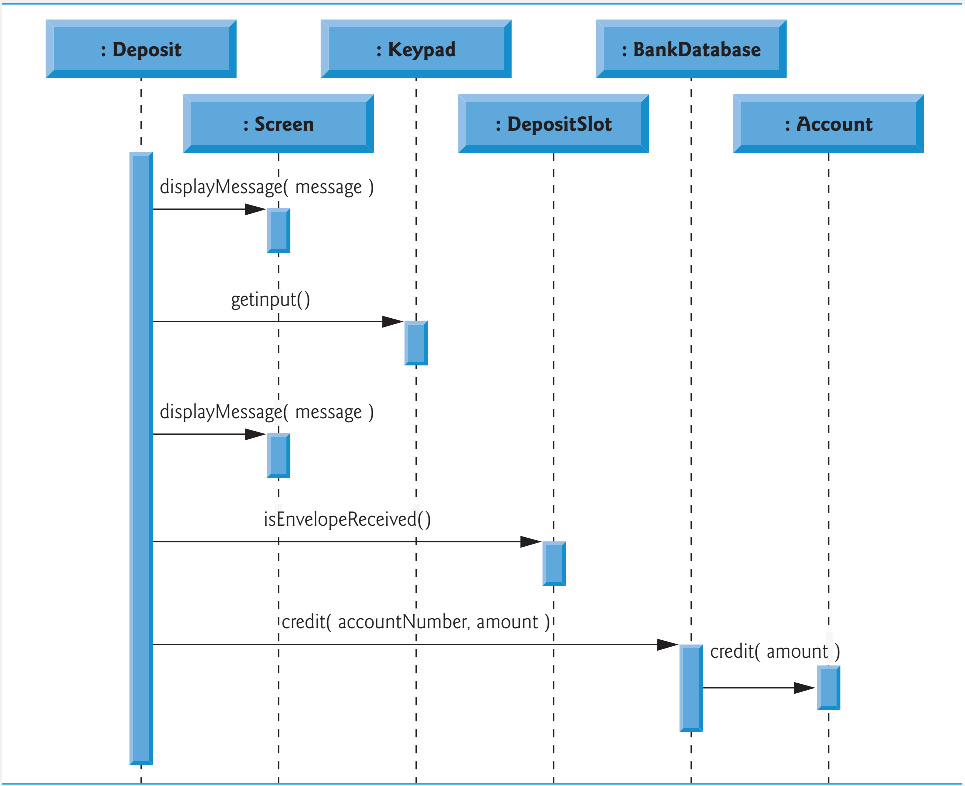


Fig. 25.30 | Sequence diagram that models a `Deposit` executing.

26

ATM Case Study, Part 2: Implementing an Object-Oriented Design

You can't work in the abstract.
—I. M. Pei

To generalize means to think.
—Georg Wilhelm Friedrich Hegel

We are all gifted. That is our inheritance.
—Ethel Waters

Let me walk through the fields of paper
touching with my wand
dry stems and stunted butterflies…
—Denise Levertov

Objectives

In this chapter you'll learn:

- Incorporate inheritance into the design of the ATM.
- Incorporate polymorphism into the design of the ATM.
- Fully implement in C++ the UML-based object-oriented design of the ATM software.
- Study a detailed code walkthrough of the ATM software system that explains the implementation issues.

Outline

26.1 Introduction

In Chapter 25, we developed an object-oriented design for our ATM system. We now begin implementing our object-oriented design in C++. In Section 26.2, we show how to convert class diagrams to C++ code. In Section 26.3, we tune the design with inheritance and polymorphism. Then we present a full C++ code implementation of the ATM software in Section 26.4. The code is carefully commented and the discussions of the implementation are thorough and precise. Studying this application provides the opportunity for you to see a more substantial application of the kind you're likely to encounter in industry.

26.2 Starting to Program the Classes of the ATM System

[*Note:* **This section can be studied after Chapter 9.**]

Visibility

We now apply access specifiers to the members of our classes. Access specifiers `public` and `private` determine the **visibility** or accessibility of an object's attributes and operations to other objects. Before we can begin implementing our design, we must consider which attributes and operations of our classes should be `public` and which should be `private`.

Previously, we observed that data members normally should be `private` and that member functions invoked by clients of a given class should be `public`. Member functions that are called only by other member functions of the class as "utility functions," however, normally should be `private`. The UML employs **visibility markers** for modeling the visibility of attributes and operations. Public visibility is indicated by placing a plus sign (+) before an operation or an attribute; a minus sign (–) indicates private visibility. Figure 26.1 shows our updated class diagram with visibility markers included. [*Note:* We do not include any operation parameters in Fig. 26.1. This is perfectly normal. Adding visibility markers does not affect the parameters already modeled in the class diagrams of Figs. 25.18–25.21.]

Navigability

Before we begin implementing our design in C++, we introduce an additional UML notation. The class diagram in Fig. 26.2 further refines the relationships among classes in the ATM system by adding navigability arrows to the association lines. **Navigability arrows** (represented as arrows with stick arrowheads in the class diagram) indicate in which direction an association can be traversed and are based on the collaborations modeled in communica-

Fig. 26.1 | Class diagram with visibility markers.

tion and sequence diagrams (see Section 25.8). When implementing a system designed using the UML, you use navigability arrows to help determine which objects need references or pointers to other objects. For example, the navigability arrow pointing from class `ATM` to class `BankDatabase` indicates that we can navigate from the former to the latter, thereby enabling the `ATM` to invoke the `BankDatabase`'s operations. However, since Fig. 26.2 does not contain a navigability arrow pointing from class `BankDatabase` to class `ATM`, the `BankDatabase` cannot access the `ATM`'s operations. Associations in a class diagram that have navigability arrows at both ends or do not have navigability arrows at all indicate **bidirectional navigability**—navigation can proceed in either direction across the association.

Like the class diagram of Fig. 25.10, the class diagram of Fig. 26.2 omits classes `BalanceInquiry` and `Deposit` to keep the diagram simple. The navigability of the associations in which these classes participate closely parallels the navigability of class `Withdrawal`'s associations. Recall from Section 25.4 that `BalanceInquiry` has an association with class `Screen`. We can navigate from class `BalanceInquiry` to class `Screen` along this association, but we cannot navigate from class `Screen` to class `BalanceInquiry`. Thus, if we were to

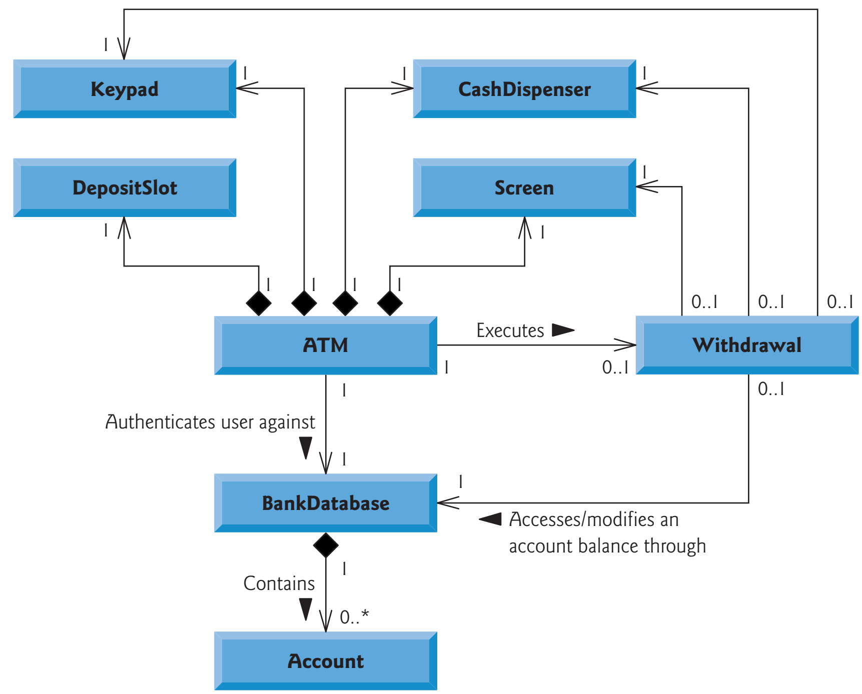


Fig. 26.2 | Class diagram with navigability arrows.

model class `BalanceInquiry` in Fig. 26.2, we would place a navigability arrow at class `Screen`'s end of this association. Also recall that class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. We can navigate from class `Deposit` to each of these classes, but not vice versa. We therefore would place navigability arrows at the `Screen`, `Keypad` and `DepositSlot` ends of these associations. [*Note:* We model these additional classes and associations in our final class diagram in Section 26.3, after we have simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

Implementing the ATM System from Its UML Design

We are now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 26.1 and Fig. 26.2 into C++ header files. This code will represent the "skeleton" of the system. In Section 26.3, we modify the header files to incorporate the object-oriented concept of inheritance. In Section 26.4, we present the complete working C++ code for our model.

As an example, we begin to develop the header file for class `Withdrawal` from our design of class `Withdrawal` in Fig. 26.1. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 26.2 to determine the associations among classes. We follow the following five guidelines for each class:

1. Use the name in the first compartment of a class in a class diagram to define the class in a header file (Fig. 26.3). Use `#ifndef`, `#define` and `#endif` preprocessor directives to prevent the header from being included more than once in a program.

```
// Fig. 26.3: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Withdrawal
{
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Fig. 26.3 | Definition of class `Withdrawal` enclosed in preprocessor wrappers.

2. Use the attributes located in the class's second compartment to declare the data members. For example, the `private` attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 26.4.

```
// Fig. 26.4: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Withdrawal
{
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Fig. 26.4 | Adding attributes to the `Withdrawal` class header file.

3. Use the associations described in the class diagram to declare references (or pointers, where appropriate) to other objects. For example, according to Fig. 26.2, `Withdrawal` can access one object of class `Screen`, one object of class `Keypad`, one object of class `CashDispenser` and one object of class `BankDatabase`. Class `Withdrawal` must maintain handles on these objects to send messages to them, so lines 19–22 of Fig. 26.5 declare four references as `private` data members. In the implementation of class `Withdrawal` in Section 26.4, a constructor initializes these data members with references to actual objects. Lines 6–9 `#include` the header files containing the definitions of classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` so that we can declare references to objects of these classes in lines 19–22.

4. It turns out that including the header files for classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` in Fig. 26.5 does more than is necessary. Class `Withdrawal` contains *references* to objects of these classes—it does not contain actual objects—and the amount of information required by the compiler to create a reference differs from that which is required to create an object. Recall that creating an object requires that you provide the compiler with a definition of the class that

```
1  // Fig. 26.5: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Screen.h" // include definition of class Screen
7  #include "Keypad.h" // include definition of class Keypad
8  #include "CashDispenser.h" // include definition of class CashDispenser
9  #include "BankDatabase.h" // include definition of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM’s screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

Fig. 26.5 | Declaring references to objects associated with class `Withdrawal`.

introduces the name of the class as a new user-defined type and indicates the data members that determine how much memory is required to store the object. Declaring a *reference* (or pointer) to an object, however, requires only that the compiler knows that the object’s class exists—it does not need to know the size of the object. Any reference (or pointer), regardless of the class of the object to which it refers, contains only the memory address of the actual object. The amount of memory required to store an address is a physical characteristic of the computer’s hardware. The compiler thus knows the size of any reference (or pointer). As a result, including a class’s full header file when declaring only a reference to an object of that class is unnecessary—we need to introduce the name of the class, but we do not need to provide the data layout of the object, because the compiler already knows the size of all references. C++ provides a statement called a **forward declaration** that signifies that a header file contains references or pointers to a class, but that the class definition lies outside the header file. We can replace the `#include`s in the `Withdrawal` class definition of Fig. 26.5 with forward declarations of classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` (lines 6–9 in Fig. 26.6). Rather than `#include` the entire header file for each of these classes, we place only a forward declaration of each class in the header file for class `Withdrawal`. If class `Withdrawal` contained actual objects instead of references (i.e., if the ampersands in lines 19–22 were omitted), then we’d need to `#include` the full header files.

Using a forward declaration (where possible) instead of including a full header file helps avoid a preprocessor problem called a **circular include**. This problem occurs when the header file for a class `A` `#include`s the header file for a class `B` and vice versa. Some preprocessors are not be able to resolve such `#include` directives, causing a compilation error. If class `A`, for example, uses only a reference to an object of class `B`, then the `#include` in class `A`'s header file can be replaced by a forward declaration of class `B` to prevent the circular include.

```
// Fig. 26.6: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Screen; // forward declaration of class Screen
class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser
class BankDatabase; // forward declaration of class BankDatabase

class Withdrawal
{
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw

   // references to associated objects
   Screen &screen; // reference to ATM’s screen
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Fig. 26.6 | Using forward declarations in place of `#include` directives.

5. Use the operations located in the third compartment of Fig. 26.1 to write the function prototypes of the class's member functions. If we've not yet specified a return type for an operation, we declare the member function with return type `void`. Refer to the class diagrams of Figs. 6.22–6.25 to declare any necessary parameters. For example, adding the `public` operation `execute` in class `Withdrawal`, which has an empty parameter list, yields the prototype in line 15 of Fig. 26.7. [*Note:* We code the definitions of member functions in `.cpp` files when we implement the complete ATM system in Section 26.4.]

Software Engineering Observation 26.1

Several UML modeling tools can convert UML-based designs into C++ code, considerably speeding the implementation process. For more information on these "automatic" code generators, refer to our UML Resource Center at `www.deitel.com/UML/`.

This concludes our discussion of the basics of generating class header files from UML diagrams. In Section 26.3, we demonstrate how to modify the header files to incorporate the object-oriented concept of inheritance.

```
// Fig. 26.7: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

class Screen; // forward declaration of class Screen
class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser
class BankDatabase; // forward declaration of class BankDatabase

class Withdrawal
{
public:
   // operations
   void execute(); // perform the transaction
private:
   // attributes
   int accountNumber; // account to withdraw funds from
   double amount; // amount to withdraw

   // references to associated objects
   Screen &screen; // reference to ATM’s screen
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Fig. 26.7 | Adding operations to the `Withdrawal` class header file.

Self-Review Exercises for Section 26.2

26.1 State whether the following statement is *true* or *false*, and if *false*, explain why: If an attribute of a class is marked with a minus sign (-) in a class diagram, the attribute is not directly accessible outside of the class.

26.2 In Fig. 26.2, the association between the `ATM` and the `Screen` indicates that:

a) we can navigate from the `Screen` to the `ATM`
b) we can navigate from the `ATM` to the `Screen`
c) Both a and b; the association is bidirectional
d) None of the above

26.3 Write C++ code to begin implementing the design for class `Account`.

26.3 Incorporating Inheritance into the ATM System

[*Note:* **This section can be studied after Chapter 13.**]

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for *commonality* among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more efficient and elegant manner that enables us to process objects of these classes polymorphically. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into C++ header files.

In Section 25.4, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—`BalanceInquiry`, `Withdrawal` and `Deposit`—to represent the transactions that the ATM system can perform. Figure 26.8 shows the attributes and operations of these classes, which have one attribute (`accountNumber`) and one operation (`execute`) in common. Each class requires attribute `accountNumber` to specify the account to which the transaction applies. Each class contains operation `execute`, which the `ATM` invokes to perform the transaction. Clearly, `BalanceInquiry`, `Withdrawal` and `Deposit` represent *types of* transactions. Figure 26.8 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing these classes. We place the common functionality in base class `Transaction` and derive classes `BalanceInquiry`, `Withdrawal` and `Deposit` from `Transaction` (Fig. 26.9).

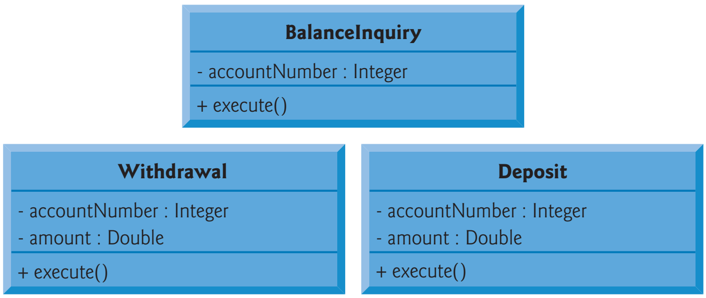


Fig. 26.8 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

The UML specifies a relationship called a **generalization** to model inheritance. Figure 26.9 is the class diagram that models the inheritance relationship between base class `Transaction` and its three derived classes. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` are derived from class

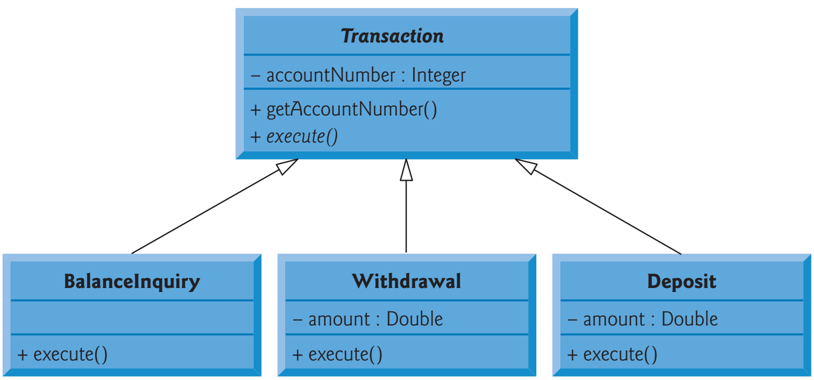


Fig. 26.9 | Class diagram modeling generalization relations\hip between base class `Transaction` and derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

`Transaction`. Class `Transaction` is said to be a generalization of its derived classes. The derived classes are said to be **specializations** of class `Transaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in base class `Transaction`. We no longer list `accountNumber` in the second compartment of each derived class, because the three derived classes inherit this attribute from `Transaction`. Recall, however, that derived classes cannot access `private` attributes of a base class. We therefore include `public` member function `getAccountNumber` in class `Transaction`. Each derived class inherits this member function, enabling the derived class to access its `accountNumber` as needed to execute a transaction.

According to Fig. 26.8, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so base class `Transaction` should contain `public` member function `execute`. However, it does not make sense to implement `execute` in class `Transaction`, because the functionality that this member function provides depends on the specific type of the actual transaction. We therefore declare member function `execute` as a pure `virtual` function in base class `Transaction`. This makes `Transaction` an *abstract class* and forces any class derived from `Transaction` that must be a *concrete class* (i.e., `BalanceInquiry`, `Withdrawal` and `Deposit`) to implement pure `virtual` member function `execute` to make the derived class concrete. The UML requires that we place abstract class names (and pure `virtual` functions—**abstract operations** in the UML) in italics, so `Transaction` and its member function `execute` appear in italics in Fig. 26.9. Operation `execute` is not italicized in derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Each derived class overrides base class `Transaction`'s `execute` member function with an appropriate implementation. Figure 26.9 includes operation `execute` in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden member function.

Processing `Transaction`*s Polymorphically*

A derived class can inherit interface and/or implementation from a base class. Compared to a hierarchy designed for implementation inheritance, one designed for interface inheritance tends to have its functionality lower in the hierarchy—a base class signifies one or more functions that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the function(s). The inheritance hierarchy designed for the ATM system takes advantage of this type of inheritance, which provides the `ATM` with an elegant way to execute all transactions "in the general." Each class derived from `Transaction` inherits some implementation details (e.g., data member `accountNumber`), but the primary benefit of incorporating inheritance into our system is that the derived classes share a common interface (e.g., pure `virtual` member function `execute`). The `ATM` can aim a `Transaction` pointer at any transaction, and when the `ATM` invokes `execute` through this pointer, the version of `execute` appropriate to that transaction (i.e., the version implemented in that derived class's `.cpp` file) runs automatically. For example, suppose a user chooses to perform a balance inquiry. The `ATM` aims a `Transaction` pointer at a new object of class `BalanceInquiry`; the compiler allows this because a `BalanceInquiry` *is a* `Transaction`. When the `ATM` uses this pointer to invoke `execute`, `BalanceInquiry`'s version of `execute` is called.

This polymorphic approach also makes the system easily *extensible*. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional `Transaction` derived class that overrides the `execute` member function with a version appropriate for the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type from the main menu and for the `ATM` to instantiate and execute objects of the new derived class. The `ATM` could execute transactions of the new type using the current code, because it executes all transactions identically.

As you learned earlier in the chapter, an abstract class like `Transaction` is one for which you never intend to instantiate objects. An abstract class simply declares common attributes and behaviors for its derived classes in an inheritance hierarchy. Class `Transaction` defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include pure `virtual` member function `execute` in class `Transaction` if `execute` lacks a concrete implementation. Conceptually, we include this member function because it's the defining behavior of all transactions—executing. Technically, we must include member function `execute` in base class `Transaction` so that the `ATM` (or any other class) can polymorphically invoke each derived class's overridden version of this function through a `Transaction` pointer or reference.

Additional Attribute of Classes `Withdrawal` *and* `Deposit`

Derived classes `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `accountNumber` from base class `Transaction`, but classes `Withdrawal` and `Deposit` contain the additional attribute `amount` that distinguishes them from class `BalanceInquiry`. Classes `Withdrawal` and `Deposit` require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class `BalanceInquiry` has no need for such an attribute and requires only an account number to execute. Even though two of the three `Transaction` derived classes share this attribute, we do not place it in base class `Transaction`—we place only features common to *all* the derived classes in the base class, so derived classes do not inherit unnecessary attributes (and operations).

Class Diagram with `Transaction` *Hierarchy Incorporated*

Figure 26.10 presents an updated class diagram of our model that incorporates inheritance and introduces class `Transaction`. We model an association between class `ATM` and class `Transaction` to show that the `ATM`, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `Transaction` exist in the system at a time). Because a `Withdrawal` is a type of `Transaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal`—derived class `Withdrawal` inherits base class `Transaction`'s association with class `ATM`. Derived classes `BalanceInquiry` and `Deposit` also inherit this association, which replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and class `ATM`. Note again the use of triangular hollow arrowheads to indicate the specializations of class `Transaction`, as indicated in Fig. 26.9.

We also add an association between class `Transaction` and the `BankDatabase` (Fig. 26.10). All `Transactions` require a reference to the `BankDatabase` so they can access and modify account information. Each `Transaction` derived class inherits this reference, so we no longer model the association between class `Withdrawal` and the `BankDatabase`. The association between class `Transaction` and the `BankDatabase` replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and the `BankDatabase`.

We include an association between class `Transaction` and the `Screen` because all `Transactions` display output to the user via the `Screen`. Each derived class inherits this

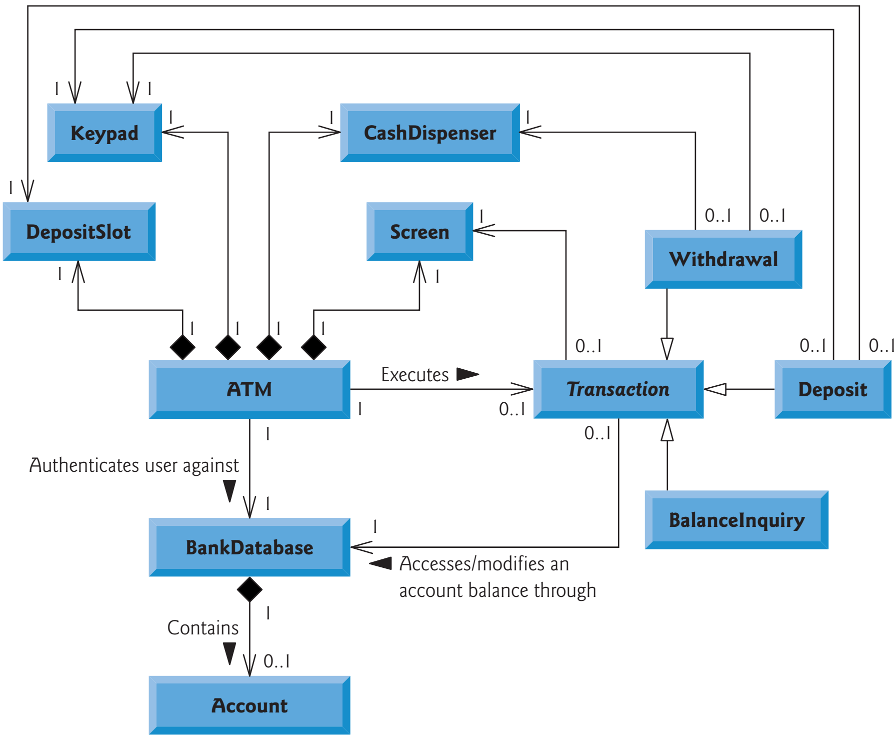


Fig. 26.10 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class name `Transaction` appears in italics.

association. Therefore, we no longer include the association previously modeled between `Withdrawal` and the `Screen`. Class `Withdrawal` still participates in associations with the `CashDispenser` and the `Keypad`. We do not move these associations to base class `Transaction`, because the association with the `Keypad` applies only to classes `Withdrawal` and `Deposit`, and the association with the `CashDispenser` applies only to class `Withdrawal`.

Our class diagram incorporating inheritance (Fig. 26.10) also models `Deposit` and `BalanceInquiry`. We show associations between `Deposit` and both the `DepositSlot` and the `Keypad`. `BalanceInquiry` takes part in no associations other than those inherited from class `Transaction`—a `BalanceInquiry` interacts only with the `BankDatabase` and the `Screen`.

Figure 26.1 showed attributes and operations with visibility markers. Now we present a modified class diagram in Fig. 26.11 that includes abstract base class `Transaction`. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 26.10), but instead shows the attributes and operations after we've employed inheritance in our system. Abstract class name `Transaction` and abstract operation name `execute` in class `Transaction` appear in *italics*. To save space, we do not include those attributes shown by associations in Fig. 26.10—we do, however, include them in the C++ implementation. We also omit all operation parameters, as we did in Fig. 26.1—incorporating inheritance does not affect the parameters already modeled in Figs. 25.18–25.21.

Fig. 26.11 | Class diagram after incorporating inheritance into the system.

Software Engineering Observation 26.2

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Fig. 26.10 and Fig. 26.11), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations. However, when examining classes modeled in this fashion, it's crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 26.10 to observe the inheritance relationship between `Transaction` *and its derived classes that is omitted from Fig. 26.11.*

Implementing the ATM System Design Incorporating Inheritance

We now modify our implementation to incorporate inheritance, using class `Withdrawal` as an example.

1. If a class `A` is a generalization of class `B`, then class `B` is derived from (and is a specialization of) class `A`. For example, abstract base class `Transaction` is a generalization of class `Withdrawal`. Thus, class `Withdrawal` is derived from (and is a specialization of) class `Transaction`. Figure 26.12 contains a portion of class `Withdrawal`'s header file, in which the class definition indicates the inheritance relationship between `Withdrawal` and `Transaction` (line 9).

```
1  // Fig. 26.12: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  // class Withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

Fig. 26.12 | `Withdrawal` class definition that derives from `Transaction`.

2. If class `A` is an abstract class and class `B` is derived from class `A`, then class `B` must implement the pure `virtual` functions of class `A` if class `B` is to be a concrete class. For example, class `Transaction` contains pure `virtual` function `execute`, so class `Withdrawal` must implement this member function if we want to instantiate a `Withdrawal` object. Figure 26.13 contains the C++ header file for class `Withdrawal` from Fig. 26.10 and Fig. 26.11. Class `Withdrawal` inherits data member `accountNumber` from base class `Transaction`, so `Withdrawal` does not declare this data member. Class `Withdrawal` also inherits references to the `Screen` and the `BankDatabase` from its base class `Transaction`, so we do not include these references in our code. Figure 26.11 specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 19 of Fig. 26.13 declares a data member for attribute `amount`. Line 16 contains the function prototype for operation `execute`. Recall that, to be a concrete class, derived class `Withdrawal` must provide a concrete implementation of the pure `virtual` function `execute` in base class `Transaction`. The prototype in line 16 signals your intent to override the base class pure `virtual` function. You must provide this prototype if you'll provide an implementation in the `.cpp` file. We present this implementation in Section 26.4. The `keypad` and `cashDispenser` references (lines 20–21) are data members derived from `Withdrawal`'s associations in Fig. 26.10. In the implementation of this class in Section 26.4, a constructor initializes these references to actual objects. Once again, to be able to compile the declarations of the references in lines 20–21, we include the forward declarations in lines 8–9.

ATM Case Study Wrap-Up

This concludes our object-oriented design of the ATM system. A complete C++ implementation of the ATM system in 850 lines of code appears in Section 26.4. This working

```
// Fig. 26.13: Withdrawal.h
// Definition of class Withdrawal that represents a withdrawal transaction
#ifndef WITHDRAWAL_H
#define WITHDRAWAL_H

#include "Transaction.h" // Transaction class definition

class Keypad; // forward declaration of class Keypad
class CashDispenser; // forward declaration of class CashDispenser

// class Withdrawal derives from base class Transaction
class Withdrawal : public Transaction
{
public:
   // member function overriding execute in base class Transaction
   virtual void execute(); // perform the transaction
private:
   // attributes
   double amount; // amount to withdraw
   Keypad &keypad; // reference to ATM's keypad
   CashDispenser &cashDispenser; // reference to ATM's cash dispenser
}; // end class Withdrawal

#endif // WITHDRAWAL_H
```

Fig. 26.13 | `Withdrawal` class header file based on Fig. 26.10 and Fig. 26.11.

implementation uses key programming notions, including classes, objects, encapsulation, visibility, composition, inheritance and polymorphism. The code is abundantly commented and conforms to the coding practices you've learned. Mastering this code is a wonderful capstone experience.

Self-Review Exercises for Section 26.3

26.4 The UML uses an arrow with a __________ to indicate a generalization relationship.
  a) solid filled arrowhead
  b) triangular hollow arrowhead
  c) diamond-shaped hollow arrowhead
  d) stick arrowhead

26.5 State whether the following statement is *true* or *false*, and if *false*, explain why: The UML requires that we underline abstract class names and operation names.

26.6 Write a C++ header file to begin implementing the design for class `Transaction` specified in Fig. 26.10 and Fig. 26.11. Be sure to include `private` references based on class `Transaction`'s associations. Also be sure to include `public` *get* functions for any of the `private` data members that the derived classes must access to perform their tasks.

26.4 ATM Case Study Implementation

This section contains the complete working implementation of the ATM system that we designed in Chapter 25 and this chapter. We consider the classes in the order in which we identified them in Section 25.4:

- ATM
- Screen
- Keypad
- CashDispenser
- DepositSlot
- Account
- BankDatabase
- Transaction
- BalanceInquiry
- Withdrawal
- Deposit

We apply the guidelines discussed in Sections 26.2 and 26.3 to code these classes based on how we modeled them in the UML class diagrams of Figs. 26.10 and 26.11. To develop the definitions of classes' member functions, we refer to the activity diagrams presented in Section 25.6 and the communication and sequence diagrams presented in Section 25.8. Note that our ATM design does not specify all the program logic and may not specify all the attributes and operations required to complete the ATM implementation. This is a normal part of the object-oriented design process. As we implement the system, we complete the program logic and add attributes and behaviors as necessary to construct the ATM system specified by the requirements specification in Section 25.3.

We conclude the discussion by presenting a C++ program (`ATMCaseStudy.cpp`) that starts the ATM and puts the other classes in the system in use. Recall that we're developing a first version of the ATM system that runs on a personal computer and uses the computer's keyboard and monitor to approximate the ATM's keypad and screen. We also only simulate the actions of the ATM's cash dispenser and deposit slot. We attempt to implement the system, however, so that real hardware versions of these devices could be integrated without significant changes in the code.

26.4.1 Class ATM

Class `ATM` (Figs. 26.14–26.15) represents the ATM as a whole. Figure 26.14 contains the `ATM` class definition, enclosed in `#ifndef`, `#define` and `#endif` preprocessor directives to ensure that this definition gets included only once in a program. We discuss lines 6–11 shortly. Lines 16–17 contain the function prototypes for the class's `public` member functions. The class diagram of Fig. 26.11 does not list any operations for class `ATM`, but we now declare a `public` member function `run` (line 17) in class `ATM` that allows an external client of the class (i.e., `ATMCaseStudy.cpp`) to tell the `ATM` to run. We also include a function prototype for a default constructor (line 16), which we discuss shortly.

```
1 // ATM.h
2 // ATM class definition. Represents an automated teller machine.
3 #ifndef ATM_H
```

Fig. 26.14 | Definition of class `ATM`, which represents the ATM. (Part 1 of 2.)

```
4  #define ATM_H
5
6  #include "Screen.h" // Screen class definition
7  #include "Keypad.h" // Keypad class definition
8  #include "CashDispenser.h" // CashDispenser class definition
9  #include "DepositSlot.h" // DepositSlot class definition
10 #include "BankDatabase.h" // BankDatabase class definition
11 class Transaction; // forward declaration of class Transaction
12
13 class ATM
14 {
15 public:
16    ATM(); // constructor initializes data members
17    void run(); // start the ATM
18 private:
19    bool userAuthenticated; // whether user is authenticated
20    int currentAccountNumber; // current user's account number
21    Screen screen; // ATM's screen
22    Keypad keypad; // ATM's keypad
23    CashDispenser cashDispenser; // ATM's cash dispenser
24    DepositSlot depositSlot; // ATM's deposit slot
25    BankDatabase bankDatabase; // account information database
26
27    // private utility functions
28    void authenticateUser(); // attempts to authenticate user
29    void performTransactions(); // performs transactions
30    int displayMainMenu() const; // displays main menu
31
32    // return object of specified Transaction derived class
33    Transaction *createTransaction( int );
34 }; // end class ATM
35
36 #endif // ATM_H
```

Fig. 26.14 | Definition of class `ATM`, which represents the ATM. (Part 2 of 2.)

Lines 19–25 of Fig. 26.14 implement the class's attributes as `private` data members. We determine all but one of these attributes from the class diagrams of Figs. 26.10– 26.11. We implement the UML `Boolean` attribute `userAuthenticated` in Fig. 26.11 as a `bool` data member in C++ (line 19). Line 20 declares a data member not found in our UML design—an `int` data member `currentAccountNumber` that keeps track of the account number of the current authenticated user. We'll soon see how the class uses this data member.

Lines 21–24 create objects to represent the parts of the ATM. Recall from the class diagram of Fig. 26.10 that class `ATM` has composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`, so class `ATM` is responsible for their creation. Line 25 creates a `BankDatabase`, with which the `ATM` interacts to access and manipulate bank account information. [*Note:* If this were a real ATM system, the `ATM` class would receive a reference to an existing database object created by the bank. However, in this implementation we are only simulating the bank's database, so class `ATM` creates the `BankDatabase` object with which it interacts.] Lines 6–10 `#include` the class definitions of

`Screen`, `Keypad`, `CashDispenser`, `DepositSlot` and `BankDatabase` so that the `ATM` can store objects of these classes.

Lines 28–30 and 33 contain function prototypes for `private` utility functions that the class uses to perform its tasks. We'll see how these functions serve the class shortly. Member function `createTransaction` (line 33) returns a `Transaction` pointer. To include the class name `Transaction` in this file, we must at least include a forward declaration of class `Transaction` (line 11). Recall that a forward declaration tells the compiler that a class exists, but that the class is defined elsewhere. A forward declaration is sufficient here, as we are using a `Transaction` pointer as a return type—if we were creating or returning an actual `Transaction` object, we would need to `#include` the full `Transaction` header file.

`ATM` Class Member-Function Definitions

Figure 26.15 contains the member-function definitions for class `ATM`. Lines 3–7 `#include` the header files required by the implementation file `ATM.cpp`. Including the `ATM` header file allows the compiler to ensure that the class's member functions are defined correctly. This also allows the member functions to use the class's data members.

```
1  // ATM.cpp
2  // Member-function definitions for class ATM.
3  #include "ATM.h" // ATM class definition
4  #include "Transaction.h" // Transaction class definition
5  #include "BalanceInquiry.h" // BalanceInquiry class definition
6  #include "Withdrawal.h" // Withdrawal class definition
7  #include "Deposit.h" // Deposit class definition
8
9  // enumeration constants represent main menu options
10 enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT };
11
12 // ATM default constructor initializes data members
13 ATM::ATM()
14    : userAuthenticated ( false ), // user is not authenticated to start
15      currentAccountNumber( 0 ) // no current account number to start
16 {
17    // empty body
18 } // end ATM default constructor
19
20 // start ATM
21 void ATM::run()
22 {
23    // welcome and authenticate user; perform transactions
24    while ( true )
25    {
26       // loop while user is not yet authenticated
27       while ( !userAuthenticated )
28       {
29          screen.displayMessageLine( "\nWelcome!" );
30          authenticateUser(); // authenticate user
31       } // end while
32
```

Fig. 26.15 | ATM class member-function definitions. (Part 1 of 4.)

```
      performTransactions(); // user is now authenticated
      userAuthenticated = false; // reset before next ATM session
      currentAccountNumber = 0; // reset before next ATM session
      screen.displayMessageLine( "\nThank you! Goodbye!" );
   } // end while
} // end function run

// attempt to authenticate user against database
void ATM::authenticateUser()
{
   screen.displayMessage( "\nPlease enter your account number: " );
   int accountNumber = keypad.getInput(); // input account number
   screen.displayMessage( "\nEnter your PIN: " ); // prompt for PIN
   int pin = keypad.getInput(); // input PIN

   // set userAuthenticated to bool value returned by database
   userAuthenticated =
      bankDatabase.authenticateUser( accountNumber, pin );

   // check whether authentication succeeded
   if ( userAuthenticated )
   {
      currentAccountNumber = accountNumber; // save user's account #
   } // end if
   else
      screen.displayMessageLine(
         "Invalid account number or PIN. Please try again." );
} // end function authenticateUser

// display the main menu and perform transactions
void ATM::performTransactions()
{
   // local pointer to store transaction currently being processed
   Transaction *currentTransactionPtr;

   bool userExited = false; // user has not chosen to exit

   // loop while user has not chosen option to exit system
   while ( !userExited )
   {
      // show main menu and get user selection
      int mainMenuSelection = displayMainMenu();

      // decide how to proceed based on user's menu selection
      switch ( mainMenuSelection )
      {
         // user chose to perform one of three transaction types
         case BALANCE_INQUIRY:
         case WITHDRAWAL:
         case DEPOSIT:
            // initialize as new object of chosen type
            currentTransactionPtr =
               createTransaction( mainMenuSelection );
```

Fig. 26.15 | ATM class member-function definitions. (Part 2 of 4.)

```

            currentTransactionPtr->execute(); // execute transaction

            // free the space for the dynamically allocated Transaction
            delete currentTransactionPtr;

            break;
         case EXIT: // user chose to terminate session
            screen.displayMessageLine( "\nExiting the system..." );
            userExited = true; // this ATM session should end
            break;
         default: // user did not enter an integer from 1-4
            screen.displayMessageLine(
               "\nYou did not enter a valid selection. Try again." );
            break;
      } // end switch
   } // end while
} // end function performTransactions

// display the main menu and return an input selection
int ATM::displayMainMenu() const
{
   screen.displayMessageLine( "\nMain menu:" );
   screen.displayMessageLine( "1 - View my balance" );
   screen.displayMessageLine( "2 - Withdraw cash" );
   screen.displayMessageLine( "3 - Deposit funds" );
   screen.displayMessageLine( "4 - Exit\n" );
   screen.displayMessage( "Enter a choice: " );
   return keypad.getInput(); // return user's selection
} // end function displayMainMenu

// return object of specified Transaction derived class
Transaction *ATM::createTransaction( int type )
{
   Transaction *tempPtr; // temporary Transaction pointer

   // determine which type of Transaction to create
   switch ( type )
   {
      case BALANCE_INQUIRY: // create new BalanceInquiry transaction
         tempPtr = new BalanceInquiry(
            currentAccountNumber, screen, bankDatabase );
         break;
      case WITHDRAWAL: // create new Withdrawal transaction
         tempPtr = new Withdrawal( currentAccountNumber, screen,
            bankDatabase, keypad, cashDispenser );
         break;
      case DEPOSIT: // create new Deposit transaction
         tempPtr = new Deposit( currentAccountNumber, screen,
            bankDatabase, keypad, depositSlot );
         break;
   } // end switch

```

Fig. 26.15 | ATM class member-function definitions. (Part 3 of 4.)

```
139    return tempPtr; // return the newly created object
140 } // end function createTransaction
```

Fig. 26.15 | ATM class member-function definitions. (Part 4 of 4.)

Line 10 declares an `enum` named `MenuOption` that contains constants corresponding to the four options in the ATM's main menu (i.e., balance inquiry, withdrawal, deposit and exit). Note that setting `BALANCE_INQUIRY` to `1` causes the subsequent enumeration constants to be assigned the values `2`, `3` and `4`, as enumeration constant values increment by `1`.

Lines 13–18 define class `ATM`'s constructor, which initializes the class's data members. When an `ATM` object is first created, no user is authenticated, so line 14 uses a member initializer to set `userAuthenticated` to `false`. Likewise, line 15 initializes `currentAccountNumber` to `0` because there is no current user yet.

ATM Member Function `run`

`ATM` member function `run` (lines 21–38) uses an infinite loop (lines 24–37) to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an authenticated user performs the desired transactions and chooses to exit, the ATM resets itself, displays a goodbye message to the user and restarts the process. We use an infinite loop here to simulate the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control). An ATM user has the option to exit the system, but does not have the ability to turn off the ATM completely.

Authenticating a User

Inside member function `run`'s infinite loop, lines 27–31 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., `!userAuthenticated` is `true`). Line 29 invokes member function `displayMessageLine` of the `ATM`'s `screen` to display a welcome message. Like `Screen` member function `displayMessage` designed in the case study, member function `displayMessageLine` (declared in line 13 of Fig. 26.16 and defined in lines 20–23 of Fig. 26.17) displays a message to the user, but this member function also outputs a newline after displaying the message. We've added this member function during implementation to give class `Screen`'s clients more control over the placement of displayed messages. Line 30 of Fig. 26.15 invokes class `ATM`'s `private` utility function `authenticateUser` (lines 41–60) to attempt to authenticate the user.

We refer to the requirements specification to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 43 of member function `authenticateUser` invokes member function `displayMessage` of the `ATM`'s `screen` to prompt the user to enter an account number. Line 44 invokes member function `getInput` of the `ATM`'s `keypad` to obtain the user's input, then stores the integer value entered by the user in a local variable `accountNumber`. Member function `authenticateUser` next prompts the user to enter a PIN (line 45), and stores the PIN input by the user in a local variable `pin` (line 46). Next, lines 49–50 attempt to authenticate the user by passing the `accountNumber` and `pin` entered by the user to the `bankDatabase`'s `authenticateUser` member function. Class `ATM` sets its `userAuthenticated` data member to the `bool` value returned by this function—`userAuthenticated` becomes `true` if authentication succeeds (i.e., `accountNumber` and `pin` match those of an existing `Account` in `bankDatabase`) and remains `false` otherwise.

If userAuthenticated is true, line 55 saves the account number entered by the user (i.e., accountNumber) in the ATM data member currentAccountNumber. The other member functions of class ATM use this variable whenever an ATM session requires access to the user's account number. If userAuthenticated is false, lines 58–59 use the screen's displayMessageLine member function to indicate that an invalid account number and/or PIN was entered and the user must try again. Note that we set currentAccountNumber only after authenticating the user's account number and the associated PIN—if the database could not authenticate the user, currentAccountNumber remains 0.

After member function run attempts to authenticate the user (line 30), if userAuthenticated is still false, the while loop in lines 27–31 executes again. If userAuthenticated is now true, the loop terminates and control continues with line 33, which calls class ATM's utility function performTransactions.

Performing Transactions

Member function performTransactions (lines 63–103) carries out an ATM session for an authenticated user. Line 66 declares a local Transaction pointer, which we aim at a BalanceInquiry, Withdrawal or Deposit object representing the ATM transaction currently being processed. We use a Transaction pointer here to allow us to take advantage of polymorphism. Also, we use the role name included in the class diagram of Fig. 25.7—currentTransaction—in naming this pointer. As per our pointer-naming convention, we append "Ptr" to the role name to form the variable name currentTransactionPtr. Line 68 declares another local variable—a bool called userExited that keeps track of whether the user has chosen to exit. This variable controls a while loop (lines 71–102) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 74 displays the main menu and obtains the user's menu selection by calling an ATM utility function displayMainMenu (defined in lines 106–115). This member function displays the main menu by invoking member functions of the ATM's screen and returns a menu selection obtained from the user through the ATM's keypad. Note that this member function is const because it does not modify the contents of the object. Line 74 stores the user's selection returned by displayMainMenu in local variable mainMenuSelection.

After obtaining a main menu selection, member function performTransactions uses a switch statement (lines 77–101) to respond to the selection appropriately. If mainMenuSelection is equal to any of the three enumeration constants representing transaction types (i.e., if the user chose to perform a transaction), lines 84–85 call utility function createTransaction (defined in lines 118–140) to return a pointer to a newly instantiated object of the type that corresponds to the selected transaction. Pointer currentTransactionPtr is assigned the pointer returned by createTransaction. Line 87 then uses currentTransactionPtr to invoke the new object's execute member function to execute the transaction. We'll discuss Transaction member function execute and the three Transaction derived classes shortly. Finally, when the Transaction derived class object is no longer needed, line 90 releases the memory dynamically allocated for it.

We aim the Transaction pointer currentTransactionPtr at an object of one of the three Transaction derived classes so that we can execute transactions *polymorphically*. For example, if the user chooses to perform a balance inquiry, mainMenuSelection equals BALANCE_INQUIRY, leading createTransaction to return a pointer to a BalanceInquiry object. Thus, currentTransactionPtr points to a BalanceInquiry, and invoking currentTransactionPtr->execute() results in BalanceInquiry's version of execute being called.

Creating a Transaction

Member function `createTransaction` (lines 118–140) uses a `switch` statement (lines 123–137) to instantiate a new `Transaction` derived class object of the type indicated by the parameter `type`. Recall that member function `performTransactions` passes `mainMenuSelection` to this member function only when `mainMenuSelection` contains a value corresponding to one of the three transaction types. Therefore `type` equals either `BALANCE_INQUIRY`, `WITHDRAWAL` or `DEPOSIT`. Each `case` in the `switch` statement aims the temporary pointer `tempPtr` at a newly created object of the appropriate `Transaction` derived class. Each constructor has a unique parameter list, based on the specific data required to initialize the derived class object. A `BalanceInquiry` requires only the account number of the current user and references to the `ATM`'s `screen` and the `bankDatabase`. In addition to these parameters, a `Withdrawal` requires references to the `ATM`'s `keypad` and `cashDispenser`, and a `Deposit` requires references to the `ATM`'s `keypad` and `depositSlot`. As you'll soon see, the `BalanceInquiry`, `Withdrawal` and `Deposit` constructors each specify reference parameters to receive the objects representing the required parts of the ATM. Thus, when member function `createTransaction` passes objects in the `ATM` (e.g., `screen` and `keypad`) to the initializer for each newly created `Transaction` derived class object, the new object actually receives *references* to the `ATM`'s composite objects. We discuss the transaction classes in more detail in Sections 26.4.8–26.4.11.

Exiting the Main Menu and Processing Invalid Selections

After executing a transaction (line 87 in `performTransactions`), `userExited` remains `false` and the `while` loop in lines 71–102 repeats, returning the user to the main menu. However, if a user does not perform a transaction and instead selects the main menu option to exit, line 95 sets `userExited` to `true`, causing the condition of the `while` loop (`!userExited`) to become `false`. This `while` is the final statement of member function `performTransactions`, so control returns to the calling function `run`. If the user enters an invalid main menu selection (i.e., not an integer from 1–4), lines 98–99 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

Awaiting the Next ATM User

When `performTransactions` returns control to member function `run`, the user has chosen to exit the system, so lines 34–35 reset the `ATM`'s data members `userAuthenticated` and `currentAccountNumber` to prepare for the next ATM user. Line 36 displays a goodbye message before the ATM starts over and welcomes the next user.

26.4.2 Class Screen

Class `Screen` (Figs. 26.16–26.17) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class `Screen` approximates a real ATM's screen with a computer monitor and outputs text messages using `cout` and the stream insertion operator (<<). In this case study, we designed class `Screen` to have one operation—`displayMessage`. For greater flexibility in displaying messages to the `Screen`, we now declare three `Screen` member functions—`displayMessage`, `displayMessageLine` and `displayDollarAmount`. The prototypes for these member functions appear in lines 12–14 of Fig. 26.16.

```cpp
// Screen.h
// Screen class definition. Represents the screen of the ATM.
#ifndef SCREEN_H
#define SCREEN_H

#include <string>
using namespace std;

class Screen
{
public:
   void displayMessage( string ) const; // output a message
   void displayMessageLine( string ) const; // output message with newline
   void displayDollarAmount( double ) const; // output a dollar amount
}; // end class Screen

#endif // SCREEN_H
```

Fig. 26.16 | Screen class definition.

```cpp
1  // Screen.cpp
2  // Member-function definitions for class Screen.
3  #include <iostream>
4  #include <iomanip>
5  #include "Screen.h" // Screen class definition
6  using namespace std;
7  
8  // output a message without a newline
9  void Screen::displayMessage( string message ) const
10 {
11    cout << message;
12 } // end function displayMessage
13 
14 // output a message with a newline
15 void Screen::displayMessageLine( string message ) const
16 {
17    cout << message << endl;
18 } // end function displayMessageLine
19 
20 // output a dollar amount
21 void Screen::displayDollarAmount( double amount ) const
22 {
23    cout << fixed << setprecision( 2 ) << "$" << amount;
24 } // end function displayDollarAmount
```

Fig. 26.17 | Screen class member-function definitions.

`Screen` Class Member-Function Definitions

Figure 26.17 contains the member-function definitions for class `Screen`. Line 5 `#includes` the `Screen` class definition. Member function `displayMessage` (lines 9–12) takes a `string` as an argument and prints it to the console using `cout` and the stream insertion operator (`<<`). The cursor stays on the same line, making this member function appropriate for displaying prompts to the user. Member function `displayMessageLine` (lines 15–18) also prints a `string`, but outputs a newline to move the cursor to the next line. Finally, member function `displayDollarAmount` (lines 21–24) outputs a properly formatted dollar amount (e.g., `$123.45`). Line 23 uses stream manipulators `fixed` and `setprecision` to output a value formatted with two decimal places.

26.4.3 Class Keypad

Class `Keypad` (Figs. 26.18–26.19) represents the keypad of the ATM and is responsible for receiving all user input. Recall that we are simulating this hardware, so we use the computer's keyboard to approximate the keypad. A computer keyboard contains many keys not found on the ATM's keypad. However, we assume that the user presses only the keys on the computer keyboard that also appear on the keypad—the keys numbered 0–9 and the *Enter* key. Line 9 of Fig. 26.18 contains the function prototype for class `Keypad`'s one member function `getInput`. This member function is declared `const` because it does not change the object.

```
1   // Keypad.h
2   // Keypad class definition. Represents the keypad of the ATM.
3   #ifndef KEYPAD_H
4   #define KEYPAD_H
5
6   class Keypad
7   {
8   public:
9      int getInput() const; // return an integer value entered by user
10  }; // end class Keypad
11
12  #endif // KEYPAD_H
```

Fig. 26.18 | Keypad class definition.

***Keypad* Class Member-Function Definition**

In the `Keypad` implementation file (Fig. 26.19), member function `getInput` (defined in lines 9–14) uses the standard input stream `cin` and the stream extraction operator (>>) to obtain input from the user. Line 11 declares a local variable to store the user's input. Line 12 reads input into local variable `input`, then line 13 returns this value. Recall that `getInput` obtains all the input used by the ATM. `Keypad`'s `getInput` member function simply returns the integer input by the user. If a client of class `Keypad` requires input that satisfies some particular criteria (i.e., a number corresponding to a valid menu option), the client must perform the appropriate error checking. [*Note:* Using the standard input stream `cin` and the stream extraction operator (>>) allows noninteger input to be read from the user. Because the real ATM's keypad permits only integer input, however, we assume that the user enters an integer and do not attempt to fix problems caused by noninteger input.]

```
1   // Keypad.cpp
2   // Member-function definition for class Keypad (the ATM's keypad).
3   #include <iostream>
```

Fig. 26.19 | Keypad class member-function definition. (Part 1 of 2.)

```
using namespace std;

#include "Keypad.h" // Keypad class definition

// return an integer value entered by user
int Keypad::getInput() const
{
   int input; // variable to store the input
   cin >> input; // we assume that user enters an integer
   return input; // return the value entered by user
} // end function getInput
```

Fig. 26.19 | Keypad class member-function definition. (Part 2 of 2.)

26.4.4 Class CashDispenser

Class `CashDispenser` (Figs. 26.20–26.21) represents the cash dispenser. Figure 26.20 contains the function prototype for a default constructor (line 9). Class `CashDispenser` declares two additional `public` member functions—`dispenseCash` (line 12) and `isSufficientCashAvailable` (line 15). The class trusts that a client (i.e., `Withdrawal`) calls `dispenseCash` only after establishing that sufficient cash is available by calling `isSufficientCashAvailable`. Thus, `dispenseCash` simply simulates dispensing the requested amount without checking whether sufficient cash is available. Line 17 declares `private` constant `INITIAL_COUNT`, which indicates the initial count of bills in the cash dispenser when the ATM starts (i.e., `500`). Line 18 implements attribute `count` (modeled in Fig. 26.11), which keeps track of the number of bills remaining in the `CashDispenser` at any time.

```
1  // CashDispenser.h
2  // CashDispenser class definition. Represents the ATM's cash dispenser.
3  #ifndef CASH_DISPENSER_H
4  #define CASH_DISPENSER_H
5
6  class CashDispenser
7  {
8  public:
9     CashDispenser(); // constructor initializes bill count to 500
10
11    // simulates dispensing of specified amount of cash
12    void dispenseCash( int );
13
14    // indicates whether cash dispenser can dispense desired amount
15    bool isSufficientCashAvailable( int ) const;
16 private:
17    static const int INITIAL_COUNT = 500;
18    int count; // number of $20 bills remaining
19 }; // end class CashDispenser
20
21 #endif // CASH_DISPENSER_H
```

Fig. 26.20 | `CashDispenser` class definition.

CashDispenser *Class Member-Function Definitions*

Figure 26.21 contains the definitions of class CashDispenser's member functions. The constructor (lines 6–9) sets count to the initial count (i.e., 500). Member function dispenseCash (lines 13–17) simulates cash dispensing. If our system were hooked up to a real hardware cash dispenser, this member function would interact with the hardware device to physically dispense cash. Our simulated version of the member function simply decreases the count of bills remaining by the number required to dispense the specified amount (line 16). Line 15 calculates the number of $20 bills required to dispense the specified amount. The ATM allows the user to choose only withdrawal amounts that are multiples of $20, so we divide amount by 20 to obtain the number of billsRequired. Also, it's the responsibility of the class's client (i.e., Withdrawal) to inform the user that cash has been dispensed—CashDispenser cannot interact directly with Screen.

```
1  // CashDispenser.cpp
2  // Member-function definitions for class CashDispenser.
3  #include "CashDispenser.h" // CashDispenser class definition
4
5  // CashDispenser default constructor initializes count to default
6  CashDispenser::CashDispenser()
7  {
8     count = INITIAL_COUNT; // set count attribute to default
9  } // end CashDispenser default constructor
10
11 // simulates dispensing of specified amount of cash; assumes enough cash
12 // is available (previous call to isSufficientCashAvailable returned true)
13 void CashDispenser::dispenseCash( int amount )
14 {
15    int billsRequired = amount / 20; // number of $20 bills required
16    count -= billsRequired; // update the count of bills
17 } // end function dispenseCash
18
19 // indicates whether cash dispenser can dispense desired amount
20 bool CashDispenser::isSufficientCashAvailable( int amount ) const
21 {
22    int billsRequired = amount / 20; // number of $20 bills required
23
24    if ( count >= billsRequired )
25       return true; // enough bills are available
26    else
27       return false; // not enough bills are available
28 } // end function isSufficientCashAvailable
```

Fig. 26.21 | CashDispenser class member-function definitions.

Member function isSufficientCashAvailable (lines 20–28) has a parameter amount that specifies the amount of cash in question. Lines 24–27 return true if the CashDispenser's count is greater than or equal to billsRequired (i.e., enough bills are available) and false otherwise (i.e., not enough bills). For example, if a user wishes to withdraw $80 (i.e., billsRequired is 4), but only three bills remain (i.e., count is 3), the member function returns false.

26.4.5 Class DepositSlot

Class `DepositSlot` (Figs. 26.22–26.23) represents the deposit slot of the ATM. Like the version of class `CashDispenser` presented here, this version of class `DepositSlot` merely simulates the functionality of a real hardware deposit slot. `DepositSlot` has no data members and only one member function—`isEnvelopeReceived` (declared in line 9 of Fig. 26.22 and defined in lines 7–10 of Fig. 26.23)—that indicates whether a deposit envelope was received.

```
1  // DepositSlot.h
2  // DepositSlot class definition. Represents the ATM's deposit slot.
3  #ifndef DEPOSIT_SLOT_H
4  #define DEPOSIT_SLOT_H
5
6  class DepositSlot
7  {
8  public:
9     bool isEnvelopeReceived() const; // tells whether envelope was received
10 }; // end class DepositSlot
11
12 #endif // DEPOSIT_SLOT_H
```

Fig. 26.22 | `DepositSlot` class definition.

```
1  // DepositSlot.cpp
2  // Member-function definition for class DepositSlot.
3  #include "DepositSlot.h" // DepositSlot class definiton
4
5  // indicates whether envelope was received (always returns true,
6  // because this is only a software simulation of a real deposit slot)
7  bool DepositSlot::isEnvelopeReceived() const
8  {
9     return true; // deposit envelope was received
10 } // end function isEnvelopeReceived
```

Fig. 26.23 | `DepositSlot` class member-function definition.

Recall from the requirements specification that the ATM allows the user up to two minutes to insert an envelope. The current version of member function `isEnvelopeReceived` simply returns `true` immediately (line 9 of Fig. 26.23), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, member function `isEnvelopeReceived` might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If `isEnvelopeReceived` were to receive such a signal within two minutes, the member function would return `true`. If two minutes elapsed and the member function still had not received a signal, then the member function would return `false`.

26.4.6 Class Account

Class `Account` (Figs. 26.24–26.25) represents a bank account. Lines 9–15 in the class definition (Fig. 26.24) contain function prototypes for the class's constructor and six member functions, which we discuss shortly. Each `Account` has four attributes (modeled in Fig. 26.11)—`accountNumber`, `pin`, `availableBalance` and `totalBalance`. Lines 17–20 implement these attributes as `private` data members. Data member `availableBalance` represents the amount of funds available for withdrawal. Data member `totalBalance` represents the amount of funds available, plus the amount of deposited funds still pending confirmation or clearance.

```
1  // Account.h
2  // Account class definition. Represents a bank account.
3  #ifndef ACCOUNT_H
4  #define ACCOUNT_H
5
6  class Account
7  {
8  public:
9     Account( int, int, double, double ); // constructor sets attributes
10    bool validatePIN( int ) const; // is user-specified PIN correct?
11    double getAvailableBalance() const; // returns available balance
12    double getTotalBalance() const; // returns total balance
13    void credit( double ); // adds an amount to the Account balance
14    void debit( double ); // subtracts an amount from the Account balance
15    int getAccountNumber() const; // returns account number
16 private:
17    int accountNumber; // account number
18    int pin; // PIN for authentication
19    double availableBalance; // funds available for withdrawal
20    double totalBalance; // funds available + funds waiting to clear
21 }; // end class Account
22
23 #endif // ACCOUNT_H
```

Fig. 26.24 | Account class definition.

`Account` Class Member-Function Definitions

Figure 26.25 presents the definitions of class `Account`'s member functions. The class's constructor (lines 6–14) takes an account number, the PIN established for the account, the initial available balance and the initial total balance as arguments. Lines 8–11 assign these values to the class's data members using member initializers.

Member function `validatePIN` (lines 17–23) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., data member `pin`). Recall that we modeled this member function's parameter `userPIN` in the UML class diagram of Fig. 25.19. If the two PINs match, the member function returns `true` (line 20); otherwise, it returns `false` (line 22).

Member functions `getAvailableBalance` (lines 26–29) and `getTotalBalance` (lines 32–35) are *get* functions that return the values of `double` data members `availableBalance` and `totalBalance`, respectively.

Member function `credit` (lines 38–41) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. Note that this member function adds the `amount` only to data member `totalBalance` (line 40). The money credited to an account during a deposit does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class `Account` includes only member functions required for carrying out ATM transactions. Therefore, we omit the member functions that some other bank system would invoke to add to data member `availableBalance` (to confirm a deposit) or subtract from data member `totalBalance` (to reject a deposit).

```
1  // Account.cpp
2  // Member-function definitions for class Account.
3  #include "Account.h" // Account class definition
4
5  // Account constructor initializes attributes
6  Account::Account( int theAccountNumber, int thePIN,
7     double theAvailableBalance, double theTotalBalance )
8     : accountNumber( theAccountNumber ),
9       pin( thePIN ),
10      availableBalance( theAvailableBalance ),
11      totalBalance( theTotalBalance )
12 {
13    // empty body
14 } // end Account constructor
15
16 // determines whether a user-specified PIN matches PIN in Account
17 bool Account::validatePIN( int userPIN ) const
18 {
19    if ( userPIN == pin )
20       return true;
21    else
22       return false;
23 } // end function validatePIN
24
25 // returns available balance
26 double Account::getAvailableBalance() const
27 {
28    return availableBalance;
29 } // end function getAvailableBalance
30
31 // returns the total balance
32 double Account::getTotalBalance() const
33 {
34    return totalBalance;
35 } // end function getTotalBalance
36
37 // credits an amount to the account
38 void Account::credit( double amount )
39 {
40    totalBalance += amount; // add to total balance
41 } // end function credit
```

Fig. 26.25 | `Account` class member-function definitions. (Part 1 of 2.)

```
42
43 // debits an amount from the account
44 void Account::debit( double amount )
45 {
46    availableBalance -= amount; // subtract from available balance
47    totalBalance -= amount; // subtract from total balance
48 } // end function debit
49
50 // returns account number
51 int Account::getAccountNumber() const
52 {
53    return accountNumber;
54 } // end function getAccountNumber
```

Fig. 26.25 | `Account` class member-function definitions. (Part 2 of 2.)

Member function `debit` (lines 44–48) subtracts an amount of money (i.e., parameter `amount`) from an `Account` as part of a withdrawal transaction. This member function subtracts the `amount` from both data member `availableBalance` (line 46) and data member `totalBalance` (line 47), because a withdrawal affects both measures of an account balance.

Member function `getAccountNumber` (lines 51–54) provides access to an `Account`'s `accountNumber`. We include this member function in our implementation so that a client of the class (i.e., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can invoke this member function on each of its `Account` objects to locate the one with a specific account number.

26.4.7 Class BankDatabase

Class `BankDatabase` (Figs. 26.26–26.27) models the bank's database with which the ATM interacts to access and modify a user's account information. The class definition (Fig. 26.26) declares function prototypes for the class's constructor and several member functions. We discuss these momentarily. The class definition also declares the `BankDatabase`'s data members. We determine one data member for class `BankDatabase` based on its composition relationship with class `Account`. Recall from Fig. 26.10 that a `BankDatabase` is composed of zero or more objects of class `Account`. Line 24 of Fig. 26.26 implements data member `accounts`—a `vector` of `Account` objects—to implement this composition relationship. Lines 6–7 allow us to use `vector` in this file. Line 27 contains the function prototype for a `private` utility function `getAccount` that allows the member functions of the class to obtain a pointer to a specific `Account` in the `accounts` `vector`.

```
1 // BankDatabase.h
2 // BankDatabase class definition. Represents the bank's database.
3 #ifndef BANK_DATABASE_H
4 #define BANK_DATABASE_H
5
6 #include <vector> // class uses vector to store Account objects
7 using namespace std;
```

Fig. 26.26 | `BankDatabase` class definition. (Part 1 of 2.)

```

#include "Account.h" // Account class definition

class BankDatabase
{
public:
   BankDatabase(); // constructor initializes accounts

   // determine whether account number and PIN match those of an Account
   bool authenticateUser( int, int ); // returns true if Account authentic

   double getAvailableBalance( int ); // get an available balance
   double getTotalBalance( int ); // get an Account's total balance
   void credit( int, double ); // add amount to Account balance
   void debit( int, double ); // subtract amount from Account balance
private:
   vector< Account > accounts; // vector of the bank's Accounts

   // private utility function
   Account * getAccount( int ); // get pointer to Account object
}; // end class BankDatabase

#endif // BANK_DATABASE_H
```

Fig. 26.26 | BankDatabase class definition. (Part 2 of 2.)

BankDatabase *Class Member-Function Definitions*

Figure 26.27 contains the member-function definitions for class `BankDatabase`. We implement the class with a default constructor (lines 6–15) that adds `Account` objects to data member `accounts`. For the sake of testing the system, we create two new `Account` objects with test data (lines 9–10), then add them to the end of the `vector` (lines 13–14). The `Account` constructor has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance.

```
1  // BankDatabase.cpp
2  // Member-function definitions for class BankDatabase.
3  #include "BankDatabase.h" // BankDatabase class definition
4
5  // BankDatabase default constructor initializes accounts
6  BankDatabase::BankDatabase()
7  {
8     // create two Account objects for testing
9     Account account1( 12345, 54321, 1000.0, 1200.0 );
10    Account account2( 98765, 56789, 200.0, 200.0 );
11
12    // add the Account objects to the vector accounts
13    accounts.push_back( account1 ); // add account1 to end of vector
14    accounts.push_back( account2 ); // add account2 to end of vector
15 } // end BankDatabase default constructor
16
```

Fig. 26.27 | BankDatabase class member-function definitions. (Part 1 of 3.)

```
// retrieve Account object containing specified account number
Account * BankDatabase::getAccount( int accountNumber )
{
   // loop through accounts searching for matching account number
   for ( size_t i = 0; i < accounts.size(); i++ )
   {
      // return current account if match found
      if ( accounts[ i ].getAccountNumber() == accountNumber )
         return &accounts[ i ];
   } // end for

   return NULL; // if no matching account was found, return NULL
} // end function getAccount

// determine whether user-specified account number and PIN match
// those of an account in the database
bool BankDatabase::authenticateUser( int userAccountNumber,
   int userPIN )
{
   // attempt to retrieve the account with the account number
   Account * const userAccountPtr = getAccount( userAccountNumber );

   // if account exists, return result of Account function validatePIN
   if ( userAccountPtr != NULL )
      return userAccountPtr->validatePIN( userPIN );
   else
      return false; // account number not found, so return false
} // end function authenticateUser

// return available balance of Account with specified account number
double BankDatabase::getAvailableBalance( int userAccountNumber )
{
   Account * const userAccountPtr = getAccount( userAccountNumber );
   return userAccountPtr->getAvailableBalance();
} // end function getAvailableBalance

// return total balance of Account with specified account number
double BankDatabase::getTotalBalance( int userAccountNumber )
{
   Account * const userAccountPtr = getAccount( userAccountNumber );
   return userAccountPtr->getTotalBalance();
} // end function getTotalBalance

// credit an amount to Account with specified account number
void BankDatabase::credit( int userAccountNumber, double amount )
{
   Account * const userAccountPtr = getAccount( userAccountNumber );
   userAccountPtr->credit( amount );
} // end function credit

// debit an amount from Account with specified account number
void BankDatabase::debit( int userAccountNumber, double amount )
{
```

Fig. 26.27 | BankDatabase class member-function definitions. (Part 2 of 3.)

```
70       Account * const userAccountPtr = getAccount( userAccountNumber );
71       userAccountPtr->debit( amount );
72    } // end function debit
```

Fig. 26.27 | `BankDatabase` class member-function definitions. (Part 3 of 3.)

Recall that class `BankDatabase` serves as an intermediary between class `ATM` and the actual `Account` objects that contain users' account information. Thus, the member functions of class `BankDatabase` do nothing more than invoke the corresponding member functions of the `Account` object belonging to the current ATM user.

We include *`private` utility function* `getAccount` (lines 18–29) to allow the `BankDatabase` to obtain a pointer to a particular `Account` within `vector accounts`. To locate the user's `Account`, the `BankDatabase` compares the value returned by member function `getAccountNumber` for each element of `accounts` to a specified account number until it finds a match. Lines 21–26 traverse the `accounts vector`. If the account number of the current `Account` (i.e., `accounts[ i ]`) equals the value of parameter `accountNumber`, the member function immediately returns the address of the current `Account` (i.e., a pointer to the current `Account`). If no account has the given account number, then line 28 returns `NULL`. Note that this member function must return a pointer, as opposed to a reference, because there is the possibility that the return value could be `NULL`—*a reference cannot be* `NULL`*, but a pointer can.*

Note that `vector` function `size` (invoked in the loop-continuation condition in line 21) returns the number of elements in a `vector` as a value of type `size_t` (which is usually `unsigned int`). As a result, we declare the control variable `i` to be of type `size_t`, too. On some compilers, declaring `i` as an `int` would cause the compiler to issue a warning message, because the loop-continuation condition would compare a `signed` value (i.e., an `int`) and an `unsigned` value (i.e., a value of type `size_t`).

Member function `authenticateUser` (lines 33–44) proves or disproves the an ATM user's identity. This function takes a user-specified account number and user-specified PIN as arguments and indicates whether they match the account number and PIN of an `Account` in the database. Line 37 calls utility function `getAccount`, which returns either a pointer to an `Account` with `userAccountNumber` as its account number or `NULL` to indicate that `userAccountNumber` is invalid. We declare `userAccountPtr` to be a `const` pointer because, once the member function aims this pointer at the user's `Account`, the pointer should not change. If `getAccount` returns a pointer to an `Account` object, line 41 returns the `bool` value returned by that object's `validatePIN` member function. `BankDatabase`'s `authenticateUser` member function does not perform the PIN comparison itself—rather, it forwards `userPIN` to the `Account` object's `validatePIN` member function to do so. The value returned by `Account` member function `validatePIN` indicates whether the user-specified PIN matches the PIN of the user's `Account`, so member function `authenticateUser` simply returns this value to the client of the class (i.e., `ATM`).

`BankDatabase` trusts the `ATM` to invoke member function `authenticateUser` and receive a return value of `true` before allowing the user to perform transactions. `BankDatabase` also trusts that each `Transaction` object created by the `ATM` contains the valid account number of the current authenticated user and that this is the account number passed to the remaining `BankDatabase` member functions as argument `userAccountNumber`.

Member functions `getAvailableBalance` (lines 47–51), `getTotalBalance` (lines 54–58), `credit` (lines 61–65) and `debit` (lines 68–72) therefore simply retrieve a pointer to the user's `Account` object with utility function `getAccount`, then use this pointer to invoke the appropriate `Account` member function on the user's `Account` object. We know that the calls to `getAccount` within these member functions will never return `NULL`, because `userAccountNumber` must refer to an existing `Account`. Note that `getAvailableBalance` and `getTotalBalance` return the values returned by the corresponding `Account` member functions. Also, `credit` and `debit` simply redirect parameter `amount` to the `Account` member functions they invoke.

26.4.8 Class `Transaction`

Class `Transaction` (Figs. 26.28–26.29) is an *abstract base class* that represents the notion of an ATM transaction. It contains the common features of derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Figure 26.28 expands upon the `Transaction` header file first developed in Section 26.3. Lines 13, 17–19 and 22 contain function prototypes for the class's constructor and four member functions, which we discuss shortly. Line 15 defines a *`virtual` destructor* with an empty body—this makes all derived-class destructors `virtual` (even those defined implicitly by the compiler) and ensures that dynamically allocated derived-class objects get destroyed properly when they are deleted via a base-class pointer. Lines 24–26 declare the class's `private` data members. Recall from the class diagram of Fig. 26.11 that class `Transaction` contains an attribute `accountNumber` (implemented in line 24) that indicates the account involved in the `Transaction`. We derive data members `screen` (line 25) and `bankDatabase` (line 26) from class `Transaction`'s associations modeled in Fig. 26.10—all transactions require access to the ATM's screen and the bank's database, so we include references to a `Screen` and a `BankDatabase` as data members of class `Transaction`. As you'll soon see, `Transaction`'s constructor initializes these references. The forward declarations in lines 6–7 signify that the header file contains references to objects of classes `Screen` and `BankDatabase`, but that the definitions of these classes lie outside the header file.

```
1  // Transaction.h
2  // Transaction abstract base class definition.
3  #ifndef TRANSACTION_H
4  #define TRANSACTION_H
5
6  class Screen; // forward declaration of class Screen
7  class BankDatabase; // forward declaration of class BankDatabase
8
9  class Transaction
10 {
11 public:
12    // constructor initializes common features of all Transactions
13    Transaction( int, Screen &, BankDatabase & );
14
15    virtual ~Transaction() { } // virtual destructor with empty body
16
```

Fig. 26.28 | `Transaction` class definition. (Part 1 of 2.)

```
   int getAccountNumber() const; // return account number
   Screen &getScreen() const; // return reference to screen
   BankDatabase &getBankDatabase() const; // return reference to database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

Fig. 26.28 | `Transaction` class definition. (Part 2 of 2.)

```
// Transaction.cpp
// Member-function definitions for class Transaction.
#include "Transaction.h" // Transaction class definition
#include "Screen.h" // Screen class definition
#include "BankDatabase.h" // BankDatabase class definition

// constructor initializes common features of all Transactions
Transaction::Transaction( int userAccountNumber, Screen &atmScreen,
   BankDatabase &atmBankDatabase )
   : accountNumber( userAccountNumber ),
     screen( atmScreen ),
     bankDatabase( atmBankDatabase )
{
   // empty body
} // end Transaction constructor

// return account number
int Transaction::getAccountNumber() const
{
   return accountNumber;
} // end function getAccountNumber

// return reference to screen
Screen &Transaction::getScreen() const
{
   return screen;
} // end function getScreen

// return reference to bank database
BankDatabase &Transaction::getBankDatabase() const
{
   return bankDatabase;
} // end function getBankDatabase
```

Fig. 26.29 | `Transaction` class member-function definitions.

Class `Transaction` has a constructor (declared in line 13 of Fig. 26.28 and defined in lines 8–15 of Fig. 26.29) that takes the current user's account number and references to the ATM's screen and the bank's database as arguments. Because `Transaction` is an abstract class, this constructor will never be called directly to instantiate `Transaction` objects. Instead, the constructors of the `Transaction` derived classes will use *base-class initializer syntax* to invoke this constructor.

Class `Transaction` has three `public` *get* functions—`getAccountNumber` (declared in line 17 of Fig. 26.28 and defined in lines 18–21 of Fig. 26.29), `getScreen` (declared in line 18 of Fig. 26.28 and defined in lines 24–27 of Fig. 26.29) and `getBankDatabase` (declared in line 19 of Fig. 26.28 and defined in lines 30–33 of Fig. 26.29). `Transaction` derived classes inherit these member functions from `Transaction` and use them to gain access to class `Transaction`'s `private` data members.

Class `Transaction` also declares a pure `virtual` function `execute` (line 22 of Fig. 26.28). It does not make sense to provide an implementation for this member function, because a generic transaction cannot be executed. Thus, we declare this member function to be a pure `virtual` function and force each `Transaction` derived class to provide its own concrete implementation that executes that particular type of transaction.

26.4.9 Class `BalanceInquiry`

Class `BalanceInquiry` (Figs. 26.30–26.31) derives from abstract base class `Transaction` and represents a balance-inquiry ATM transaction. `BalanceInquiry` does not have any data members of its own, but it inherits `Transaction` data members `accountNumber`, `screen` and `bankDatabase`, which are accessible through `Transaction`'s `public` *get* functions. Line 6 `#includes` the definition of base class `Transaction`. The `BalanceInquiry` constructor (declared in line 11 of Fig. 26.30 and defined in lines 8–13 of Fig. 26.31) takes arguments corresponding to the `Transaction` data members and simply forwards them to `Transaction`'s constructor, using *base-class initializer syntax* (line 10 of Fig. 26.31). Line 12 of Fig. 26.30 contains the function prototype for member function `execute`, which is required to indicate the intention to override the base class's pure `virtual` function of the same name.

```
1  // BalanceInquiry.h
2  // BalanceInquiry class definition. Represents a balance inquiry.
3  #ifndef BALANCE_INQUIRY_H
4  #define BALANCE_INQUIRY_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class BalanceInquiry : public Transaction
9  {
10 public:
11    BalanceInquiry( int, Screen &, BankDatabase & ); // constructor
12    virtual void execute(); // perform the transaction
13 }; // end class BalanceInquiry
14
15 #endif // BALANCE_INQUIRY_H
```

Fig. 26.30 | `BalanceInquiry` class definition.

```
1  // BalanceInquiry.cpp
2  // Member-function definitions for class BalanceInquiry.
3  #include "BalanceInquiry.h" // BalanceInquiry class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6
7  // BalanceInquiry constructor initializes base-class data members
8  BalanceInquiry:: BalanceInquiry( int userAccountNumber, Screen &atmScreen,
9     BankDatabase &atmBankDatabase )
10    : Transaction( userAccountNumber, atmScreen, atmBankDatabase )
11 {
12    // empty body
13 } // end BalanceInquiry constructor
14
15 // performs transaction; overrides Transaction's pure virtual function
16 void BalanceInquiry::execute()
17 {
18    // get references to bank database and screen
19    BankDatabase &bankDatabase = getBankDatabase();
20    Screen &screen = getScreen();
21
22    // get the available balance for the current user's Account
23    double availableBalance =
24       bankDatabase.getAvailableBalance( getAccountNumber() );
25
26    // get the total balance for the current user's Account
27    double totalBalance =
28       bankDatabase.getTotalBalance( getAccountNumber() );
29
30    // display the balance information on the screen
31    screen.displayMessageLine( "\nBalance Information:" );
32    screen.displayMessage( " - Available balance: " );
33    screen.displayDollarAmount( availableBalance );
34    screen.displayMessage( "\n - Total balance:     " );
35    screen.displayDollarAmount( totalBalance );
36    screen.displayMessageLine( "" );
37 } // end function execute
```

Fig. 26.31 | `BalanceInquiry` class member-function definitions.

Class `BalanceInquiry` overrides `Transaction`'s pure `virtual` function `execute` to provide a concrete implementation (lines 16–37 of Fig. 26.31) that performs the steps involved in a balance inquiry. Lines 19–20 get references to the bank database and the ATM's screen by invoking member functions inherited from base class `Transaction`. Lines 23–24 retrieve the available balance of the account involved by invoking member function `getAvailableBalance` of `bankDatabase`. Line 24 uses inherited member function `getAccountNumber` to get the account number of the current user, which it then passes to `getAvailableBalance`. Lines 27–28 retrieve the total balance of the current user's account. Lines 31–36 display the balance information on the ATM's screen. Recall that `displayDollarAmount` takes a `double` argument and outputs it to the screen formatted as a dollar amount. For example, if a user's `availableBalance` is `700.5`, line 33 outputs `$700.50`. Line 36 inserts a blank line of output to separate the balance information from subsequent output (i.e., the main menu repeated by class `ATM` after executing the `BalanceInquiry`).

26.4.10 Class `Withdrawal`

Class `Withdrawal` (Figs. 26.32–26.33) derives from `Transaction` and represents a withdrawal ATM transaction. Figure 26.32 expands upon the header file for this class developed in Fig. 26.13. Class `Withdrawal` has a constructor and one member function `execute`, which we discuss shortly. Recall from the class diagram of Fig. 26.11 that class `Withdrawal` has one attribute, `amount`, which line 16 implements as an `int` data member. Figure 26.10 models associations between class `Withdrawal` and classes `Keypad` and `CashDispenser`, for which lines 17–18 implement references `keypad` and `cashDispenser`, respectively. Line 19 is the function prototype of a `private` utility function that we soon discuss.

```
1   // Withdrawal.h
2   // Withdrawal class definition. Represents a withdrawal transaction.
3   #ifndef WITHDRAWAL_H
4   #define WITHDRAWAL_H
5
6   #include "Transaction.h" // Transaction class definition
7   class Keypad; // forward declaration of class Keypad
8   class CashDispenser; // forward declaration of class CashDispenser
9
10  class Withdrawal : public Transaction
11  {
12  public:
13     Withdrawal( int, Screen &, BankDatabase &, Keypad &, CashDispenser & );
14     virtual void execute(); // perform the transaction
15  private:
16     int amount; // amount to withdraw
17     Keypad &keypad; // reference to ATM's keypad
18     CashDispenser &cashDispenser; // reference to ATM's cash dispenser
19     int displayMenuOfAmounts() const; // display the withdrawal menu
20  }; // end class Withdrawal
21
22  #endif // WITHDRAWAL_H
```

Fig. 26.32 | `Withdrawal` class definition.

`Withdrawal` Class Member-Function Definitions

Figure 26.33 contains the member-function definitions for class `Withdrawal`. Line 3 `#includes` the class's definition, and lines 4–7 `#include` the definitions of the other classes used in `Withdrawal`'s member functions. Line 11 declares a global constant corresponding to the cancel option on the withdrawal menu. We'll soon discuss how the class uses this constant.

```
1   // Withdrawal.cpp
2   // Member-function definitions for class Withdrawal.
3   #include "Withdrawal.h" // Withdrawal class definition
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 1 of 4.)

```
#include "Screen.h" // Screen class definition
#include "BankDatabase.h" // BankDatabase class definition
#include "Keypad.h" // Keypad class definition
#include "CashDispenser.h" // CashDispenser class definition

// global constant that corresponds to menu option to cancel
static const int CANCELED = 6;

// Withdrawal constructor initialize class's data members
Withdrawal::Withdrawal( int userAccountNumber, Screen &atmScreen,
   BankDatabase &atmBankDatabase, Keypad &atmKeypad,
   CashDispenser &atmCashDispenser )
   : Transaction( userAccountNumber, atmScreen, atmBankDatabase ),
     keypad( atmKeypad ), cashDispenser( atmCashDispenser )
{
   // empty body
} // end Withdrawal constructor

// perform transaction; overrides Transaction's pure virtual function
void Withdrawal::execute()
{
   bool cashDispensed = false; // cash was not dispensed yet
   bool transactionCanceled = false; // transaction was not canceled yet

   // get references to bank database and screen
   BankDatabase &bankDatabase = getBankDatabase();
   Screen &screen = getScreen();

   // loop until cash is dispensed or the user cancels
   do
   {
      // obtain the chosen withdrawal amount from the user
      int selection = displayMenuOfAmounts();

      // check whether user chose a withdrawal amount or canceled
      if ( selection != CANCELED )
      {
         amount = selection; // set amount to the selected dollar amount

         // get available balance of account involved
         double availableBalance =
            bankDatabase.getAvailableBalance( getAccountNumber() );

         // check whether the user has enough money in the account
         if ( amount <= availableBalance )
         {
            // check whether the cash dispenser has enough money
            if ( cashDispenser.isSufficientCashAvailable( amount ) )
            {
               // update the account involved to reflect withdrawal
               bankDatabase.debit( getAccountNumber(), amount );

               cashDispenser.dispenseCash( amount ); // dispense cash
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 2 of 4.)

```
                  cashDispensed = true; // cash was dispensed

                  // instruct user to take cash
                  screen.displayMessageLine(
                     "\nPlease take your cash from the cash dispenser." );
               } // end if
               else // cash dispenser does not have enough cash
                  screen.displayMessageLine(
                     "\nInsufficient cash available in the ATM."
                     "\n\nPlease choose a smaller amount." );
            } // end if
            else // not enough money available in user's account
            {
               screen.displayMessageLine(
                  "\nInsufficient funds in your account."
                  "\n\nPlease choose a smaller amount." );
            } // end else
         } // end if
         else // user chose cancel menu option
         {
            screen.displayMessageLine( "\nCanceling transaction..." );
            transactionCanceled = true; // user canceled the transaction
         } // end else
      } while ( !cashDispensed && !transactionCanceled ); // end do...while
   } // end function execute

   // display a menu of withdrawal amounts and the option to cancel;
   // return the chosen amount or 0 if the user chooses to cancel
   int Withdrawal::displayMenuOfAmounts() const
   {
      int userChoice = 0; // local variable to store return value

      Screen &screen = getScreen(); // get screen reference

      // array of amounts to correspond to menu numbers
      int amounts[] = { 0, 20, 40, 60, 100, 200 };

      // loop while no valid choice has been made
      while ( userChoice == 0 )
      {
         // display the menu
         screen.displayMessageLine( "\nWithdrawal options:" );
         screen.displayMessageLine( "1 - $20" );
         screen.displayMessageLine( "2 - $40" );
         screen.displayMessageLine( "3 - $60" );
         screen.displayMessageLine( "4 - $100" );
         screen.displayMessageLine( "5 - $200" );
         screen.displayMessageLine( "6 - Cancel transaction" );
         screen.displayMessage( "\nChoose a withdrawal option (1-6): " );

         int input = keypad.getInput(); // get user input through keypad

```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 3 of 4.)

```
109        // determine how to proceed based on the input value
110        switch ( input )
111        {
112           case 1: // if the user chose a withdrawal amount
113           case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
114           case 3: // corresponding amount from amounts array
115           case 4:
116           case 5:
117              userChoice = amounts[ input ]; // save user's choice
118              break;
119           case CANCELED: // the user chose to cancel
120              userChoice = CANCELED; // save user's choice
121              break;
122           default: // the user did not enter a value from 1-6
123              screen.displayMessageLine(
124                 "\nIvalid selection. Try again." );
125        } // end switch
126     } // end while
127
128     return userChoice; // return withdrawal amount or CANCELED
129  } // end function displayMenuOfAmounts
```

Fig. 26.33 | `Withdrawal` class member-function definitions. (Part 4 of 4.)

Class `Withdrawal`'s constructor (defined in lines 13–20 of Fig. 26.33) has five parameters. It uses a base-class initializer in line 16 to pass parameters `userAccountNumber`, `atmScreen` and `atmBankDatabase` to base class `Transaction`'s constructor to set the data members that `Withdrawal` inherits from `Transaction`. The constructor also takes references `atmKeypad` and `atmCashDispenser` as parameters and assigns them to reference data members `keypad` and `cashDispenser` using member initializers (line 17).

Class `Withdrawal` overrides `Transaction`'s pure `virtual` function `execute` with a concrete implementation (lines 23–81) that performs the steps involved in a withdrawal. Line 25 declares and initializes a local `bool` variable `cashDispensed`. This variable indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 26 declares and initializes to `false` a `bool` variable `transactionCanceled` that indicates whether the transaction has been canceled by the user. Lines 29–30 get references to the bank database and the ATM's screen by invoking member functions inherited from base class `Transaction`.

Lines 33–80 contain a `do…while` statement that executes its body until cash is dispensed (i.e., until `cashDispensed` becomes `true`) or until the user chooses to cancel (i.e., until `transactionCanceled` becomes `true`). This loop continuously returns the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 36 displays a menu of withdrawal amounts and obtains a user selection by calling `private` utility function `displayMenuOfAmounts` (defined in lines 85–129). This function displays the menu of amounts and returns either an `int` withdrawal amount or the `int` constant `CANCELED` to indicate that the user has chosen to cancel the transaction.

Member function `displayMenuOfAmounts` (lines 85–129) first declares local variable `userChoice` (initially `0`) to store the value that the member function will return (line 87).

Line 89 gets a reference to the screen by calling member function `getScreen` inherited from base class `Transaction`. Line 92 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index `0`) because the menu has no option 0. The `while` statement in lines 95–126 repeats until `userChoice` takes on a value other than `0`. We'll see shortly that this occurs when the user makes a valid selection from the menu. Lines 98–105 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 107 obtains integer `input` through the keypad. The `switch` statement in lines 110–125 determines how to proceed based on the user's input. If the user selects a number between `1` and `5`, line 117 sets `userChoice` to the value of the element in `amounts` at index `input`. For example, if the user enters `3` to withdraw $60, line 117 sets `userChoice` to the value of `amounts[ 3 ]` (i.e., `60`). Line 118 terminates the `switch`. Variable `userChoice` no longer equals `0`, so the `while` in lines 95–126 terminates and line 128 returns `userChoice`. If the user selects the cancel menu option, lines 120–121 execute, setting `userChoice` to `CANCELED` and causing the member function to return this value. If the user does not enter a valid menu selection, lines 123–124 display an error message and the user is returned to the withdrawal menu.

The `if` statement in line 39 in member function `execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 77–78 execute to display an appropriate message to the user and set `transactionCanceled` to `true`. This causes the loop-continuation test in line 80 to fail and control to return to the calling member function (i.e., `ATM` member function `performTransactions`). If the user has chosen a withdrawal amount, line 41 assigns local variable `selection` to data member `amount`. Lines 44–45 retrieve the available balance of the current user's `Account` and store it in a local `double` variable `availableBalance`. Next, the `if` statement in line 48 determines whether the selected amount is less than or equal to the user's available balance. If it isn't, lines 70–72 display an appropriate error message. Control then continues to the end of the `do…while`, and the loop repeats because both `cashDispensed` and `transactionCanceled` are still `false`. If the user's balance is high enough, the `if` statement in line 51 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the `cashDispenser`'s `isSufficientCashAvailable` member function. If this member function returns `false`, lines 64–66 display an appropriate error message and the `do…while` repeats. If sufficient cash is available, then the requirements for the withdrawal are satisfied, and line 54 debits `amount` from the user's account in the database. Lines 56–57 then instruct the cash dispenser to dispense the cash to the user and set `cashDispensed` to `true`. Finally, lines 60–61 display a message to the user that cash has been dispensed. Because `cashDispensed` is now `true`, control continues after the `do…while`. No additional statements appear below the loop, so the member function returns control to class `ATM`.

In the function calls in lines 64–66 and lines 70–72, we divide the argument to `Screen` member function `displayMessageLine` into two string literals, each placed on a separate line in the program. We do so because each argument is too long to fit on a single line. *C++ concatenates (i.e., combines) string literals adjacent to each other, even if they are on separate lines.* For example, if you write `"Happy " "Birthday"` in a program, C++ will view these two adjacent string literals as the single string literal `"Happy Birthday"`. As a result, when lines 64–66 execute, `displayMessageLine` receives a single `string` as a parameter, even though the argument in the function call appears as two string literals.

26.4.11 Class Deposit

Class `Deposit` (Figs. 26.34–26.35) derives from `Transaction` and represents a deposit ATM transaction. Figure 26.34 contains the `Deposit` class definition. Like derived classes `BalanceInquiry` and `Withdrawal`, `Deposit` declares a constructor (line 13) and member function `execute` (line 14)—we discuss these momentarily. Recall from the class diagram of Fig. 26.11 that class `Deposit` has one attribute `amount`, which line 16 implements as an `int` data member. Lines 17–18 create reference data members `keypad` and `depositSlot` that implement the associations between class `Deposit` and classes `Keypad` and `DepositSlot` modeled in Fig. 26.10. Line 19 contains the function prototype for a `private` utility function `promptForDepositAmount` that we'll discuss shortly.

```
1  // Deposit.h
2  // Deposit class definition. Represents a deposit transaction.
3  #ifndef DEPOSIT_H
4  #define DEPOSIT_H
5
6  #include "Transaction.h" // Transaction class definition
7  class Keypad; // forward declaration of class Keypad
8  class DepositSlot; // forward declaration of class DepositSlot
9
10 class Deposit : public Transaction
11 {
12 public:
13    Deposit( int, Screen &, BankDatabase &, Keypad &, DepositSlot & );
14    virtual void execute(); // perform the transaction
15 private:
16    double amount; // amount to deposit
17    Keypad &keypad; // reference to ATM's keypad
18    DepositSlot &depositSlot; // reference to ATM's deposit slot
19    double promptForDepositAmount() const; // get deposit amount from user
20 }; // end class Deposit
21
22 #endif // DEPOSIT_H
```

Fig. 26.34 | `Deposit` class definition.

`Deposit` Class Member-Function Definitions

Figure 26.35 presents the `Deposit` class implementation. Line 3 `#include`s the `Deposit` class definition, and lines 4–7 `#include` the class definitions of the other classes used in `Deposit`'s member functions. Line 9 declares a constant `CANCELED` that corresponds to the value a user enters to cancel a deposit. We'll soon discuss how the class uses this constant.

```
1  // Deposit.cpp
2  // Member-function definitions for class Deposit.
3  #include "Deposit.h" // Deposit class definition
4  #include "Screen.h" // Screen class definition
5  #include "BankDatabase.h" // BankDatabase class definition
6  #include "Keypad.h" // Keypad class definition
```

Fig. 26.35 | `Deposit` class member-function definitions. (Part 1 of 3.)

```
#include "DepositSlot.h" // DepositSlot class definition

static const int CANCELED = 0; // constant representing cancel option

// Deposit constructor initializes class's data members
Deposit::Deposit( int userAccountNumber, Screen &atmScreen,
   BankDatabase &atmBankDatabase, Keypad &atmKeypad,
   DepositSlot &atmDepositSlot )
   : Transaction( userAccountNumber, atmScreen, atmBankDatabase ),
     keypad( atmKeypad ), depositSlot( atmDepositSlot )
{
   // empty body
} // end Deposit constructor

// performs transaction; overrides Transaction's pure virtual function
void Deposit::execute()
{
   BankDatabase &bankDatabase = getBankDatabase(); // get reference
   Screen &screen = getScreen(); // get reference

   amount = promptForDepositAmount(); // get deposit amount from user

   // check whether user entered a deposit amount or canceled
   if ( amount != CANCELED )
   {
      // request deposit envelope containing specified amount
      screen.displayMessage(
         "\nPlease insert a deposit envelope containing " );
      screen.displayDollarAmount( amount );
      screen.displayMessageLine( " in the deposit slot." );

      // receive deposit envelope
      bool envelopeReceived = depositSlot.isEnvelopeReceived();

      // check whether deposit envelope was received
      if ( envelopeReceived )
      {
         screen.displayMessageLine( "\nYour envelope has been received."
            "\nNOTE: The money deposited will not be available until we"
            "\nverify the amount of any enclosed cash, and any enclosed "
            "checks clear." );

         // credit account to reflect the deposit
         bankDatabase.credit( getAccountNumber(), amount );
      } // end if
      else // deposit envelope not received
      {
         screen.displayMessageLine( "\nYou did not insert an "
            "envelope, so the ATM has canceled your transaction." );
      } // end else
   } // end if
   else // user canceled instead of entering amount
   {
```

Fig. 26.35 | Deposit class member-function definitions. (Part 2 of 3.)

```
60         screen.displayMessageLine( "\nCanceling transaction..." );
61      } // end else
62   } // end function execute
63
64   // prompt user to enter a deposit amount in cents
65   double Deposit::promptForDepositAmount() const
66   {
67      Screen &screen = getScreen(); // get reference to screen
68
69      // display the prompt and receive input
70      screen.displayMessage( "\nPlease enter a deposit amount in "
71         "CENTS (or 0 to cancel): " );
72      int input = keypad.getInput(); // receive input of deposit amount
73
74      // check whether the user canceled or entered a valid amount
75      if ( input == CANCELED )
76         return CANCELED;
77      else
78      {
79         return static_cast< double >( input ) / 100; // return dollar amount
80      } // end else
81   } // end function promptForDepositAmount
```

Fig. 26.35 | `Deposit` class member-function definitions. (Part 3 of 3.)

Like class `Withdrawal`, class `Deposit` contains a constructor (lines 12–19) that passes three parameters to base class `Transaction`'s constructor using a base-class initializer (line 15). The constructor also has parameters `atmKeypad` and `atmDepositSlot`, which it assigns to its corresponding data members (line 16).

Member function `execute` (lines 22–62) overrides pure `virtual` function `execute` in base class `Transaction` with a concrete implementation that performs the steps required in a deposit transaction. Lines 24–25 get references to the database and the screen. Line 27 prompts the user to enter a deposit amount by invoking `private` utility function `promptForDepositAmount` (defined in lines 65–81) and sets data member `amount` to the value returned. Member function `promptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM's keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the `double` value representing the dollar amount to be deposited.

Line 67 in member function `promptForDepositAmount` gets a reference to the ATM's screen. Lines 70–71 display a message on the screen asking the user to input a deposit amount as a number of cents or "0" to cancel the transaction. Line 72 receives the user's input from the keypad. The `if` statement in lines 75–80 determines whether the user has entered a real deposit amount or chosen to cancel. If the user chooses to cancel, line 76 returns the constant `CANCELED`. Otherwise, line 79 returns the deposit amount after converting from the number of cents to a dollar amount by casting `input` to a `double`, then dividing by `100`. For example, if the user enters `125` as the number of cents, line 79 returns `125.0` divided by `100`, or `1.25`—125 cents is $1.25.

The `if` statement in lines 30–61 in member function `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 60 displays an appropriate message, and the member function returns. If the user enters a deposit amount, lines 33–36 instruct the user to insert a deposit envelope with the correct amount. Recall that `Screen` member function `displayDollarAmount` outputs a `double` formatted as a dollar amount.

Line 39 sets a local `bool` variable to the value returned by `depositSlot`'s `isEnvelopeReceived` member function, indicating whether a deposit envelope has been received. Recall that we coded `isEnvelopeReceived` (lines 7–10 of Fig. 26.23) to always return `true`, because we are simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code member function `execute` of class `Deposit` to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for all possible return values. Thus, class `Deposit` is prepared for future versions of `isEnvelopeReceived` that could return `false`. Lines 44–50 execute if the deposit slot receives an envelope. Lines 44–47 display an appropriate message to the user. Line 50 then credits the deposit amount to the user's account in the database. Lines 54–55 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The member function then returns without modifying the user's account.

26.4.12 Test Program `ATMCaseStudy.cpp`

`ATMCaseStudy.cpp` (Fig. 26.36) is a simple C++ program that allows us to start, or "turn on," the ATM and test the implementation of our ATM system model. The program's `main` function (lines 6–11) does nothing more than instantiate a new `ATM` object named `atm` (line 8) and invoke its `run` member function (line 9) to start the ATM.

```
1   // ATMCaseStudy.cpp
2   // Driver program for the ATM case study.
3   #include "ATM.h" // ATM class definition
4
5   // main function creates and runs the ATM
6   int main()
7   {
8      ATM atm; // create an ATM object
9      atm.run(); // tell the ATM to start
10  } // end main
```

Fig. 26.36 | `ATMCaseStudy.cpp` starts the ATM system.

26.5 Wrap-Up

In this chapter, you used inheritance to tune the design of the ATM software system, and you fully implemented the ATM in C++. Congratulations on completing the entire ATM case study! We hope you found this experience to be valuable and that it reinforced many of the object-oriented programming concepts that you've learned.

Answers to Self-Review Exercises

26.1 True. The minus sign (-) indicates private visibility. We've mentioned "friendship" as an exception to private visibility. Friendship is discussed in Chapter 10.

26.2 b.

26.3 The design for class `Account` yields the header file in Fig. 26.37.

```
// Fig. 26.37: Account.h
// Account class definition. Represents a bank account.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
   bool validatePIN( int ); // is user-specified PIN correct?
   double getAvailableBalance(); // returns available balance
   double getTotalBalance(); // returns total balance
   void credit( double ); // adds an amount to the Account
   void debit( double ); // subtracts an amount from the Account
private:
   int accountNumber; // account number
   int pin; // PIN for authentication
   double availableBalance; // funds available for withdrawal
   double totalBalance; // funds available + funds waiting to clear
}; // end class Account

#endif // ACCOUNT_H
```

Fig. 26.37 | `Account` class header file based on Fig. 26.1 and Fig. 26.2.

26.4 b.

26.5 False. The UML requires that we italicize abstract class names and operation names.

26.6 The design for class `Transaction` yields the header file in Fig. 26.38. In the implementation, a constructor initializes `private` reference attributes `screen` and `bankDatabase` to actual objects, and member functions `getScreen` and `getBankDatabase` access these attributes. These member functions allow classes derived from `Transaction` to access the ATM's screen and interact with the bank's database.

```
// Fig. 36.38: Transaction.h
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   int getAccountNumber(); // return account number
   Screen &getScreen(); // return reference to screen
   BankDatabase &getBankDatabase(); // return reference to bank database
```

Fig. 26.38 | `Transaction` class header file based on Fig. 26.10 and Fig. 26.11. (Part 1 of 2.)

```

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

Fig. 26.38 | `Transaction` class header file based on Fig. 26.10 and Fig. 26.11. (Part 2 of 2.)

This page intentionally left blank

F

C Legacy Code Topics

We'll use a signal I have tried and found far-reaching and easy to yell. Waa-hoo!
—Zane Grey

It is quite a three-pipe problem.
—Sir Arthur Conan Doyle

But yet an union in partition.
—William Shakespeare

Objectives

In this appendix you'll learn

- To redirect keyboard input to come from a file and redirect screen output to a file.
- To write functions that use variable-length argument lists.
- To process command-line arguments.
- To process unexpected events within a program.
- To allocate memory dynamically for arrays, using C-style dynamic memory allocation.
- To resize memory dynamically allocated using C-style dynamic memory allocation.

Outline

F.1 Introduction

This chapter presents several topics not ordinarily covered in introductory courses. Many of the capabilities discussed here are specific to particular operating systems, especially UNIX/LINUX/Mac OS X and/or Windows. Much of the material is for the benefit of C++ programmers who will need to work with older C legacy code.

F.2 Redirecting Input/Output on UNIX/Linux/Mac OS X and Windows Systems

Normally, the input to a program is from the keyboard (standard input), and the output from a program is displayed on the screen (standard output). On most computer systems—UNIX, LINUX, Mac OS X and Windows systems in particular—it is possible to **redirect** inputs to come from a file, and redirect outputs to be placed in a file. Both forms of redirection can be accomplished without using the file-processing capabilities of the standard library.

There are several ways to redirect input and output from the UNIX command line. Consider the executable file `sum` that inputs integers one at a time, keeps a running total of the values until the end-of-file indicator is set, then prints the result. Normally the user inputs integers from the keyboard and enters the end-of-file key combination to indicate that no further values will be input. With input redirection, the input can be stored in a file. For example, if the data are stored in file `input`, the command line

```
$ sum < input
```

causes program `sum` to be executed; the **redirect input symbol (`<`)** indicates that the data in file `input` (instead of the keyboard) is to be used as input by the program. Redirecting input in a Windows **Command Prompt** is performed identically.

Note that `$` represents the UNIX command-line prompt. (UNIX prompts vary from system to system and between shells on a single system.) Redirection is an operating-system function, not another C++ feature.

The second method of redirecting input is **piping**. A **pipe (`|`)** causes the output of one program to be redirected as the input to another program. Suppose program `random` outputs a series of random integers; the output of `random` can be "piped" directly to program `sum` using the UNIX command line

```
$ random | sum
```

This causes the sum of the integers produced by `random` to be calculated. Piping can be performed in UNIX, LINUX, Mac OS X and Windows.

Program output can be redirected to a file by using the **redirect output symbol (>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to redirect the output of program `random` to a new file called `out`, use

```
$ random > out
```

Finally, program output can be appended to the end of an existing file by using the **append output symbol (>>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to append the output from program `random` to file `out` created in the preceding command line, use the command line

```
$ random >> out
```

F.3 Variable-Length Argument Lists

It is possible to create functions that receive an unspecified number of arguments. An ellipsis (`...`) in a function's prototype indicates that the function receives a variable number of arguments of any type.[1] Note that the ellipsis must always be placed at the end of the parameter list, and there must be at least one argument before the ellipsis. The macros and definitions of the **variable arguments header `<cstdarg>`** (Fig. F.1) provide the capabilities necessary to build functions with variable-length argument lists.

Identifier	Description
`va_list`	A type suitable for holding information needed by macros `va_start`, `va_arg` and `va_end`. To access the arguments in a variable-length argument list, an object of type `va_list` must be declared.
`va_start`	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with `va_list` for use by the `va_arg` and `va_end` macros.
`va_arg`	A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of `va_arg` modifies the object declared with `va_list` so that the object points to the next argument in the list.
`va_end`	A macro that performs termination housekeeping in a function whose variable-length argument list was referred to by the `va_start` macro.

Fig. F.1 | The type and the macros defined in header `<cstdarg>`.

1. In C++, programmers use function overloading to accomplish much of what C programmers accomplish with variable-length argument lists.

Figure F.2 demonstrates function average that receives a variable number of arguments. The first argument of average is always the number of values to be averaged, and the remainder of the arguments must all be of type double.

Function average uses all the definitions and macros of header <cstdarg>. Object list, of type **va_list**, is used by macros **va_start**, **va_arg** and **va_end** to process the vari-

```
// Fig. F.2: figF_02.cpp
// Using variable-length argument lists.
#include <iostream>
#include <iomanip>
#include <cstdarg>
using namespace std;

double average( int, ... );

int main()
{
   double double1 = 37.5;
   double double2 = 22.5;
   double double3 = 1.7;
   double double4 = 10.2;

   cout << fixed << setprecision( 1 ) << "double1 = "
        << double1 << "\ndouble2 = " << double2 << "\ndouble3 = "
        << double3 << "\ndouble4 = " << double4 << endl
        << setprecision( 3 )
        << "\nThe average of double1 and double2 is "
        << average( 2, double1, double2 )
        << "\nThe average of double1, double2, and double3 is "
        << average( 3, double1, double2, double3 )
        << "\nThe average of double1, double2, double3"
        << " and double4 is "
        << average( 4, double1, double2, double3, double4 )
        << endl;
}  // end main

// calculate average
double average( int count, ... )
{
   double total = 0;
   va_list list;  // for storing information needed by va_start

   va_start( list, count );

   // process variable-length argument list
   for ( int i = 1; i <= count; i++ )
      total += va_arg( list, double );

   va_end( list ); // end the va_start
   return total / count;
}  // end function average
```

Fig. F.2 | Using variable-length argument lists. (Part 1 of 2.)

```
double1 = 37.5
double2 = 22.5
double3 = 1.7
double4 = 10.2

The average of double1 and double2 is 30.000
The average of double1, double2, and double3 is 20.567
The average of double1, double2, double3 and double4 is 17.975
```

Fig. F.2 | Using variable-length argument lists. (Part 2 of 2.)

able-length argument list of function `average`. The function invokes `va_start` to initialize object `list` for use in `va_arg` and `va_end`. The macro receives two arguments—object `list` and the identifier of the rightmost argument in the argument list before the ellipsis—`count` in this case (`va_start` uses `count` here to determine where the variable-length argument list begins).

Next, function `average` repeatedly adds the arguments in the variable-length argument list to the `total`. The value to be added to `total` is retrieved from the argument list by invoking macro `va_arg`. Macro `va_arg` receives two arguments—object `list` and the type of the value expected in the argument list (`double` in this case)—and returns the value of the argument. Function `average` invokes macro `va_end` with object `list` as an argument before returning. Finally, the average is calculated and returned to `main`. Note that we used only `double` arguments for the variable-length portion of the argument list.

Variable-length argument lists promote variables of type `float` to type `double`. These argument lists also promote integral variables that are smaller than `int` to type `int` (variables of type `int`, `unsigned`, `long` and `unsigned long` are left alone).

Software Engineering Observation F.1

Variable-length argument lists can be used only with fundamental-type arguments and with `struct`-type arguments that do not contain C++ specific features such as `virtual` functions, constructors, destructors, references, `const` data members and `virtual` base classes.

Common Programming Error F.1

Placing an ellipsis in the middle of a function parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

F.4 Using Command-Line Arguments

On many systems it is possible to pass arguments to `main` from a command line by including parameters `int argc` and `char *argv[]` in the parameter list of `main`. Parameter **`argc`** receives the number of command-line arguments. Parameter **`argv`** is an array of `char *`'s pointing to strings in which the actual command-line arguments are stored. Common uses of command-line arguments include printing the arguments, passing options to a program and passing filenames to a program.

Figure F.3 copies a file into another file one character at a time. The executable file for the program is called `copyFile` (i.e., the executable name for the file). A typical command line for the `copyFile` program on a UNIX system is

```
$ copyFile input output
```

```
1  // Fig. F.3: figF_03.cpp
2  // Using command-line arguments
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main( int argc, char *argv[] )1
8  {
9     // check number of command-line arguments
10    if ( argc != 3 )
11       cout << "Usage: copyFile infile_name outfile_name" << endl;
12    else
13    {
14       ifstream inFile( argv[ 1 ], ios::in );
15
16       // input file could not be opened
17       if ( !inFile )
18       {
19          cout << argv[ 1 ] << " could not be opened" << endl;
20          return -1;
21       }  // end if
22
23       ofstream outFile( argv[ 2 ], ios::out );
24
25       // output file could not be opened
26       if ( !outFile )
27       {
28          cout << argv[ 2 ] << " could not be opened" << endl;
29          inFile.close();
30          return -2;
31       } // end if
32
33       char c = inFile.get(); // read first character
34
35       while ( inFile )
36       {
37          outFile.put( c );   // output character
38          c = inFile.get();   // read next character
39       }  // end while
40    }  // end else
41 }  // end main
```

Fig. F.3 | Using command-line arguments.

This command line indicates that file `input` is to be copied to file `output`. When the program executes, if `argc` is not `3` (`copyFile` counts as one of the arguments), the program prints an error message (line 11). Otherwise, array `argv` contains the strings `"copyFile"`, `"input"` and `"output"`. The second and third arguments on the command line are used as file names by the program. The files are opened by creating `ifstream` object `inFile` and `ofstream` object `outFile` (lines 14 and 23). If both files are opened successfully, characters are read from file `input` with member function `get` and written to file `output` with member function `put` until the end-of-file indicator for file `input` is set (lines 35–39). Then the program terminates. The result is an exact copy of file `input`. Note that not all computer systems support command-line arguments as easily as UNIX, LINUX, Mac OS X and Windows. Some VMS and older Macintosh systems, for example, require special settings for processing command-line arguments. See the manuals for your system for more information on command-line arguments.

F.5 Notes on Compiling Multiple-Source-File Programs

As stated earlier in the text, it is normal to build programs that consist of multiple source files (see Chapter 9). There are several considerations when creating programs in multiple files. For example, the definition of a function must be entirely contained in one file—it cannot span two or more files.

In Chapter 6, we introduced the concepts of storage class and scope. We learned that variables declared outside any function definition are of storage class `static` by default and are referred to as global variables. Global variables are accessible to any function defined in the same file after the variable is declared. Global variables also are accessible to functions in other files; however, the global variables must be declared in each file in which they are used. For example, if we define global integer variable `flag` in one file, and refer to it in a second file, the second file must contain the declaration

```
extern int flag;
```

prior to the variable's use in that file. In the preceding declaration, the storage class-specifier `extern` indicates to the compiler that variable `flag` is defined either later in the same file or in a different file. The compiler informs the linker that unresolved references to variable `flag` appear in the file. (The compiler does not know where `flag` is defined, so it lets the linker attempt to find `flag`.) If the linker cannot locate a definition of `flag`, a linker error is reported. If a proper global definition is located, the linker resolves the references by indicating where `flag` is located.

Performance Tip F.1

Global variables increase performance because they can be accessed directly by any function—the overhead of passing data to functions is eliminated.

Software Engineering Observation F.2

Global variables should be avoided unless application performance is critical or the variable represents a shared global resource such as `cin`, because they violate the principle of least privilege, and they make software difficult to maintain.

Just as `extern` declarations can be used to declare global variables to other program files, function prototypes can be used to declare functions in other program files. (The `extern` specifier is not required in prototypes.) This is accomplished by including the function prototype in each file in which the function is invoked, then compiling each source file and linking the resulting object code files together. Function prototypes indicate to the compiler that the specified function is defined either later in the same file or in a different file. The compiler does not attempt to resolve references to such a function—that task is left to the linker. If the linker cannot locate a function definition, an error is generated.

As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <cstring>`. This directive includes in a file the function prototypes for functions such as `strcmp` and `strcat`. Other functions in the file can use `strcmp` and `strcat` to accomplish their tasks. The `strcmp` and `strcat` functions are defined for us separately. We do not need to know where they are defined. We are simply reusing the code in our programs. The linker resolves our references to these functions. This process enables us to use the functions in the standard library.

Software Engineering Observation F.3

Creating programs in multiple source files facilitates software reusability and good software engineering. Functions may be common to many applications. In such instances, those functions should be stored in their own source files, and each source file should have a corresponding header file containing function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their application with the corresponding source file.

Portability Tip F.1

Some systems do not support global variable names or function names of more than six characters. This should be considered when writing programs that will be ported to multiple platforms.

It is possible to restrict the scope of a global variable or function to the file in which it is defined. The storage-class specifier `static`, when applied to a global namespace scope variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as **internal linkage**. Global variables (except those that are `const`) and functions that are not preceded by `static` in their definitions have **external linkage**—they can be accessed in other files if those files contain proper declarations and/or function prototypes.

The global variable declaration

```
static double pi = 3.14159;
```

creates variable `pi` of type `double`, initializes it to `3.14159` and indicates that `pi` is known only to functions in the file in which it is defined.

The `static` specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`. If a function is defined before it is used in a file, `static` should be applied to the function definition. Otherwise, `static` should be applied to the function prototype. Identifiers defined in the unnamed `namespace` also have internal linkage. The C++ standard recommends using the unnamed `namespace` rather than `static`.

When building large programs from multiple source files, compiling the program becomes tedious if making small changes to one file means that the entire program must be recompiled. Many systems provide special utilities that recompile only source files dependent on the modified program file. On UNIX systems, the utility is called `make`. Utility `make` reads a file called `Makefile` that contains instructions for compiling and linking the program. Systems such as Borland C++ and Microsoft Visual C++ for PCs provide `make` utilities and "projects." For more information on `make` utilities, see the manual for your particular system.

F.6 Program Termination with exit and atexit

The general utilities library (<cstdlib>) provides methods of terminating program execution other than a conventional return from main. Function exit forces a program to terminate as if it executed normally. The function often is used to terminate a program when an error is detected or if a file to be processed by the program cannot be opened.

Function atexit **registers** a function in the program to be called when the program terminates by reaching the end of main or when exit is invoked. Function **atexit** takes a pointer to a function (i.e., the function name) as an argument. Functions called at program termination cannot have arguments and cannot return a value.

Function exit takes one argument. The argument is normally the symbolic constant **EXIT_SUCCESS** or **EXIT_FAILURE**. If exit is called with EXIT_SUCCESS, the implementation-defined value for successful termination is returned to the calling environment. If exit is called with EXIT_FAILURE, the implementation-defined value for unsuccessful termination is returned. When function exit is invoked, any functions previously registered with atexit are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed, and control returns to the host environment. Figure F.4 tests functions exit and atexit. The program prompts the user to determine whether the program should be terminated with exit or by reaching the end of main. Note that function print is executed at program termination in each case.

```
// Fig. F.4: figF_04.cpp
// Using the exit and atexit functions
#include <iostream>
#include <cstdlib>
using namespace std;

void print();

int main()
{
   atexit( print ); // register function print

   cout << "Enter 1 to terminate program with function exit"
        << "\nEnter 2 to terminate program normally\n";

   int answer;
   cin >> answer;

   // exit if answer is 1
   if ( answer == 1 )
   {
      cout << "\nTerminating program with function exit\n";
      exit( EXIT_SUCCESS );
   } // end if

   cout << "\nTerminating program by reaching the end of main"
        << endl;
} // end main
```

Fig. F.4 | Using functions exit and atexit. (Part 1 of 2.)

```
// display message before termination
void print()
{
   cout << "Executing function print at program termination\n"
        << "Program terminated" << endl;
}  // end function print
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

Fig. F.4 | Using functions `exit` and `atexit`. (Part 2 of 2.)

Terminating a program with function `exit` runs the destructors for only the static and global objects in the program. Terminating with function `abort` ends the program without running any destructors.

F.7 Type Qualifier `volatile`

The **`volatile`** type qualifier is applied to a definition of a variable that may be altered from outside the program (i.e., the variable is not completely under the control of the program). Thus, the compiler cannot perform optimizations (such as speeding program execution or reducing memory consumption, for example) that depend on "knowing that a variable's behavior is influenced only by program activities the compiler can observe."

F.8 Suffixes for Integer and Floating-Point Constants

C++ provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are: **`u`** or **`U`** for an **`unsigned`** integer, **`l`** or **`L`** for a **`long`** integer, and **`ul`** or **`UL`** for an **`unsigned long`** integer. The following constants are of type `unsigned`, `long` and `unsigned long`, respectively:

```
174u
8358L
28373ul
```

If an integer constant is not suffixed, its type is `int`; if the constant cannot be stored in an `int`, it is stored in a `long`.

The floating-point suffixes are **f** or **F** for a float and **l** or **L** for a **long double**. The following constants are of type long double and float, respectively:

```
3.14159L
1.28f
```

A floating-point constant that is not suffixed is of type double. A constant with an improper suffix results in either a compiler warning or an error.

F.9 Signal Handling

An unexpected event, or **signal**, can terminate a program prematurely. Such events include **interrupts** (pressing <*Ctrl*> *C* on a UNIX, LINUX, Mac OS X or Windows system), **illegal instructions**, **segmentation violations**, **termination orders from the operating system** and **floating-point exceptions** (division by zero or multiplying large floating-point values). The **signal-handling library** provides function signal to **trap unexpected events**. Function **signal** receives two arguments—an integer signal number and a pointer to a signal-handling function. Signals can be generated by function **raise**, which takes an integer signal number as an argument. Figure F.5 summarizes the standard signals defined in header file **<csignal>**. The next example demonstrates functions signal and raise.

Figure F.6 traps an interactive signal (SIGINT) with function signal. The program calls signal with SIGINT and a pointer to function signalHandler. (Recall that a function's name is a pointer to that function.) When a signal of type SIGINT occurs, function signalHandler is called, a message is printed and the user is given the option to continue normal program execution. If the user wishes to continue execution, the signal handler is reinitialized by calling signal again (some systems require the signal handler to be reinitialized), and control returns to the point in the program at which the signal was detected. In this program, function raise is used to simulate an interactive signal. A random number between 1 and 50 is chosen. If the number is 25, then raise is called to generate the signal. Normally, interactive signals are initiated outside the program. For example, pressing <*Ctrl*> *C* during program execution on a UNIX, LINUX, Mac OS X or Windows system generates an interactive signal that terminates program execution. Signal handling can be used to trap the interactive signal and prevent the program from terminating.

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to abort).
SIGFPE	An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

Fig. F.5 | Signals defined in header <csignal>.

```
// Fig. F.6: figF_06.cpp
// Using signal handling
#include <iostream>
#include <iomanip>
#include <csignal>
#include <cstdlib>
#include <ctime>
using namespace std;

void signalHandler( int );

int main()
{
   signal( SIGINT, signalHandler );
   srand( time( 0 ) );

   // create and output random numbers
   for ( int i = 1; i <= 100; i++ )
   {
      int x = 1 + rand() % 50;

      if ( x == 25 )
         raise( SIGINT ); // raise SIGINT when x is 25

      cout << setw( 4 ) << i;

      if ( i % 10 == 0 )
         cout << endl; // output endl when i is a multiple of 10
   } // end for
} // end main

// handles signal
void signalHandler( int signalValue )
{
   cout << "\nInterrupt signal (" << signalValue
        << ") received.\n"
        << "Do you wish to continue (1 = yes or 2 = no)? ";

   int response;

   cin >> response;

   // check for invalid responses
   while ( response != 1 && response != 2 )
   {
      cout << "(1 = yes or 2 = no)? ";
      cin >> response;
   }  // end while

   // determine if it is time to exit
   if ( response != 1 )
      exit( EXIT_SUCCESS );

```

Fig. F.6 | Using signal handling. (Part 1 of 2.)

```
    // call signal and pass it SIGINT and address of signalHandler
    signal( SIGINT, signalHandler );
} // end function signalHandler
```

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93  94  95  96  97  98  99
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 100
```

```
   1   2   3   4
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Fig. F.6 | Using signal handling. (Part 2 of 2.)

F.10 Dynamic Memory Allocation with `calloc` and `realloc`

In Chapter 11, we discussed C++-style dynamic memory allocation with `new` and `delete`. C++ programmers should use `new` and `delete`, rather than C's functions **`malloc`** and **`free`** (header `<cstdlib>`). However, most C++ programmers will find themselves reading a great deal of C legacy code, and therefore we include this additional discussion on C-style dynamic memory allocation.

The general utilities library (`<cstdlib>`) provides two other functions for dynamic memory allocation—`calloc` and `realloc`. These functions can be used to create and modify **dynamic arrays**. As shown in Chapter 8, a pointer to an array can be subscripted like an array. Thus, a pointer to a contiguous portion of memory created by `calloc` can be manipulated as an array. Function **`calloc`** dynamically allocates memory for an array and initializes the memory to zeros. The prototype for `calloc` is

```
void *calloc( size_t nmemb, size_t size );
```

It receives two arguments—the number of elements (`nmemb`) and the size of each element (`size`)—and initializes the elements of the array to zero. The function returns a pointer to the allocated memory or a null pointer (`0`) if the memory is not allocated.

Function **`realloc`** changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`. The original object's contents are not modified, provided that the memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the size of the new object. The prototype for `realloc` is

```
void *realloc( void *ptr, size_t size );
```

Function `realloc` takes two arguments—a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is `0`, `realloc` works identically to `malloc`. If `size` is `0` and `ptr` is not `0`, the memory for the object is freed. Otherwise, if `ptr` is not `0` and size is greater than zero, `realloc` tries to allocate a new block of memory. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory or a null pointer.

Common Programming Error F.2

Runtime errors may occur if you use the `delete` *operator on a pointer resulting from* `malloc`*,* `calloc` *or* `realloc`*, or if you use* `realloc` *or* `free` *on a pointer resulting from the* `new` *operator.*

F.11 Unconditional Branch: goto

Throughout the text we've stressed the importance of using structured programming techniques to build reliable software that is easy to debug, maintain and modify. In some cases, performance is more important than strict adherence to structured-programming techniques. In these cases, some unstructured programming techniques may be used. For example, we can use `break` to terminate execution of a repetition structure before the loop-continuation condition becomes false. This saves unnecessary repetitions of the loop if the task is completed before loop termination.

Another instance of unstructured programming is the **goto statement**—an unconditional branch. The result of the `goto` statement is a change in the flow of control of the program to the first statement after the **label** specified in the `goto` statement. A label is an identifier followed by a colon. A label must appear in the same function as the `goto` statement that refers to it. Figure F.7 uses `goto` statements to loop 10 times and print the counter value each time. After initializing `count` to `1`, the program tests `count` to determine whether it is greater than `10`. (The label `start` is skipped, because labels do not perform any action.) If so, control is transferred from the `goto` to the first statement after the label `end`. Otherwise, `count` is printed and incremented, and control is transferred from the `goto` to the first statement after the label `start`.

```
// Fig. F.7: figF_07.cpp
// Using goto.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int count = 1;

   start: // label
      // goto end when count exceeds 10
      if ( count > 10 )
         goto end;

```

Fig. F.7 | Using goto. (Part 1 of 2.)

```
        cout << setw( 2 ) << left << count;
        ++count;

        // goto start on line 17
        goto start;

    end: // label
        cout << endl;
} // end main
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. F.7 | Using `goto`. (Part 2 of 2.)

In Chapters 4–5, we stated that only three control structures are required to write any program—sequence, selection and repetition. When the rules of structured programming are followed, it is possible to create deeply nested control structures from which it is difficult to escape efficiently. Some programmers use `goto` statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure.

Performance Tip F.2

The `goto` *statement can be used to exit deeply nested control structures efficiently but can make code more difficult to read and maintain. Its use is strongly discouraged.*

Error-Prevention Tip F.1

The `goto` *statement should be used only in performance-oriented applications. The* `goto` *statement is unstructured and can lead to programs that are more difficult to debug, maintain, modify and understand.*

F.12 Unions

A **union** (defined with keyword **`union`**) is a region of memory that, over time, can contain objects of a variety of types. However, at any moment, a `union` can contain a maximum of one object, because the members of a `union` share the same storage space. It is your responsibility to ensure that the data in a `union` is referenced with a member name of the proper data type.

Common Programming Error F.3

The result of referencing a `union` *member other than the last one stored is undefined. It treats the stored data as a different type.*

Portability Tip F.2

If data are stored in a `union` *as one type and referenced as another type, the results are implementation dependent.*

At different times during a program's execution, some objects might not be relevant, while one other object is—so a `union` shares the space instead of wasting storage on objects that are not being used. The number of bytes used to store a `union` will be at least enough to hold the largest member.

Performance Tip F.3

Using `union`s conserves storage.

Portability Tip F.3

The amount of storage required to store a `union` is implementation dependent.

A `union` is declared in the same format as a `struct` or a `class`. For example,

```
union Number
{
   int x;
   double y;
};
```

indicates that `Number` is a `union` type with members `int x` and `double y`. The `union` definition must precede all functions in which it will be used.

Software Engineering Observation F.4

Like a `struct` or a `class` declaration, a `union` declaration creates a new type. Placing a `union` or `struct` declaration outside any function does not create a global variable.

The only valid built-in operations that can be performed on a `union` are assigning a `union` to another `union` of the same type, taking the address (`&`) of a `union` and accessing `union` members using the structure member operator (`.`) and the structure pointer operator (`->`). `union`s cannot be compared.

Common Programming Error F.4

Comparing `union`s is a compilation error, because the compiler does not know which member of each is active and hence which member of one to compare to which member of the other.

A `union` is similar to a class in that it can have a constructor to initialize any of its members. A `union` that has no constructor can be initialized with another `union` of the same type, with an expression of the type of the first member of the `union` or with an initializer (enclosed in braces) of the type of the first member of the `union`. `union`s can have other member functions, such as destructors, but a `union`'s member functions cannot be declared `virtual`. The members of a `union` are `public` by default.

Common Programming Error F.5

Initializing a `union` in a declaration with a value or an expression whose type is different from the type of the `union`'s first member is a compilation error.

A `union` cannot be used as a base class in inheritance (i.e., classes cannot be derived from `union`s). `union`s can have objects as members only if these objects do not have a constructor, a destructor or an overloaded assignment operator. None of a `union`'s data members can be declared `static`.

Figure F.8 uses the variable `value` of type `union Number` to display the value stored in the `union` as both an `int` and a `double`. The program output is implementation dependent. The program output shows that the internal representation of a `double` value can be quite different from the representation of an `int`.

```cpp
// Fig. F.8: figF_08.cpp
// An example of a union.
#include <iostream>
using namespace std;

// define union Number
union Number
{
   int integer1;
   double double1;
};  // end union Number

int main()
{
   Number value; // union variable

   value.integer1 = 100; // assign 100 to member integer1

   cout << "Put a value in the integer member\n"
        << "and print both members.\nint:   "
        << value.integer1 << "\ndouble: " << value.double1
        << endl;

   value.double1 = 100.0;  // assign 100.0 to member double1

   cout << "Put a value in the floating member\n"
        << "and print both members.\nint:   "
        << value.integer1 << "\ndouble: " << value.double1
        << endl;
}  // end main
```

```
Put a value in the integer member
and print both members.
int:   100
double: -9.25596e+061
Put a value in the floating member
and print both members.
int:   0
double: 100
```

Fig. F.8 | Printing the value of a union in both member data types.

An **anonymous** `union` is a `union` without a type name that does not attempt to define objects or pointers before its terminating semicolon. Such a `union` does not create a type but does create an unnamed object. An anonymous `union`'s members may be accessed directly in the scope in which the anonymous `union` is declared just as are any other local variables—there is no need to use the dot (`.`) or arrow (`->`) operators.

Anonymous unions have some restrictions. Anonymous unions can contain only data members. All members of an anonymous union must be public. And an anonymous union declared globally (i.e., at global namespace scope) must be explicitly declared static. Figure F.9 illustrates the use of an anonymous union.

```
// Fig. F.9: figF_09.cpp
// Using an anonymous union.
#include <iostream>
using namespace std;

int main()
{
   // declare an anonymous union
   // members integer1, double1 and charPtr share the same space
   union
   {
      int integer1;
      double double1;
      char *charPtr;
   };  // end anonymous union

   // declare local variables
   int integer2 = 1;
   double double2 = 3.3;
   char *char2Ptr = "Anonymous";

   // assign value to each union member
   // successively and print each
   cout << integer2 << ' ';
   integer1 = 2;
   cout << integer1 << endl;

   cout << double2 << ' ';
   double1 = 4.4;
   cout << double1 << endl;

   cout << char2Ptr << ' ';
   charPtr = "union";
   cout << charPtr << endl;
}  // end main
```

```
1 2
3.3 4.4
Anonymous union
```

Fig. F.9 | Using an anonymous union.

F.13 Linkage Specifications

It is possible from a C++ program to call functions written and compiled with a C compiler. As stated in Section 6.17, C++ specially encodes function names for type-safe linkage. C, however, does not encode its function names. Thus, a function compiled in C will not be recognized when an attempt is made to link C code with C++ code, because the C++ code expects a specially encoded function name. C++ enables you to provide **linkage specifications** to inform the compiler that a function was compiled on a C compiler and to prevent the name of the function from being encoded by the C++ compiler. Linkage specifications are useful when large libraries of specialized functions have been developed, and the user either does not have access to the source code for recompilation into C++ or does not have time to convert the library functions from C to C++.

To inform the compiler that one or several functions have been compiled in C, write the function prototypes as follows:

```
extern "C" function prototype    // single function

extern "C"   // multiple functions
{
   function prototypes
}
```

These declarations inform the compiler that the specified functions are not compiled in C++, so name encoding should not be performed on the functions listed in the linkage specification. These functions can then be linked properly with the program. C++ environments normally include the standard C libraries and do not require you to use linkage specifications for those functions.

F.14 Wrap-Up

This appendix introduced a number of C legacy-code topics. We discussed redirecting keyboard input to come from a file and redirecting screen output to a file. We also introduced variable-length argument lists, command-line arguments and processing of unexpected events. You also learned about allocating and resizing memory dynamically.

Summary

Section F.2 Redirecting Input/Output on UNIX/Linux/Mac OS X and Windows Systems

- On many systems—UNIX, LINUX, Mac OS X or Windows systems in particular—it is possible to redirect input to a program and output from a program. Input is redirected from the UNIX, LINUX, Mac OS X or Windows command lines using the redirect input symbol (<) or a pipe (|). Output is redirected from the UNIX, LINUX, Mac OS X or Windows command lines using the redirect output symbol (>) or the append output symbol (>>). The redirect output symbol simply stores the program output in a file, and the append output symbol appends the output to the end of a file.

Section F.3 Variable-Length Argument Lists

- The macros and definitions of the variable arguments header `<cstdarg>` provide the capabilities necessary to build functions with variable-length argument lists.
- An ellipsis (…) in a function prototype indicates that the function receives a variable number of arguments.

- Type `va_list` is suitable for holding information needed by macros `va_start`, `va_arg` and `va_end`. To access the arguments in a variable-length argument list, an object of type `va_list` must be declared.
- Macro `va_start` is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with `va_list` for use by macros `va_arg` and `va_end`.
- Macro `va_arg` expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of `va_arg` modifies the `va_list` object so that the object points to the next argument in the list.
- Macro `va_end` facilitates a normal return from a function whose variable argument list was referred to by the `va_start` macro.

Section F.4 Using Command-Line Arguments

- On many systems—UNIX, LINUX, Mac OS X and Windows in particular—it is possible to pass command-line arguments to `main` by including in `main`'s parameter list the parameters `int argc` and `char *argv[]`. Parameter `argc` is the number of command-line arguments. Parameter `argv` is an array of `char *`'s containing the command-line arguments.

Section F.5 Notes on Compiling Multiple-Source-File Programs

- A function definition must be entirely contained in one file—it cannot span two or more files.
- Global variables must be declared in each file in which they are used.
- Function prototypes can be used to declare functions in other program files. (The `extern` specifier is not required in a function prototype.) This is accomplished by including the function prototype in each file in which the function is invoked, then linking the compiled files together.
- The storage-class specifier `static`, when applied to a global namespace scope variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as internal linkage. Global variables and functions that are not preceded by `static` in their definitions have external linkage—they can be accessed in other files if those files contain proper declarations and/or function prototypes.
- The `static` specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`.
- When building large programs from multiple source files, compiling the program becomes tedious if making small changes to one file means that the entire program must be recompiled. Many systems provide special utilities that recompile only the modified program file. On UNIX systems, the utility is called `make`. Utility `make` reads a file called `Makefile` that contains instructions for compiling and linking the program.

Section F.6 Program Termination with `exit` and `atexit`

- Function `exit` forces a program to terminate as if it had executed normally.
- Function `atexit` registers a function to be called upon normal termination of the program—i.e., either when the program terminates by reaching the end of `main`, or when `exit` is invoked.
- Function `atexit` takes a pointer to a function (e.g., a function name) as an argument. Functions called at program termination cannot have arguments and cannot return a value.
- Function `exit` takes one argument—normally the symbolic constant `EXIT_SUCCESS` or the symbolic constant `EXIT_FAILURE`. If `exit` is called with `EXIT_SUCCESS`, the implementation-defined value for successful termination is returned to the calling environment. If `exit` is called with `EXIT_FAILURE`, the implementation-defined value for unsuccessful termination is returned.

- When function `exit` is invoked, any functions registered with `atexit` are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed and control returns to the host environment.

Section F.7 Type Qualifier `volatile`

- The `volatile` qualifier is used to prevent optimizations of a variable, because it can be modified from outside the program's scope.

Section F.8 Suffixes for Integer and Floating-Point Constants

- C++ provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are `u` or `U` for an `unsigned` integer, `l` or `L` for a `long` integer and `ul` or `UL` for an `unsigned long` integer. If an integer constant is not suffixed, its type is determined by the first type capable of storing a value of that size (first `int`, then `long int`). The floating-point suffixes are `f` or `F` for a `float` and `l` or `L` for a `long double`. A floating-point constant that is not suffixed is of type `double`.

Section F.9 Signal Handling

- The signal-handling library provides the capability to register a function to trap unexpected events with function `signal`. Function `signal` receives two arguments—an integer signal number and a pointer to the signal-handling function.
- Signals can also be generated with function `raise` and an integer argument.

Section F.10 Dynamic Memory Allocation with `calloc` *and* `realloc`

- The general-utilities library (`cstdlib`) provides functions `calloc` and `realloc` for dynamic memory allocation. These functions can be used to create dynamic arrays.
- Function `calloc` receives two arguments—the number of elements (`nmemb`) and the size of each element (`size`)—and initializes the elements of the array to zero. The function returns a pointer to the allocated memory or if the memory is not allocated, the function returns a null pointer.
- Function `realloc` changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`. The original object's contents are not modified, provided that the amount of memory allocated is larger than the amount allocated previously.
- Function `realloc` takes two arguments—a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is null, `realloc` works identically to `malloc`. If `size` is `0` and the pointer received is not null, the memory for the object is freed. Otherwise, if `ptr` is not null and `size` is greater than zero, `realloc` tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory or a null pointer.

Section F.11 Unconditional Branch: `goto`

- The result of the `goto` statement is a change in the program's flow of control. Program execution continues at the first statement after the label in the `goto` statement.
- A label is an identifier followed by a colon. A label must appear in the same function as the `goto` statement that refers to it.

Section F.12 Unions

- A `union` is a data type whose members share the same storage space. The members can be almost any type. The storage reserved for a `union` is large enough to store its largest member. In most cases, `union`s contain two or more data types. Only one member, and thus one data type, can be referenced at a time.

- A `union` is declared in the same format as a structure.
- A `union` can be initialized only with a value of the type of its first member or another object of the same union type.

Section F.13 Linkage Specifications

- C++ enables you to provide linkage specifications to inform the compiler that a function was compiled on a C compiler and to prevent the name of the function from being encoded by the C++ compiler.
- To inform the compiler that one or several functions have been compiled in C, write the function prototypes as follows:

```
extern "C" function prototype    // single function

extern "C"   // multiple functions
{
   function prototypes
}
```

  These declarations inform the compiler that the specified functions are not compiled in C++, so name encoding should not be performed on the functions listed in the linkage specification. These functions can then be linked properly with the program.
- C++ environments normally include the standard C libraries and do not require you to use linkage specifications for those functions.

Self-Review Exercise

F.1 Fill in the blanks in each of the following:

a) Symbol ________ redirects input data from the keyboard to come from a file.
b) The ________ symbol is used to redirect the screen output to be placed in a file.
c) The ________ symbol is used to append the output of a program to the end of a file.
d) A(n) ________ is used to direct the output of a program as the input of another program.
e) A(n) ________ in the parameter list of a function indicates that the function can receive a variable number of arguments.
f) Macro ________ must be invoked before the arguments in a variable-length argument list can be accessed.
g) Macro ________ is used to access the individual arguments of a variable-length argument list.
h) Macro ________ performs termination housekeeping in a function whose variable argument list was referred to by macro `va_start`.
i) Argument ________ of `main` receives the number of arguments in a command line.
j) Argument ________ of `main` stores command-line arguments as character strings.
k) The UNIX utility ________ reads a file called ________ that contains instructions for compiling and linking a program consisting of multiple source files. The utility recompiles a file only if the file (or a header it uses) has been modified since it was last compiled.
l) Function ________ forces a program to terminate execution.
m) Function ________ registers a function to be called upon normal termination of the program.
n) An integer or floating-point ________ can be appended to an integer or floating-point constant to specify the exact type of the constant.
o) Function ________ can be used to register a function to trap unexpected events.
p) Function ________ generates a signal from within a program.

q) Function ________ dynamically allocates memory for an array and initializes the elements to zero.
r) Function ________ changes the size of a block of dynamically allocated memory.
s) A(n) ________ is an entity containing a collection of variables that occupy the same memory, but at different times.
t) The ________ keyword is used to introduce a union definition.

Answers to Self-Review Exercise

F.1 a) redirect input (<). b) redirect output (>). c) append output (>>). d) pipe (|). e) ellipsis (...). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make`, `Makefile`. l) `exit`. m) `atexit`. n) suffix. o) `signal`. p) `raise`. q) `calloc`. r) `realloc`. s) union. t) `union`.

Exercises

F.2 Write a program that calculates the product of a series of integers that are passed to function `product` using a variable-length argument list. Test your function with several calls, each with a different number of arguments.

F.3 Write a program that prints the command-line arguments of the program.

F.4 Write a program that sorts an integer array into ascending order or descending order. The program should use command-line arguments to pass either argument `-a` for ascending order or `-d` for descending order. [*Note:* This is the standard format for passing options to a program in UNIX.]

F.5 Read the manuals for your system to determine what signals are supported by the signal-handling library (`<csignal>`). Write a program with signal handlers for the signals `SIGABRT` and `SIGINT`. The program should test the trapping of these signals by calling function `abort` to generate a signal of type `SIGABRT` and by pressing <*Ctrl*> *C* to generate a signal of type `SIGINT`.

F.6 Write a program that dynamically allocates an array of integers using a function from `<cstdlib>`, not the `new` operator. The size of the array should be input from the keyboard. The elements of the array should be assigned values input from the keyboard. Print the values of the array. Next, reallocate the memory for the array to half of the current number of elements. Print the values remaining in the array to confirm that they match the first half of the values in the original array.

F.7 Write a program that takes two filenames as command-line arguments, reads the characters from the first file one at a time and writes the characters in reverse order to the second file.

F.8 Write a program that uses `goto` statements to simulate a nested looping structure that prints a square of asterisks. The program should use only the following three output statements:

```
cout << '*';
cout << ' ';
cout << endl;
```

F.9 Provide the definition for `union Data` containing `char charcter1`, `short short1`, `long long1`, `float float1` and `double double1`.

F.10 Create `union Integer` with members `char c`, `short s`, `int i` and `long l`. Write a program that inputs values of type `char`, `short`, `int` and `long` and stores the values in `union` variables of type `union Integer`. Each `union` variable should be printed as a `char`, a `short`, an `int` and a `long`. Do the values always print correctly?

F.11 Create `union FloatingPoint` with members `float float1`, `double double1` and `long double longDouble`. Write a program that inputs values of type `float`, `double` and `long double` and stores the values in `union` variables of type `union FloatingPoint`. Each `union` variable should be printed as a `float`, a `double` and a `long double`. Do the values always print correctly?

F.12 Given the `union`

```
union A
{
   double y;
   char *zPtr;
};
```

which of the following are correct statements for initializing the `union`?

a) `A p = b;  // b is of type A`
b) `A q = x;  // x is a double`
c) `A r = 3.14159;`
d) `A s = { 79.63 };`
e) `A t = { "Hi There!" };`
f) `A u = { 3.14159, "Pi" };`
g) `A v = { y = -7.843, zPtr = &x };`

G

UML 2: Additional Diagram Types

G.1 Introduction

If you've read the optional Software Engineering Case Study in Chapters 25–26, you should now have a comfortable grasp of the UML diagram types that we use to model our ATM system. The case study is intended for use in first- or second-semester courses, so we limit our discussion to a concise subset of the UML. The UML 2 provides a total of 13 diagram types. The end of Section 25.3 summarizes the six diagram types that we use in the case study. This appendix lists and briefly defines the seven remaining diagram types.

G.2 Additional Diagram Types

The following are the seven diagram types that we've chosen not to use in our Software Engineering Case Study.

- **Object diagrams** model a "snapshot" of the system by modeling a system's objects and their relationships at a specific point in time. Each object represents an instance of a class from a class diagram, and several objects may be created from one class. For our ATM system, an object diagram could show several distinct `Account` objects side by side, illustrating that they are all part of the bank's account database.
- **Component diagrams** model the **artifacts** and **components**—resources (which include source files)—that make up the system.
- **Deployment diagrams** model the system's runtime requirements (such as the computer or computers on which the system will reside), memory requirements, or other devices the system requires during execution.
- **Package diagrams** model the hierarchical structure of **packages** (which are groups of classes) in the system at compile time and the relationships that exist between the packages.
- **Composite structure diagrams** model the internal structure of a complex object at runtime. New in UML 2, they allow system designers to hierarchically decompose a complex object into smaller parts. Composite structure diagrams are beyond the scope of our case study. They are more appropriate for larger industrial applications, which exhibit complex groupings of objects at execution time.

- **Interaction overview diagrams**, new in UML 2, provide a summary of control flow in the system by combining elements of several types of behavioral diagrams (e.g., activity diagrams, sequence diagrams).
- **Timing diagrams**, also new in UML 2, model the timing constraints imposed on stage changes and interactions between objects in a system.

To learn more about these diagrams and advanced UML topics, please visit our UML Resource Center at `www.deitel.com/UML/`.

Using the Visual Studio Debugger

H

And so shall I catch the fly.
—William Shakespeare

We are built to make mistakes, coded for error.
—Lewis Thomas

What we anticipate seldom occurs; what we least expect generally happens.
—Benjamin Disraeli

He can run but he can't hide.
—Joe Louis

It is one thing to show a man that he is in error, and another to put him in possession of truth.
—John Locke

Objectives

In this appendix you'll learn:

- To set breakpoints and run a program in the debugger.
- To use the **Continue** command to continue execution.
- To use the **Locals** window to view and modify the values of variables.
- To use the **Watch** window to evaluate expressions.
- To use the **Step Into**, **Step Out** and **Step Over** commands to control execution.
- To use the **Autos** window to view variables that are used in the surrounding statements.

Outline

H.1 Introduction

In Chapter 2, you learned that there are two types of errors—compilation errors and logic errors—and you learned how to eliminate compilation errors from your code. Logic errors (also called **bugs**) do not prevent a program from compiling successfully, but can cause the program to produce erroneous results when it runs. Most C++ compiler vendors provide software called a **debugger**, which allows you to monitor the execution of your programs to locate and remove logic errors. The debugger will be one of your most important program development tools. This appendix demonstrates key features of the Visual Studio debugger. Appendix I discusses the features and capabilities of the GNU C++ debugger

H.2 Breakpoints and the Continue Command

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether a logic error exists. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Note that attempting to set a breakpoint at a line of code that is not executable (such as a comment) will actually set the breakpoint at the next executable line of code in that function.

To illustrate the features of the debugger, we use the program listed in Fig. H.3, which creates and manipulates an object of class `Account` (Figs. H.1–H.2). Execution begins in `main` (lines 10–27 of Fig. H.3). Line 12 creates an `Account` object with an initial balance of $50.00. `Account`'s constructor (lines 9–21 of Fig. H.2) accepts one argument, which specifies the `Account`'s initial `balance`. Line 15 of Fig. H.3 outputs the initial account balance using `Account` member function `getBalance`. Line 17 declares a local variable `withdrawalAmount`, which stores a withdrawal amount read from the user. Line 19 prompts the user for the withdrawal amount, and line 20 inputs the amount into `withdrawalAmount`. Line 23 subtracts the withdrawal from the `Account`'s `balance` using its `debit` member function. Finally, line 26 displays the new `balance`.

```
// Fig. H.1: Account.h
// Definition of Account class.
class Account
{
```

Fig. H.1 | Header file for the `Account` class. (Part 1 of 2.)

```
public:
   Account( int ); // constructor initializes balance
   void credit( int ); // add an amount to the account balance
   void debit( int ); // subtract an amount from the account balance
   int getBalance(); // return the account balance
private:
   int balance; // data member that stores the balance
}; // end class Account
```

Fig. H.1 | Header file for the Account class. (Part 2 of 2.)

```
// Fig. H.2: Account.cpp
// Member-function definitions for class Account.
#include <iostream>
using namespace std;

#include "Account.h" // include definition of class Account

// Account constructor initializes data member balance
Account::Account( int initialBalance )
{
   balance = 0; // assume that the balance begins at 0

   // if initialBalance is greater than 0, set this value as the
   // balance of the account; otherwise, balance remains 0
   if ( initialBalance > 0 )
      balance = initialBalance;

   // if initialBalance is negative, print error message
   if ( initialBalance < 0 )
      cout << "Error: Initial balance cannot be negative.\n" << endl;
} // end Account constructor

// credit (add) an amount to the account balance
void Account::credit( int amount )
{
   balance = balance + amount; // add amount to balance
} // end function credit

// debit (subtract) an amount from the account balance
void Account::debit( int amount )
{
   if ( amount <= balance ) // debit amount does not exceed balance
      balance = balance - amount;
   else // debit amount exceeds balance
      cout << "Debit amount exceeded account balance.\n" << endl;
} // end function debit

// return the account balance
int Account::getBalance()
{
```

Fig. H.2 | Definition for the Account class. (Part 1 of 2.)

```
   return balance; // gives the value of balance to the calling function
} // end function getBalance
```

Fig. H.2 | Definition for the Account class. (Part 2 of 2.)

```
// Fig. H.3: figL_03.cpp
// Create and manipulate Account objects.
#include <iostream>
using namespace std;

// include definition of class Account from Account.h
#include "Account.h"

// function main begins program execution
int main()
{
   Account account1( 50 ); // create Account object

   // display initial balance of each object
   cout << "account1 balance: $" << account1.getBalance() << endl;

   int withdrawalAmount; // stores withdrawal amount read from user

   cout << "\nEnter withdrawal amount for account1: "; // prompt
   cin >> withdrawalAmount; // obtain user input
   cout << "\nattempting to subtract " << withdrawalAmount
      << " from account1 balance\n\n";
   account1.debit( withdrawalAmount ); // try to subtract from account1

   // display balances
   cout << "account1 balance: $" << account1.getBalance() << endl;
} // end main
```

Fig. H.3 | Test class for debugging.

Creating a Project in Visual C++ 2010 Express

In the following steps, you'll create a project that includes the code from Figs. H.1–H.3.

1. In Visual C++ 2010 Express select **File > New > Project...** to display the **New Project** dialog.
2. In the left columns, select **Win32**, and in the center column, select **Win32 Console Application**.
3. In the **Name:** field, enter a name for your project and in the **Location:** field, specify where you'd like to save the project on your computer, then click **OK**.
4. In the **Win32 Application Wizard** dialog, click **Next >**.
5. Under **Application type:**, select **Console application**, and under **Additional options:**, uncheck **Precompiled header**, select **Empty project**, then click **Finish**.
6. In the Visual C++'s **Solution Explorer** window, right click your project's **Source Files** folder and select **Add > Existing Item...** to display the **Add Existing Item** dialog.

7. Locate the folder containing the Appendix H example code, select all three files and click **Add**.

Enabling Debug Mode and Inserting Breakpoints

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in Fig. H.3.

1. ***Enabling the debugger.*** The debugger is normally enabled by default. If it isn't enabled, you have to change the settings of the ***Solution Configurations* combo box** (Fig. H.4) in the toolbar. To do this, click the combo box's down arrow, then select **Debug**.

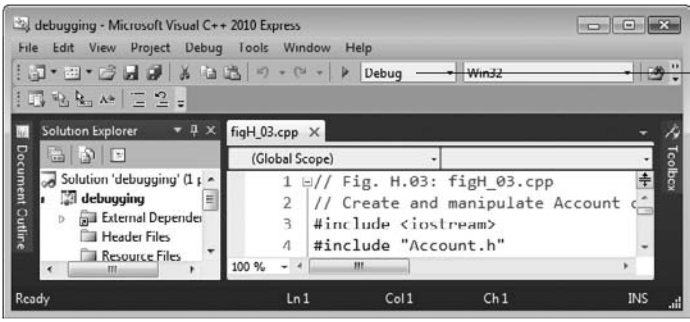


Fig. H.4 | Enabling the debugger.

2. ***Inserting breakpoints.*** Open the file `figH_03.cpp` by double-clicking it in the **Solution Explorer**. To insert a breakpoint, click inside the **margin indicator bar** (the gray margin at the left of the code window in Fig. H.5) next to the line of code at which you wish to break or right click that line of code and select **Breakpoint > Insert Breakpoint**. You can set as many breakpoints as necessary. Set breakpoints at lines 17 and 21 of your code. A red circle appears in the margin indicator bar where you clicked, indicating that a breakpoint has been set (Fig. H.5). When the program runs, the debugger pauses execution at any line that contains a breakpoint. The program is said to be in **break mode** when the debugger pauses the program. Breakpoints can be set before running a program, in break mode and while a program is running.

3. ***Starting to debug.*** After setting breakpoints in the code editor, select **Debug > Build Solution** to compile the program, then select **Debug > Start Debugging** to begin the debugging process. [*Note:* If you do not compile the program first, it will still be compiled when you select **Debug > Start Debugging**.] When you debug a console application, a **Command Prompt** window appears (Fig. H.6) in which you can specify program input and view program output. The debugger enters break mode when execution reaches the breakpoint at line 17.

4. ***Examining program execution.*** Upon entering break mode at the first breakpoint (line 17), the IDE becomes the active window (Fig. H.7). The **yellow arrow** to the left of line 17 indicates that this line contains the next statement to execute.

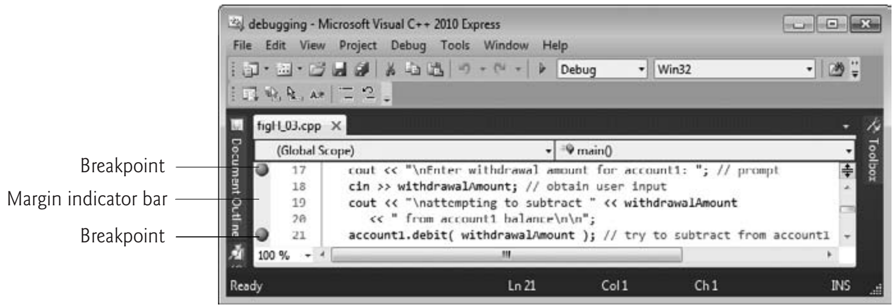


Fig. H.5 | Setting two breakpoints.

Fig. H.6 | **Inventory** program running.

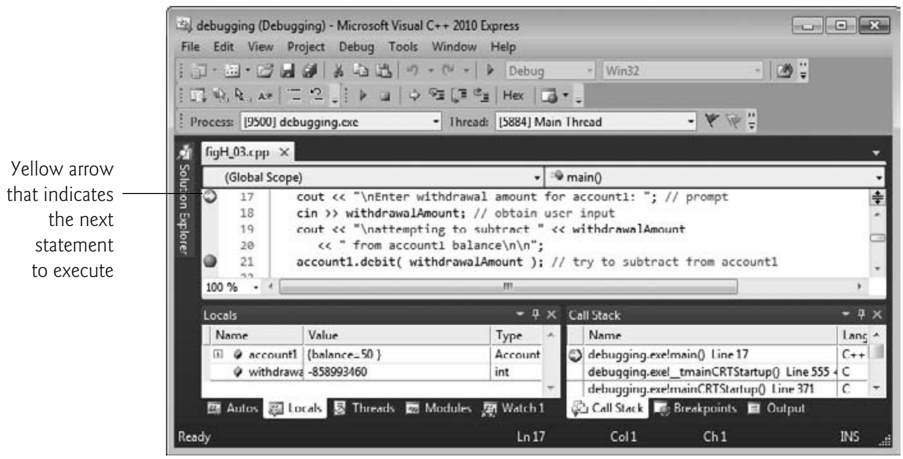
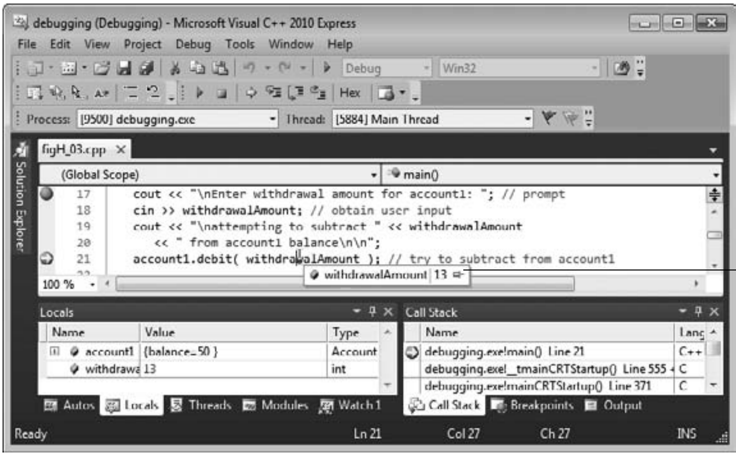


Fig. H.7 | Program execution suspended at the first breakpoint.

5. ***Using the* Continue *command to resume execution.*** To resume execution, select **Debug > Continue**. The **Continue command** resumes program execution until the next breakpoint or the end of `main` is encountered, whichever comes first. The program continues executing and pauses for input at line 18. Enter `13` as the withdrawal amount. The program executes until it stops at the next breakpoint (line 21). Notice that when you place your mouse pointer over the variable name `withdrawalAmount`, the value stored in the variable is displayed in a ***Quick Info* box** (Fig. H.8). As you'll see, this can help you spot logic errors in your programs.



Quick Info box

Fig. H.8 | *Quick Info* box showing the value of a variable.

6. ***Setting a breakpoint at* `main`*'s closing brace.*** Set a breakpoint at line 25 in the source code by clicking in the margin indicator bar to the left of line 25. This will prevent the program from closing immediately after displaying its result. When there are no more breakpoints at which to suspend execution, the program will execute to completion and the **Command Prompt** window will close. If you do not set this breakpoint, you won't be able to view the program's output before the console window closes.
7. ***Continuing program execution.*** Use the **Debug > Continue** command to execute the code up to the next breakpoint. The program displays the result of its calculation (Fig. H.9).

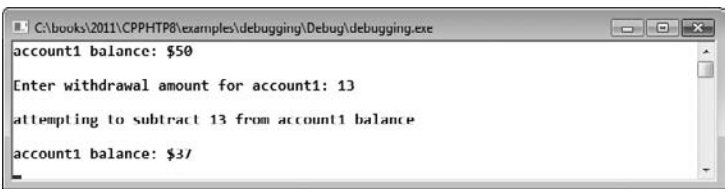


Fig. H.9 | Program output.

8. ***Removing a breakpoint.*** Click the breakpoint in the margin indicator bar.
9. ***Finishing program execution.*** Select **Debug > Continue** to execute the program to completion.

In this section, you learned how to enable the debugger and set breakpoints so that you can examine the results of code while a program is running. You also learned how to continue execution after a program suspends execution at a breakpoint and how to remove breakpoints.

H.3 Locals and Watch Windows

In the preceding section, you learned that the *Quick Info* feature allows you to examine a variable's value. In this section, you'll learn to use the **Locals window** to assign new values to variables while your program is running. You'll also use the **Watch window** to examine the value of more complex expressions.

1. ***Inserting breakpoints.*** Clear the existing breakpoints. Then, set a breakpoint at line 21 in the source code by clicking in the margin indicator bar to the left of line 21 (Fig. H.10). Set another breakpoint at line 24 by clicking in the margin indicator bar to the left of line 24.

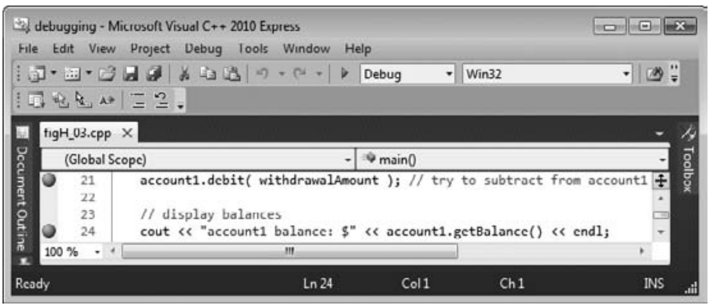


Fig. H.10 | Setting breakpoints at lines 25 and 28.

2. ***Starting debugging.*** Select **Debug > Start**. Type 13 at the **Enter withdrawal amount for account1:** prompt and press *Enter* so that your program reads the value you just entered. The program executes until the breakpoint at line 21.

3. ***Suspending program execution.*** The debugger enters break mode at line 21 (Fig. H.11). At this point, line 18 has input the `withdrawalAmount` that you entered (13), lines 19–20 have output that the program will attempt to withdraw money and line 21 is the next statement that will execute.

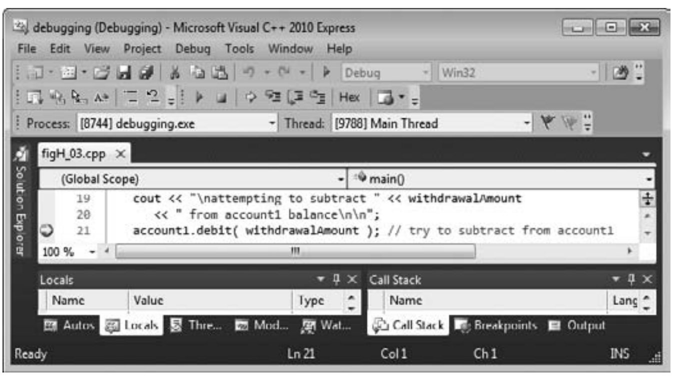


Fig. H.11 | Program execution suspended when debugger reaches the breakpoint at line 25.

4. *Examining data.* In break mode, you can explore the values of your local variables using the debugger's **Locals** window, which is normally displayed at the bottom left of the IDE when you are debugging. If it is not shown, you can view the **Locals** window, select **Debug > Windows > Locals**. Figure H.12 shows the values for `main`'s local variables `account1` and `withdrawalAmount` (`13`).

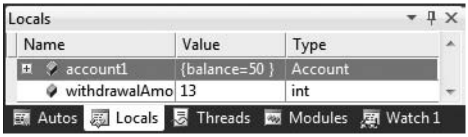


Fig. H.12 | Examining variable `withdrawalAmount`.

5. ***Evaluating arithmetic and boolean expressions.*** You can evaluate arithmetic and boolean expressions using the **Watch** window. You can display up to four **Watch** windows. Select **Debug > Windows > Watch 1**. In the first row of the **Name** column, type `(withdrawalAmount + 3) * 5`, then press *Enter*. The value of this expression (`80` in this case) is displayed in the **Value** column (Fig. H.13). In the next row of the **Name** column, type `withdrawalAmount == 3`, then press *Enter*. This expression determines whether the value of `withdrawalAmount` is `3`. Expressions containing the `==` operator (or any other relational or equality operator) are treated as `bool` expressions. The value of the expression in this case is `false` (Fig. H.13), because `withdrawalAmount` currently contains `13`, not `3`.

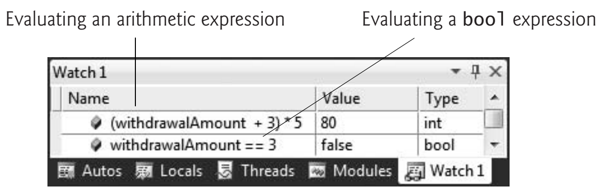


Fig. H.13 | Examining the values of expressions.

6. *Resuming execution.* Select **Debug > Continue** to resume execution. Line 21 debits the account by the withdrawal amount, and the debugger reenters break mode at line 24. Select **Debug > Windows > Locals** or click the **Locals** tab at the bottom of Visual Studio to redisplay the **Locals** window. The updated `balance` value in `account1` is now displayed in red (Fig. H.14) to indicate that it has been modified since the last breakpoint. Click the plus box to the left of `account1` in the **Name** column of the **Locals** window. This allows you to view each of `account1`'s data member values individually—this is particularly useful for objects that have several data members.

7. *Modifying values.* Based on the value input by the user (`13`), the account balance output by the program should be `$37`. However, you can use the **Locals** window to change the values of variables during the program's execution. This can be

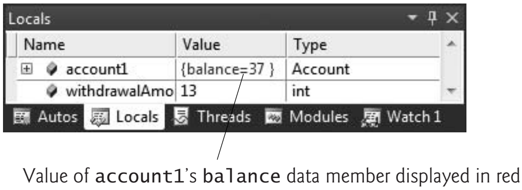


Fig. H.14 | Displaying the value of local variables.

valuable for experimenting with different values and for locating logic errors. In the **Locals** window, expand the account1 node and double click the **Value** field in the `balance` row to select the value `37`. Type `33`, then press *Enter*. The debugger changes the value of `balance` and displays its new value in red (Fig. H.15).

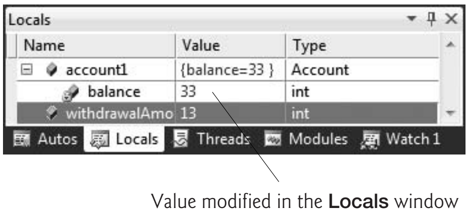


Fig. H.15 | Modifying the value of a variable.

8. ***Setting a breakpoint at at*** **`main`*'s closing brace.*** Set a breakpoint at line 25 in the source code to prevent the program from closing immediately after displaying its result. If you do not set this breakpoint, you won't be able to view the program's output before the console window closes.
9. ***Viewing the program result.*** Select **Debug > Continue** to continue program execution. Function `main` executes until the `return` statement in line 29 and displays the result. Notice that the result is `$33` (Fig. H.16). This shows that *Step 7* changed the value of `balance` from the calculated value (`37`) to `33`.

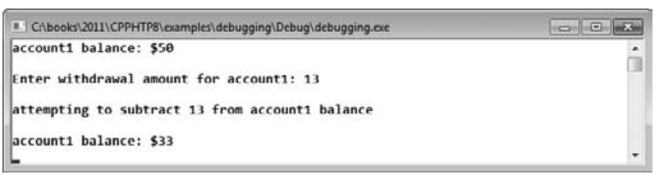


Fig. H.16 | Output displayed after modifying the `account1` variable.

10. ***Stopping the debugging session.*** Select **Debug > Stop Debugging**. This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Watch** and **Locals** windows to evaluate arithmetic and boolean expressions. You also learned how to modify the value of a variable during your program's execution.

H.4 Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands

Sometimes executing a program line by line can help you verify that a function's code executes correctly, and can help you find and fix logic errors. The commands you learn in this section allow you to execute a function line by line, execute all the statements of a function at once or execute only the remaining statements of a function (if you've already executed some statements within the function).

1. *Setting a breakpoint.* Set a breakpoint at line 21 by clicking in the margin indicator bar to the left of the line.
2. *Starting the debugger.* Select **Debug > Start.** Enter the value `13` at the **Enter withdrawal amount for account1:** prompt. Execution will halt when the program reaches the breakpoint at line 21.
3. *Using the* **Step Into** *command.* The **Step Into command** executes the next statement in the program (line 21), then immediately halts. If that statement is a function call (as is the case here), control transfers into the called function. This enables you to execute each statement inside the function individually to confirm the function's execution. Select **Debug > Step Into** to enter the `debit` function. Then, Select **Debug > Step Into** again so the yellow arrow is positioned at line 31 of `Account.cpp`, as shown in Fig. H.17.

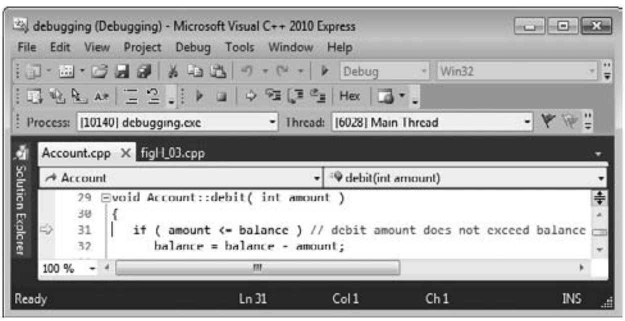


Fig. H.17 | Stepping into the `debit` function.

4. *Using the* **Step Over** *command.* Select **Debug > Step Over** to execute the current statement (line 31 in Fig. H.17) and transfer control to line 32 (Fig. H.18). The **Step Over command** behaves like the **Step Into** command when the next statement to execute does not contain a function call. You'll see how the **Step Over** command differs from the **Step Into** command in *Step 10*.
5. *Using the* **Step Out** *command.* Select **Debug > Step Out** to execute the remaining statements in the function and return control to the next executable statement (line 28 in Fig. H.3). Often, in lengthy functions, you'll want to look at a few key lines of code, then continue debugging the caller's code. The **Step Out command** enables you to continue program execution in the caller without having to step through the entire called function line by line.

Control transfers to the next statement

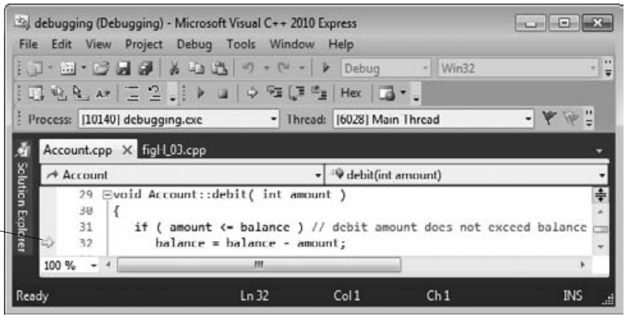


Fig. H.18 | Stepping over a statement in the `debit` function.

6. ***Setting a breakpoint.*** Set a breakpoint at the end of `main` at line 25 of Fig. H.3. You'll make use of this breakpoint in the next step.
7. ***Using the* Continue *command.*** Select **Debug > Continue** to execute until the next breakpoint is reached at line 25. Using the **Continue** command is useful when you wish to execute all the code up to the next breakpoint.
8. ***Stopping the debugger.*** Select **Debug > Stop Debugging** to end the debugging session. This will close the **Command Prompt** window.
9. ***Starting the debugger.*** Before we can demonstrate the next debugger feature, you must start the debugger again. Start it, as you did in *Step 2*, and enter `13` in response to the prompt. The debugger enters break mode at line 21.
10. ***Using the* Step Over *command.*** Select **Debug > Step Over** (Fig. H.19) Recall that this command behaves like the **Step Into** command when the next statement to execute does not contain a function call. If the next statement to execute contains a function call, the called function executes in its entirety (without pausing execution at any statement inside the function), and the yellow arrow advances to the next executable line (after the function call) in the current function. In this case, the debugger executes line 21, located in `main` (Fig. H.3). Line 21 calls the `debit` function. The debugger then pauses execution at line 24, the next executable line in the current function, `main`.

The `debit` function call executes to completion when the **Step Over** command is selected

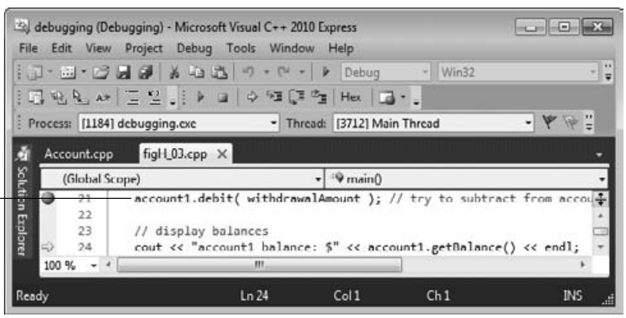


Fig. H.19 | Using the debugger's **Step Over** command.

11. *Stopping the debugger.* Select **Debug > Stop Debugging.** This will close the **Command Prompt** window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's **Step Into** command to debug functions called during your program's execution. You saw how the **Step Over** command can be used to step over a function call. You used the **Step Out** command to continue execution until the end of the current function. You also learned that the **Continue** command continues execution until another breakpoint is found or the program exits.

H.5 Autos Window

The **Autos window** displays the variables used in the previous statement executed (including the return value of a function, if there is one) and the variables in the next statement to execute.

1. *Setting breakpoints.* Set breakpoints at lines 10 and 18 in `main` by clicking in the margin indicator bar.
2. *Using the* **Autos** *window.* Start the debugger by selecting **Debug > Start.** When the debugger enters break mode at line 10, open the **Autos** window (Fig. H.20). Since we are just beginning the program's execution, the **Autos** window lists only the variables in the next statement that will execute—in this case, the `account1` object, its value and its type. Viewing the values stored in an object lets you verify that your program is manipulating these variables correctly. Notice that `account1` contains a large negative value. This value, which may be different each time the program executes, is `account1`'s uninitialized value. This unpredictable (and often undesirable) value demonstrates why it is important to initialize all C++ variables before they are used.

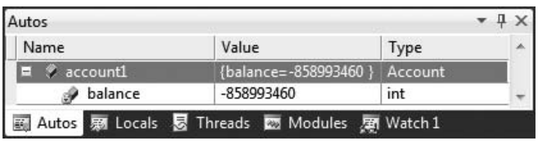


Fig. H.20 | **Autos** window displaying the state of `account1` object.

3. *Using the* **Step Over** *command.* Select **Debug > Step Over** to execute line 10. The **Autos** window updates the value of `account1`'s `balance` data member (Fig. H.21) after it is initialized. The value of `balance` is displayed in red to indicate that it just changed.

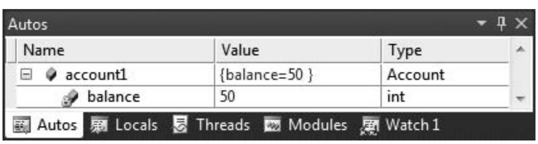


Fig. H.21 | **Autos** window displaying the state of `account1` object after initialization.

4. *Continuing execution.* Select **Debug > Continue** to execute the program until the second breakpoint at line 18. The **Autos** window displays uninitialized local variable `withdrawalAmount` (Fig. H.22), which has a large negative value.

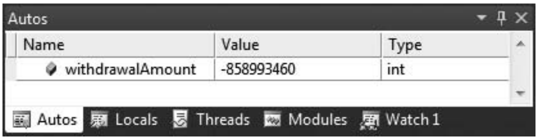


Fig. H.22 | **Autos** window displaying local variable `withdrawalAmount`.

5. *Entering data.* Select **Debug > Step Over** to execute line 18. At the program's input prompt, enter a value for the withdrawal amount. The **Autos** window updates the value of local variable `withdrawalAmount` with the value you entered (Fig. H.23).

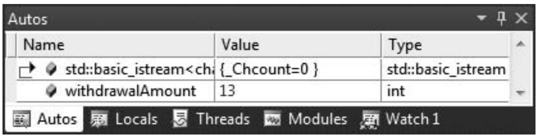


Fig. H.23 | **Autos** window displaying updated local variable `withdrawalAmount`.

6. *Stopping the debugger.* Select **Debug > Stop Debugging** to end the debugging session. Remove all remaining breakpoints.

H.6 Wrap-U

In this appendix, you learned how to insert, disable and remove breakpoints in the Visual Studio debugger. Breakpoints allow you to pause program execution so you can examine variable values. This capability will help you locate and fix logic errors in your programs. You saw how to use the **Locals** and **Watch** windows to examine the value of an expression and how to change the value of a variable. You also learned debugger commands **Step Into**, **Step Over**, **Step Out** and **Continue** that can be used to determine whether a function is executing correctly. Finally, you learned how to use the **Autos** window to examine variables used specifically in the previous and next commands.

Summary

Section H.1 Introduction

- Most C++ compiler vendors provide software called a debugger, which allows you to monitor the execution of your programs to locate and remove logic errors.
- Breakpoints are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses.

- The debugger is enabled by default. If it isn't enabled, you have to change the settings of the *Solution Configurations* combo box.

Section H.2 Breakpoints and the Continue *Command*

- To insert a breakpoint, either click inside the margin indicator bar next to the line of code or right click that line of code and select **Breakpoint > Insert Breakpoint**. A red circle appears where you clicked, indicating that a breakpoint has been set.
- When the program runs, it suspends execution at any line that contains a breakpoint. It is then said to be in break mode.
- A yellow arrow indicates that this line contains the next statement to execute.
- When you place your mouse pointer over a variable name, the value that the variable stores is displayed in a *Quick Info* box.
- To disable a breakpoint, right click a line of code on which a breakpoint has been set and select **Breakpoint > Disable Breakpoint**. The disabled breakpoint is indicated by a hollow circle.
- To remove a breakpoint that you no longer need, right click a line of code on which a breakpoint has been set and select **Breakpoint > Delete Breakpoint**. You also can remove a breakpoint by clicking the circle in the margin indicator bar.

Section H.3 Locals *and* Watch *Windows*

- Once the program has entered break mode, you can explore the values of your variables using the debugger's **Locals** window. To view the **Locals** window, select **Debug > Windows > Locals**.
- You can evaluate arithmetic and boolean expressions using the **Watch** window.
- Updated variables are displayed in red to indicate that they've been modified.
- Clicking the plus box next to an object in the **Name** column of the **Locals** window allows you to view each of object's data member values individually.
- You can click the **Value** field of a variable to change its value in the **Locals** window.

Section H.4 Using the Step Into, Step Over, Step Out *and* Continue *Commands*

- The **Step Into** command executes the next statement (the yellow highlighted line) in the program. If the next statement is to execute a function call and you select **Step Into**, control is transferred to the called function.
- The **Step Over** command behaves like the **Step Into** command when the next statement to execute does not contain a function call. If the next statement to execute contains a function call, the called function executes in its entirety, and the yellow arrow advances to the next executable line in the current function.
- The **Step Over** command executes the remaining statements in the function and returns control to the function call.
- The **Continue** command will execute any statements between the next executable statement and the next breakpoint or the end of `main`, whichever comes first.

Section H.5 Autos *Window*

- The **Autos** window allows you to view the contents of the variables used in the last statement that was executed. The **Autos** window also lists the values in the next statement to be executed.

Self-Review Exercises

H.1 Fill in the blanks in each of the following statements:

a) When the debugger suspends program execution at a breakpoint, the program is said to be in ________ mode.

b) The ________ feature in Visual Studio 2005 allows you to look at the value of a variable by positioning the mouse over the variable name in the code.

c) You can examine the value of an expression by using the debugger's ________ window.

d) The ________ command behaves like the **Step Into** command when the next statement to execute does not contain a function call.

H.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When program execution suspends at a breakpoint, the next statement to be executed is the statement after the breakpoint.

b) When a variable's value is changed, it becomes yellow in the **Autos** and **Locals** windows.

c) During debugging, the **Step Out** command executes the remaining statements in the current function and returns program control to the place where the function was called.

Answers to Self-Review Exercises

H.1 a) break. b) *Quick Info* box. c) **Watch**. d) **Step Over**.

H.2 a) False. When program execution suspends at a breakpoint, the next statement to be executed is the statement at the breakpoint. b) False. A variable turns red when its value is changed. c) True.

Using the GNU C++ Debugger

I

And so shall I catch the fly.
—William Shakespeare

We are built to make mistakes, coded for error.
—Lewis Thomas

What we anticipate seldom occurs; what we least expect generally happens.
—Benjamin Disraeli

Objectives

In this appendix you'll learn:

- To use the `run` command to run a program in the debugger.
- To use the `break` command to set a breakpoint.
- To use the `continue` command to continue execution.
- To use the `print` command to evaluate expressions.
- To use the `set` command to change variable values during program execution.
- To use the `step`, `finish` and `next` commands to control execution.
- To use the `watch` command to see how a data member is modified during program execution.
- To use the `delete` command to remove a breakpoint or a watchpoint.

Outline

I.1 Introduction

In Chapter 2, you learned that there are two types of errors—compilation errors and logic errors—and you learned how to eliminate compilation errors from your code. Logic errors do not prevent a program from compiling successfully, but they can cause the program to produce erroneous results when it runs. GNU includes software called a **debugger** that allows you to monitor the execution of your programs so you can locate and remove logic errors.

The debugger is one of the most important program development tools. Many IDEs provide their own debuggers similar to the one included in GNU or provide a graphical user interface to GNU's debugger. This appendix demonstrates key features of GNU's debugger. Appendix H discusses the features and capabilities of the Visual Studio debugger. Our C++ Resource Center (`www.deitel.com/cplusplus/`) provides links to tutorials that can help students and instructors familiarize themselves with the debuggers provided with various other development tools.

I.2 Breakpoints and the `run`, `stop`, `continue` and `print` Commands

We begin our study of the debugger by investigating **breakpoints**, which are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether a logic error exists. For example, you can examine the value of a variable that stores the result of a calculation to determine whether the calculation was performed correctly. Note that attempting to set a breakpoint at a line of code that is not executable (such as a comment) will actually set the breakpoint at the next executable line of code in that function.

To illustrate the features of the debugger, we use class `Account` (Figs. I.1–I.2) and the program listed in Fig. I.3, which creates and manipulates an object of class `Account`. Execution begins in `main` (lines 12–30 of Fig. I.3). Line 10 creates an `Account` object with an initial balance of $50.00. `Account`'s constructor (lines 8–20 of Fig. I.2) accepts one argument, which specifies the `Account`'s initial `balance`. Line 13 of Fig. I.3 outputs the initial account balance using `Account` member function `getBalance`. Line 15 declares a local variable `withdrawalAmount` which stores a withdrawal amount input by the user. Line 17 prompts the user for the withdrawal amount; line 18 inputs the `withdrawalAmount`. Line 21 uses the `Account`'s `debit` member function to subtract the `withdrawalAmount` from the `Account`'s `balance`. Finally, line 24 displays the new `balance`.

```cpp
// Fig. I.1: Account.h
// Definition of Account class.
class Account
{
public:
   Account( int ); // constructor initializes balance
   void credit( int ); // add an amount to the account balance
   void debit( int ); // subtract an amount from the account balance
   int getBalance(); // return the account balance
private:
   int balance; // data member that stores the balance
}; // end class Account
```

Fig. I.1 | Header file for the Account class.

```cpp
// Fig. I.2: Account.cpp
// Member-function definitions for class Account.
#include <iostream>
#include "Account.h" // include definition of class Account
using namespace std;

// Account constructor initializes data member balance
Account::Account( int initialBalance )
{
   balance = 0; // assume that the balance begins at 0

   // if initialBalance is greater than 0, set this value as the
   // balance of the Account; otherwise, balance remains 0
   if ( initialBalance > 0 )
      balance = initialBalance;

   // if initialBalance is negative, print error message
   if ( initialBalance < 0 )
      cout << "Error: Initial balance cannot be negative.\n" << endl;
} // end Account constructor

// credit (add) an amount to the account balance
void Account::credit( int amount )
{
   balance = balance + amount; // add amount to balance
} // end function credit

// debit (subtract) an amount from the account balance
void Account::debit( int amount )
{
   if ( amount <= balance ) // debit amount does not exceed balance
      balance = balance - amount;
   else // debit amount exceeds balance
      cout << "Debit amount exceeded account balance.\n" << endl;
} // end function debit

```

Fig. I.2 | Definition for the Account class. (Part 1 of 2.)

```
// return the account balance
int Account::getBalance()
{
   return balance; // gives the value of balance to the calling function
} // end function getBalance
```

Fig. I.2 | Definition for the `Account` class. (Part 2 of 2.)

```
1  // Fig. I.3: figI_03.cpp
2  // Create and manipulate Account objects.
3  #include <iostream>
4  #include "Account.h"
5  using namespace std;
6
7  // function main begins program execution
8  int main()
9  {
10    Account account1( 50 ); // create Account object
11
12    // display initial balance of each object
13    cout << "account1 balance: $" << account1.getBalance() << endl;
14
15    int withdrawalAmount; // stores withdrawal amount read from user
16
17    cout << "\nEnter withdrawal amount for account1: "; // prompt
18    cin >> withdrawalAmount; // obtain user input
19    cout << "\nattempting to subtract " << withdrawalAmount
20       << " from account1 balance\n\n";
21    account1.debit( withdrawalAmount ); // try to subtract from account1
22
23    // display balances
24    cout << "account1 balance: $" << account1.getBalance() << endl;
25 } // end main
```

Fig. I.3 | Test class for debugging.

In the following steps, you'll use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in line 15 of Fig. I.3.

1. *Compiling the program for debugging.* To use the debugger, you must compile your program with the **`-g`** option, which generates additional information that the debugger needs to help you debug your programs. To do so, type

```
g++ -g -o figI_03 figI_03.cpp Account.cpp
```

2. *Starting the debugger.* Type gdb `figI_03` (Fig. I.4). The **gdb command** starts the debugger and displays the `(gdb)` prompt at which you can enter commands.
3. *Running a program in the debugger.* Run the program through the debugger by typing **`run`** (Fig. I.5). If you do not set any breakpoints before running your program in the debugger, the program will run to completion.
4. *Inserting breakpoints using the GNU debugger.* Set a breakpoint at line 13 of `FigI_03.cpp` by typing `break 13`. The **`break` command** inserts a breakpoint at

```
$ gdb FigI_03
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db
library "/ lib/tls/i686/cmov/libthread_db.so.1".

(gdb)
```

Fig. I.4 | Starting the debugger to run the program.

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

Fig. I.5 | Running the program with no breakpoints set.

the line number specified as its argument (i.e., `13`). You can set as many breakpoints as necessary. Each breakpoint is identified by the order in which it was created. The first breakpoint is known as `Breakpoint 1`. Set another breakpoint at line 21 by typing `break 21` (Fig. I.6). This new breakpoint is known as `Breakpoint 2`. When the program runs, it suspends execution at any line that contains a breakpoint and the debugger enters **break mode**. Breakpoints can be set even after the debugging process has begun. [*Note:* If you do not have a numbered listing for your code, you can use the **`list` command** to output your code with line numbers. For more information about the `list` command type **`help`** `list` from the `gdb` prompt.]

```
(gdb) break 13
Breakpoint 1 at 0x80486f6: file FigI_03.cpp, line 13.
(gdb) break 21
Breakpoint 2 at 0x8048799: file FigI_03.cpp, line 21.
(gdb)
```

Fig. I.6 | Setting two breakpoints in the program.

5. ***Running the program and beginning the debugging process.*** Type `run` to execute your program and begin the debugging process (Fig. I.7). The debugger enters

break mode when execution reaches the breakpoint at line 13. At this point, the debugger notifies you that a breakpoint has been reached and displays the source code at that line (13), which will be the next statement to execute.

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:13
13        cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

Fig. I.7 | Running the program until it reaches the first breakpoint.

6. ***Using the `continue` command to resume execution.*** Type `continue`. The **`continue` command** causes the program to continue running until the next breakpoint is reached (line 21). Enter `13` at the prompt. The debugger notifies you when execution reaches the second breakpoint (Fig. I.8). Note that `figI_03`'s normal output appears between messages from the debugger.

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 2, main () at FigI_03.cpp:21
21        account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.8 | Continuing execution until the second breakpoint is reached.

7. ***Examining a variable's value.*** Type `print withdrawalAmount` to display the current value stored in the `withdrawalAmount` variable (Fig. I.9). The **`print` command** allows you to peek inside the computer at the value of one of your variables. This can be used to help you find and eliminate logic errors in your code. In this case, the variable's value is `13`—the value you entered that was assigned to variable `withdrawalAmount` in line 18 of Fig. I.3. Next, use `print` to display the contents of the `account1` object. When an object is displayed with `print`, braces are placed around the object's data members. In this case, there is a single data member—`balance`—which has a value of `50`.
8. ***Using convenience variables.*** When you use `print`, the result is stored in a convenience variable such as `$1`. Convenience variables are temporary variables created by the debugger that are named using a dollar sign followed by an integer. Convenience variables can be used to perform arithmetic and evaluate boolean expressions. Type `print $1`. The debugger displays the value of `$1` (Fig. I.10),

```
(gdb) print withdrawalAmount
$2 = 13
(gdb) print account1
$3 = {balance = 50}
(gdb)
```

Fig. I.9 | Printing the values of variables.

```
(gdb) print $1
$3 = 13
(gdb)
```

Fig. I.10 | Printing a convenience variable.

which contains the value of `withdrawalAmount`. Note that printing the value of `$1` creates a new convenience variable—`$3`.

9. ***Continuing program execution.*** Type `continue` to continue the program's execution. The debugger encounters no additional breakpoints, so it continues executing and eventually terminates (Fig. I.11).

```
(gdb) continue
Continuing.
account1 balance: $37

Program exited normally.
(gdb)
```

Fig. I.11 | Finishing execution of the program.

10. ***Removing a breakpoint.*** You can display a list of all of the breakpoints in the program by typing **`info break`**. To remove a breakpoint, type **`delete`**, followed by a space and the number of the breakpoint to remove. Remove the first breakpoint by typing `delete 1`. Remove the second breakpoint as well. Now type `info break` to list the remaining breakpoints in the program. The debugger should indicate that no breakpoints are set (Fig. I.12).

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080486f6 in main at FigI_03.cpp:13
        breakpoint already hit 1 time
2   breakpoint     keep y   0x08048799 in main at FigI_03.cpp:21
        breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Fig. I.12 | Viewing and removing breakpoints.

11. ***Executing the program without breakpoints.*** Type `run` to execute the program. Enter the value `13` at the prompt. Because you successfully removed the two breakpoints, the program's output is displayed without the debugger entering break mode (Fig. I.13).

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

Fig. I.13 | Program executing with no breakpoints set.

12. ***Using the `quit` command.*** Use the **`quit` command** to end the debugging session (Fig. I.14). This command causes the debugger to terminate.

```
(gdb) quit
$
```

Fig. I.14 | Exiting the debugger using the `quit` command.

In this section, you used the `gdb` command to start the debugger and the `run` command to start debugging a program. You set a breakpoint at a particular line number in the `main` function. The `break` command can also be used to set a breakpoint at a line number in another file or at a particular function. Typing `break`, then the filename, a colon and the line number will set a breakpoint at a line in another file. Typing `break`, then a function name will cause the debugger to enter the break mode whenever that function is called.

Also in this section, you saw how the `help list` command will provide more information on the `list` command. If you have any questions about the debugger or any of its commands, type `help` or `help` followed by the command name for more information.

Finally, you examined variables with the `print` command and remove breakpoints with the `delete` command. You learned how to use the `continue` command to continue execution after a breakpoint is reached and the `quit` command to end the debugger.

I.3 `print` and `set` Commands

In the preceding section, you learned how to use the debugger's `print` command to examine the value of a variable during program execution. In this section, you'll learn how to use the `print` command to examine the value of more complex expressions. You'll also learn the **`set` command**, which allows you to assign new values to variables. We assume you are working in the directory containing this appendix's examples and have compiled for debugging with the `-g` compiler option.

1. ***Starting debugging.*** Type `gdb figI_03` to start the GNU debugger.
2. ***Inserting a breakpoint.*** Set a breakpoint at line 21 in the source code by typing `break 21` (Fig. I.15).

```
(gdb) break 21
Breakpoint 1 at 0x8048799: file FigI_03.cpp, line 21.
(gdb)
```

Fig. I.15 | Setting a breakpoint in the program.

3. ***Running the program and reaching a breakpoint.*** Type `run` to begin the debugging process (Fig. I.16). This will cause `main` to execute until the breakpoint at line 21 is reached. This suspends program execution and switches the program into break mode. The statement in line 25 is the next statement that will execute.

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at FigI_03.cpp:21
21        account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.16 | Running the program until the breakpoint at line 25 is reached.

4. ***Evaluating arithmetic and boolean expressions.*** Recall from Section I.2 that once the debugger enters break mode, you can explore the values of the program's variables using the `print` command. You can also use `print` to evaluate arithmetic and boolean expressions. Type `print withdrawalAmount - 2`. This expression returns the value `11` (Fig. I.17), but does not actually change the value of `withdrawalAmount`. Type `print withdrawalAmount == 11`. Expressions containing the `==` symbol return `bool` values. The value returned is `false` (Fig. I.17) because `withdrawalAmount withdrawalAmount` still contains `13`.

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

Fig. I.17 | Printing expressions with the debugger.

5. ***Modifying values.*** You can change the values of variables during the program's execution in the debugger. This can be valuable for experimenting with different values and for locating logic errors. You can use the debugger's `set` command to change a variable's value. Type `set withdrawalAmount = 42` to change the value of `withdrawalAmount`, then type `print withdrawalAmount` to display its new value (Fig. I.18).

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

Fig. I.18 | Setting the value of a variable while in break mode.

6. ***Viewing the program result.*** Type `continue` to continue program execution. Line 21 of Fig. I.3 executes, passing `withdrawalAmount` to `Account` member function `debit`. Function `main` then displays the new balance. Note that the result is `$8` (Fig. I.19). This shows that the preceding step changed the value of `withdrawalAmount` from the value `13` that you input to `42`.

```
(gdb) continue
Continuing.
account1 balance: $8

Program exited normally.
(gdb)
```

Fig. I.19 | Using a modified variable in the execution of a program.

7. ***Using the `quit` command.*** Use the `quit` command to end the debugging session (Fig. I.20). This command causes the debugger to terminate.

```
(gdb) quit
$
```

Fig. I.20 | Exiting the debugger using the `quit` command.

In this section, you used the debugger's `print` command to evaluate arithmetic and boolean expressions. You also learned how to use the `set` command to modify the value of a variable during your program's execution.

I.4 Controlling Execution Using the `step`, `finish` and `next` Commands

Sometimes you'll need to execute a program line by line to find and fix errors. Walking through a portion of your program this way can help you verify that a function's code executes correctly. The commands in this section allow you to execute a function line by line, execute all the statements of a function at once or execute only the remaining statements of a function (if you've already executed some statements within the function).

1. ***Starting the debugger.*** Start the debugger by typing `gdb figI_03`.
2. ***Setting a breakpoint.*** Type `break 21` to set a breakpoint at line 21.
3. ***Running the program.*** Run the program by typing `run`, then enter 13 at the prompt. After the program displays its two output messages, the debugger indicates that the breakpoint has been reached and displays the code at line 21. The debugger then pauses and wait for the next command to be entered.
4. ***Using the `step` command.*** The **`step` command** executes the next statement in the program. If the next statement to execute is a function call, control transfers to the called function. The `step` command enables you to enter a function and study its individual statements. For instance, you can use the `print` and `set` commands to view and modify the variables within the function. Type `step` to enter the `debit` member function of class `Account` (Fig. I.2). The debugger indicates that the step has been completed and displays the next executable statement (Fig. I.21)—in this case, line 31 of class `Account` (Fig. I.2).

```
(gdb) step
Account::debit (this=0xbff81700, amount=13) at Account.cpp:31
31         if ( amount <= balance ) // debit amount does not exceed balance
(gdb)
```

Fig. I.21 | Using the `step` command to enter a function.

5. ***Using the `finish` command.*** After you've stepped into the `debit` member function, type **`finish`**. This command executes the remaining statements in the function and returns control to the place where the function was called. The `finish` command executes the remaining statements in member function `debit`, then pauses at line 24 in `main` (Fig. I.22). In lengthy functions, you may want to look at a few key lines of code, then continue debugging the caller's code. The `finish` command is useful for situations in which you do not want to step through the remainder of a function line by line.

```
(gdb) finish
Run till exit from #0  Account::debit (this=0xbff81700, amount=13) at
   Account.cpp:31
0x080487a9 in main () at FigI_03.cpp:21
21         account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.22 | Using the `finish` command to complete execution of a function and return to the calling function.

6. ***Using the `continue` command to continue execution.*** Enter the `continue` command to continue execution until the program terminates.

7. ***Running the program again.*** Breakpoints persist until the end of the debugging session in which they are set. So, the breakpoint you set in *Step 2* is still set. Type `run` to run the program and enter `13` at the prompt. As in *Step 3*, the program runs until the breakpoint at line 21 is reached, then the debugger pauses and waits for the next command (Fig. I.23).

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance


Breakpoint 1, main () at FigI_03.cpp:21
21          account1.debit( withdrawalAmount ); // try to subtract from account1
(gdb)
```

Fig. I.23 | Restarting the program.

8. ***Using the* `next` *command.*** Type **`next`**. This command behaves like the `step` command, except when the next statement to execute contains a function call. In that case, the called function executes in its entirety and the program advances to the next executable line after the function call (Fig. I.24). In *Step 4*, the `step` command enters the called function. In this example, the `next` command executes `Account` member function `debit`, then the debugger pauses at line 24.

```
(gdb) next
24          cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

Fig. I.24 | Using the `next` command to execute a function in its entirety.

9. ***Using the* `quit` *command.*** Use the `quit` command to end the debugging session (Fig. I.25). While the program is running, this command causes the program to immediately terminate rather than execute the remaining statements in `main`.

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

Fig. I.25 | Exiting the debugger using the `quit` command.

In this section, you used the debugger's `step` and `finish` commands to debug functions called during your program's execution. You saw how the `next` command can step over a function call. You also learned that the `quit` command ends a debugging session.

I.5 watch Command

The **watch command** tells the debugger to watch a data member. When that data member is about to change, the debugger will notify you. In this section, you'll use the `watch` command to see how the `Account` object's data member `balance` is modified during execution.

1. *Starting the debugger.* Start the debugger by typing `gdb figI_03`.
2. *Setting a breakpoint and running the program.* Type `break 10` to set a breakpoint at line 10. Then, run the program with the command `run`. The debugger and program will pause at the breakpoint at line 10 (Fig. I.26).

```
(gdb) break 10
Breakpoint 1 at 0x80486e5: file FigI_03.cpp, line 10.
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:10
10          Account account1( 50 ); // create Account object
(gdb)
```

Fig. I.26 | Running the program until the first breakpoint.

3. *Watching a class's data member.* Set a watch on `account1`'s `balance` data member by typing `watch account1.balance` (Fig. I.27). This watch is labeled as `watchpoint 2` because watchpoints are labeled with the same sequence of numbers as breakpoints. You can set a watch on any variable or data member of an object currently in scope. Whenever the value of a watched variable changes, the debugger enters break mode and notifies you that the value has changed.

```
(gdb) watch account1.balance
Hardware watchpoint 2: account1.balance
(gdb)
```

Fig. I.27 | Setting a watchpoint on a data member.

4. *Executing the constructor.* Use the `next` command to execute the constructor and initialize the `account1` object's `balance` data member. The debugger indicates that the `balance` data member's value changed, shows the old and new values and enters break mode at line 18 (Fig. I.28).

```
(gdb) next
Hardware watchpoint 2: account1.balance

Old value = 0
New value = 50
Account (this=0xbfcd6b90, initialBalance=50) at Account.cpp:18
18          if ( initialBalance < 0 )
(gdb)
```

Fig. I.28 | Stepping into the constructor.

5. *Exiting the constructor.* Type `finish` to complete the constructor's execution and return to `main`.
6. *Withdrawing money from the account.* Type `continue` to continue execution and enter a withdrawal value at the prompt. The program executes normally. Line 21 of Fig. I.3 calls `Account` member function `debit` to reduce the `Account` object's `balance` by a specified `amount`. Line 32 of Fig. I.2 inside function `debit` changes the value of `balance`. The debugger notifies you of this change and enters break mode (Fig. I.29).

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x0804893b in Account::debit (this=0xbfcd6b90, amount=13) at Account.cpp:32
32              balance = balance - amount;
(gdb)
```

Fig. I.29 | Entering break mode when a variable is changed.

7. *Continuing execution.* Type `continue`—the program will finish executing function `main` because the program does not attempt any additional changes to `balance`. The debugger removes the watch on `account1`'s `balance` data member because the `account1` object goes out of scope when function `main` ends. Removing the watchpoint causes the debugger to enter break mode. Type `continue` again to finish execution of the program (Fig. I.30).

```
(gdb) continue
Continuing.
account1 balance: $37

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0xb7da0595 in exit () from /lib/tls/i686/cmov/libc.so.6
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

Fig. I.30 | Continuing to the end of the program.

8. *Restarting the debugger and resetting the watch on the variable.* Type `run` to restart the debugger. Once again, set a watch on `account1` data member `balance`

by typing `watch account1.balance`. This watchpoint is labeled as `watchpoint 3`. Type `continue` to continue execution (Fig. I.31).

```
(gdb) run
Starting program: /home/nuke/AppJ/FigI_03

Breakpoint 1, main () at FigI_03.cpp:10
10          Account account1( 50 ); // create Account object
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 0
New value = 50
Account (this=0xbfd8eb90, initialBalance=50) at Account.cpp:18
18          if ( initialBalance < 0 )
(gdb)
```

Fig. I.31 | Resetting the watch on a data member.

9. ***Removing the watch on the data member.*** Suppose you want to watch a data member for only part of a program's execution. You can remove the debugger's watch on variable `balance` by typing `delete 3` (Fig. I.32). Type `continue`—the program will finish executing without reentering break mode.

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

Fig. I.32 | Removing a watch.

In this section, you used the `watch` command to enable the debugger to notify you when the value of a variable changes. You used the `delete` command to remove a watch on a data member before the end of the program.

I.6 Wrap-Up

In this appendix, you learned how to insert and remove breakpoints in the debugger. Breakpoints allow you to pause program execution so you can examine variable values with the debugger's `print` command, which can help you locate and fix logic errors. You used the `print` command to examine the value of an expression, and you used the `set` command to change the value of a variable. You also learned debugger commands (including the `step`, `finish` and `next` commands) that can be used to determine whether a function is executing correctly. You learned how to use the `watch` command to keep track of a data member throughout the scope of that data member. Finally, you learned how to use the `info break` command to list all the breakpoints and watchpoints set for a program and the `delete` command to remove individual breakpoints and watchpoints.

Summary

Section I.1 Introduction

- GNU includes software called a debugger, which allows you to monitor the execution of your programs to locate and remove logic errors.

*Section I.2 Breakpoints and the **`run`**, **`stop`**, **`continue`** and **`print`** Commands*

- The GNU debugger works only with executable files that were compiled with the `-g` compiler option, which generates information that is used by the debugger to help you debug your programs.
- The `gdb` command will start the GNU debugger and enable you to use its features. The `run` command will run a program through the debugger.
- Breakpoints are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses.
- The `break` command inserts a breakpoint at the line number specified after the command.
- When the program runs, it suspends execution at any line that contains a breakpoint and is said to be in break mode.
- The `continue` command causes the program to continue running until the next breakpoint is reached.
- The `print` command allows you to peek inside the computer at the value of one of your variables.
- When the `print` command is used, the result is stored in a convenience variable such as `$1`. Convenience variables are temporary variables that can be used in the debugging process to perform arithmetic and evaluate boolean expressions.
- You can display a list of all of the breakpoints in the program by typing `info break`.
- To remove a breakpoint, type `delete`, followed by a space and the number of the breakpoint to remove.

*Section I.3 **`print`** and **`set`** Commands*

- Use the `quit` command to end the debugging session.
- The `set` command allows you to assign new values to variables.

*Section I.4 Controlling Execution Using the **`step`**, **`finish`** and **`next`** Commands*

- The `step` command executes the next statement in the program. If the next statement to execute is a function call, control transfers to the called function. The `step` command enables you to enter a function and study the individual statements of that function.
- The `finish` command executes the remaining statements in the function and returns control to the place where the function was called.

- The `next` command behaves like the `step` command, except when the next statement to execute contains a function call. In that case, the called function executes in its entirety and the program advances to the next executable line after the function call.

Section I.5 **`watch`** *Command*

- The `watch` command sets a watch on any variable or data member of an object currently in scope during execution of the debugger. Whenever the value of a watched variable changes, the debugger enters break mode and notifies you that the value has changed.

Self-Review Exercises

I.1 Fill in the blanks in each of the following statements:

a) A breakpoint cannot be set at a(n) ________.
b) You can examine the value of an expression by using the debugger's ________ command.
c) You can modify the value of a variable by using the debugger's ________ command.
d) During debugging, the ________ command executes the remaining statements in the current function and returns program control to the place where the function was called.
e) The debugger's ________ command behaves like the `step` command when the next statement to execute does not contain a function call.
f) The `watch` debugger command allows you to view all changes to a(n) ________.

I.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When program execution suspends at a breakpoint, the next statement to be executed is the statement after the breakpoint.
b) Watches can be removed using the debugger's `remove` command.
c) The `-g` compiler option must be used when compiling programs for debugging.

Answers to Self-Review Exercises

I.1 a) non-executable line. b) `print`. c) `set`. d) `finish`. e) `next`. f) data member.

I.2 a) False. When program execution suspends at a breakpoint, the next statement to be executed is the statement at the breakpoint. b) False. Watches can be removed using the debugger's `delete` command. c) True.

27 Game Programming with Ogre

Come, Watson, come! The game is afoot.
—Sir Arthur Conan Doyle

For it's one, two, three strikes you're out at the old ball game.
—Jack Norworth

The game is up.
—William Shakespeare

If you wish to avoid foreign collision, you had better abandon the ocean.
—Henry Clay

Objectives

In this chapter you'll learn:

- Some basics of game programming.
- To create games using Ogre.
- To perform collision detection.
- To use Ogre to import and display graphics.
- To use OgreAL to integrate the OpenAL audio library into your games.
- To have Ogre accept keyboard input.
- To create the simple game Pong® with Ogre and OgreAL.
- To use Ogre to regulate the speed of a game.

Outline

27.1 Introduction

We now introduce game programming and graphics with the Ogre 3D graphics engine. Created in 2000 by Steve Streeting, Ogre is an open-source project maintained by the Ogre team at `www.ogre3d.org`. First, we discuss basic issues involved in game programming. Then we show how to use Ogre to create a simple game featuring a play mechanic similar to the classic video game Pong®, originally developed by Atari in 1972. We demonstrate how to create a scene with colored 3D graphics, smoothly animate moving objects, use timers to control animation speed, detect collisions between objects, add sound, accept keyboard input and display text output.

27.2 Installing Ogre, OgreAL and OpenAL

Ogre is powerful and easy to use, but the installation is a bit involved and varies for different platforms and compilers. There are two installation options—you can install the Ogre SDK or download the source code and compile it. We use Ogre 1.4 (Eihort) in this text. You must also install OgreAL and the OpenAL audio library. OpenAL handles the sound functionality and OgreAL allows you to integrate the OpenAL capabilities with the Ogre code. We provide detailed instructions for installing the Ogre SDK and all the OgreAL components. The instructions also explain how to configure a Visual C++ 2008 project to use Ogre and OgreAL. You can find the instructions in the Additional Resources section of the book's website at `www.deitel.com/books/cpphtp7`.

27.3 Basics of Game Programming

This section introduces the general fundamentals of game programming. Section 27.4 presents an implementation of the game of Pong, using the Ogre and OgreAL libraries.

Graphics

Graphics are perhaps the most crucial feature of any video game. Once a specialty, graphics programming is becoming more accessible even to novices. There are many **3D graphics engines** available—these frameworks hide the often tedious and complex programming required with graphics APIs and allow you to manage graphics more easily.

Ogre (Object-oriented Graphics Rendering Engine), one of the leading graphics engines, has been used in many commercial products including video games. It provides an object-oriented interface for 3D graphics programming. It supports the Direct3D and OpenGL graphics APIs and runs on the Windows, Linux and Mac platforms. **Direct3D** is Microsoft's Windows 3D graphics API. **OpenGL** is a graphics specification implemented by many video card vendors across all major platforms, including Windows.

Ogre is strictly a graphics rendering engine—it does not directly support sound, physics, collision detection, networking or other game-related needs. The Ogre community has produced many add-ons that allow users to integrate other libraries with Ogre to support those features.

3D Models

A **3D model** is a computer representation of an object which can be drawn on the screen—a process called **rendering**. **Materials** determine an object's appearance by setting lighting properties, colors and textures. A **texture** is an image that is wrapped around the model.

Most objects displayed in 3D graphics, everything from the terrain to the characters and the buildings, are 3D models. Many models are created in **3D modeling tools**. Some popular 3D modeling tools are Maya (`usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018`), SoftImage XSI (`www.softimage.com/`) and Blender (`www.blender.org/`). They're all available for Windows, Linux and Mac platforms. Blender is free and Maya offers a free version. SoftImage has a free 30-day trial available. The Ogre community has also produced several tools to allow users to **export 3D models** from these and other popular modeling tools into Ogre.

Materials, Textures and Colors

Colors are determined by red, green and blue light intensities, which can range from 0 to 1.0—a color value of `(1.0, 0, 0)` will create a bright red color, `(0, 1.0, 0)` will create a bright green, and `(0, 0, 1.0)` will make a bright blue. The first value is the red intensity, the second is the green intensity and the third is the blue intensity. To create white, use the maximum intensities of all three color values, `(1.0, 1.0, 1.0)`. To create black—the absence of all color—use `(0, 0, 0)`. Color values sometimes include an **alpha channel** to represent transparency, also ranging from 0 to 1.0, 0 being completely transparent and 1 completely opaque. Figure 27.1 shows common colors and their red, green, and blue intensity values. You can find color charts on the web as well, such as the color chart at `www.tayloredmktg.com/rgb/`.

Color	Red value	Green value	Blue value
Red	1.0	0.0	0.0
Green	0.0	1.0	0.0
Blue	0.0	0.0	1.0
Orange	1.0	0.784	0.0
Pink	1.0	0.686	0.686
Cyan	0.0	1.0	1.0

Fig. 27.1 | Red, green and blue intensities of common colors in Ogre. (Part 1 of 2.)

Color	Red value	Green value	Blue value
Magenta	1.0	0.0	1.0
Yellow	1.0	1.0	0.0
Black	0.0	0.0	0.0
White	1.0	1.0	1.0
Gray	0.5	0.5	0.5
Light gray	0.75	0.75	0.75
Dark gray	0.25	0.25	0.25

Fig. 27.1 | Red, green and blue intensities of common colors in Ogre. (Part 2 of 2.)

In 3D graphics, materials are used to determine the color of a 3D model. A material determines how the model should reflect different types of light and applies textures to the model. Materials can be set to use different **levels of detail (LoD)** depending on how far away from the viewer the model is. When close up, the model should be rendered with as much detail as possible. When the object in the scene is far away, there is no point in wasting computing power rendering details that the viewer can't see. To increase performance, the object can be rendered with much less detail.

Lighting

There are four different types of light in a 3D scene—ambient, diffuse, emissive and specular.[1] **Ambient light** is the general lighting in the scene that has been reflected off so many surfaces that it doesn't appear to have any definite source. **Diffuse light** appears to come from a particular direction and is reflected evenly off any surfaces it hits. **Emissive light** appears to come from an object in the scene. Emissive light won't affect the objects around it but will make the object emitting it seem brighter. **Specular light** comes from a particular direction and is reflected off an object based on the direction to the viewer. This is used to make an object appear shiny.

Collision Detection and Response

Collision detection is the process of determining whether two objects in a game are touching. You must know which objects to test and must deal with some complex mathematics. Checking whether one square hits another is relatively simple if each is parallel to flat ground. Checking circles and spheres is more difficult—the mathematics of curved surfaces is more complex.

Objects need to react appropriately when they collide with other objects. Some objects, such as walls, are stationary, while others move throughout the scene. Modeling the physics of moving objects can be complex. There are collision detection and physics modeling libraries that handle these complexities for you. Such libraries help to create a realistic game-playing experience.

---

1. D. Astel and K. Hawkins, *Beginning OpenGL: Game Programming*. Boston, MA: Thompson, 2004, pp. 104–110.

Sound

Sound is crucial to the game-playing experience. Gamers want to hear the lasers on their ships blasting away or the engines of their street racers revving up as they "peel out" at the starting line. Audio libraries help you enrich your games with sound. Many of those libraries support **3D sound**. In a 3D scene, objects emitting sound may be at various distances and directions from the user. The sound libraries take these factors into account when playing sounds. A sound from an object close to the listener will be louder than the sound from an object farther away. Also, sounds from one side of the listener will be played differently than sounds from the other.

Text

Games often communicate with the user by displaying text. This can range from giving the user instructions, to simply reporting how many points he or she has scored so far. In many games, text is a crucial form of communication between players. You can find free text fonts to use in your games at `www.1001freefonts.com`.

Timers

The speed at which a game runs can vary between systems due to differences in processor speeds. To solve this problem, game programmers use **timers** to control animation speed. If an object moves the same distance every **frame** (each time the screen is redrawn), then it may move at different speeds on different computers. A game running at 100 frames per second (fps) would be twice as fast as the same game running at 50 frames per second. Timers help keep the game play consistent by regulating the speed.

User Experience

Games should be fun to play and should appeal to the player in as many ways as possible. The basics we've discussed contribute to the overall user experience. You can get the player's attention through graphics and sound. Actions in games often have sounds associated with them. Many web sites offer free sounds you can use in your games. Some popular sound sites are Sound Hunter (`www.soundhunter.com`), Absolute Sound Effects Archive (`www.grsites.com/sounds`) and the search engine FindSounds (`www.findsounds.com`). You can also play a sound track in the background. Be sure to get permission to use any copyrighted songs if you plan on releasing your game as a product.

Players need to interact with games. User input devices include the keyboard, mouse, joystick and game controller. Keep the controls simple—the game should be easy to use, but not easy to beat. You can communicate with the player using text.

27.4 The Game of Pong: Code Walkthrough

In the next several sections, we present a complete C++/Ogre implementation of a simple game featuring a play mechanic similar to the classic video game Pong®, originally developed by Atari in 1972. (See the original Pong coin-op game in Fig. 27.2.) We walk through the code, explaining the Ogre capabilities as they're encountered. This is one of the largest example programs in the book. You should test drive the program thoroughly before reading the code walkthrough. You'll find all the files for this program in the `ch27` folder of the book's examples. Copy the `PongResources` folder to the `OgreSDK\media` folder. Ogre will throw a runtime exception if any of the resources can't be found. Open the

Fig. 27.2 | Atari's Pong coin-op went into commercial production in November of 1972, selling a total of 38,000 machines. The game was designed entirely in TTL circuits, using no CPU or game code software. Its "code" was implemented using simple "gate" chips, timers and counter chips. This simple, fun and addictive game was the cornerstone of what became the video game industry. (PONG® classic video game courtesy of Atari Interactive, Inc. © 2007 Atari Interactive, Inc. All rights reserved. Used with permission.)

Pong Visual C++ project. If you followed the Ogre and OgreAL installation instructions correctly, the project should build successfully. The project's executable file is copied to your `OgreSDK\bin\debug` folder (or `OgreSDK\bin\release` if you build in Release mode). *Important:* Remember to copy the `OgreAL_d.dll` (or `OgreAL.dll` for Release mode) and the `alut.dll` files into the folder containing the Pong executable.

Pong has four major game objects, a ball, two paddles and a rectangular box (Fig. 27.3). The ball will bounce across the screen inside the box while the players control the paddles to keep the ball from hitting the left or right sides. If the ball hits the left or right side of the box, the player "attacking" that side is awarded a point. The score is displayed at the top of the screen.

Play the game for a while, using the *A* and *Z* keys to control the left paddle and the up and down arrow keys to control the right paddle. Notice the colors of the objects, the ball interacting with the other objects and the score displayed at the top of the screen. Hit the *Esc* key to quit—closing the window won't stop the program.

27.4.1 Ogre Initialization

Class `Pong` (Figs. 27.4–27.5) represents the game of Pong. Figure 27.4 contains the `Pong` class definition. Line 6 includes the `Ogre.h` header file. This file automatically includes the most commonly used Ogre header files. Line 22 is the function prototype for the `run` function, which will start and run the game. Lines 25–26 contain the function prototypes for

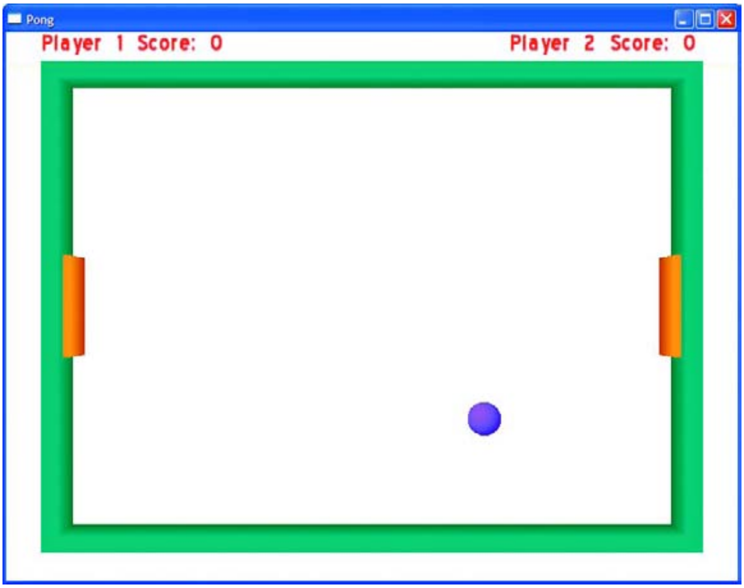


Fig. 27.3 | Pong game objects.

handling user input from the keyboard. Lines 29–30 define the function prototypes for performing game logic between frames. We also declare pointers to important Ogre objects (lines 38–42), input handling objects (lines 45–46) and the game objects (lines 49–51). Lines 54–56 define some variables used to control the behavior of the game—we'll discuss these later.

```
1  // Pong.h
2  // Pong class definition (represents a game of Pong).
3  #ifndef PONG_H
4  #define PONG_H
5
6  #include <Ogre.h> // Ogre class definitions
7  #include <OIS/OISEvents.h> // OISEvents class definition
8  #include <OIS/OISInputManager.h> // OISInputManager class definition
9  #include <OIS/OISKeyboard.h> // OISKeyboard class definition
10 using namespace Ogre;
11
12 class Ball; // forward declaration of class Ball
13 class Paddle; // forward declaration of class Paddle
14
15 enum Players { PLAYER1, PLAYER2 };
16
17 class Pong : public FrameListener, public OIS::KeyListener
18 {
19 public:
20    Pong(); // constructor
```

Fig. 27.4 | Pong class definition (represents a game of Pong). (Part 1 of 2.)

```
   ~Pong(); // destructor
   void run(); // run a game of Pong

   // handle keyPressed and keyReleased events
   bool keyPressed( const OIS::KeyEvent &keyEventRef );
   bool keyReleased( const OIS::KeyEvent &keyEventRef );

   // move the game objects and control interactions between frames
   virtual bool frameStarted( const FrameEvent &frameEvent );
   virtual bool frameEnded( const FrameEvent &frameEvent );
   static void updateScore( Players player ); // update the score

private:
   void createScene(); // create the scene to be rendered
   static void updateScoreText(); // update the score on the screen

   // Ogre objects
   Root *rootPtr; // pointer to Ogre's Root object
   SceneManager *sceneManagerPtr; // pointer to the SceneManager
   RenderWindow *windowPtr; // pointer to RenderWindow to render scene in
   Viewport *viewportPtr; // pointer to Viewport, area that a camera sees
   Camera *cameraPtr; // pointer to a Camera in the scene

   // OIS input objects
   OIS::InputManager *inputManagerPtr; // pointer to the InputManager
   OIS::Keyboard *keyboardPtr; // pointer to the Keyboard

   // game objects
   Ball *ballPtr; // pointer to the Ball
   Paddle *leftPaddlePtr; // pointer to player 1's Paddle
   Paddle *rightPaddlePtr; // pointer to player 2's Paddle

   // variables to control game states
   bool quit, pause; // did user quit or pause the game?
   Real time; // used to delay the motion of a new Ball
   static bool wait; // should the Ball's movement be delayed?

   static int player1Score; // player 1's score
   static int player2Score; // player 2's score
}; // end class Pong

#endif // PONG_H
```

Fig. 27.4 | Pong class definition (represents a game of Pong). (Part 2 of 2.)

```
// Pong.cpp
// Pong class member-function definitions.
#include <sstream>
#include <stdexcept>
#include <Ogre.h> // Ogre class definitions
#include <OgreAL.h> // OgreAL class definitions
```

Fig. 27.5 | Pong class member-function definitions. (Part 1 of 7.)

```
#include <OgreStringConverter.h> // OgreStringConverter class definition
#include <OIS/OISEvents.h> // OISEvents class definition
#include <OIS/OISInputManager.h> // OISInputManager class definition
#include <OIS/OISKeyboard.h> // OISKeyboard class definition
#include "Ball.h" // Ball class definition
#include "Paddle.h" // Paddle class definition
#include "Pong.h" // Pong class definition
using namespace std;
using namespace Ogre;

int Pong::player1Score = 0; // initialize player 1's score to 0
int Pong::player2Score = 0; // initialize player 2's score to 0
bool Pong::wait = false; // initialize wait to false

// directions to move the Paddles
const Vector3 PADDLE_DOWN = Vector3( 0, -15, 0 );
const Vector3 PADDLE_UP = Vector3( 0, 15, 0 );

// constructor
Pong::Pong()
{
   rootPtr = new Root(); // initialize Root object

   // use the Ogre Config Dialog Box to choose the settings
   if ( !( rootPtr->showConfigDialog() ) ) // user canceled the dialog box
      throw runtime_error( "User Canceled Ogre Setup Dialog Box." );

   // get a pointer to the RenderWindow
   windowPtr = rootPtr->initialise( true, "Pong" );

   // create the SceneManager
   sceneManagerPtr = rootPtr->createSceneManager( ST_GENERIC );

   // create the Camera
   cameraPtr = sceneManagerPtr->createCamera( "PongCamera" );
   cameraPtr->setPosition( Vector3( 0, 0, 200 ) ); // set Camera position
   cameraPtr->lookAt( Vector3( 0, 0, 0 ) ); // set where Camera looks
   cameraPtr->setNearClipDistance( 5 ); // near distance Camera can see
   cameraPtr->setFarClipDistance( 1000 ); // far distance Camera can see

   // create the Viewport
   viewportPtr = windowPtr->addViewport( cameraPtr );
   viewportPtr->setBackgroundColour( ColourValue( 0, 0, 0 ) );

   // set the Camera's aspect ratio
   cameraPtr->setAspectRatio( Real( viewportPtr->getActualWidth() ) /
      ( viewportPtr->getActualHeight() ) );

   // set the scene's ambient light
   sceneManagerPtr->setAmbientLight( ColourValue( 0.75, 0.75, 0.75 )

   // create the Light
```

Fig. 27.5 | Pong class member-function definitions. (Part 2 of 7.)

```
   Light *lightPtr = sceneManagerPtr->createLight( "Light" ); // a Light
   lightPtr->setPosition( 0, 0, 50 ); // set the Light's position

   unsigned long hWnd; // variable to hold the window handle
   windowPtr->getCustomAttribute( "WINDOW", &hWnd ); // get window handle
   OIS::ParamList paramList; // create an OIS ParamList

   // add the window to the ParamList
   paramList.insert( OIS::ParamList::value_type( "WINDOW",
      Ogre::StringConverter::toString( hWnd ) ) );

   // create the InputManager
   inputManagerPtr = OIS::InputManager::createInputSystem( paramList );
   keyboardPtr = static_cast< OIS::Keyboard* >( inputManagerPtr->
      createInputObject( OIS::OISKeyboard, true ) ); // create a Keyboard
   keyboardPtr->setEventCallback( this ); // add a KeyListener

   rootPtr->addFrameListener( this ); // add this Pong as a FrameListener

   // load resources for Pong
   ResourceGroupManager::getSingleton().addResourceLocation(
      "resources", "FileSystem", "Pong" );
   ResourceGroupManager::getSingleton().initialiseAllResourceGroups();

   quit = pause = false; // player has not quit or paused the game
   time = 0; // initialize the time since Ball was reset to 0
} // end Pong constructor

// Pong destructor erases objects contained in a Pong object
Pong::~Pong()
{
   // free dynamically allocated memory for Keyboard
   inputManagerPtr->destroyInputObject( keyboardPtr );
   OIS::InputManager::destroyInputSystem( inputManagerPtr );

   // free dynamically allocated memory for Root
   delete rootPtr; // release memory pointer points to
   rootPtr = 0; // point pointer at 0

   // free dynamically allocated memory for Ball
   delete ballPtr; // release memory pointer points to
   ballPtr = 0; // point pointer at 0

   // free dynamically allocated memory for Paddle
   delete leftPaddlePtr; // release memory pointer points to
   leftPaddlePtr = 0; // point pointer at 0

   // free dynamically allocated memory for Paddle
   delete rightPaddlePtr; // release memory pointer points to
   rightPaddlePtr = 0; // point pointer at 0
} // end Pong destructor

```

Fig. 27.5 | Pong class member-function definitions. (Part 3 of 7.)

```
// create the scene to be displayed
void Pong::createScene()
{
   // get a pointer to the Score Overlay
   Overlay *scoreOverlayPtr =
      OverlayManager::getSingleton().getByName( "Score" );
   scoreOverlayPtr->show(); // show the Overlay

   // make the game objects
   ballPtr = new Ball( sceneManagerPtr ); // make the Ball
   ballPtr->addToScene(); // add the Ball to the scene
   rightPaddlePtr = new Paddle( sceneManagerPtr, "RightPaddle", 90 );
   rightPaddlePtr->addToScene(); // add a Paddle to the scene
   leftPaddlePtr = new Paddle( sceneManagerPtr, "LeftPaddle", -90 );
   leftPaddlePtr->addToScene(); // add a Paddle to the scene

   // create the walls
   Entity *entityPtr = sceneManagerPtr->
      createEntity( "WallLeft", "cube.mesh" ); // create the left wall
   entityPtr->setMaterialName( "wall" ); // set material for left wall
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled

   // create the SceneNode for the left wall
   SceneNode *nodePtr = sceneManagerPtr->getRootSceneNode()->
      createChildSceneNode( "WallLeft" );
   nodePtr->attachObject( entityPtr ); // attach left wall to SceneNode
   nodePtr->setPosition( -95, 0, 0 ); // set the left wall's position
   nodePtr->setScale( .05, 1.45, .1 ); // set the left wall's size
   entityPtr = sceneManagerPtr->createEntity( "WallRight", "cube.mesh" );
   entityPtr->setMaterialName( "wall" ); // set material for right wall
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled

   // create the SceneNode for the right wall
   nodePtr = sceneManagerPtr->getRootSceneNode()->
      createChildSceneNode( "WallRight" );
   nodePtr->attachObject( entityPtr ); // attach right wall to SceneNode
   nodePtr->setPosition( 95, 0, 0 ); // set the right wall's position
   nodePtr->setScale( .05, 1.45, .1 ); // set the right wall's size
   entityPtr = sceneManagerPtr->createEntity( "WallBottom", "cube.mesh" );
   entityPtr->setMaterialName( "wall" ); // set material for bottom wall
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled

   // create the SceneNode for the bottom wall
   nodePtr = sceneManagerPtr->getRootSceneNode()->
      createChildSceneNode( "WallBottom" );
   nodePtr->attachObject( entityPtr ); // attach bottom wall to SceneNode
   nodePtr->setPosition( 0, -70, 0 ); // set the bottom wall's position
   nodePtr->setScale( 1.95, .05, .1 ); // set bottom wall's size
   entityPtr = sceneManagerPtr->createEntity( "WallTop", "cube.mesh" );
   entityPtr->setMaterialName( "wall" ); // set the material for top wall
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled

```

Fig. 27.5 | Pong class member-function definitions. (Part 4 of 7.)

```
   // create the SceneNode for the top wall
   nodePtr = sceneManagerPtr->getRootSceneNode()->
      createChildSceneNode( "WallTop" );
   nodePtr->attachObject( entityPtr ); // attach top wall to the SceneNode
   nodePtr->setPosition( 0, 70, 0 ); // set the top wall's position
   nodePtr->setScale( 1.95, .05, .1 ); // set the top wall's size
} // end function createScene

// start a game of Pong
void Pong::run()
{
   createScene(); // create the scene
   rootPtr->startRendering(); // start rendering frames
} // end function run

// update the score
void Pong::updateScore( Players player )
{
   // increase the correct player's score
   if ( player == PLAYER1 )
      player1Score++;
   else
      player2Score++;

   wait = true; // the game should wait to restart the Ball
   updateScoreText(); // update the score text on the screen
} // end function updateScore

// update the score text
void Pong::updateScoreText()
{
   ostringstream scoreText; // text to be displayed

   scoreText << "Player 2 Score: " << player2Score; // make the text

   // get the right player's TextArea
   OverlayElement *textElementPtr =
      OverlayManager::getSingletonPtr()->getOverlayElement( "right" );
   textElementPtr->setCaption( scoreText.str() ); // set the text

   scoreText.str( "" ); // reset the text in scoreText
   scoreText << "Player 1 Score: " << player1Score; // make the text

   // get the left player's TextArea
   textElementPtr =
      OverlayManager::getSingletonPtr()->getOverlayElement( "left" );
   textElementPtr->setCaption( scoreText.str() ); // set the text
} // end function updateScoreText

// respond to user keyboard input
bool Pong::keyPressed( const OIS::KeyEvent &keyEventRef )
{
```

Fig. 27.5 | Pong class member-function definitions. (Part 5 of 7.)

```
   // if the game is not paused
   if ( !pause )
   {
      // process KeyEvents that apply when the game is not paused
      switch ( keyEventRef.key )
      {
      case OIS::KC_ESCAPE: // escape key hit: quit
         quit = true;
         break;
      case OIS::KC_UP: // up-arrow key hit: move the right Paddle up
         rightPaddlePtr->movePaddle( PADDLE_UP );
         break;
      case OIS::KC_DOWN: //down-arrow key hit: move the right Paddle down
         rightPaddlePtr->movePaddle( PADDLE_DOWN );
         break;
      case OIS::KC_A: // A key hit: move the left Paddle up
         leftPaddlePtr->movePaddle( PADDLE_UP );
         break;
      case OIS::KC_Z: // Z key hit: move the left Paddle down
         leftPaddlePtr->movePaddle( PADDLE_DOWN );
         break;
      case OIS::KC_P: // P key hit: pause the game
         pause = true; // set pause to true when the user pauses the game
         Overlay *pauseOverlayPtr =
            OverlayManager::getSingleton().getByName( "PauseOverlay" );
         pauseOverlayPtr->show(); // show the pause Overlay
         break;
      } // end switch
   } // end if
   else // game is paused
   {
      // user hit 'R' on the keyboard
      if ( keyEventRef.key == OIS::KC_R )
      {
         pause = false; // set pause to false when user resumes the game
         Overlay *pauseOverlayPtr =
            OverlayManager::getSingleton().getByName( "PauseOverlay" );
         pauseOverlayPtr->hide(); // hide the pause Overlay
      } // end if
   } // end else
   return true;
} // end function keyPressed

// process key released events
bool Pong::keyReleased( const OIS::KeyEvent &keyEventRef )
{
   return true;
} // end function keyReleased

// return true if the program should render the next frame
bool Pong::frameEnded( const FrameEvent &frameEvent )
{
```

Fig. 27.5 | Pong class member-function definitions. (Part 6 of 7.)

```
267    return !quit; // quit = false if the user hasn't quit yet
268 } // end function frameEnded
269
270 // process game logic, return true if the next frame should be rendered
271 bool Pong::frameStarted( const FrameEvent &frameEvent )
272 {
273    keyboardPtr->capture(); // get keyboard events
274    // the game is not paused and the Ball should move
275    if ( !wait && !pause )
276    {
277       // move the Ball
278       ballPtr->moveBall( frameEvent.timeSinceLastFrame );
279    } // end if
280    // don't move the Ball if wait is true
281    else if ( wait )
282    {
283       // increase time if it is less than 4 seconds
284       if ( time < 4 )
285          // add the seconds since the last frame
286          time += frameEvent.timeSinceLastFrame;
287       else
288       {
289          wait = false; // shouldn't wait to move the Ball any more
290          time = 0; // reset the control variable to 0
291       } // end else
292    } // end else
293
294    return !quit; // quit = false if the user hasn't quit yet
295 } // end function frameStarted
```

Fig. 27.5 | `Pong` class member-function definitions. (Part 7 of 7.)

Before we can display any graphics, we need to initialize the Ogre engine settings and create certain Ogre objects. The **OGRE Engine Rendering Setup** dialog box (Fig. 27.6) enables the user to choose the rendering settings, including which **rendering subsystem** to use (Direct3D or OpenGL), resolution, color depth, full-screen mode, and other options that are beyond the scope of this chapter. Direct3D is exclusively for Windows. OpenGL is supported on all major platforms. The **resolution** is defined by two values, width and height, which determine the number of pixels used to draw the scene. The resolution options for both rendering subsystems can range from 640×400 to 1680×1050 or higher, depending on your hardware. A higher resolution will produce more detailed graphics. If you choose to turn off full-screen mode, the resolution will also determine the size of the window in which the game is displayed. We run the game at a resolution of 800×600. A **color depth** of n bits means that 2^n possible colors can be displayed on the screen. A lower color depth will make the program require less memory, but the graphics may not be as good. Direct3D and OpenGL each support 16-bit and 32-bit color depths. In 32-bit colors, only 24 bits are used for the color; the other eight bits represent the alpha value (i.e., transparency).

To display the dialog box, you must create a **`Root` object** (Fig. 27.5, line 28)—the Ogre object used to start the engine. The only Ogre object that can be created before `Root` is the `LogManager`—but that is beyond the scope of this book. Next, we call the

Fig. 27.6 | **OGRE Engine Rendering Setup** dialog box.

`showConfigDialog` function of the `Root` class (line 31) to display the dialog. Once you hit **OK**, Ogre saves the settings and uses them as the default settings the next time the dialog box is displayed. The program should end if the user selects **Cancel**, because the settings may not be properly configured and could cause errors. We throw a `runtime_error` if `showConfigDialog` returns `false` (i.e., the user selected **Cancel**).

Once the rendering subsystem and window options have been set, we can create the **`RenderWindow`**, a window in which Ogre will render graphics, by calling the `initialise` function of class `Root` (line 35). The first parameter, `true`, tells Ogre to create the window with the settings the user chose in the dialog box. Passing `false` to this parameter allows you to manually create the window at a later time. The second, `"Pong"`, is the name of the window within the engine and will also appear in the title bar of the window if it isn't full screen. Notice the British spelling of `initialise` and of *colour* in Fig. 27.6, reflecting Ogre's origin in the United Kingdom.

27.4.2 Creating a Scene

Now that we've initialized Ogre and set up a window to render our graphics in, we'll add some objects to create our **scene**—the collection of images that we display on the screen.

SceneManager

To control the scene we use Ogre's **`SceneManager` object** (line 38). The `SceneManager` manages the **scene graph**, a data structure that contains all the objects in the scene, both visible and nonvisible. The `SceneManager` is used to create objects that will be added to the scene graph and determines which objects will be rendered. Excluding objects that are not within the visible scene from being rendered, known as **culling**, decreases rendering time and increases performance. This is done automatically. We'll keep a pointer to this SceneManager object, as it will be used extensively throughout the game.

Several types of SceneManagers have been designed to handle different types of scenes, such as indoor scenes or expansive landscape scenes. An Ogre application can use more than one SceneManager, separately or at the same time. For the purposes of this chapter, we use only one instance of the generic scene type (ST_GENERIC), a SceneManager that's not optimized for any particular type of scene.

Camera

Once we have a SceneManager, we can start constructing our scene. First we add a Camera. A **Camera** in Ogre is the eye through which you view the scene. A 3D scene is usually too large to be displayed in one window. The Camera looks into the scene and tells Ogre what part you can actually see. Cameras can be placed at any location in the scene or attached to SceneNodes—we'll discuss these shortly. If attached to a SceneNode, the Camera will follow that node if it moves within the scene. Ogre supports multiple Cameras in a single scene, but we need only one.

We use the SceneManager to create the Camera (line 41), then we set the position, orientation, clip distances and Viewport (lines 42–49). Position is the location of the Camera within the scene. We position the Camera 200 units from the origin along the positive *z*-axis, toward the player. This places the Camera far enough away from the origin of the scene so that we can center the game around it. We can set all of the *z*-coordinates to 0 and not have to change them. The orientation is the direction in which the Camera is looking. We have the Camera look at the origin (line 43) because we center the game around that point. The clip distances define how near and how far the Camera can see. If something is closer to the Camera than the near clip distance, or farther away than the far clip distance, then the Camera won't be able to see it. The **Viewport** is the area of the screen used to display what the Camera can see. We set the Viewport's background color to white. A Camera can have more than one Viewport, but we'll use only one for our game. Cameras have many other features and functions, but this is all we need for our game.

Light

One of the most important aspects of a 3D scene is lighting. Ogre has three types of **Lights**—Point, Spot and Directional. **Point lights** have a position in space and radiate light in all directions. **Spot lights** have a position in space like Point lights, but radiate light in only one direction; the strength of the light fades as the distance from the source increases. **Directional lights** do not have a position in space, they have only a direction in which they shine—the light is assumed to come from the same direction no matter where you are in the scene.

We use a Point light in our game. Lines 59–60 use the SceneManager to create the Light and set its position within the scene. The argument to the createLight function is the name by which we'll refer to the Light. We set the Light's position by specifying its *x*-, *y*- and *z*-coordinates. Our scene is now ready for use.

27.4.3 Adding to the Scene

As we mentioned earlier, Pong has four major game objects—a ball, two paddles and a rectangular box. All of these elements must be added to the scene before they can be displayed.

Add the `Ball`

Class `Ball` (Figs. 27.7–27.8) represents the ball that bounces around the screen. We must add the `Ball` to the scene before it can be displayed. Member function `addToScene` (Fig. 27.8, lines 26–51) creates an `Entity` that represents the `Ball`, adds it to the scene and creates sounds associated with the `Ball`; we'll discuss the sounds later. An **`Entity`** is a mesh instance within the scene. A **mesh** is a file that contains the geometry information of a 3D model. Many `Entity` objects can be based on the same mesh as long as each `Entity` has a unique name. Lines 29–30 create the `Entity`, which is referenced through a pointer. The first argument is the name of the `Entity`. The second argument, `"sphere.mesh"`, is the mesh file used to determine the `Entity`'s shape. We use the sphere mesh provided with the Ogre SDK. You can find the mesh in the `OgreSDK\media\models` folder on your computer.

```
// Ball.h
// Ball class definition (represents a bouncing ball).
#ifndef BALL_H
#define BALL_H

#include <Ogre.h> // Ogre class definitions
#include <OgreAL.h> // OgreAL class definitions
using namespace Ogre;

class Paddle; // forward declaration of class Paddle

const int RADIUS = 5; // the radius of the Ball

class Ball
{
public:
   Ball( SceneManager *sceneManagerPtr ); // constructor
   ~Ball(); // destructor
   void addToScene(); // add the Ball to the scene
   void moveBall( Real time ); // move the Ball across the screen

private:
   SceneManager *sceneManagerPtr; // pointer to the SceneManager
   SceneNode *nodePtr; // pointer to the SceneNode
   OgreAL::SoundManager *soundManagerPtr; // pointer to the SoundManager
   OgreAL::Sound *wallSoundPtr; // sound played when Ball hits a wall
   OgreAL::Sound *paddleSoundPtr; // sound played when Ball hits a Paddle
   OgreAL::Sound *scoreSoundPtr; // sound played when someone scores
   int speed; // speed of the Ball
   Vector3 direction; // direction of the Ball

   // private utility functions
   void reverseHorizontalDirection(); // change horizontal direction
   void reverseVerticalDirection(); // change vertical direction
   void hitPaddle(); // control the Ball hitting the Paddles
}; // end class Ball

#endif // BALL_H
```

Fig. 27.7 | `Ball` class definition (represents a bouncing ball).

```
// Ball.cpp
// Ball class member-function definitions.
#include <Ogre.h> // Ogre class definitions
#include <OgreAL.h> // OgreAL class definitions
#include "Ball.h" // Ball class definition
#include "Paddle.h" // Paddle class definition
#include "Pong.h" // Pong class definition
using namespace Ogre;

// Ball constructor
Ball::Ball( SceneManager *ptr )
{
   sceneManagerPtr = ptr; // set pointer to the SceneManager
   soundManagerPtr = new OgreAL::SoundManager(); // create SoundManager
   speed = 100; // speed of the Ball
   direction = Vector3( 1, -1, 0 ); // direction of the Ball
} // end Ball constructor

// Ball destructor
Ball::~Ball()
{
   // empty body
} // end Ball destructor

// add the Ball to the scene
void Ball::addToScene()
{
   // create Entity and attach it to a node in the scene
   Entity *entityPtr =
      sceneManagerPtr->createEntity( "Ball", "sphere.mesh" );
   entityPtr->setMaterialName( "ball" ); // set material for the Ball
   entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
   nodePtr = sceneManagerPtr->getRootSceneNode()->
      createChildSceneNode( "Ball" ); // create a SceneNode
   nodePtr->attachObject( entityPtr ); // attach the Entity to SceneNode
   nodePtr->setScale( .05, .05, .05 ); // scale SceneNode

   // attach sounds to Ball so they will play from where Ball is
   wallSoundPtr = soundManagerPtr->
      createSound( "wallSound", "wallSound.wav", false );
   nodePtr->attachObject( wallSoundPtr ); // attach a sound to SceneNode
   paddleSoundPtr = soundManagerPtr->
      createSound( "paddleSound", "paddleSound.wav", false );
   nodePtr->attachObject( paddleSoundPtr ); // attach sound to SceneNode
   scoreSoundPtr = soundManagerPtr->
      createSound( "cheer", "cheer.wav", false ); // create a Sound

   // attach the score sound to its own node centered at ( 0, 0, 0 )
   sceneManagerPtr->getRootSceneNode()->createChildSceneNode( "score" )->
      attachObject( scoreSoundPtr );
} // end function addToScene

```

Fig. 27.8 | `Ball` class member-function definitions. (Part 1 of 4.)

```
// move the Ball across the screen
void Ball::moveBall( Real time )
{
   nodePtr->translate( ( direction * ( speed * time ) ) ); // move Ball
   Vector3 position = nodePtr->getPosition(); // get Ball's new position

   // get the positions of the four walls
   Vector3 topPosition = sceneManagerPtr->
      getSceneNode( "WallTop" )->getPosition();
   Vector3 bottomPosition = sceneManagerPtr->
      getSceneNode( "WallBottom" )->getPosition();
   Vector3 leftPosition = sceneManagerPtr->
      getSceneNode( "WallLeft" )->getPosition();
   Vector3 rightPosition = sceneManagerPtr->
      getSceneNode( "WallRight" )->getPosition();

   const int WALL_WIDTH = 5; // the width of the walls

   // check if the Ball hit the left side
   if ( ( position.x  - RADIUS ) <= leftPosition.x + ( WALL_WIDTH / 2 ) )
   {
      nodePtr->setPosition( 0, 0, 0 ); // place Ball in center of screen
      Pong::updateScore( PLAYER2 ); // update the score
      scoreSoundPtr->play(); // play a sound when player 2 scores
   } // end if
   // check if the Ball hit the right side
   else if (
      ( position.x + RADIUS ) >= rightPosition.x - ( WALL_WIDTH / 2 ) )
   {
      nodePtr->setPosition( 0, 0, 0 ); // place Ball in center of screen
      Pong::updateScore( PLAYER1 ); // update the score
      scoreSoundPtr->play(); // play a sound when player 1 scores
   } // end else
   // check if the Ball hit the bottom wall
   else if (
      ( position.y  - RADIUS ) <= bottomPosition.y + ( WALL_WIDTH / 2 ) &&
      direction.y < 0 )
   {
      // place the Ball on the bottom wall
      nodePtr->setPosition( position.x,
         ( bottomPosition.y + ( WALL_WIDTH / 2 ) + RADIUS ), position.z );
      reverseVerticalDirection(); // make the Ball start moving up
   } // end else
   // check if the Ball hit the top wall
   else if (
      ( position.y + RADIUS ) >= topPosition.y - ( WALL_WIDTH / 2 ) &&
      direction.y > 0 )
   {
      // place the Ball on the top wall
      nodePtr->setPosition( position.x,
         ( topPosition.y - ( WALL_WIDTH / 2 ) - RADIUS ), position.z );
      reverseVerticalDirection(); // make the Ball start moving down
   } // end else
```

Fig. 27.8 | `Ball` class member-function definitions. (Part 2 of 4.)

```

   hitPaddle(); // check if the Ball hit a Paddle
} // end function moveBall

// reverse the Ball's horizontal direction
void Ball::reverseHorizontalDirection()
{
   direction *= Vector3( -1, 1, 1 ); // reverse the horizontal direction
   paddleSoundPtr->play(); // play the "paddleSound" sound effect
} // end function reverseHorizontalDirection

// reverse the Ball's vertical direction
void Ball::reverseVerticalDirection()
{
   direction *= Vector3( 1, -1, 1 ); // reverse the vertical direction
   wallSoundPtr->play(); // play the "wallSound" sound effect
} // end function reverseVerticalDirection

// control the Ball colliding with the Paddle
void Ball::hitPaddle()
{
   // get the position of the Paddles and the Ball
   Vector3 leftPaddlePosition = sceneManagerPtr->
      getSceneNode( "LeftPaddle" )->getPosition(); // left Paddle
   Vector3 rightPaddlePosition = sceneManagerPtr->
      getSceneNode( "RightPaddle" )->getPosition(); // right Paddle
   Vector3 ballPosition = nodePtr->getPosition(); // the Ball's position

   const int PADDLE_WIDTH = 2; // width of the Paddle
   const int PADDLE_HEIGHT = 30; // height of the Paddle

   // is the Ball in range of the left Paddle?
   if ( ( ballPosition.x - RADIUS ) <
      ( leftPaddlePosition.x + ( PADDLE_WIDTH / 2 ) ) )
   {
      // check for collision with left Paddle
      if ( ( ballPosition.y - RADIUS ) <
         ( leftPaddlePosition.y + ( PADDLE_HEIGHT / 2 ) ) &&
         ( ballPosition.y + RADIUS ) >
         ( leftPaddlePosition.y - ( PADDLE_HEIGHT / 2 ) ) )
      {
         reverseHorizontalDirection(); // reverse the Ball's direction

         // place the Ball at the edge of the Paddle
         nodePtr->setPosition(
            ( leftPaddlePosition.x + ( PADDLE_WIDTH / 2 ) + RADIUS ),
            ballPosition.y, ballPosition.z );
      } // end if
   } // end if
   // is the Ball in range of the right Paddle?
   else if ( ( ballPosition.x + RADIUS ) >
      ( rightPaddlePosition.x - ( PADDLE_WIDTH / 2 ) ) )
   {
```

Fig. 27.8 | `Ball` class member-function definitions. (Part 3 of 4.)

```
159         // check for collision with right Paddle
160         if ( ( ballPosition.y - RADIUS ) <
161            ( rightPaddlePosition.y + ( PADDLE_HEIGHT / 2 ) ) &&
162            ( ballPosition.y + RADIUS ) >
163            ( rightPaddlePosition.y - ( PADDLE_HEIGHT / 2 ) ) )
164         {
165            reverseHorizontalDirection(); // reverse the Ball's direction
166
167            // place the Ball at the edge of the Paddle
168            nodePtr->setPosition(
169               ( rightPaddlePosition.x - ( PADDLE_WIDTH / 2 ) - RADIUS ),
170               ballPosition.y, ballPosition.z );
171         } // end if
172      } // end else
173   } // end function hitPaddle
```

Fig. 27.8 | `Ball` class member-function definitions. (Part 4 of 4.)

Line 31 sets the material used to color the `Entity`. The argument, `"ball"`, is the name of the material used to color the `Ball`. In Ogre, a material is usually created with a script, though it can also be created directly in the program. A material **script** (Fig. 27.9) is a text file that Ogre uses to create a material. Save the text file with a `.material` extension.

```
1   // ball material definition
2   material ball
3   {
4      // define one technique for rendering the Ball
5      technique
6      {
7         // render the Ball in one pass
8         pass
9         {
10           // color the Ball violet
11           ambient 0.58 0 0.827
12           diffuse 0.58 0 0.827
13           specular 0.58 0 0.827 120
14        }
15     }
16  }
```

Fig. 27.9 | `ball` material script.

Notice that the `ball` material's structure looks similar to C++ code, with curly braces enclosing each section. Line 2 indicates that we're defining a `material` called `ball`. Within the `material`, we define a `technique`—i.e., how to render the object (lines 5–15). You can define multiple `technique`s for a material, but that is beyond our scope. Within each `technique`, one or more `pass`es are defined (lines 8–14). Each `pass` defines a single step in the `material`'s rendering process. Using multiple `pass`es is beyond our scope. The color is determined in the `pass` by setting color values for the different types of lighting in the scene. We want the `Ball` to be violet, so set the color values to (0.58, 0, 0.827) (lines 11–13). The numbers after each type of light (ambient, diffuse and specular) represent color values. The fourth number in `specular` (`120`) determines how shiny the `Ball` is. It can be any value greater than 0. The higher the value the shinier the object appears.

Note that `"ball"` is not the name of the file that the `material` is defined in, but rather the name of the `material` within that file. A `material` file can define multiple `materials`. The same `material` can be used for multiple `Entity` objects, but each `material` defined must have a unique name.

We've created the `Entity` for our `Ball`, but it isn't part of the scene yet. Lines 33–35 (Fig. 27.8) add it to the scene so it will be rendered on the screen. We use the `SceneManager` to create a **SceneNode**—a `Node` within the scene graph that holds information about an object and its position in the scene, visible or nonvisible. A `SceneNode` may have many child `SceneNodes` attached to it, but can have only one parent `Node`. The argument, `"Ball"`, is the name by which the `SceneNode` will be referred to in the Ogre engine. Every `SceneNode` in the scene graph must have a unique name. The call to `getRootSceneNode` retrieves the topmost node within the scene graph. The **root node** is the parent of all other nodes. When you create a child of the root node, its initial position is (0, 0, 0). Line 35 attaches the `Entity` representing the `Ball` to the newly created `SceneNode`. The `Ball` is now part of the scene and will be rendered. Notice that all of the functions used to add the `Ball` to the scene are member functions of the `SceneManager`. That is why the `Ball` constructor takes a pointer to the `SceneManager` as a parameter—the `Ball` object must be able to access the `SceneManager` to add itself to the scene.

The sphere mesh provided with the Ogre SDK has a radius of 100. This is much larger than we need. Line 36 changes the size of the `Entity` attached to the `SceneNode`, but it does not affect the size of the actual mesh that the node's `Entity` is based on. We supply a scaling factor for each axis (x, y and z). We pass `.05` as the scaling factor for all three axes. Using the same scaling factor for all three axes uniformly scales the mesh so it maintains its original shape. The function multiplies the radius of the sphere on each axis by the scaling factor to change the radius from 100 to 5. When you scale a mesh, the lighting effects become somewhat distorted. We fix that by having the `Entity` calculate the new normals for the mesh each time it's scaled (line 32). [*Note:* In Ogre 1.6 (Shoggoth), this function doesn't exist—the operation is performed automatically. You will need to comment or remove this line and similar ones in other files to get the example to compile.] A **normal** in this case refers to the direction each facet (i.e., small section) of the object's surface is facing. If the facet is facing toward the light, it's brighter. If it's facing away from the light, it's darker.

There is also a function `scale` that will change the size of the `Entity`. The difference is that `scale` changes the size based on the current size, while `setScale` changes it based on the original size of the node. These functions also scale all children of the `SceneNode` by the same factor. This can be changed by telling each child of the parent node that you do not wish to have it scaled when the parent is scaled—to do that, call the `setInheritScale` function and pass it `false`.

Add the `Paddles`

Class `Paddle` (Figs. 27.10–27.11) represents the `Paddles`. We add a `Paddle` to the scene in much the same way that we added the `Ball`. The member function `addToScene` (Fig. 27.11, lines 23–34) uses the same first five Ogre functions, but with different arguments.

```
// Paddle.h
// Paddle class definition (represents a paddle in the game).
#ifndef PADDLE_H
#define PADDLE_H

#include <Ogre.h> // Ogre class definitions
using namespace Ogre;

class Paddle
{
public:
   // constructor
   Paddle( SceneManager *sceneManagerPtr, String paddleName,
      int positionX );
   ~Paddle(); // destructor
   void addToScene(); // add a Paddle to the scene
   void movePaddle( const Vector3 &direction ); // move a Paddle

private:
   SceneManager* sceneManagerPtr; // pointer to the SceneManager
   SceneNode *nodePtr; // pointer to a SceneNode
   String name; // name of the Paddle
   int x; // x-coordinate of the Paddle
}; // end of class Paddle

#endif // PADDLE_H
```

Fig. 27.10 | `Paddle` class definition (represents a paddle in the game).

```
// Paddle.cpp
// Paddle class member-function definitions.
#include <Ogre.h> // Ogre class definitions
#include "Paddle.h" // Paddle class definition
using namespace Ogre;

// constructor
Paddle::Paddle( SceneManager *ptr, String paddleName,
   int positionX )
{
   sceneManagerPtr = ptr; // set the pointer to the SceneManager
   name = paddleName; // set the Paddle's name
   x = positionX; // set the Paddle's x-coordinate
} // end Paddle constructor

// destructor
Paddle::~Paddle()
{
   // empty body
} // end Paddle default destructor

```

Fig. 27.11 | `Paddle` class member-function definitions. (Part 1 of 2.)

```
22 // add the Paddle to the scene
23 void Paddle::addToScene()
24 {
25    Entity *entityPtr = sceneManagerPtr->
26       createEntity( name, "cube.mesh" ); // create an Entity
27    entityPtr->setMaterialName( "paddle" ); // set the Paddle's material
28    entityPtr->setNormaliseNormals( true ); // fix the normals when scaled
29    nodePtr = sceneManagerPtr->getRootSceneNode()->
30       createChildSceneNode( name ); // create a SceneNode for the Paddle
31    nodePtr->attachObject( entityPtr ); // attach Paddle to the SceneNode
32    nodePtr->setScale( .02, .3, .1 ); // set the Paddle's size
33    nodePtr->setPosition( x, 0, 0 ); // set the Paddle's position
34 } // end function addToScene
35
36 // move the Paddle up and down the screen
37 void Paddle::movePaddle( const Vector3 &direction )
38 {
39    nodePtr->translate( direction ); // move the Paddle
40    if ( nodePtr->getPosition().y > 52.5 ) // top of the box
41       nodePtr->setPosition( x, 52.5, 0 ); // place Paddle at top of box
42    else if ( nodePtr->getPosition().y < -52.5 ) // bottom of the box
43
44       // place the Paddle at the bottom of the box
45       nodePtr->setPosition( x, -52.5, 0 );
46 } // end function movePaddle
```

Fig. 27.11 | `Paddle` class member-function definitions. (Part 2 of 2.)

First we create an `Entity` to represent the `Paddle` on the screen (lines 25–26). Here we use the name supplied to the constructor as the `Entity` name. We can't just use `"Paddle"` as we used `"Ball"` because each `Entity` must have a unique name, and there are two `Paddles` in the game. We use the cube mesh provided with the Ogre SDK as the model for the `Paddle`. The cube mesh is located in the `OgreSDK\media\models` folder. We color both `Paddles` dark orange with the same `material` (Fig. 27.12). This `material` script is almost identical to the script used for the ball. The only differences are the name of the `material` (line 2) and the color values (lines 11–13).

Then we create a child `SceneNode` of the root node to hold the data for the `Paddle` (Fig. 27.11, lines 29–30). We use the name provided to the constructor for the `Node`'s name as we did for the `Entity`. This is allowed because `Nodes` and `Entity` objects are separate types, so there is not a name conflict.

Next we attach the `Entity` to the node (line 31) and scale the node to an appropriate size (line 32). The cube mesh is $100 \times 100 \times 100$, but we scale it to $2 \times 30 \times 10$ to make it an appropriate size for a `Paddle`. We also set the `Entity` to recalculate its normals (line 28) as we did with the `Ball`. The only new Ogre function we use is `setPosition` (line 33). This function places the node at the given coordinates in the scene relative to its parent. We didn't need to use this function in the `Ball` class, because we wanted `Ball`'s `SceneNode` to start at (0, 0, 0), which is the default position of any node attached to the root node. We want the `Paddle` to be positioned at the edge of the screen, so we have to move it there. In line 33, `x` is a data member of class `Paddle` that defines the `Paddle`'s *x*-coordinate.

```
// paddle material definition
material paddle
{
   // define one technique for rendering a Paddle
   technique
   {
      // render a Paddle in one pass
      pass
      {
         // color the Paddle dark orange
         ambient 1 0.549 0
         diffuse 1 0.549 0
         specular 1 0.549 0 120
      }
   }
}
```

Fig. 27.12 | `Paddle` material script.

Add the Walls

Now we'll add the box that contains the bouncing `Ball` and the moving `Paddles`. We create this box in the `createScene` function of class `Pong` (Fig. 27.5, lines 128–168). We use the same cube mesh, provided with the Ogre SDK, for all four walls—scaling the walls appropriately to make the box and recalculating the normals for lighting. The walls are added to the scene similarly to the `Ball` and `Paddles`. We create an `Entity` using the cube mesh to represent each wall. We use a simple material to color all the walls cyan (Fig. 27.13). The material script looks just like the other two we've seen, only the name and color values differ.

```
// wall material definition
material wall
{
   // define one technique for rendering a wall
   technique
   {
      // render a wall in one pass
      pass
      {
         // color the wall cyan
         ambient 0 0.545 0.545
         diffuse 0 0.545 0.545
         specular 0 0.545 0.545 120
      }
   }
}
```

Fig. 27.13 | Wall material script.

Now position and scale the walls. The left and right walls are each placed 95 units from the origin in the *x*-direction. The top and bottom walls are each placed 70 units from the origin in the *y*-direction. Each wall is then scaled to the correct size. The top and bottom walls are positioned 140 units apart in the *y*-direction. We give both a width of 5 units. This width is an arbitrary value. You can change the width to make the game look as you want it to. If you change this, you'll also have to change the collision detection code—you'll see why when we discuss the collision logic. For the left and right walls to stretch between the top and bottom walls, they must be 145 units long (140 plus the half-width of each wall). So the *x*-scaling factor for the left and right walls is 1.45. The left and right walls are also given a width of 5 units and are positioned 185 units apart in the *x*-direction. For the top and bottom walls to stretch between the left and right walls, they must be 195 units long, so their *y*-scaling factor is 1.95.

Add the Text

We use an Ogre `Overlay` to display the score of the game as text. An **`Overlay`** refers to anything you want to render on top of the 3D elements of the scene. We use `Overlays` only for text in this chapter. The `Overlay` is defined by a script saved in a `.overlay` file.

`Overlays` are composed of `OverlayElements`. The first element in an `Overlay` must be an `OverlayContainer`. An **`OverlayContainer`** can hold any type of `OverlayElement`. A **`TextAreaOverlayElement`** holds the text that will be displayed on the screen. Every object in an `Overlay` has three main attributes—metrics mode, position and size. The position is determined by the top-left corner of the object and is always relative to the parent `OverlayContainer` of the object. Size is determined by width and height. The **metrics mode** determines how the object is positioned and sized. **Pixel mode** will size the object based on the width and height declared in pixels. **Relative mode** will position and size the object relative to the size of the parent `OverlayContainer` (or the window if it's the outermost `OverlayContainer`). In relative mode, size and position values range from 0.0 to 1.0—think of it as a percentage of the parent `OverlayElement`'s size. If you position an element at (0.0, 0.0), it will be at the top-left corner of the parent element; (0.5, 0.0) would be 50 percent across the top.

To display the score, we create an `Overlay` (Fig. 27.14). Line 2 names the `Overlay` `Score`. A single overlay file can hold several `Overlay` definitions. Ogre will reference each `Overlay` by the name rather than the file. The ***z*-order** of the `Overlay` (line 5) is used to define what this `Overlay` should be rendered over. When using multiple `Overlays`, an `Overlay` with a higher *z*-order will be rendered on top of an `Overlay` with a lower *z*-order. Lines 8–58 create a **`PanelOverlayElement`** container that holds two `TextAreaOverlayElements`. The `OverlayContainer` is positioned in the top-left corner of the screen (lines 13–14) and runs along the entire width (line 17). The container is 10 percent of the height of the window (line 18). The first `TextAreaOverlayElement` (lines 21–38) is positioned at the top of the container 5 percent away from the left side, runs half the width and is the same height as the container (lines 28–31). The other `TextAreaOverlayElement` (lines 40–57) is positioned 69 percent of the way across the top of the container and runs to the end. The `TextAreaOverlayElements` also declare a font to use (which is defined by a script in a `.fontdef` file in the `PongResources` folder), the character height, font color (note the British spelling, "colour") and the caption (lines 34–37 and 53–56).

Figure 27.15 is the `.fontdef` file that defines the `BlueBold` font. Line 2 gives the font a name that Ogre will refer to. Line 5 tells Ogre what type of font it is. True type is a common font file format (a `.ttf` file). The source (line 8) is the file that contains the font. We put the `.ttf` file in the same folder as the `.fontdef` file. If you place the two files in

```
// An Overlay to display the score
Score
{
   // set a high z-order, displays on top of anything with lower z-order
   zorder 500

   // create a PanelOverlayElement container to hold the text areas
   container Panel(ScorePanel)
   {
      // use relative metrics mode to position this container at the
      // top-left corner of the screen
      metrics_mode relative
      left 0.0
      top 0.0

      // make it 1/10 the height and the full width of the screen
      width 1.0
      height .1

      // create a TextAreaOverlayElement to display player 1's score
      element TextArea(left)
      {
         // position and size the element relative to the container
         metrics_mode relative

         // position it at the top of the container 5% from the left and
         // make it the same height and half as long as the container
         left 0.05
         top 0.0
         width 0.5
         height 1.0

         // set font used for caption and set the size and color
         font_name BlueBold
         char_height .05
         colour 1.0 0 0
         caption Player 1 Score: 0
      }
      // create a TextAreaOverlayElement to display player 2's score
      element TextArea(right)
      {
         // position and size the element relative to the container
         metrics_mode relative

         // position it at the top of the container 69% from the left and
         // make it the same height as the container, stretch to the end
         left 0.69
         top 0.0
         width 0.5
         height 1.0

```

Fig. 27.14 | `Overlay` script to display the score. (Part 1 of 2.)

```
            // set font used for caption and set the size and color
            font_name BlueBold
            char_height 0.05
            colour 1.0 0 0
            caption Player 2 Score: 0
         }
      }
   }
```

Fig. 27.14 | `Overlay` script to display the score. (Part 2 of 2.)

different locations, you'll have to specify the path to the `.ttf` file in line 8 or add the folder containing the `.ttf` file as a resource location (discussed in Section 27.4.8).

Lines 115–117 of class Pong (Fig. 27.5) display the score on the screen. We use the static member function `getSingleton` of class `OverlayManager` to get a pointer to the `OverlayManager` object. We use that object to get a pointer to the score `Overlay`, then we call the `show` function to display it on the screen. When a player scores, we need to update the text within the `Overlay` to reflect the change (lines 192–210). First we create the new text. Then we get a pointer to the appropriate `TextAreaOverlayElement` from the `OverlayManager` and use the `TextAreaOverlayElement` member function `setCaption` to replace the text.

```
1  // define the BlueBold font
2  BlueBold
3  {
4     // define the font type
5     type truetype
6
7     // set the source file for the font
8     source bluebold.ttf
9
10    // set the font size
11    size 16
12
13    // set the font resolution (96 is standard)
14    resolution 96
15 }
```

Fig. 27.15 | BlueBold font definition script.

27.4.4 Animation and Timers

Now that we know how to draw a `Ball` on the screen, animating it and making it move around the screen is straightforward. The function `moveBall` (Fig. 27.8, lines 54–108) moves the `Ball` around the screen. In most Pong games, the ball can travel at many different angles. However, since we are just starting out with Ogre, we want to keep things as simple as possible. For this reason, in our Pong game, the ball has only four possible directions of travel: down-right, up-right, down-left, and up-left—all at 45-degree angles to the *x*- and *y*-axes in our program.

Line 56 actually makes the Ball move; the rest of the function controls collisions with various objects within the scene, as we'll see shortly. The translate function takes as an argument a Vector3, which is a three-dimensional vector type defined by Ogre. The vector represents the direction and distance to move the Ball. We pass to the translate function the Ball's direction multiplied by the distance to travel (speed × time) to determine the final vector. The speed parameter is the number of units the ball will move per second. The time parameter is the number of seconds since the last time the Ball was moved. We'll see where this comes from in just a moment. SceneNode translations are done in **parent space** by default. That means that the node is translated with respect to its parent node's position and orientation (i.e., the direction the node is facing). Translations can also be performed in world or local space by adding another parameter to the translate function (TS_LOCAL or TS_WORLD). Translations in **world space** are done with respect to the origin of the scene (0, 0, 0). Translations in **local space** are done with respect to the node's origin (wherever the node is positioned and whichever direction it's facing).[2]

To continuously move the Ball across the screen, call the moveBall function each time a new frame is rendered. Figure 27.4 defines the Pong class, our game's main driving class, which inherits from the Ogre class FrameListener. A **FrameListener** is a class that processes Ogre::FrameEvents. A **FrameEvent** occurs every time a frame begins or ends. Every FrameListener has two functions, frameStarted and frameEnded (lines 29–30). These functions both return a bool. Ogre keeps rendering frames until one of these functions returns false. We use the frameStarted function (Fig. 27.5, lines 271–295) to control the animation of our Ball, specifically line 278. This function is called by Ogre before each new frame is rendered. Before every frame, the frameStarted function calls the Ball class member function moveBall, which continuously moves the Ball across the screen. As discussed earlier, controlling the speed of the animation is vital to creating smooth motion. Frame rates (i.e., how quickly the scene gets redrawn) may vary greatly on different machines, so the Ball could move at a different speed on each one. For that reason we pass the FrameEvent's data member timeSinceLastFrame, in seconds, to the moveBall function. We multiply this time by the Ball's speed to determine the distance the Ball should move across the screen. This is an example of using a timer to control animation.

27.4.5 User Input

Now we discuss moving the Paddle up and down on the screen with the movePaddle function of class Paddle (Fig. 27.11, lines 37–46). To move the Paddle, we again use the SceneNode function translate (line 39). Rather than moving the Paddle based on time, we move it based on user input from the keyboard. The user specifies a direction, up or down, by pressing the corresponding key, and the Paddle moves accordingly. The direction is passed to movePaddle as a Vector3.

Ogre does not directly support user input from devices such as the keyboard, mouse or joystick. The Ogre SDK does come with the **Object Oriented Input System (OIS)** for handling user input. It isn't required that you use OIS for input with Ogre, but it's a good choice.

We need to create an InputManager, a Keyboard and a KeyListener to handle the user input and control the calls to movePaddle. The **InputManager** is used to create the

2. Junker, Gregory, *Pro OGRE 3D Programming*. New York: Springer-Verlag, 2006, pp. 82—89.

various input devices. We create the `InputManager` in class `Pong`'s constructor (Fig. 27.5, line 71). To create the `InputManager` we must provide it with a window in which to collect (lines 62–68).

We create a **`Keyboard`** object which represents the actual keyboard. To collect `KeyEvent`s, we must call the `capture` function of class `Keyboard`. We want to call this function repeatedly, so we place it in the `frameStarted` function, which is called at the beginning of every frame. Class `Pong` inherits from class **`KeyListener`**, an OIS class that handles input from the keyboard. We register it with the `Keyboard` (line 74) to receive **`KeyEvents`**, i.e., indications that the player has hit a key. A `KeyListener` defines two member functions (Fig. 27.4, lines 25–26)—we use only one of these (line 25). We must implement the other one, though, because they are both declared pure `virtual` in the class `KeyListener`.

The implementation of the key-handling function is in lines 213–256. Every time we capture a key press, the `Keyboard` sends the `KeyEvent` to this member function. OIS defines an enumeration of all the keys on the keyboard, which we use to determine which key was pressed. The `switch` statement (lines 219–242) responds only to certain keys. We extract the key enumeration from the `KeyEvent` and pass it to the `switch` statement (line 219). If the *A* or *Z* key is being pressed, the `Paddle` on the left side should move up or down, respectively. Likewise, if the user presses the up- or down-arrow key, the `Paddle` on the right side should move in the corresponding direction. The directions passed to the `movePaddle` function are defined as constant `Vector3`s (lines 22–23).

We allow the user to pause the game by hitting the *P* key (lines 236–241), which sets the `Pong` data member `pause` to `true`. The `if` statement (line 216) will skip the `switch` statement that controls the `Paddle` movement when `pause` is `true`. The `pause` data member will also stop the `Ball` from moving when it's `true` (line 275). We also use an `Overlay` (Fig. 27.16) to display `"Game Paused"` on the screen. The game resumes when the player hits the *R* key.

```
// An Overlay to display "Game Paused" when the player pauses the game
PauseOverlay
{
   // set a high z-order, displays on top of anything with lower z-order
   zorder 500

   // create a PanelOverlayElement container to hold the text area
   container Panel(Pause)
   {
      // use relative metrics mode to position and size this container
      metrics_mode relative

      // place the container at the top-left corner of the window
      left 0.0
      top 0.0

      // make the container the same size as the window
      width 1.0
      height 1.0
```

Fig. 27.16 | `Overlay` script to display `"Game Paused"` when player pauses the game. (Part 1 of 2.)

```

      // create a TextAreaOverlayElement to display the text
      element TextArea(pauseText)
      {
         // position and size the element relative to its container
         metrics_mode relative

         // center it vertically in the container
         vert_align center

         // put the left corner 4/10 from the left of the container and
         // make it 2/10 the width of the container and 1/10 the height
         left 0.4
         width 0.2
         height 0.1

         // set the font used for caption and set the size and color
         font_name BlueBold
         char_height 0.05
         colour 0 0 0
         caption Game Paused
      }
   }
}
```

Fig. 27.16 | `Overlay` script to display `"Game Paused"` when player pauses the game. (Part 2 of 2.)

If the user hits the *Esc* key, the game exits by setting the `quit` data member to `true` (Fig. 27.5 lines 221–223). Recall that Ogre continues to render frames until the `frameStarted` or `frameEnded` function returns `false`. These both return `!quit`, so when we set `quit` to `true`, the functions return `false` and tell Ogre to shut down. If you don't use the *Esc* key to quit, the program won't stop properly; it keeps running in the background. Be sure to use the *Esc* key.

27.4.6 Collision Detection

The `Ball` collides with a number of objects as it bounces around the screen. We need to detect these collisions to make the `Ball` interact correctly with its surroundings. Lines 60–105 of Fig. 27.8 control collisions between the `Ball` and the walls of the playing area. The call to `SceneNode` member function `getPosition` returns a `Vector3` representing the node's position relative to its parent node. Because all of our nodes are direct children of the root node, whose position is (0, 0, 0), the position returned is always relative to the origin. There is also a `_getDerivedPosition` member function that will return the position relative to the origin of any node regardless of its parent's position.

You can retrieve any node within the scene graph by passing the name of the node to the `SceneManager` member function `getSceneNode`. We retrieve the nodes of the four walls (lines 60–67) and use their positions to check for collisions with the `Ball`. If the `Ball`'s *x*-coordinate (minus the radius) is less than or equal to the left wall's *x*-coordinate (plus half the wall's width), then the `Ball` has collided with the left wall. Once the collision is determined, the proper actions are taken. The `Ball` is placed in the middle of the screen for the next round (line 74). Player 2 is given a point (line 75) by calling the `Pong` class member function `updateScore`. We made the `updateScore` function `static` so we can call it from the `Ball` class without a reference to an instance of the `Pong` class. Finally, a sound is played to indicate that a player has scored (line 76)—we'll explain that function call in Section 27.4.7. The process is the same to determine if the `Ball` hit the right side. The `Ball`'s *x*-coordinate is checked against the right wall's *x*-coordinate. If the `Ball` hits the right side, the appropriate actions are taken. Figure 27.17 shows player 1 scoring a point. The `Ball` is not actually going into the wall; it's an illusion caused by the 3D graphics.

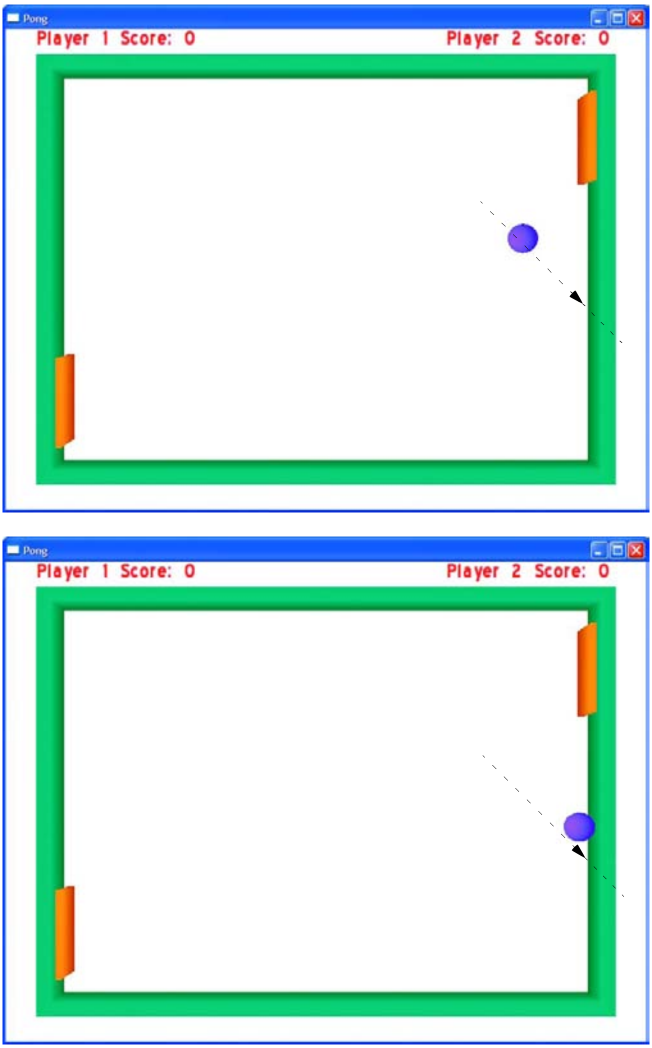


Fig. 27.17 | Player 1 scoring a point. (Part 1 of 2.)

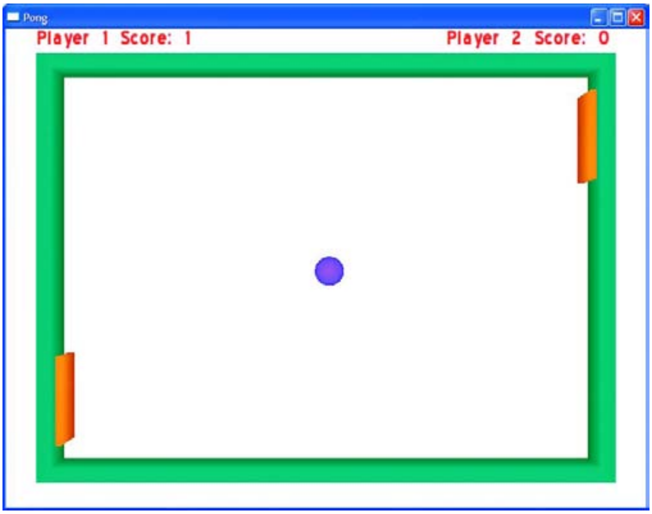


Fig. 27.17 | Player 1 scoring a point. (Part 2 of 2.)

The `Ball` is then checked against the top and bottom walls. The same collision logic is used. If the `Ball`'s *y*-coordinate (plus or minus the radius, depending on which wall it hits) after being moved would cross the wall's inner *y*-coordinate (which is physically impossible because they are both solid objects), the `Ball` has collided with either the top or bottom wall. To prevent the `Ball` from overlapping the wall, we place it at the edge of the wall after a collision (lines 92–93 and 102–103). Technically this violates the physics of the `Ball` by changing the distance it moved in the given time interval. To be accurate, we would have to determine the distance and direction it moved after hitting the wall and draw it at that point. In the interest of keeping the code simple, we don't deal with this issue. The scene gets redrawn so quickly that the distance the Ball moves each frame is extremely small. The effect on the `Ball`'s movement is imperceptible. Figure 27.18 shows the `Ball` bouncing off the top wall.

At the end of `moveBall` we call the function `hitPaddle` (lines 125–173) to check for collisions between the `Ball` and the `Paddles` and take appropriate actions when one occurs. Lines 128–129 retrieve the node that the left `Paddle` is attached to, then return the position of the node. Lines 130–131 do the same thing for the right `Paddle`. We use these positions to detect collisions between the `Paddles` and the `Ball`. The logic is similar to the logic used for checking the walls. We first check if the `Ball`'s *x*-coordinate is past the `Paddles`'. We also check that the `Ball` is within the `Paddles`' *y*-coordinates. Figure 27.19 shows the `Ball` bouncing off a `Paddle`.

Consider the lines that change the ball's direction. Line 113 causes the `Ball` to start moving left if it's currently moving right, and start moving right if it's currently moving left. Line 120 makes the `Ball` start moving up if it's currently moving down, and down if it's currently moving up. Why does this work? The direction of the `Ball` is determined by a `Vector3`. Each value represents a distance along the *x*-, *y*- or *z*-axis. A positive *x*-value means the `Ball` will move right along the *x*-axis, and a negative value will move the `Ball`

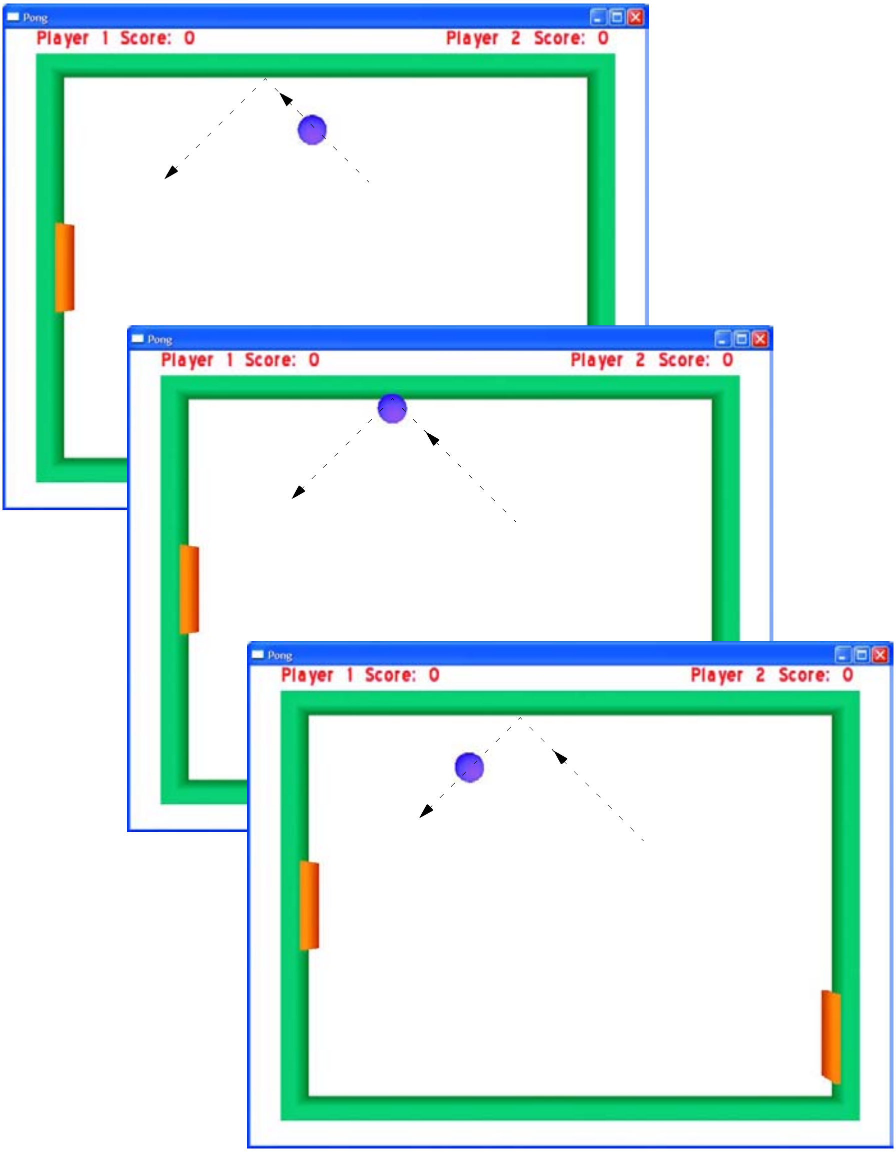


Fig. 27.18 | The `Ball` bouncing off the top wall.

left. If the `Ball` is moving right, multiplying its *x*-value by –1 will change the sign and reverse the direction. We do the same thing to change the vertical direction.

The collisions in our game are fairly simple cases, so we've kept the logic simple. There are whole libraries dedicated to handling collisions and physics, such as Open Dynamics Engine (ODE, `www.ode.org/`), Bullet (`www.continuousphysics.com/Bullet/`), Newton Game Dynamics (`www.newtondynamics.com/`) and PhysX (`www.ageia.com/`). These libraries have Ogre bindings available on the Ogre Community Add-on page.

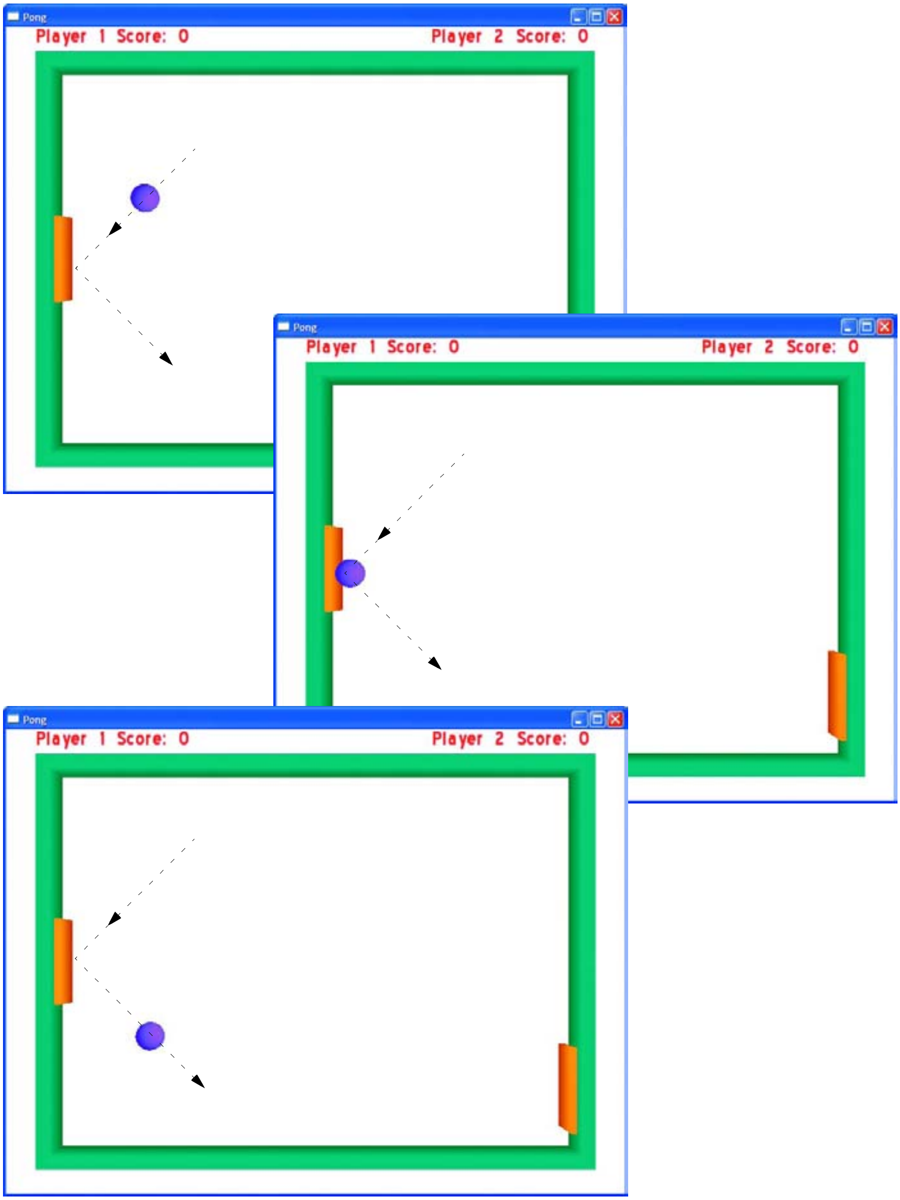


Fig. 27.19 | The `Ball` bouncing off the left `Paddle`.

27.4.7 Sound

We now discuss importing sounds and playing sound files in Ogre programs, which we'll use to "juice up" our Pong game. We play a "boing" sound whenever the `Ball` hits a wall, we play a different "boing" sound whenever the `Ball` hits a `Paddle` and we play a cheering sound whenever a player scores.

We'll use OgreAL to add sound. **OgreAL** is a wrapper around the **OpenAL audio library**. OgreAL was created by Casey Borders (`www.mooproductions.org`), a member of the Ogre community. The OpenAL library is maintained by Creative Labs, `developer.creative.com`. The wrapper allows us to integrate sound functionality into the Ogre code by attaching the sounds to nodes within the scene graph. Because all of the sounds we play relate to the `Ball`, we place the OgreAL code in the `Ball` class. The OgreAL functions used to import and play sounds are analogous to those used for importing and displaying models in Ogre.

As with Ogre, we include the `OgreAL.h` header. OgreAL manages the sounds using a **`SoundManager`** class, following the Ogre resource management scheme. We create a `SoundManager` in the `Ball` constructor (Fig. 27.8, line 14). There can be only one `SoundManager`. The `SoundManager` is used to create instances of **`Sounds`**, the OgreAL objects that contain the sound data. We create three `Sounds` and attach them to nodes (lines 39–50). The `createSound` function takes three parameters. The first is an `Ogre::String` that will be the name of the `Sound`. The second is the name of the sound file associated with the `Sound`. The third is a `bool` that determines if the `Sound` should be looped to continue playing. Passing `false` will play the `Sound` through once, then stop. Passing `true` will continuously loop the sound until you stop it. We attach the `Sounds` to a node in the same way we would attach an `Entity` to a node with the `attachObject` function.

The first `Sound` we create (lines 39–40) will be played whenever the `Ball` bounces off the top or bottom wall. We attach it to the `Ball`'s node. OpenAL supports 3D sound, so when the `Sound` is attached to the `Ball`, it will play from wherever the `Ball` is. Our scene is relatively small, so you may not notice the `Sound` being played in 3D, but if you listen closely, it will sound slightly different. Because we placed the call to the `play` function (line 121) inside our function that reverses the `Ball`'s vertical direction, the "boing" sound will play whenever the `Ball`'s vertical direction is reversed—in other words, whenever the ball hits the top or bottom wall.

The second `Sound` is created the same way as the first, and again is attached to the `Ball`'s node. This sound will play whenever the `Ball` bounces off one of the `Paddles`. We play this `Sound` within the `reverseHorizontalDirection` function (line 114) for the same reason we play the other sound from `reverseVerticalDirection`.

The third `Sound` will play whenever a player scores. There is no particular location to play this `Sound` from—we attach it directly to the root node of the scene graph, which is positioned at the origin. We play the `Sound` from the `moveBall` function every time it's determined that a player has scored (lines 76 and 84).

There are a couple of things to note about `Sounds` in OgreAL. Each `Sound` must have a unique name, just like `Entity` objects and `Nodes`. A `Sound` must finish playing before it can be played again.

27.4.8 Resources

As mentioned earlier, Ogre uses scripts to create `Materials`, `Overlays` and some other advanced features that are beyond the scope of this text. Ogre also uses `.mesh` files to represent 3D objects. OgreAL uses sound files. All of these resources must be loaded before we can use them. Ogre will throw a runtime exception if you try to use a resource that hasn't been loaded. To manage the game's resources we use a **`ResourceGroupManager`**. To load the resources for our game, we first tell the `ResourceGroupManager` where to find them.

The `addResourceLocation` function (Figure 27.5, lines 79–80) takes three `Ogre::String` arguments. The first is the location of the resources. We placed all the resources in a folder called `resources` within the example's directory. Normally you'd organize the resources in different folders by type, e.g. materials, models and overlays. But for simplicity we maintain one folder to hold all the resources we'll need for the game. The second argument is the type of file the resources are in. The third is the resource group these files belong to. We'll put these files in the `"Pong"` group. Now we load the resources in the location we just added (line 81).

27.4.9 Pong Driver

The last step is to write a `main` function (Fig. 27.20). Ogre supports various platforms, so you should try not to write platform-specific code when you can avoid it. The preprocessor `if else` wrapper (lines 8–17) will determine whether the program is running on a Windows platform. If it is, it will include the `windows.h` header and define the `WinMain` function. If not, it will define the normal `main` function. This allows the code to run on various platforms without having to be changed. You may not have seen the Windows-specific code before. The preprocessor directive to include the `windows.h` header gives the program the necessary access to the Windows API to run our program. The definition of `WIN32_LEAN_AND_MEAN` (line 9) will exclude rarely used headers in the `windows.h` header. This will speed up the compilation time for our program.

```
1  // PongMain.cpp
2  // Driver program for the game of Pong
3  #include <iostream>
4  #include "Pong.h" // Pong class definition
5  using namespace std;
6
7  // If running on Windows, include windows.h and define WinMain function
8  #if OGRE_PLATFORM==PLATFORM_WIN32 || OGRE_PLATFORM==OGRE_PLATFORM_WIN32
9  #define WIN32_LEAN_AND_MEAN
10 #include "windows.h"
11
12 int WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
13
14 // If not, define normal main function
15 #else
16 int main()
17 #endif
18 {
19    try
20    {
21       Pong game; // create a Pong object
22       game.run(); // start the Pong game
23    } // end try
24    catch ( runtime_error &error )
25    {
26 #if OGRE_PLATFORM==PLATFORM_WIN32 || OGRE_PLATFORM==OGRE_PLATFORM_WIN32
27       MessageBoxA( NULL, error.what(), "Exception Thrown!",
28          MB_OK | MB_ICONERROR | MB_TASKMODAL );
```

Fig. 27.20 | Driver program for the game of Pong. (Part 1 of 2.)

```
29  #else
30          cerr << "Exception Thrown: " << error.what() << endl;
31  #endif
32        } // end catch
33  } // end main
```

Fig. 27.20 | Driver program for the game of Pong. (Part 2 of 2.)

The `main` function creates the initial `Pong` object in a `try` block (lines 19–23). Recall that the `Pong` constructor throws an exception if the user cancels the Ogre configuration dialog box. If the user hits **OK** in the dialog box, the `Pong` object is created and we call the `run` member function (line 22). Class `Pong` member function `run` (Fig. 27.5, lines 172–176) first creates the game's scene (line 174), then calls the `startRendering` member function (line 175) of class `Root` to render the scene repeatedly until either the `frameStarted` or the `frameEnded` function returns `false`.

27.5 Wrap-Up

In this chapter you learned the basics of creating computer games with Ogre. We discussed the basic concepts of graphics, briefly describing models, lighting and colors. You saw how to use the free Ogre 3D rendering engine to produce a 3D game. We showed you how to use the `SceneManager` to create and manage your scene. You learned how to use a `Camera` to view your scene. We discussed responding to user input from the keyboard with OIS. We demonstrated how to move an object at a constant speed. We covered the basics of collision detection and showed how important it is to game programming. You learned how to display text on the screen using `Overlays`. We showed how Ogre uses scripts to manage materials and `Overlays` without having to recompile every time you change them.We also showed how to add sound to your games using the OgreAL wrapper for OpenAL.

This chapter should be viewed only as an introduction. We presented you with a basic example of Pong. Use it as a foundation for your own version. Go out and find your own sounds to use. Add new features to the game. Explore Ogre's other capabilities and create some cool visual effects. Really make this game your own. Game programming is all about being creative.

27.6 Ogre Web Resources

`www.ogre3d.org/`
The Ogre home page. Here you can find the latest Ogre news, download Ogre or Ogre-related tools, browse the documentation or check out projects that use Ogre.

`www.ogre3d.org/index.php?option=com_content&task=view&id=411&Itemid=131`
Prebuilt SDK download page. There are SDKs available for Code::Blocks + MingGW C++ Toolbox, Visual C++ .Net 2003 and Visual C++ .Net 2005 (must install Service Pack 1).

`www.ogre3d.org/index.php?option=com_content&task=view&id=412&Itemid=132`
Ogre source-code download page. Source code available for Windows, Linux and Mac OS X. Also download the third-party dependencies package for your platform. Also has a link to a guide to building Ogre from source.

`www.ogre3d.org/index.php?option=com_content&task=view&id=415&Itemid=144`
Instructions on getting the Ogre source code from the CVS directory.

`www.ogre3d.org/wiki/index.php/Installing_An_SDK`
Installation instructions for the Ogre SDK on Windows with Visual C++, Code::Blocks + MinGW, Code::Blocks + MinGW + STLPort, Eclipse + MinGW + STLPort and GCC & Make/Any IDE. Linux, Debian, Gentoo, Fedora and Ubuntu. Mac OS X.

`www.ogre3d.org/wiki/index.php/Building_From_Source`
Instructions for building the Ogre source code on Windows with Visual C++, Visual C++ Toolkit 2003 & Code::Blocks, and GCC. Linux with GCC & Make, Debian, Fedora, Gentoo, Ubuntu/Kubuntu. Mac OS X with Xcode.

`www.ogre3d.org/wiki/index.php/BuildFAQ`
Solutions to common errors when building from the source code. Errors include not being able to find files, unresolved external symbols and other types of errors.

`www.ogre3d.org/wiki/index.php/SettingUpAnApplication`
Guide to setting up an Ogre Application project on Visual C++, Code::Blocks, GCC, Autotools, Scons, Eclipse, Anjuta IDE, KDevelop IDE.

`www.ogre3d.org/phpBB2addons/viewtopic.php?t=3293`
OgreAL download and installation instructions.

`developer.creative.com/landing.asp?cat=1&sbcat=31&top=38`
OpenAL download and installation links.

`www.openal.org/downloads.html`
OpenAL download page.

`www.wreckedgames.com/wiki/index.php/WreckedLibs:OIS`
Object Oriented Input System (OIS) wiki page includes links to the OIS manual and API reference.

`www.tayloredmktg.com/rgb/`
Color-code chart. Gives RGB values in hex and decimal. Colors are divided into general color range (e.g., grays, blues, greens, oranges).

`www.htmlcenter.com/tutorials/tutorials.cfm/89/General/`
Color chart gives RGB and hex color values.

Tutorials

`www.ogre3d.org/wiki/index.php/Ogre_Tutorials`
Ogre tutorials page. Tutorials range from basic to advanced on topics including introduction to Ogre, FrameListeners, animation multiple SceneManagers and content creation.

`www.blender.org/tutorials-help/`
Blender tutorials page.

`en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro`
The "Blender 3D: Noob to Pro" wikibook guides new Blender users through the process of 3D modeling. Teaches you how to work with models, lighting, rendering, animation, particles and soft bodies. Also has advanced tutorials on python scripting and advanced animation.

`www.cegui.org.uk/wiki/index.php/Tutorials`
Many tutorials on using Crazy Eddie's GUI System (CEGUI) that is supported by Ogre.

Tools

`www.ogre3d.org/index.php?option=com_content&task=view&id=413&Itemid=133`
Download model export tools for Blender, Maya, Softimage XSI and 3DS Max.

`usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7639525`
Autodesk Maya Personal Learning Edition page. Free version of Maya.

`www.softimage.com/downloads/default.aspx`
SoftImage XSI download page. Free 30-day trial available.

`www.blender.org/download/get-blender/`
Blender download page.

Code Examples

`www.ogre3d.org/wiki/index.php/CodeSnippits#HOWTO`
The Ogre Cookbook contains code samples explaining how to do various tasks relating to geometry, rendering, materials, textures, animation, input GUI and sound.

`www.ogre3d.org/phpBB2/viewtopic.php?t=27326`
Asteroid Wars. A game written using Ogre for graphics. Source code is available.

`www.ogre3d.org/phpBB2/viewtopic.php?t=27806`
Five games written with Ogre. The source code for all the games is available.

Books

`www.amazon.com/Pro-OGRE-3D-Programming/dp/1590597109/ref=pd_bbs_sr_1/102-2583408-2260151?ie=UTF8&s=books&qid=1173888297&sr=1-1`
Pro OGRE 3D Programming, by Gregory Junker.

Forums

`www.ogre3d.org/phpBB2/viewtopic.php?t=5706`
A forum post describing how to install Ogre on Debian GNU/Linux.

`www.ogre3d.org/phpBB2/viewforum.php?f=2`
Ogre Help forum. Get help from Ogre users on any problems you encounter while using Ogre.

`www.ogre3d.org/phpBB2/`
A number of forums on the Ogre site, including help, using Ogre in practice, content creation, programming basics and more.

`www.ogre3d.org/phpBB2addons/`
Ogre add-ons forums. A number of forums dedicated to the more popular Ogre add-ons, including OgreAL, OgreODE, NxOgre, PyOgre and more.

`www.ogre3d.org/phpBB2addons/viewforum.php?f=10`
OgreAL forum on the Ogre site. Information on installing and using OgreAL. Great place to find help.

`www.wreckedgames.com/forum/viewforum.php?f=6&sid=dc5f903554a80ac5194213329f5e46e4`
OIS forum. Get help on using OIS.

Summary

Section 27.3 Basics of Game Programming

- 3D graphics engines hide the tedious and complex programming required with graphics APIs.
- Ogre supports the Direct3D and OpenGL graphics APIs and runs on the Windows, Linux and Mac platforms.
- Ogre is strictly a graphics rendering engine. The Ogre community has produced many add-ons that allow users to integrate other libraries with Ogre to support those features.
- A 3D model is a computer representation of an object which can be drawn on the screen.
- Materials determine an object's appearance by setting lighting properties, colors and textures.
- A texture is an image that is wrapped around the model.

- Colors are determined by red, green and blue light intensities and an optional alpha value to represent transparency. Values can range from 0 to 1.0.
- There are four different types of light in a 3D scene—ambient, diffuse, emissive and specular.
- Collision detection is the process of determining whether two objects in a game are touching and reacting appropriately.
- There are collision detection and physics modeling libraries that handle the complexities for you.
- Audio libraries enrich your games with sound. Many of those libraries support 3D sound.
- Games often communicate with the user by displaying text.
- Timers control animation speed and make animations look more natural.
- User input devices include the keyboard, mouse, joystick and the game controller.

Section 27.4.1 Ogre Initialization

- `Root` is the base object used in Ogre used to start the engine. No Ogre calls can be made until the `Root` object has been created.
- Call the `showConfigDialog` function of the `Root` class to display the dialog. The **OGRE Engine Rendering Setup** dialog box enables the user to choose the rendering settings.
- The resolution is defined by two values, width and height, which determine the number of pixels used to draw the scene. A higher resolution will produce more detailed graphics.
- A color depth of n bits means that 2^n possible colors can be displayed on the screen.
- The `RenderWindow` is a window in which Ogre will render graphics.

Section 27.4.2 Creating a Scene

- A scene is a collection of images that make up our graphics.
- The `SceneManager` manages the scene graph, a data structure that contains all the scene's objects.
- The `SceneManager` is used to create objects and determine which objects will be rendered. An Ogre application can use more than one `SceneManager`.
- A `Camera` is the eye through which you view the scene. `Camera`s can be placed at any location in the scene or attached to `SceneNode`s. Ogre supports multiple `Camera`s in a single scene.
- The `Viewport` is the area of the screen used to display what the `Camera` can see. A `Camera` can have more than one `Viewport`.
- Ogre has three types of `Light`s—`Point`, `Spot` and `Directional`. `Light`s are created with the `createLight` function of class `SceneManager`.

Section 27.4.3 Adding to the Scene

- An `Entity` is an instance of a mesh within the scene. A mesh is a file that contains the geometry information of a 3D model. Many `Entity` objects can be based on the same mesh, as long as each `Entity` has a unique name.
- Use the `SceneManager` to create `SceneNode`s that hold information about an object and its position in the scene.
- The root node is the parent of all other nodes. When you create a child of the root node, its initial position is (0, 0, 0).
- Attach `Entity` objects to `SceneNode`s with the `attachObject` function of class `SceneNode`.
- `scale` changes the size of the `Entity` attached to the `SceneNode`, but it does not affect the size of the actual mesh that the node's `Entity` is based on. `setScale` changes the size based on the orig-

inal size of the `Entity`. These functions also scale all children of the `SceneNode` by the same factor. To change that, call the `setInheritScale` function and pass it `false`.

- `setPosition` function places the node at the given coordinates in the scene.
- Ogre uses a `material` script to create a `material`. Save the file with a `.material` extension. A `material` file can define multiple materials; every `material` must have a unique name.
- An `Overlay` is defined by a script saved in an `.overlay` file. A single `.overlay` file can hold several `Overlay` definitions. Every object in an `Overlay` has three main attributes—metrics mode, position and size.
- `Overlays` are composed of `OverlayElements`. The first element in an `Overlay` must be an `OverlayContainer`. An `OverlayContainer` can hold any `OverlayElement`. A `TextAreaOverlayElement` holds text. Call the `show` function to display the `Overlay` on the screen
- Use `TextAreaOverlayElement` to display text. Call `setCaption` to change text on the screen.
- An `Overlay` with a higher *z*-order will be rendered on top of an `Overlay` with a lower *z*-order.
- Fonts are defined by a script in a `fontdef` file.
- Use the static member function `getSingleton` of class `OverlayManager` to get the `OverlayManager` object.

Section 27.4.4 Animation and Timers

- The `translate` function moves a `SceneNode`.
- `SceneNode` translations are done in parent space by default. Translations in parent space are done with respect to the parent's origin. Translations in world space are done with respect to the origin of the scene (0, 0, 0). Translations in local space are done with respect to the node's origin.
- A `FrameListener` processes `Ogre::FrameEvents`. A `FrameEvent` occurs when a frame begins or ends.

Section 27.4.5 User Input

- Ogre does not directly support user input from devices such as the keyboard, mouse or joystick.
- Use the Object Oriented Input System (OIS) for handling user input.
- The `InputManager` is used to create the various input devices. To create the `InputManager` we must provide it with a window in which to collect input.
- A `Keyboard` object collects `KeyEvents` and sends them to a `KeyListener`.
- OIS defines an enumeration of all the keys on the keyboard, which we use to determine which key was pressed.

Section 27.4.6 Collision Detection

- `getPosition` returns a `Vector3` representing the node's position relative to its parent node; `_getDerivedPosition` returns the position relative to the origin.
- The `SceneManager` can retrieve any node within the scene graph by referencing the name given to the node when it was created.
- The direction of the `Ball` is determined by a `Vector3`. A positive *x*-value means the `Ball` will move right along the *x*-axis, and a negative value will move the `Ball` left. If the `Ball` is moving right, multiplying its *x*-value by –1 will change the sign and reverse the direction.
- There are whole libraries dedicated to handling collisions and physics.

Section 27.4.7 Sound

- OgreAL is a wrapper around the OpenAL audio library. The wrapper allows us to integrate sound functionality into the Ogre code by attaching the sounds to nodes within the scene graph.

- We must have the preprocessor directive to include the `OgreAL.h` header.
- `Sound` is the OgreAL object that contains the sound data. Use the `createSound` function of class `SoundManager` to create sounds. There can be only one `SoundManager`.
- The `createSound` function takes three parameters. The first is an `Ogre::String` that will be the name of the `Sound` within the OgreAL system. The second is the name of the sound file associated with the `Sound`. The third is a `bool` that determines whether the `Sound` should be looped to continue playing. Passing `false` will play the `Sound` through once, then stop. Passing `true` will continuously loop the sound until you stop it.
- Attach the `Sounds` to a node with the `attachObject` function.
- Each `Sound` must have a unique name.
- A `Sound` must finish playing before it can be played again.

Section 27.4.8 Resources

- All of the resources must be loaded before we can use them.
- Use a `ResourceGroupManager` to manage the game's resources.
- The `addResourceLocation` function takes three `Ogre::String` arguments. The first is the location of the resources. The second is the type of file the resources are in. The third is the resource group these files belong to.

Section 27.4.9 Pong Driver

- Ogre supports various platforms, so you should try not to write platform-specific code when you can avoid it.

Terminology

Self-Review Exercises

27.1 Fill in the blanks in each of the following statements:

a) The ________ header includes the most commonly used Ogre header files.
b) The ________ object must be created before any other Ogre function (other than logging) is called.
c) The main type defined by OgreAL for pointing to sound-file data is ________.
d) A(n) ________ object is used to represent a color in Ogre.
e) The ________ header includes the most commonly used OgreAL header files.
f) ________ are used to define materials and overlays for Ogre programs.
g) The ________ object is used to load resources for Ogre programs.
h) Ogre uses a(n) ________ object to manage the scene.
i) A 3D model is defined in an Ogre ________ file.

27.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The coordinates (0, 0) refer to the bottom-left corner of an `OverlayContainer`.
b) If Ogre attempts to load an external file that does not exist, a runtime error will occur.
c) Color values in Ogre range from 0 to 255.
d) Passing a value of `false` to the `createSound` function will cause the sound file to play continuously.
e) An `Overlay` that draws text on the screen must specify a font in which that text should be drawn.
f) Every `Entity` must have a unique name.

27.3 Write statements to accomplish each of the following:

a) Attach an `Entity` pointer named `entityPtr` to a `SceneNode` pointer name `nodePtr`.
b) Scale the `Entity` from the previous question to half its original size.
c) Create the `Sound sample` that loops the `sound.wav` file.
d) If the spacebar is being pressed, set the value of the `int number` to 0.
e) Set an `Overlay Element` to position itself relative to the size of its parent `Container`.
f) Add a folder named `sounds` in the `media` folder as a `"General"` resource location.
g) Move a SceneNode 15 units left, 4 units up and 8 units toward you.

27.4 Find the error in each of the following:

a) `SceneNode node;`
b) `ColourValue( 0, 0, 255 );`
c)
```
Root *rootPtr = new Root();
rootPtr->initialize( true, "Window" );
```
d) `viewportPtr = sceneManagerPtr->addViewport( cameraPtr );`

Answers to Self-Review Exercises

27.1 a) `Ogre.h`. b) `Root`. c) `Sound`. d) `ColourValue`. e) `OgreAL.h`. f) scripts. g) `ResourceGroupManager` or `ResourceManager`. h) `SceneManager`. i) `.mesh`.

27.2
a) False. The coordinates (0, 0) refer to the top-left corner of an `OverlayContainer`.
b) True.
c) False. Color values in Ogre range from 0.0 to 1.0.
d) False. The sound will play once, then stop.
e) True.
f) True.

27.3
a) `nodePtr->attachObject( entityPtr );`
b) `nodePtr->setScale( .5, .5, .5 );`
c) `soundManagerPtr->createSound( "sample", "sound.wav", true );`
d)
```
if ( keyEvent.key == OIS::KC_SPACE )
   number = 0;
```
e) `metrics_mode relative;`
f)
```
ResourceGroupManager::getSingleton().addResourceLocation( "media/sounds",
   "FileSystem", "General" );
```
g) `sceneNodePtr->translate( -15, 4, 8 );`

27.4
a) The variable `node` should instead be declared as a pointer to a `SceneNode`. All of Ogre's `SceneNode` functions either take a pointer as a parameter or return a pointer.
b) The `ColourValue` object can accept parameters only with values between 0 and 1.
c) Ogre uses British spelling, the function is spelled `initialise`. Also, the render settings must be set before you call `initialise`.
d) `addViewport` is a function of the `RenderWindow` class, not `SceneManager`.

Exercises

27.5 Look through the resources available in our Game Programming Resource Center at `www.deitel.com/computergames/gameprogramming/` and the C++ Game Programming Resource Center at `www.deitel.com/CplusplusGameProgramming/`.

27.6 *(Pong Win Condition)* Modify the Pong game so that when a player reaches 21 points, the game ends and displays a message that the left or right player has won.

27.7 *(Ball Speed Increase)* In most Pong games, when a rally between the two players lasts for a long time, the ball begins to speed up in order to prevent a stalemate. Modify the Pong game so that the ball's speed increases for every ten times that it is hit in a rally. When either player scores, the ball should return to its original speed.

27.8 *(Paddle Speed Decrease)* Some Pong games also modify the speed of one or both players' paddles in an effort to keep the game balanced. Modify the Pong game so that when one player has a lead of at least 5 points, his or her paddle begins to slow down. The greater that player's lead, the slower his or her paddle should move. If the player's lead falls to under 5 points, his or her paddle should return to normal speed.

27.9 *(Pong Menu)* Modify the Pong game so that before the game begins, a menu appears on the screen that allows the players to choose from several different ball and paddle speeds.

27.10 *(Rotating Sphere)* Write a program that draws the mesh `sphere.mesh` in the center of the screen. When the user presses one of the arrow keys, the mesh should move ten units in that direction.

27.11 *(Rotating Sphere Modification)* Modify the program from Exercise 27.10 so that if the user holds down an arrow key, the sphere will move only once every second.

27.12 (*The Game of Snake*) The object of the game of snake is to maneuver the snake throughout the game area trying to eat bits of food. The snake is represented with a string of contiguous spheres in the game area, which is a two-dimensional grid. The snake can move up, down, left or right. If the snake eats a piece of food (shown by the "F"), it grows by adding another sphere to the end (Fig. 27.21). If the snake hits a wall of the game area (i.e., would be out of the array), the player loses (Fig. 27.22). If the snake runs into itself, the player loses (Fig. 27.23).

27.13 *(Snake with Obstacles)* Modify the program from Exercise 27.12 to add obstacles to the game area (Fig. 27.24). If the snake runs into an obstacle, the player loses.

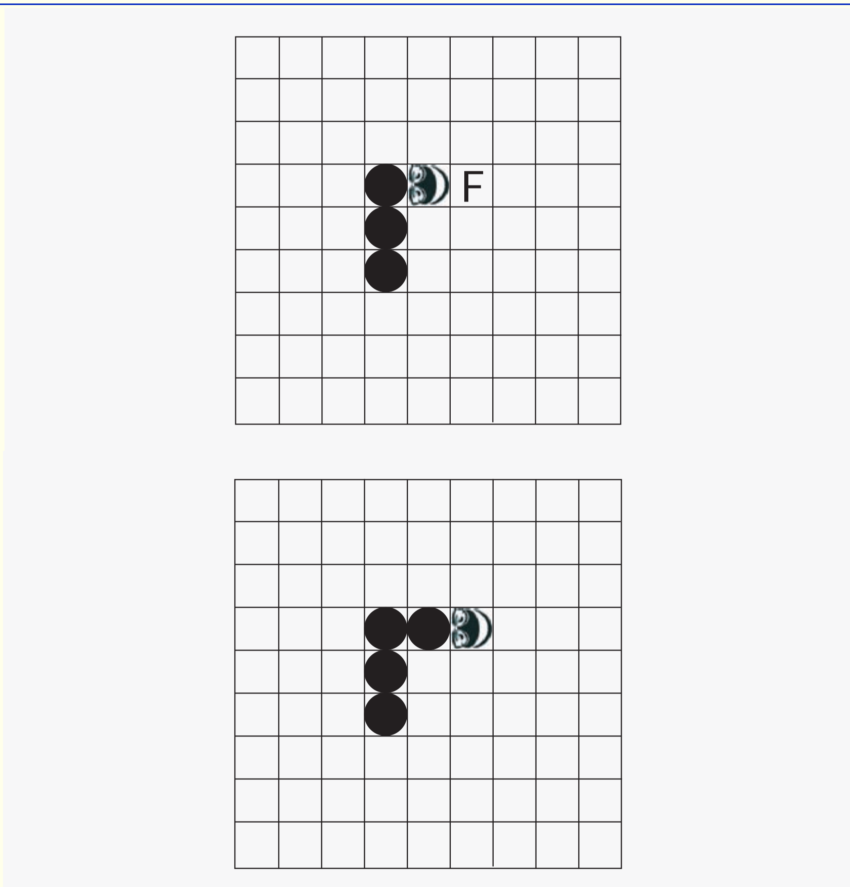


Fig. 27.21 | The snake grows when it eats.

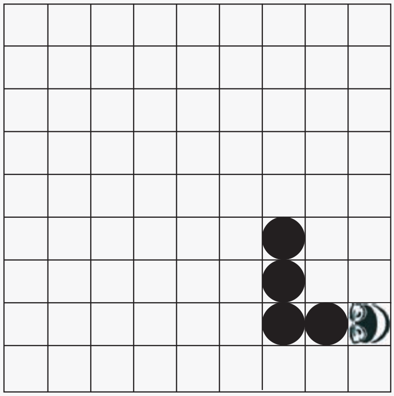


Fig. 27.22 | The snake dies if it hits a wall.

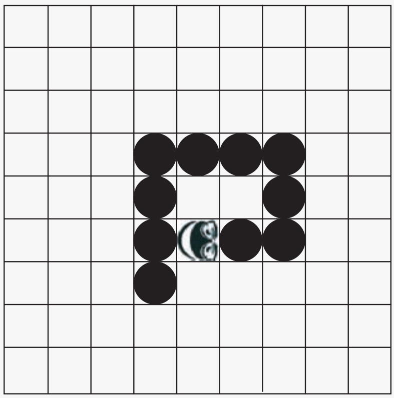


Fig. 27.23 | The snake dies if it hits itself.

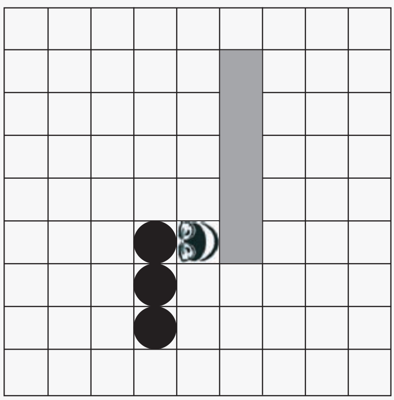


Fig. 27.24 | The snake dies if it hits an obstacle.

Continued from Back Cover

"*I really like the Making a Difference programming exercises. The game programming [in the Functions chapter] gets students excited.*"**—Virginia Bailey, Jackson State University**

"*It's great that the text introduces object-oriented programming early. The car analogy was well-thought out. An extremely meticulous treatment of control structures. The virtual function figure and corresponding explanation in the Polymorphism chapter is thorough and truly commendable .*"**—Gregory Dai, eBay, Inc.**

"*An excellent 'objects first' coverage of C++ that remains accessible to beginners. The example-driven presentation is enriched by the optional OO design case study that contextualizes the material in a software engineering project.*"
—Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

"*It is excellent that [the authors] use the STL and standard libraries early.*"**—John Dibling, SpryWare**

"*Provides a complete basis of fundamental instruction in all core aspects of C++. Examples provide a solid grounding in the construction of C++ programs. A solid overview of C++ Stream I/O.*"
—Peter DePasquale, The College of New Jersey

"*Great discussion about the mistakes resulted from using = for == and vice versa!*"
—Wing-Ning Li, University of Arkansas

"*Thorough and detailed coverage of exceptions from an object-oriented point of view.*"
—Dean Mathias, Utah State University

"*Chapter 20 (Data Structures) is very good. The examples are accessible to CS, IT, software engineering and business students.*"**—Thomas J. Borrelli, Rochester Institute of Technology**

"*The Simpletron exercises are brilliant. The Polymorphism chapter explains one of the hardest topics to understand in OOP in a clear manner. Great job! The writing is excellent, the examples are well developed and the exercises are interesting.*"**— José Antonio González Seco, Parliament of Andalusia, Spain**

"*Introducing the UML early is a great idea.*"**—Raymond Stephenson, Microsoft**

"*Good use of diagrams, especially of the activation call stack and recursive functions.*"
—Amar Raheja, California State Polytechnic University, Pomona

"*Terrific discussion of pointers—the best I have seen.*"**—Anne B. Horton, Lockheed Martin**

"*Great coverage of polymorphism and how the compiler implements polymorphism 'under the hood.' I wish I had such a clear presentation of data structures when I was a student.*"**—Ed James-Beckham, Borland**

"*A nice introduction to searching and sorting, and Big-O.*"**—Robert Myers, Florida State University**

"*Will get you up and running quickly with the memory management and regular expression libraries.*"
—Ed Brey, Kohler Co.

"*Excellent introduction to the Standard Template Library (STL). The best book on C++ programming for the serious student!*"**—Richard Albright, Goldey-Beacom College**

"*Each code example is completely reviewed. This is a critical step for students to learn good programming practices.*"
—Jack R. Hagemeister, Washington State University

"*The most thorough C++ treatment I've seen. Replete with real-world case studies covering the full software development lifecycle. Code examples are extraordinary!*"**—Terrell Hull, Logicalis Integration Solutions**