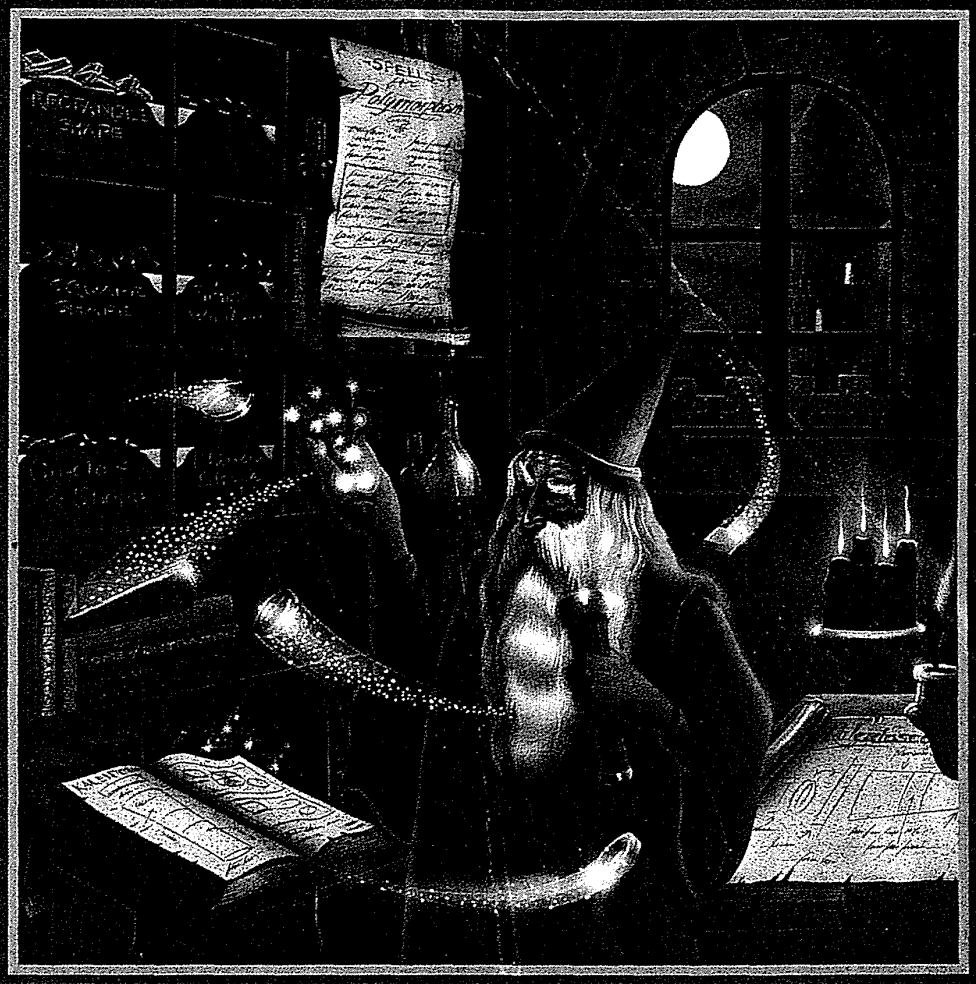
C++ PROGRAM DESIGN

An Introduction to Programming and Object-Oriented Design



COHOON & DAVIDSON

3rd Edition

C++ PROGRAM DESIGN

An Introduction to Programming and Object-Oriented Design

OHOON

AVIDSON

rd Edition

McGraw Hill

LG 1020

C++ PROGRAM DESIGN

An Introduction to Programming and Object-Oriented Design

OHOON

AVIDSON

rd Edition

Mc Graw Hill

C++ PROGRAM DESIGN

An Introduction to Programming and Object-Oriented Design

COHOON & DAVIDSON

3rd Edition

C++ Program Design

An Introduction to Programming and Object-Oriented Design

Third Edition

James P. Cohoon
University of Virginia

Jack W. Davidson
University of Virginia

McGraw Hill

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education
A Division of The McGraw-Hill Companies

C++ PROGRAM DESIGN: AN INTRODUCTION TO PROGRAMMING
AND OBJECT-ORIENTED DESIGN, THIRD EDTION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2002, 1999, 1997 by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9 8 7 6 5 4 3 2 1

ISBN 0–07–241163–5
ISBN 0–07–112182–X (ISE)

General manager: *Thomas E. Casson*
Publisher: *Elizabeth A. Jones*
Senior developmental editor: *Kelley Butcher*
Executive marketing manager: *John Wannemacher*
Senior project manager: *Kay J. Brimeyer*
Production supervisor: *Enboge Chong*
Coordinator of freelance design: *David W. Hash*
Cover illustration: *Paul D. Turnbaugh*
Cover design: *Z Graphics*
Senior photo research coordinator: *Lori Hancock*
Supplement producer: *Sandra M. Schnee*
Media technology senior producer: *Phillip Meek*
Compositor: *Interactive Composition Corporation*
Typeface: *10/12 Times Roman*
Printer: *R. R. Donnelley & Sons Company/Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Cohoon, James P.
C++ program design : an introduction to programming and object-oriented design / James P. Cohoon, Jack W. Davidson.—3rd ed.
p. cm.
Includes index.
ISBN 0–07–241163–5
1. C++ (Computer program language) 2. Object-oriented programming (Computer science) I. Davidson, Jack W. (Jack Winfred) II. Title.

QA76.73.C153 C653 2002
005.13′3—dc21 2001030617
CIP

INTERNATIONAL EDITION ISBN 0–07–112182–X
Copyright © 2002. Exclusive rights by The McGraw-Hill Companies, Inc., for manufacture and export. This book cannot be re-exported from the country to which it is sold by McGraw-Hill.
The International Edition is not available in North America.

www.mhhe.com

Dedicated to:

To Audrey and Joanne, our parents, families, and teachers from whom we have learned so much and owe our gratitude.

Preface

INTRODUCTION

Computers are an inescapable fixture in our lives. They control complex systems such as financial networks, mass transit, telephone systems, and power plants. Tens of millions of people use the Internet to access information, shop, recreate, communicate, and conduct business. Because the computer has become such an intrinsic component of modern life, we believe that everyone should have a basic working knowledge of how computers are programmed.

This textbook is about the fundamentals of programming and software development using C++, a popular high-level programming language developed by Bjarne Stroustrup of AT&T Bell Laboratories. We chose C++ for teaching programming because it supports the development of software using the object-oriented approach. An advantage of object-oriented development is that it lets us build complex software systems employing many of the techniques that have been used for constructing complex physical systems, such as cars, airplanes, or buildings. This book is targeted for a first programming course, and it has been designed to be appropriate for people from all disciplines. We assume no prior programming skills and use mathematics and science at a level appropriate to first-year college students.

Some of this book's important features are

- *The* C++ *standard is given broad coverage.* Our original naive intent was to offer complete coverage of C++. However, such a presentation would be overwhelming for the beginning student. For example, the language definition describes more than 150 standard classes and libraries. Rather than being encyclopedic, we provide in-depth coverage of all materials that an introductory course would need, introduce much of the remaining material, and give pointers to the rest. We also provide integrated coverage of the important additions and modifications to the C++ language, such as type **bool**, Standard Template Library, namespaces, and exceptions. The breadth of our coverage provides flexibility for the instructor. For example, an instructor may choose not to cover inheritance, but instead to cover

templates. For the students, the coverage allows advanced learners to go further in the language, and it makes the book valuable as a reference source.

- *Classes are introduced early.* Chapter 1 includes a gentle introduction to the object-oriented paradigm. Material is presented there to whet students' appetites. We believe that students must first be client users of objects before they can appreciate the difficulties of designing flexible, usable objects. All proficient designers started as users. The next several chapters introduce and use some standard stream class objects, such as `cout` and `cin`, `string` objects using the Standard Template Library, and a limited number of objects derived from a graphical library developed for the textbook. This experience helps reinforce the concepts of encapsulation, software reuse, and the object-oriented programming paradigm. After this solid introduction to the use of objects, we present approximately 50 classes and ADTs over the final eight chapters.
- *We present the use of a graphical Application Programmer Interface (API) designed specifically for beginning programmers to develop interesting programs.* We provide a portable, object-oriented graphical library, named EzWindows, for the easy display of simple geometric, bitmap, and text objects. We supply implementations of the API for popular Windows and UNIX compilers. Using the API provides several important experiences for the student. First, students are client users of a software library. As mentioned earlier, using well-designed objects helps novice programmers begin to appreciate good object-oriented design. Their experience as users forms the basis for becoming designers. Second, using the API introduces students to the real-world practice of developing programs using an application-specific library. Third, using EzWindows to perform graphical input and output exposes the student to event-based programming and the dominant mode of input and output used in real applications, and it permits development of exciting and visually interesting programs. This experience motivates the students, and it provides a visually concrete set of objects that help students understand the object-oriented paradigm. EzWindows is simple enough to allow even the first programming assignments to be graphical. Examples using EzWindows are sprinkled throughout the text. However, the presentation is done in a fashion that accommodates instructors who prefer to cover only ANSI materials. For students and instructors who want more advanced graphic capabilities, see Appendix E, which is a complete EzWindows reference. For sample assignments and supplementary materials, visit our Web sites `http://www.mhhe.com/c++programdesign` and `http://www.cplusplus-programdesign.com`.
- *Software-engineering design concepts are introduced via problem studies and software projects.* Besides containing numerous small examples for introducing C++ and object-oriented design concepts, each chapter considers one or more problems in detail. As appropriate, there are object-oriented analysis and design, algorithm development, and code to realize

the design. In addition, two chapters are devoted to the principles of software project development using our EzWindows API (Chapters 10 and 15). These chapters are springboards for software reuse and for projects suitable for individual and group work.

- *Programming and style tips are presented in boxes that clearly separate this material from the main text.* In addition to explaining C++ and object-oriented programming, we also give advice on how to be a better and more knowledgeable programmer and designer. For example, there are important tips on avoiding common programming errors, writing readable code, and understanding the new directions the standard took, as well as tips on performance and software engineering. Boxes also present one or two pages per chapter of historical information on computing.
- *Integrated use of the Standard Template Library.* An important component of the C++ language is its Standard Template Library, or as it is more commonly known, the STL. This library provides a rich collection of container classes for representing lists and strings and a set of generic algorithms for important programming tasks such as sorting, searching, and list traversal. Readers first encounter the STL in Chapter 3 when we introduce the string class. In subsequent chapters, additional features are introduced. In particular, Chapter 9 thoroughly explores list representation using the STL's vector container class, and Chapters 11 and 14 give insight on how the STL container classes can be implemented and use their generic algorithms to solve several common programming tasks.

THIRD EDITION HIGHLIGHTS

This third edition incorporates many of the suggestions we have received from both instructors and students. Some of the notable improvements and additions in this edition are:

- *Self-check questions.* At appropriate points in each chapter there are self-check questions that students can use to check their mastery of the material. The self-check questions are in addition to the exercises at the end of the chapter. The solutions to the self-check questions are available at our Web sites `www.mhhe.com/c++programdesign` and `www.cplusplus-programdesign.com`. The self-check questions include both short answer questions as well as programming exercises.
- *Earlier coverage of classes.* The material on classes is now covered a chapter earlier (Chapter 7). This change reflects the growing consensus that early coverage of classes is possible and the right approach for teaching object-oriented programming using C++.
- *Coverage of testing and debugging.* An important skill for programmers is how to test and debug the programs they write. Previous editions of the text did not cover this important topic. Chapter 12 provides an introduction to testing and debugging. The chapter discusses various testing techniques such as unit testing, integration testing, and code inspections. The sections

on debugging focus on teaching students how to use the scientific method to find bugs. The chapter also discusses common bugs encountered by beginning programmers and how to recognize them.

- *Improved explanations.* The entire text has been reexamined and, based on our own analysis and user feedback, additional examples have been inserted, clarifying figures have been created, and, as appropriate, explanations have been revised and expanded.

CONTEXT

In the early 1990s, with the support of the National Science Foundation, the Department of Computer Science at the University of Virginia began developing a new computer science curriculum. We carefully examined our existing curriculum and those of several other peer schools.

What we found were curricula that emphasized the following:

- Use of a programming language that is rarely used outside of undergraduate courses.
- Construction of small programs, consisting of at most a few hundred lines.
- Development in isolation of text-based programs "from scratch" for each assignment.
- Development in an environment lacking modern tools.
- An informal development with the belief that if a program "works," it is acceptable.

Comparing this situation with the real world, we saw considerable differences. Practicing computer professionals:

- Use programming languages designed for developing large applications that are often thousands or even millions of source lines long.
- Are involved most often in modifying and maintaining such systems rather than in developing them.
- Work in teams, not as single programmers.
- Do system development according to mandated specifications.
- Build systems that use graphical user interfaces to do input and output.
- Use existing libraries and tools to build systems.

To better prepare our students for real-world programming, we developed the first edition of this book. This third edition reflects feedback we have received from both instructors and students as well as our own experience using the book in large introductory programming courses and in a second course on programming and software engineering.

Programming

Most of the important concepts and problems in computer science cannot be appreciated unless one has a good understanding of what a program is and how to write one. Unfortunately, learning to program is difficult. Programming

well, like writing well, takes years of practice. In fact, teaching programming and teaching writing are, in some respects, very similar.

Students are taught writing by reading examples of good prose and by writing, writing, writing. In the process, they learn the important skill of how to organize ideas so they can be presented effectively. As students develop their skills, they move from writing and editing a paragraph or several paragraphs to creating larger pieces of prose, such as essays, short stories, and reports.

Our approach to teaching programming is similar to teaching writing but with one very important addition. Throughout the text, we present and discuss many examples of both good and bad programming. Programming exercises give the student the opportunity to practice organizing and writing code. In addition, we offer examples that facilitate learning the practical skill of modifying existing code. This is done through the use of code that is specifically designed to be modified by the student. We have found this mechanism to be effective because it forces the student to read and understand the provided code. In the text, a CD-ROM/World Wide Web icon signals that this code is available on the CD-ROM included with the book and at our Web site.

Why C++?

As we began our new curriculum development, one of the first issues was choosing which programming language to use. Like many departments, we had been using Pascal. Although it was unanimously decided that Pascal should be replaced, the choice of a replacement was the subject of much heated debate. Some of the languages we considered were C, C++, Modula-3, Scheme, and Smalltalk. A deciding factor was that we wanted to use a language that we ourselves use professionally. This decision narrowed the choices to C or C++. Although the decision was not unanimous, we chose C++ based on the belief that the object-oriented paradigm would be the dominant programming paradigm of the future.

In hindsight, we made the correct choice. C++ has continued to grow in popularity, and many companies use it as their development language. Indeed, many of our graduates report that when they interview for a job, a question they are often asked is whether they know C++. We believe that we will see a continuing shift to C++ as the introductory programming language of choice. We have also been pleasantly surprised by the effect on our students. The students in our upper-level courses who have completed our software development sequence can tackle much larger and harder problems than the students who had completed the comparable sequence in our old curriculum. In addition, we have seen substantial migration of other disciplines to C++. For example, all students in the commerce school at our university now take C++, and the engineering disciplines that had previously required Fortran now require C++.

Some of you may be wondering about Java. Java is definitely an interesting language. However, serious software development tools are not yet mature, and the language and its libraries are still undergoing serious revisions. The conventional wisdom of the professionals in our research areas is that Java

might be the language for developing graphical user interfaces (GUIs) but that C++ remains the language for application development. If we are to meet our goal of most ably preparing students for computing careers, then C++ is the right educational vehicle.

Introduce objects early

Our experience of teaching C++ over the past 8 years shows that the object-oriented paradigm can be successfully introduced to beginning programmers. In our initial course offerings, we introduced objects near the end of the course and did superficial coverage of objects, classes, overloading, and inheritance. Essentially, we taught C using C++ syntax and input and output mechanisms. This approach failed. It introduced a new concept too late in the course—students were not able to integrate the material. We revised our course to introduce objects earlier and found that this approach worked much better. Students now have time to absorb this material because it is used and reinforced throughout the course rather than just at the end. The objects-early approach is reflected in this text. Students begin using standard objects in Chapter 2. Chapters 3 through 7 introduce the students to the use of graphical objects from the EzWindows API. After this solid introduction to using objects, Chapter 7 introduces classes and the design of objects, and it logically follows the chapters that introduce control structures, functions, and libraries (Chapters 4 through 6). We strongly believe that this is the proper sequencing of the material in an introductory textbook. The students certainly like it! By their second and third assignment they are producing useful software with graphic capabilities.

Software projects

As noted, what we had been teaching in the past was not at all close to what was happening in the real world. To educate future computer scientists in the skills that support the engineering and comprehension of large software systems, reengineering of existing systems, and application of innovative techniques (such as software reuse), our department deemed it necessary to begin introducing this material in the first course. Our software project chapters (Chapters 10 and 15) are vehicles for this introduction. These chapters provide several important experiences for the student. First, both projects use our EzWindows API. Using the API to do event-based programming and graphical input and output exposes students to the programming model typically used in real-world applications, and it permits students to develop more exciting and interesting programs. If desired, the software projects facilitate students' working together in groups of up to four. Again, this practice mirrors the real world, where it is rare for a lone programmer to develop an application. The software projects also illustrate software maintenance. Many of the exercises at the end of the software project chapters call for the student to make major modifications or nontrivial extensions to the project program.

CHAPTER SUMMARY

- *Chapter 1: Computing and the object-oriented design methodology*—basic computing terminology, machine organization, software, software development, software engineering, object-oriented design and programming.
- *Chapter 2: C++: the fundamentals*—program organization, function main(), include statement, comments, definitions, writing readable code, interactive input and output, fundamental types, literals, constants, declarations, expressions, conversions, precedence.
- *Chapter 3: Modifying objects*—assignment statement and conversions, extractions, const objects, increment and decrement, insertion and extractions, string class, Standard Template Library, graphical objects and the EzWindows API.
- *Chapter 4: Control constructs*—logical values and operators, truth tables, bool, relational operators, general precedence, short-circuit evaluation, if statement, if-else statement, sorting, switch statement, enum, while statement, for statement, invariants, do statement, text processing, scientific visualization.
- *Chapter 5: Function usage basics and libraries*—functions, value parameters, formal parameters, actual parameters, invocation, flow of control, activation records, pseudorandom numbers, prototyping, preprocessor, inclusion directives, header files, conditional compilation, software reuse, using libraries, standard streams, manipulators, file streams, file processing, iostream, iomanip, fstream, ctype, string, stdlib, and assert libraries.
- *Chapter 6: Programmer-defined functions*—function definitions, parameters, invocation, flow of control, return statement, scope, local objects, global objects, reference parameters, constant parameters, default parameters, parameter casting, function overloading, initialization, name reuse, top-down design, recursion, in-memory streams, utility functions, Standard Template Library, integrating a quadratic polynomial, financial visualization.
- *Chapter 7: The class construct and object-oriented design*—programmer-defined data types, class construct, information hiding, encapsulation, object-oriented analysis and design, access specification, data members, member functions, constructors, kaleidoscope program, object-oriented factory automation simulator/trainer.
- *Chapter 8: Implementing abstract data types*—data abstraction, object-oriented design, default and copy constructors, inspectors, mutators, facilitators, auxiliary functions, memberwise assignment, const member functions, arithmetic operator overloading, reference return, insertion and extraction overloading, pseudorandom-number generation, ADTs for rational and pseudorandom numbers, and the red-yellow-green game.

- *Chapter 9: Lists*—one-dimensional arrays, subscripting, parameter passing, initialization, character strings, multidimensional lists, tables, matrices, Standard Template Library, container classes, adapter classes, vector class, vector member functions, sorting, InsertionSort, QuickSort, binary search, two-dimensional search, list representation, initialization lists, iterators, ADTs for maze-traversing robot problem.

- *Chapter 10: The EzWindows API: a detailed examination*—Application Programmer Interfaces, graphical user interface, event-based programming, window coordinate system, callbacks, mouse and timer events, EzWindows API mechanics, ADTs for simple windows, bitmaps, text labels, and a Simon Says game.

- *Chapter 11: Pointers and dynamic memory*—lvalues, rvalues, pointer types, addressing, indirection, pointers as parameters, pointers to pointers, constant pointers, equivalence of array and pointer notation, character string processing, command-line parameters, pointers to functions, dynamic objects, free store, new and delete operators, dangling pointers, memory leak, destructors, member assignment, this, ADT for a list of integers.

- *Chapter 12: Testing and debugging*—black-box testing, white-box testing, inspections, unit testing, integration testing, system testing, statement coverage, equivalence partitioning, regression testing, boundary conditions, code reviews, test harness, path coverage.

- *Chapter 13: Inheritance*—object-oriented design, reuse, base class, derived class, single inheritance, is-a relationship, has-a relationship, uses-a relationship, shape hierarchy, controlling inheritance, protected members, multiple inheritance, ADTs for rectangles, circles, ellipses, and triangles, an object-oriented kaleidoscope program.

- *Chapter 14: Templates and polymorphism*—generic actions and types, function template, class template, container class, sequential lists, linked list, iterator class, friends, polymorphism, virtual function, pure virtual function, abstract base class, virtually derived class, virtual multiple inheritance, list ADTs, random-access list, sequential lists, list iterators, singly linked lists, doubly linked lists.

- *Chapter 15: Software project—bug hunt!*—encapsulation, inheritance, virtual functions, object-oriented design, Bug Hunt game, ADTs for various kinds of bugs and a game controller.

- *Appendixes* — ASCII character set, general precedence table, iostream, stdlib, time, string and algorithm libraries, vector and other container classes, string class, namespaces, using statements, exceptions, friends, EzWindows API, project and make files.

USING THIS BOOK

This text has more material than can be covered in a single course. The extra coverage was deliberate—it allows instructors to select their choice of topics on programming and software development. The book was also designed for flexibility in teaching. For example, if an instructor desires to move the introduction of classes earlier in the course, he or she can cover iteration after classes and our development of the rational number ADT. If an instructor desires to introduce classes after arrays, then Sections 9.1 to 9.5 and Section 9.12 of Chapter 9 can precede Chapters 7 and 8. Also, the discussion of inheritance in Chapter 13 can precede the coverage of pointers and dynamic objects in Chapter 11. Instructors who do a breadth-first coverage of computer science may choose to omit the software project chapters and substitute material from sources that cover topics such as the social and ethical aspects of computing or elementary formal logic. The testing material of Chapter 12 (Section 12.1) can be covered anytime after the material on classes (Chapter 7) has been introduced. The section on debugging (Section 12.2) relies on array examples, and therefore should be covered after Chapters 9 and 11.

We use the following layout for our course.

Week	Topic	Readings
1	Computing and object-oriented design	Chapter 1
2	Programming fundamentals	Chapter 2
3	Object manipulation	Chapter 3
4	Conditional statements	Chapter 4 (Sections 4.1–4.6)
5	Iteration statements	Chapter 4 (Sections 4.7–4.12)
6	Functions and reuse	Chapter 5, Chapter 6
7	Parameter passing	Chapter 6
8	OO analysis and design	Chapter 7
9–10	ADTs	Chapter 8 (Sections 8.1–8.6)
11	Arrays	Chapter 9 (Sections 9.1–9.4, Section 9.12)
12	Vectors	Chapter 9 (Sections 9.5–9.10)
13	Project—Simon Says, OOA/OOD	Chapter 10
14	Inheritance	Chapter 13

Depending on faculty interests, the material covered in week 13 can vary. In the introductory course at our university, we spend one week every semester on a problem in detail. Generally, this examination contributes to the final project.

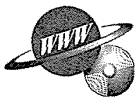
SUPPLEMENTARY MATERIALS

In addition to the included CD-ROM, which contains source code and supplementary files for many of our programs and listings, we have developed other

materials. For example, there is a set of slide transparencies (approximately 300 slides). The course we teach also has a closed-laboratory component that meets once a week for reinforcing current course topics. For these laboratories, we have developed a student laboratory manual. These materials are available from the publisher. For more detailed information, visit our Web site at `http://www.mhhe.com/c++programdesign`. In particular, we maintain a frequently asked question list (FAQ) and links to helpful C++ and educational sites. Other educational supplements are also available at our class Web site `http://www.cs.virginia.edu/cs101`.

SYMBOLS

The following icons are used in the margins throughout the text.

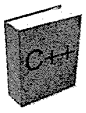


The World Wide Web (WWW) and CD-ROM icon is associated with some code listings and programs. This icon indicates that the code is available both on the CD-ROM supplied with the book and at our Web site `http://www.mhhe.com/c++programdesign`. When the icon is associated with the label Program, the program consists of a single file. If the icon is associated with the label Listing, a library file or one file in a multifile program is being made available.

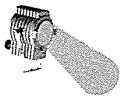
The exclamation icon indicates a warning about programming. Often these are tips on how to avoid common programming errors.

The detour icon indicates a set of self-check exercises. The answers to the exercises can be found at our Web site `www.mhhe.com/c++programdesign` or `www.cplusplus-programdesign.com`. The self-check exercises include both short answer questions as well as programming exercises.

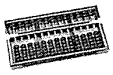
The sunglass icon indicates that the associated material is related to programming style. At the current time, a number of conventions are being used. The manner that code is presented in this text generally reflects the dominant convention. (Of course, our variation is the best!)



The book icon indicates that the associated material is concerned with the C++ programming language itself. The two typical uses of this icon are for advanced C++ topics or for describing a recent language extension that can have an impact on software development.



The spotlight icon indicates programming tips or highlights material that presents a more detailed discussion or a sidebar to the current topic.



The abacus icon indicates a discussion on the history of computing. Many people often mistakenly think that computing is simply writing programs. While designing and writing programs is certainly an important part of computing, it is by no means the only thing encompassed by computing. Each chapter contains at least one anecdote regarding triumphs and failures of the pioneers in computing.

THE AUTHORS

Jim Cohoon is a professor in the computer science department at the University of Virginia and is a former member of the technical staff at AT&T Bell Laboratories. He joined the faculty after receiving his Ph.D. from the University of Minnesota. He has been nominated twice by the department for the university's best-teaching award. In 1994, Professor Cohoon was awarded a Fulbright Fellowship to Germany, where he lectured on C++ and software engineering. Professor Cohoon's research interests include algorithms, computer-aided design of electronic systems, optimization strategies, and computer science education. He is the author of more than 60 papers in these fields. He is a member of the Association of Computing Machinery (ACM), the ACM Special Interest Group on Design Automation (SIGDA), the ACM Special Interest Group on Computer Science Education (SIGCSE), the Institute of Electrical and Electronics Engineers (IEEE), and the IEEE Circuits and Systems Society. He is a member of the ACM Publications and SIG Boards and is past chair of SIGDA. He can be reached at `cohoon@virginia.edu`. His Web homepage is `http://www.cs.virginia.edu/~cohoon`.

Jack Davidson is also a professor in the computer science department at the University of Virginia. He joined the faculty after receiving his Ph.D. from the University of Arizona. Professor Davidson has received NCR's Faculty Innovation Award for innovation in teaching. Professor Davidson's research interests include compilers, computer architecture, systems software, and computer science education. He is the author of more than 80 papers in these fields. He is a member of the ACM, the ACM Special Interest Group on Programming Languages (SIGPLAN), the ACM Special Interest Group on Computer Architecture (SIGARCH), SIGCSE, the IEEE, and the IEEE Computer Society. He served as an associate editor of *Transactions on Programming Languages and Systems*, ACM's flagship journal on programming languages and systems, from 1994 to 2000. He was chair of the 1998 Programming Language Design and Implementation Conference (PLDI '98) and program co-chair of the 2000 SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems

(LCTES 2000). He can be reached at `jwd@virginia.edu`. His Web homepage is `http://www.cs.virginia.edu/~jwd`.

DELVING FURTHER

The following are primary references on the C++ language.

- International Standard for Information Systems—Programming Language C++, ISO/IEC FDIS 14882, Washington, DC: American National Standards Institute, 1998.
- B. Stroustrup, *The C++ Programming Language,* 3rd ed., Reading, MA: Addison-Wesley, 1998.

The following are good sources on libraries and more-advanced object-oriented design, program development, and the Standard Template Library.

- J. Bergin, *Data Abstraction: The Object-Oriented Approach Using C++*, New York: McGraw-Hill, 1994.
- M. D. Carroll and M. A. Ellis, *Designing and Coding Reusable C++*, Reading, MA: Addison-Wesley, 1995.
- M. P. Cline and G. A. Lomow, *C++ FAQs*, Reading, MA: Addison-Wesley, 1995.
- A. Koenig and B. Moo, *Ruminations on C++*, Reading, MA: Addison-Wesley, 1997.
- S. B. Lippman and J. Lajoie, *C++ Primer,* 3rd ed., Reading, MA: Addison-Wesley, 1998.
- S. Maguire, *Writing Solid Code*, Redmond, WA: Microsoft Press, 1993.
- S. Meyers, *Effective C++*, Reading, MA: Addison-Wesley, 1998.
- S. Meyers, *More Effective C++*, Reading, MA: Addison-Wesley, 1996.
- D. R. Musser and A. Saini, *STL Tutorial and Reference Guide*, Reading, MA: Addison-Wesley, 1995.
- P. J. Plauger, A. Stepanov, M. Lee, and D. R. Musser, *The Standard Template Library*, Englewood Cliffs, NJ: Prentice-Hall, 1998.
- B. Stroustrup, *The Design and Evolution of C++*, Reading, MA: Addison-Wesley, 1994.

The following are good sources for learning more about the history and future of computing.

- S. Augarten, *Bit by Bit: An Illustrated History of Computers*, New York: Ticknor & Fields, 1984.
- P. J. Denning and B. Metcalfe (eds.), *Beyond Calculation: The Next Fifty Years of Computing*, New York: Copernicus Press, Springer-Verlag, 1997.
- J. A. N. Lee, *Computer Pioneers*, Piscattaway, NJ: IEEE Press, 1995.
- J. Palfreman and D. Swade, *The Dream Machine: Exploring the Computer Age*, London: BBC Books, 1991.

- H. G. Stine, *The Untold Story of the Computer Revolution*, New York: Arbor House, 1985.
- M. R. Williams, *A History of Computing Technology*, Englewood Cliffs, NJ: Prentice-Hall, 1985.

ACKNOWLEDGMENTS

We thank the University of Virginia for providing an environment that made this book possible. In particular, we thank Jack Stankovic for his tireless efforts in leading the computer science department to national prominence. We also thank Mark Bailey, Alan Batson, Joanne Cohoon, Clark Coleman, John Karro, Sean McCulloch, James Ortega, Jane Prey, Paul Reynolds, and Alfred Weaver for their comments. Thanks also goes to Bruce Childers who helped design and implement the original EzWindows API and Peter Valle who helped revise the EzWindows API for this edition.

We are grateful to Rich Rashid and Amitabh Srivastava of Microsoft Research for providing an environment that allowed the third edition to be completed. Microsoft Research is a great place to think, write, and do research. A very special thanks go to David Hanson, Todd Proebsting, and Chris Fraser of the Programming Language Systems group at Microsoft Research for making us feel welcome and at home. We will miss the interesting and stimulating lunch-time conversations—especially the lunch trips to Hole-in-the-Wall Barbeque for Meatloaf Monday and to Acapalco Fresh for Burrito Thursday.

We thank all of the people at McGraw-Hill for their efforts in making this edition a reality. In particular, we thank Tom Casson, for his support and encouragement; Kay Brimeyer, for her behind-the-scenes product-management skills; John Wannemacher, for his creative marketing ideas; David Hash, for leading the cover-design team; June Waldman, for copyediting the second edition; Jill Barrie for copyediting the third edition; and Janelle Pregler for her careful proofreading of the third edition. Special thanks go again to Elizabeth (Betsy) Jones, our executive editor, for support, direction, and focus throughout this project, and Kelley Butcher, our senior developmental editor, for managing and synthesizing the reviewing process.

We thank the following class testers, readers, and reviewers for their valuable comments and suggestions on the second edition of this text:

Kenneth Bayse, Clark University
Leslie Blackford, Wheaton College
Jacobo Carrasquel, Carnegie Mellon University
John Dailey Jr., University of Illinois
Suzanne Miller Dorney, Grand Valley State University
Gerald Dueck, Brandon University
H. E. Dunsmore, Purdue University
Elizabeth Lee Falta, Louisiana Tech University
William Filter, Sandia National Laboratories
Ann Ford, University of Michigan

Kyle Gillette, George Washington High School of Denver
Robert Holloway, University of Wisconsin
Van Howbert, Colorado State University
Liewen Huang, University of Virginia
Leon Jololian, New Jersey Institute of Technology
Michael Jones, Encyclopedia Brittanica, Inc.
Edward Keefe, Des Moines Area Community College
Andrea Kerslake, Frederick High School
Nancy Kinnersley, University of Kansas
Chih-Fen Koh, State University of New York, Stony Brook
Valeri Kolesnikov, Slippery Rock University
Rhoda Baggs Koss, Florida Institute of Technology
Joel Kraft, Case Western Reserve University
R. Raymond Lang, Xavier University
Jens Lienig, Tanner Research
Rowan Lindley, Westchester Community College
John Lowther, Michigan Technological University
Lewis Lum, University of Portland
Bruce Maxim, University of Michigan at Dearborn
Jin Mazumdar, State University of New York at Fredonia
Michael McCarthy, University of Pittsburgh
Douglas Morris, University of Pittsburgh
Robert O'Neil, Utah State University
Glenn Pavlicek, Bridgewater State College
Howard Pyron, University of Missouri at Rolla
Donna Reese, Mississippi State University
Charles Riedesel, University of Nebraska
Carol Roberts, University of Maine
Paul Schatz, University of Virginia
W. Brent Seales, University of Kentucky
Shashi Shekhar, University of Minnesota
Neelam Soundarajan, Ohio State University
M. A. Sridhar, University of South Carolina
Kirk Stephens, Southwestern Community College
Phil Sweany, Michigan Technological University
Peter Theron, Herzing Community College
Ralph Tomlinson, Iowa State University
Kevin Treu, Furman University
Jane Turk, LaSalle University
Frank Vedro, University of Pittsburgh
David Whalley, Florida State University
Brian Wharry, Elmhurst College

We are also grateful to the following reviewers who provided comments and feedback on the third edition:

Richard Albright, University of Delaware
Ramzi Bualuan, University of Notre Dame

Drue Coles, Boston University
Joseph DeLibero, Arizona State University
Ann Ford, University of Michigan
Juan E. Gilbert, Auburn University
Larry Gordon, University of Alaska, Anchorage
Sherri Harms, University of Nebraska at Lincoln
Robert A. Hovis, Ohio Northern University
Thomas Ibry, University of North Texas
Debbie Keen, University of Kentucky
Andrew Kinley, Rose-Hulman Institute of Technology
Stephen P. Leach, Florida State University
Kelsey Lick, University of California at Riverside
Rose M. Lowe, Clemson University
Robert J. McGlinn, Southern Illinois University
Wayne Miller, University of Kansas
Jay Morris, Old Dominion University
Lawrence J. Osborne, Lamar University
Young Park, Bradley University
Wolfgang Pelz, The University of Akron
Norman Pestaina, Florida International University
Clayton Price, University of Missouri—Rolla
Thomas Scanla, University of Wisconsin
Sung Shin, South Dakota State University
Brenda Sonderegger, Montana State University
Michael Stinson, Central Michigan University
Robert Underwood, Colorado School of Mines
Charles Welty, University of Southern Maine
Rick Zaccone, Bucknell University

We thank our spouses, Audrey and Joanne, and our children for their efforts, cooperation, and sacrifices in making this book happen.

Finally, we thank the users of this book. Many of the changes in this edition were prompted by suggestions we received via electronic mail from users of the text. We welcome your comments, suggestions, and ideas for improving this material. Please write in care of the publisher, McGraw-Hill, or send electronic mail to `cohoon@virginia.edu` or `jwd@virginia.edu`.

J. P. C
J. W. D

Contents

Preface, v

Chapter 1

Computing and the object-oriented design methodology 1

1.1	Basic computing terminology	2
	Self-check questions	13
1.2	Software	18
1.3	Engineering software	24
1.4	Object-oriented design	32
	Self-check questions	40
1.5	Points to remember	41
1.6	To delve further	43
1.7	Exercises	44

Chapter 2

C++: The fundamentals 49

2.1	Program organization	50
2.2	A first program	50
2.3	A second program	52
2.4	Comments	53
	Self-check questions	56
2.5	Assigning a value	56
2.6	Fundamental C++ objects	57
2.7	Constants	61
	Self-check questions	69
2.8	Names	69
2.9	Definitions	73
	Self-check questions	76

2.10 Expressions 77
Self-check questions 87
2.11 Output statements 88
2.12 Computing average velocity 91
Self-check questions 93
2.13 Points to remember 95
2.14 Exercises 98

Chapter 3

Modifying objects 103

3.1 Assignment 104
3.2 Const definitions 108
3.3 Input statements 109
Self-check questions 111
3.4 Computing the number of molecules in a hydrocarbon 112
3.5 Compound assignment 115
3.6 Increment and decrement 117
Self-check questions 119
3.7 Estimating yearly savings of change 120
3.8 The string class 122
Self-check questions 128
3.9 EzWindows 130
3.10 Mowing lawns 134
Self-check questions 141
3.11 Points to remember 141
3.12 Exercises 146

Chapter 4

Control constructs 153

4.1 Boolean algebra 154
4.2 A Boolean type 156
Self-check questions 161
4.3 Conditional execution using the if statement 164
Self-check questions 173
4.4 Conditional execution using the switch statement 176
4.5 Computing a requested expression 179
4.6 Validating a date 181
Self-check questions 188
4.7 Iteration using the while statement 189
4.8 Simple string and character processing 195
Self-check questions 203
4.9 Iteration using the for construct 204

4.10 Simple data visualization 212
4.11 Solving the lazy hobo riddle 214
4.12 Iteration using the do construct 215
Self-check questions 217
4.13 Points to remember 218
4.14 Exercises 222

Chapter 5

Function basics 233

5.1 Function basics 234
5.2 The preprocessor 243
Self-check questions 246
5.3 Using software libraries 246
5.4 The iostream library 248
5.5 The iomanip library 251
5.6 The fstream library 258
5.7 Random numbers 263
5.8 The assert library 268
Self-check questions 269
5.9 Points to remember 272
5.10 To delve further 275
5.11 Exercises 275

Chapter 6

Programmer-defined functions 281

6.1 Basics 282
6.2 A tasty problem 285
6.3 Some useful functions 290
6.4 Integrating a quadratic polynomial 293
6.5 The local scope 296
6.6 The global scope 299
Self-check questions 301
6.7 Reference parameters 304
6.8 Passing objects by reference 312
6.9 Validating telephone access codes 314
Self-check questions 317
6.10 Constant parameters 319
6.11 Default parameters 321
6.12 Casting of function parameters 323
6.13 Function overloading 324
Self-check questions 326
6.14 Recursive functions 328
Self-check questions 332

6.15 Displaying a price-interval stock chart 333
6.16 Points to Remember 342
6.17 To delve further 345
6.18 Exercises 345

Chapter 7

The class construct and object-oriented design 363

7.1 Introducing a programmer-defined data type 364
7.2 The RectangleShape class 366
Self-check questions 374
7.3 Using the RectangleShape class 375
7.4 Constructors 379
Self-check questions 380
7.5 Building a kaleidoscope 382
7.6 Object-oriented analysis and design 387
7.7 Points to remember 397
7.8 To delve further 399
7.9 Exercises 400

Chapter 8

Implementing abstract data types 405

8.1 Introducing abstract data types 406
8.2 Rational ADT basics 406
8.3 Rational interface description 413
Self-check questions 421
8.4 Implementing the rational class 423
8.5 Copy construction, member assignment, and destruction 430
Self-check questions 435
8.6 ADT for pseudorandom integers 437
8.7 Red-yellow-green game 444
8.8 Points to remember 461
8.9 Exercises 465

Chapter 9

Lists 471

9.1 Named collections 472
9.2 One-dimensional arrays 472
Self-check questions 483
9.3 Arrays as parameters 485

9.4 Sorting 488
Self-check questions 493
9.5 Container classes 494
9.6 Class vector 497
9.7 QuickSort 511
9.8 Binary searching 516
9.9 String class revisited 518
Self-check questions 520
9.10 Find that word—exploring a two-dimensional list 521
9.11 Maze runner 525
9.12 Multidimensional arrays 549
Self-check questions 552
9.13 Points to remember 553
9.14 Exercises 558

Chapter 10

The EzWindows API: a detailed examination 565

10.1 Application programmer interfaces 566
10.2 A simple window class 567
10.3 The bitmap class 577
10.4 Mouse events 579
10.5 Bitmaps and mouse events 583
10.6 Timer events 586
10.7 Alert messages 589
Self-check questions 590
10.8 Simon says 591
10.9 Points to remember 607
10.10 Exercises 608

Chapter 11

Pointers and dynamic memory 615

11.1 Lvalues and rvalues 616
11.2 Pointer basics 616
Self-check questions 626
11.3 Constant pointers and pointers to constants 629
11.4 Arrays and pointers 630
11.5 Character string processing 633
11.6 Program command-line parameters 637
Self-check questions 639
11.7 Pointers to functions 641
Self-check questions 644
11.8 Dynamic objects 645

11.9 A simple ADT for representing lists of integer values 651
Self-check questions 660
11.10 Points to remember 661
11.11 Exercises 665

Chapter 12

Testing and debugging 673
12.1 Testing 674
Self-check questions 682
Self-check questions 693
12.2 Debugging 693
Self-check questions 704
12.3 Points to remember 705
12.4 To Delve Further 705
12.5 Exercises 706

Chapter 13

Inheritance 707
13.1 Object-oriented design using inheritance 708
13.2 Reuse via inheritance 709
13.3 A hierarchy of shapes 711
Self-check questions 725
13.4 Protected members and inheritance 727
13.5 Controlling inheritance 730
Self-check questions 732
13.6 Multiple inheritance 735
Self-check questions 743
13.7 A prettier kaleidoscope 743
13.8 Points to remember 756
13.9 Exercises 759

Chapter 14

Templates and polymorphism 765
14.1 Generic actions and types 766
14.2 Function templates 766
14.3 Class templates 769
14.4 A simple list class using a class template 770
Self-check questions 778
14.5 Sequential lists 779
Self-check questions 801

14.6 Polymorphism 801
14.7 Virtual function nuances 804
14.8 Abstract base classes 807
14.9 Virtual multiple inheritance 810
Self-check questions 812
14.10 Points to remember 815
14.11 Exercises 817

Chapter 15

Software project—Bug Hunt! 823
15.1 Bug hunt 824
15.2 Base class Bug 824
15.3 Class GameController 839
15.4 Bug hunt 842
Self-check questions 843
15.5 Points to remember 844
15.6 Exercises 847

Appendix A

Tables 849
A.1 ASCII character set 850
A.2 Operator precedence 851

Appendix B

Standard libraries 853
B.1 Library naming and access 854
B.2 Iostream library 854
B.3 Stdlib library 856
B.4 Math library 856
B.5 Time library 857
B.6 Cstring library 858
B.7 Algorithm library 860

Appendix C

Standard classes 865
C.1 Container Classes 866
C.2 Class string 871

Appendix D

Advanced topics 875
D.1 Namespaces 876
D.2 Exception handling 881
D.3 Friends 888

Appendix E

EzWindows API reference manual 891
E.1 Enumerated types 892
E.2 Coordinate system 892
E.3 Class Position 894
E.4 Class SimpleWindow 894
E.5 Class WindowObject 898
E.6 Class RaySegment 898
E.7 Class Shape 900
E.8 Class EllipseShape 901
E.9 Class CircleShape 902
E.10 Class RectangleShape 902
E.11 Class TriangleShape 904
E.12 Class SquareShape 904
E.13 Class Label 905
E.14 Class BitMap 907
E.15 Class RandomInt 908
E.16 Miscellaneous functions 909

Appendix F

Projects and makefiles 911
F.1 Project and makefile fundamentals 912
F.2 Borland C++ IDE 912
F.3 Microsoft Visual C++ IDE 920
F.4 UNIX Makefiles 926

Index, 933

CHAPTER 11

Pointers and dynamic memory

Introduction

Many problem solutions require that a significant amount of information be represented. These tasks can be accomplished with C++ mechanisms that provide for the dynamic creation and management of objects during run time. Such mechanisms allow programs to be flexible and to represent arbitrarily sized data instances. Dynamic objects are created using the **new** operator and are returned to the system using the **delete** operator. Access to a dynamic object is through a pointer, where a pointer is an object whose value is the location of another object. Pointers have similarities to iterators. In fact, an iterator can be viewed as a pointer abstraction. To support pointers, C++ provides two complementary operators—the address operator & and the dereferencing operator *. The address operator allows the location of an object to be computed, and the dereferencing operator allows the value stored at a location to be computed. Our discussion begins with pointers.

Key Concepts

- lvalues
- rvalues
- pointer types
- null address
- dereferencing operator *
- indirect member selector operator ->
- address operator &
- pointer assignment
- indirect assignment
- pointers as parameters
- pointers to pointers
- constant pointers
- pointers to constants
- arrays and pointers
- command-line parameters
- pointers to function
- dynamic objects
- free store
- operators **new** and **delete**
- exception handling
- `set_new_handler()`
- dangling pointers
- memory leak
- destructors
- copy constructor
- member assignment
- **this** pointer
- qualifier **const**
- pointers to function
- exception handling

11.1 LVALUES AND RVALUES

In C++ there are two kinds of expressions: those that represent objects that can be both evaluated and modified and those that represent objects can only be evaluated. Suppose the following definitions are in effect.

```
int a = 1;
int b;
int c[3];
```

Then the following code segment contains both kinds of expressions.

```
b = 5;
cout << b << endl;
c[0] = 2*a;
```

The expression b represents an object whose value can be evaluated or modified depending on the circumstances. In the first assignment statement, the object represented by expression b is being modified. In the insertion statement, the object represented by expression b is being evaluated. In the second assignment, the expression c[0] represents an object whose value can also be evaluated or modified.

Expressions representing objects that can be both evaluated and modified are *lvalues* (pronounced "el-values"). Thus object names—such as a and b in the preceding code segment—are lvalues, but not all lvalues are object names, as discussed below.

Although the typical assignment statement uses the name of an object as its left operand, the syntax of an assignment statement requires only that the left operand be an lvalue. This flexibility is important because not all objects have names. For example, although an array or a vector has a name, the individual elements do not—individual elements are referenced using lvalue subscript expressions. Thus the assignment to c[0] in the preceding code segment is an example of assignment to an lvalue that corresponds to an object without a name.

The two right operands of our assignment statements, 5 and 2*a, are not lvalues—it would not make sense to allow either 5 or 2*a to be the target of an assignment. These kinds of expressions are *rvalues* (pronounced "are-values"); they cannot be used as the left operand of an assignment. Lvalues do not have such restrictions. Lvalue expressions can be used as either left or right operands of an assignment.

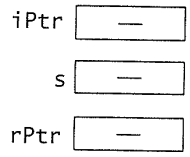
11.2 POINTER BASICS

A pointer is an object whose value represents the location of another object. Pointer objects are defined in conjunction with the unary *indirection* operator *

or, as it is more commonly known, the *dereferencing* operator. The following code segment defines three pointer objects: `iPtr`, `s`, and `rPtr`.

```
int *iPtr;
char *s;
Rational *rPtr;
```

Object `iPtr` is of type pointer to **`int`**, object `s` is of type pointer to **`char`**, and object `rPtr` is of type pointer to `Rational`. These pointer types are all different just as **`int`**, **`char`**, and `Rational` are all different types. Because these definitions did not include initialization, the three objects are uninitialized. The following figure depicts the result of these definitions (a dash indicates an uninitialized value).

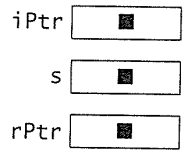


For explanatory purposes in referring to a pointer type, we use the associated type concatenated with the dereferencing operator; that is, **`int`*** should be read as "pointer to **`int`**," and **`char`*** should be read as "pointer to **`char`**."

There is one literal value that can be assigned to any pointer object. This literal value is 0, which in this context is known as the *null address*. For example, the following statements all assign the null address.

```
int *iPtr = 0;
char s = 0;
Rational rPtr = 0;
```

A pointer object whose value is the null address is not pointing to an object that can be accessed. In depicting pointer objects whose values are null, we use a filled-in square.



In the next section we discuss operators that allow a pointer to be associated with specific objects. In doing so, we will be able to achieve representations like that shown in Figure 11.1.

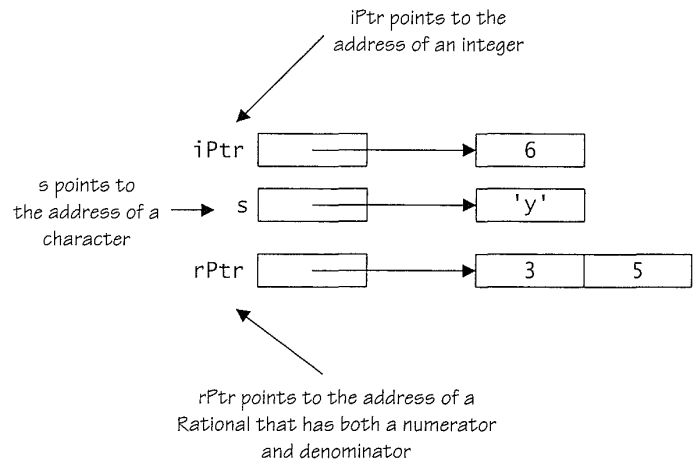
The assignment operator is defined for pointer objects. If `Ptr1` and `Ptr2` have the same pointer type, then the following assignment is legal.

```
Ptr1 = Ptr2;
```

The meaning of the preceding assignment is the same as it is for other objects—the value represented by `Ptr1` is modified to hold the value represented by `Ptr2`. As a result of this assignment, objects `Ptr1` and `Ptr2` point to the same object. The preceding example uses the name of a pointer object as

Figure 11.1

Three pointer objects and the values to which they point



the right operand, but the right operand can, in fact, be any expression that evaluates to the same type as the left operand.

Suppose the following definitions are in effect.

```
int i = 1;
int *iPtr;
char *cPtr;
```

Then none of the following three assignment statements is legal—their right operands do not produce values of the proper type.

```
iPtr = i;     // illegal: i is not an int*
i = iPtr;     // illegal: iPtr is not an int
iPtr = cPtr;  // illegal: cPtr is not an int*
```

In the first of these illegal assignment statements, `iPtr` is the target of the assignment. Therefore, the right operand must evaluate to an **`int*`** value, but `i` is an **`int`**, which makes the assignment illegal. In the second assignment, the right operand must evaluate to an **`int`**. This evaluation is not possible with `iPtr`, which is an **`int*`**. In the third assignment, the right operand needs to evaluate to an **`int*`**. However, `cPtr` evaluates to **`char*`**. Thus none of the assignments met the C++ requirement that the right and left operands of an assignment evaluate to the same type.

11.2.1 Addressing and indirection

C++ provides the unary *address* operator & so that the location of an object can be computed and used by a program. When the address operator is applied to

Programmer Alert

Always define one object per statement

When defining a pointer object, some programmers juxtapose the dereferencing operator with the type name rather than the object name as in the following definition:

```
char* Ptr;
```

This form is appealing because it clearly identifies that the type being used is a pointer type. However, it can lead to a misunderstanding. For example, consider the following statement:

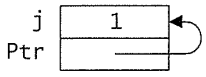
```
char* s, t; // really char *s; char t
```

The statement first creates an object `s` that is a pointer to a **char**. It then creates an object `t` that is a **char**. The different object types are the result of the right associativity of the dereferencing operator—even though the dereferencing operator is juxtaposed with the type **char**, it is still associated with object `s`. You should recall our earlier warning that objects should always be defined one per statement. Avoiding this type of error is a principal reason for that warning.

an object in an expression, the value produced by the addressing operation is a pointer to that object. Suppose the following code segment has been executed.

```
int j = 1;
int *Ptr;
Ptr = &j;
```

The first definition initializes `j` to the value `1`; the assignment statement sets `Ptr` to the address of the memory location for `j`. The following figure depicts the result of these actions.



The value of `j` can now be accessed directly through `j` *or* indirectly through pointer object `Ptr`. The indirect method requires the use of the dereferencing operator *.

When the dereferencing operator is used on a pointer object, the dereferencing operation computes the lvalue of the object to which that pointer object points. For example, in the following insertion statement, the value `1` is displayed to the standard output stream.

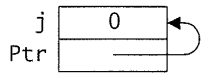
```
cout << *Ptr << endl; // displaying object j
```

The value `1` is displayed because `*Ptr` is the object to which `Ptr` points. Because `Ptr` points to the memory location where object `j` is stored, `*Ptr` evaluates to value `1`. Thus when the dereferencing operator acts on a pointer object, the operator's behavior is analogous to when the operator acts on an iterator. In fact, iterators can be viewed as pointer abstractions.

As noted in our initial discussion of lvalues, the left operand of the assignment operator can be any lvalue expression. This flexibility is demonstrated in the following assignment:

```
*Ptr = 0;                // modifying object j
```

The right operand is the value 0, and the left operand is the expression `*Ptr`. The expression `*Ptr` is an lvalue that refers to `j`. Therefore, this assignment operation indirectly modifies `j` so that it now has the value 0. A picture of memory for objects `j` and `Ptr` is the following:



If the following two insertion statements were executed, two zeros would be inserted to the standard output stream.

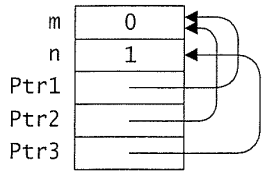
```
cout << j<< endl;
cout << *Ptr << endl;
```

Like other objects, the definitions of pointer objects can include various initialization expressions, as illustrated in the following code segment:

```
int m = 0;
int n = 1;
int *Ptr1 = &m;
int *Ptr2 = Ptr1;
int *Ptr3 = &n;
```

The first two definitions create and initialize **`int`** objects `m` and `n`. The next three definitions create and initialize **`int`**`*` objects `Ptr1`, `Ptr2`, and `Ptr3`.

Pointer `Ptr1` is initialized so that it points to `m`. Pointer `Ptr2` is initialized so that it is a copy of `Ptr1`'s current value. Therefore, `Ptr2` also points to `m`. Finally, `Ptr3` is initialized so that it points to `n`. A picture of the memory for the five objects is the following:

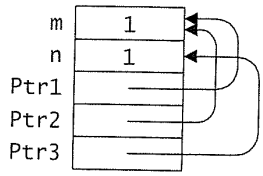


Suppose the following code segment is executed. What would be the changes in the various objects?

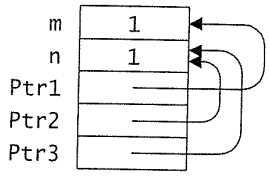
```
*Ptr1 = *Ptr3;
Ptr2 = Ptr3;
```

The right operand of the first assignment is the expression `*Ptr3`. Because `Ptr3` is a pointer to the memory location of `n`, expression `*Ptr3` evaluates to 1. The left operand of the first assignment is the expression `*Ptr1`. Because `Ptr1` is a pointer to the memory location of `m`, the expression `*Ptr1` is a legal lvalue expression referring to that value of `m`. Therefore, the first assignment

indirectly modifies m so that it is a copy of n. A picture of the memory at this point is the following:



The second assignment in our code segment replaces the value of Ptr2 with the value of Ptr3. Because Ptr3 points to n, Ptr2 now also points to n. The picture of the memory would now be the following:



If a pointer object points to class-type objects then, by using the dereferencing and selection operators, individual members can be accessed. Because the selection operator has higher precedence, the dereferencing operator must be nested within parentheses.

```
Rational a(4,3);
Rational *aPtr = &a;
(*aPtr).Insert(cout); // invokes member Insert() of
                      // object *aPtr. Parentheses
                      // are necessary.
```

Because this syntax is clumsy, C++ includes the indirect member selector operator -> that combines dereferencing with selection. Thus the following statement also displays the Rational object to which aPtr points.

```
aPtr->Insert(cout);
```

The dereferencing and indirect member selector operators are not tools for overcoming class access permissions. A nonpublic data member cannot be accessed by a client using either of these operators. For example, the following statements are illegal.

```
(*aPtr).NumeratorValue = 1; // illegal: private member
aPtr->DenominatorValue = 2; // illegal: private member
```

A pointer object whose value is the null address cannot be dereferenced. For example, the following code segment is illegal.

```
int *NullPtr = 0;
*NullPtr = 1; // illegal: NullPtr is not pointing to
              // a location of an int object
```

11.2.2 Pointers to pointers

C++ allows pointer types whose objects are pointers to pointers. C++ also allows pointer types whose objects are pointers to pointers to pointers, and so on. Although the concept might seem strange, it does have its applications (e.g., dynamic multidimensional lists). The following definition specifies a pointer object `PtrPtr` that points to an `int*` object.

```
int **PtrPtr;
```

Each asterisk in a definition indicates another level of dereferencing. For example, the two asterisks in front of `PtrPtr` indicate a pointer to a pointer, and three asterisks would have indicated a pointer to a pointer to a pointer.

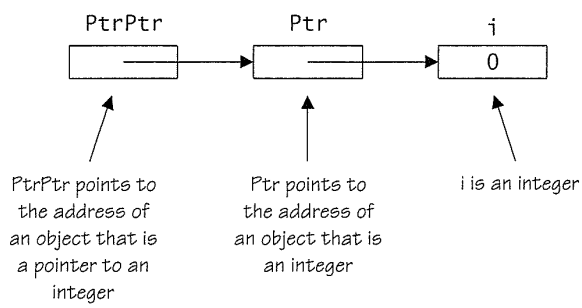
Suppose the following definitions are also in effect.

```
int i = 0;
int *Ptr= &i;
```

Then the following assignment is correct because `Ptr` is an `int*`, which makes expression `&Ptr` evaluate to an `int**`.

```
PtrPtr = &Ptr;
```

As the result of this assignment, the objects `i`, `Ptr`, and `PtrPtr` have the following relationship:



As noted previously, C++ considers two pointer objects to be of different types if the types of values to which they point are different. Therefore, the following assignment is illegal.

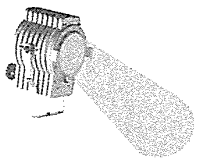
```
PtrPtr = Ptr;  // illegal: PtrPtr needs an int**
```

The assignment is illegal because the right operand is of type `int*`, but `PtrPtr` expects a value of type `int**`.

11.2.3 Pointers as parameters

Up to this point, our examples have been isolated code segments. We now consider a small program, Program 11.1, which invokes a `void` function `IndirectSwap()`. Function `IndirectSwap()` has two formal parameters `Ptr1` and `Ptr2`. Both parameters are value parameters of type `char*`.

Programming Tip



Pointer power

The ability to access and change the value of an object whose name is not part of the assignment statement is the characteristic that makes pointer objects so powerful but also so confusing. Program errors involving pointers are often subtle and difficult to track down, and so it pays to think carefully about how you are using pointers. A simple technique that helps understand pointers is to draw a picture showing the objects and what points to what. We will use this technique throughout this chapter to help explain how pointers work. The technique of drawing pictures for data structures is so helpful even experienced programmers use this technique when trying to understand or debug code involving pointers.

Program 11.1

Demonstrates that pointers can be used to simulate reference parameters

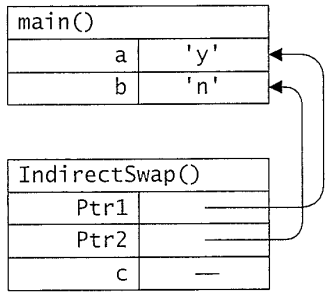
```
// Program 11.1: Swapping objects using indirection
#include <iostream>
#include <string>
using namespace std;
void IndirectSwap(char *Ptr1, char *Ptr2) {
   // swap the contents of the char objects to which
   // Ptr1 and Ptr2 point
   char c = *Ptr1;
   *Ptr1 = *Ptr2;
   *Ptr2 = c;
}
int main() {
   char a = 'y';
   char b = 'n';
   // pass the lvalues of a and b to IndirectSwap()
   IndirectSwap(&a, &b);
   // display the new values of a and b
   cout << a << b << endl;
   return 0;
}
```

When the program begins, objects a and b are initialized to `'y'` and `'n'`, respectively. The following figure depicts the activation record of `main()` after executing the definitions of a and b.

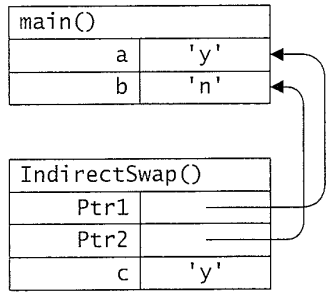
main()	
a	'y'
b	'n'

The invocation of function `IndirectSwap()` in Program 11.1 is `IndirectSwap(&a, &b)`. The actual parameters for this invocation are the values of the expressions &a and &b. Because expression &a is a pointer to a, value parameter `Ptr1` points to a; and because expression &b is a pointer to b, value

parameter `Ptr2` points to `b`. An initial picture of the memory associated with this invocation of `IndirectSwap()` is the following:



In function `IndirectSwap()`, local object `c` is initialized with the value of the expression `*Ptr1`. In particular, in this invocation of `IndirectSwap()`, object `c` is initialized to `'y'` because `Ptr1` points to object `a` of `main()` that has the value `'y'`.

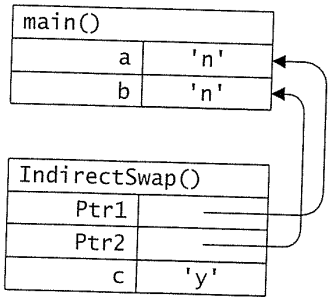


The two assignment statements in `IndirectSwap()` then modify the objects to which `Ptr1` and `Ptr2` point.

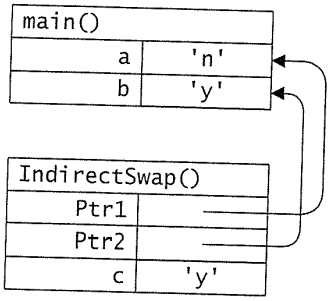
```
*Ptr1 = *Ptr2;
*Ptr2 = c;
```

Because `Ptr1` points to object `a` of `main()`, the first assignment in modifying the lvalue `*Ptr1` in fact modifies `a`. Thus the assignment does not alter any of the values in the activation record of `IndirectSwap()`. The new value of `a` is

the lvalue of `*Ptr2`. Because `Ptr2` points to object `b` from `main()`, this assignment in effect gives `a` the value `'n'`.

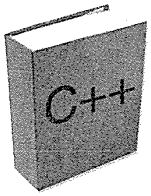


For the second assignment in `IndirectSwap()`, the left operand is `*Ptr2`. Because `Ptr2` points to object `b` of `main()`, this assignment in effect modifies `b`. Thus this assignment, like the previous assignment, does not alter any of the values in the activation record of `IndirectSwap()`, but again updates the contents of the activation record for `main()`. The new value of `b` is a copy of `c`, which is a `'y'`.



The execution of `IndirectSwap()` is completed with the second assignment, and the flow of control then returns to `main()`. Thus the result of the invocation of `IndirectSwap()` is a swapping of the values of the objects to which the actual parameters point. Function `IndirectSwap()` is equivalent to function `Swap()` of Chapter 6.

C++ Language



Importance of reference parameters

Program 11.1 demonstrates that through the use of the dereferencing and address operators and value parameters, we can achieve a side effect that allows us to mimic reference parameters! Since you are now accustomed to C++ programming, this feature may not seem important. However, it is crucial to a C programmer because C does not provide reference parameters. Imagine being a C programmer and every time you need a function with reference parameters, you must instead write a function that performs pointer manipulation while also making sure that the actual parameters to the function are pointers to the objects to be modified. These requirements are both burdensome and error prone. The inclusion of reference parameters in C++ is a critical software-engineering aspect of the language.

Self-check Questions

1. Give a C++ statement that makes the pointer `IntPtr` point to the integer object `Counter`.

2. Give a C++ statement that makes the pointer `IntPtr1` point to the same location as the pointer `IntPtr2`.

3. Give the C++ statement that copies the value pointed to by `Ptr1` to the memory location pointed to by `Ptr2`.

4. Give a C++ statement that sets the floating-point object pointed to by `FloatPtr1` to 34.5.

5. Assume that `Ptr1` and `Ptr2` are pointers to two **`double`** objects. Write the code fragment that swaps `Ptr1` and `Ptr2`. That is, `Ptr2` should point to what `Ptr1` previously pointed to, and `Ptr1` should point to what `Ptr2` previously pointed to.

6. Assume that `Ptr1` and `Ptr2` are pointers to two **`double`** objects. Write the code fragment that swaps the values in the memory locations pointed to by `Ptr1` and `Ptr2`.

7. Give the output of the following program.

```
#include <iostream>
int main() {
    int Value = 10;
    int *ptr = &Value;
    cout << *ptr << endl;
    return 0;
}
```

8. Consider the following declarations and code.

```
int s[10];
int *iptr1;
int *iptr2;
int **iptr3;
for (int i = 0; i < 10; i++)
    s[i] = 9 - i;
iptr1 = s;
iptr2 = &s[3];
iptr3 = &iptr1;
*iptr1 = 5;
*iptr2 = 11;
**iptr3 = 14;
```

In the following picture, write the values (rvalues and lvalues) that are stored in memory after the code is executed. Assume integers and pointers take 4 bytes of memory.

s[0]		1000
s[1]		1004
s[2]		1008
s[3]		1012
s[4]		1016
s[5]		1020
s[6]		1024
s[7]		1028
s[8]		1032
s[9]		1036
iptr1		1040
iptr2		1044
iptr3		1048

9. Give the output of the following program.

```
#include <iostream>
int main() {
    int Value = 10;
    int *ptr = &Value;
    *ptr = 0;
    cout << Value << endl;
    return 0;
}
```

10. Give the output of the following program.

```
#include <iostream>
int main() {
    int Value1 = 10;
    int Value2 = 40;
    int *ptr1 = &Value1;
    int *ptr2 = &Value2;
    ptr1 = ptr2;
    cout << *ptr1 << endl;
    cout << *ptr2 << endl;
    return 0;
}
```

11. Give the output of the following program.

```
#include <iostream>
void Test(int *p, int v) {
    *p = 5;
    return;
}
int main() {
    int Count = 5;
    Test(Count, Count);
    cout << Count << endl;
    return 0;
}
```

12. Give the output of the following program.

```
#include <iostream>
void Swap(int *p1, int *p2) {
    int *temp = p1;
    p1 = p2;
    p2 = temp;
    return;
}
int main() {
    int Value1 = 10;
    int Value2 = 30;
    int *Ptr1 = &Value1;
```

```
    int *Ptr2 = &Value2;
    Swap(Ptr1, Ptr2);
    cout << Value1 << Value2 << endl;
    return 0;
}
```

11.3 CONSTANT POINTERS AND POINTERS TO CONSTANTS

The modifier **const** can also be applied in pointer definitions. Depending on where the modifier **const** is placed, either the pointer is a constant or the contents of the location to which the pointer points are considered to be constant.

Suppose the following definitions are in effect.

```
char c1 = 'a';
char c2 = 'b';
```

Then the following code segment illustrates the three legal places in a pointer definition where the modifier **const** may be applied.

```
const char *Ptr1 = &c1; // *Ptr1 is considered to
                        // be a constant; Ptr1 is
                        // not a constant
char const *Ptr2 = &c1; // *Ptr2 is considered to
                        // be a constant; Ptr2 is
                        // not constant
char *const Ptr3 = &c1; // Ptr3 is a constant;
                        // *Ptr3 is not constant
```

The modifier **const** can appear immediately before or after the type; the **const** can also appear after the unary operator *. According to the language definition, a type modifier can appear either before or after the type without changing the meaning of the definition. Therefore, the first two definitions in the preceding code segment define pointer objects with similar capabilities.

In the preceding discussion, we used the phrase *considered to be a constant*. For example, the phrase was used with the definition of `Ptr1`. As a result of that particular pointer definition, the value of the object to which `Ptr1` points cannot be changed through an assignment to `*Ptr1`. However, the value of `c1`—the object to which `Ptr1` points—can still be changed through a direct assignment to `c1`.

```
*Ptr1 = 'A';  // illegal: *Ptr1 is a constant
c1 = 'A';     // legal: c1 is not a constant
```

A mnemonic that helps remind you what is being defined as constant is reading the definition backwards, substituting *is a pointer to a* for the * in the definition. Thus the definition of `Ptr1` reads "`Ptr1` is a pointer to a **char** constant," the definition of `Ptr2` reads "`Ptr2` is a pointer to a constant **char**," and the definition of `Ptr3` reads "`Ptr3` constant is a pointer to a **char**." Although the last phrase is grammatically poor, this method should help you determine

what is constant and what is not. For the preceding definitions, `*Ptr1`, `*Ptr2`, and `Ptr3` are constants. As a result, the following statements are illegal.

```
*Ptr1 = 'A';  // illegal: *Ptr1 cannot be changed
*Ptr2 = 'B';  // illegal: *Ptr2 cannot be changed
Ptr3 = Ptr2;  // illegal:  Ptr3 cannot be changed
```

Because `Ptr1`, `Ptr2`, and `*Ptr3` are not constants, the following statements are legal.

```
Ptr1 = Ptr2;  // legal:  Ptr1 can be changed
Ptr2 = Ptr1;  // legal:  Ptr2 can be changed
*Ptr3 = 'C';  // legal: *Ptr3 can be changed
```

In the following definition, we define a constant pointer to a location whose contents are considered to be a constant.

```
const char *const Ptr4 = &c1;
```

As a result, the following assignments are illegal.

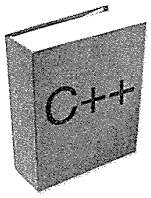
```
Ptr4 = &c2;   // illegal:  Ptr cannot be changed
*Ptr4 = 'D';  // illegal: *Ptr cannot be changed
```

The first assignment is illegal because it attempts to modify `Ptr4`. The second assignment is illegal because it attempts to change the contents to which `Ptr4` points.

It is also illegal for a pointer to a constant to be assigned to a pointer that is not pointing to a constant.

```
char *Ptr5 = Ptr4;  // illegal: potentially allows
                    // constant *Ptr4 to be modified
```

C++ Language



Understanding C++ declarations

The syntax of C++ can make understanding declarations difficult. One simple trick is to read the declarations backwards—that is, from right to left. Consider the following declaration

```
char *const Ptr3;
```

Starting from the right we can read as follows "`Ptr3` is a constant pointer to a **`char`**." When we see the keyword **`const`** we say "constant", and when we see a * we say pointer. Here's another one.

```
char const *Ptr1;
```

So `Ptr1` is a pointer to a constant `char`. This little trick really helps when you encounter a really complicated declaration.

11.4 ARRAYS AND POINTERS

In C++ the name of an array is considered to be a constant pointer; that is, the name of an array is associated with a particular memory location, and that association cannot be changed by statements in the program. In fact, the name

of an array is associated with the memory location of the first element in the array. For example, consider the following six definitions:

```
int A[5];
int B[10];
int *Ptr1 = A;      // Ptr1 points to A[0]
int *Ptr2 = B;      // Ptr2 points to B[0]
int *Ptr3 = &B[0]; // Ptr3 points to B[0]
int *Ptr4 = &A[4]; // Ptr4 points to A[4]
```

The first two definitions create 5-element and 10-element **int** arrays A and B, respectively. The last four definitions create pointer objects of type **int***. Based on the preceding discussion, the initialization expression A for Ptr1 is equivalent to a pointer to the first element in array A. Similarly, the definition of Ptr2 initializes it to point to the first element of B. The definition of Ptr3 explicitly initializes it to also point to the first element of B. The definition of Ptr4 initializes it to point to the last element in array A.

The equality and relational operators are defined for pointers of the same type; therefore, given the preceding definitions, the comparisons in the following code segment are legal.

```
if (A == B)
   cout << "A and B: same values" << endl;
else
   cout << "A and B: different values" << endl;
if (Ptr1 == A)
   cout << "A and Ptr1: same values" << endl;
else
   cout << "A and Ptr1: different values" << endl;
if (Ptr2 != Ptr3)
   cout << "Ptr2 and Ptr3: different values"
    << endl;
else
   cout << "Ptr2 and Ptr3: same values" << endl;
if (Ptr1 < Ptr4)
   cout << "Ptr1 and Ptr4: Ptr1 is first" << endl;
else
   cout << "Ptr1 and Ptr4: Ptr1 is not first"
    << endl;
```

The output of the code segment is the following:

```
A and B: different values
A and Ptr1: same values
Ptr2 and Ptr3: same values
Ptr1 and Ptr4: Ptr1 is first
```

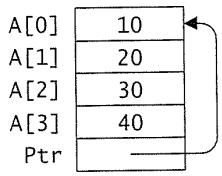
Note that the relational operators <, <=, >, and >= are defined only for pointers that point to the same array or to the object that occurs in memory immediately after the array.

The increment and decrement operators are defined for pointer objects. The effect of the increment operator ++ on a pointer object is to have the new address to which it points be the starting location of the object immediately after the object to which it previously pointed. Thus the increment operator works on a pointer object in a manner analogous to its operation on an iterator object.

Consider these definitions for A and Ptr.

```
int A[4] = {10, 20, 30, 40};
int *Ptr = A;
```

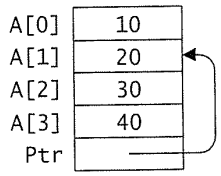
The following figure depicts the initial values in the memory associated with these definitions.



If we were to execute the following code segment, it would display the value 20 to output stream cout.

```
++Ptr;
cout << *Ptr << endl;
```

This output occurs because the pointer increment operator updates its operand to point to the object immediately after the object to which its operand previously pointed. As Ptr was pointing to A[0], the increment causes it to now point to the next element in the list, which is A[1]. The memory associated with A and Ptr would have the following depiction:

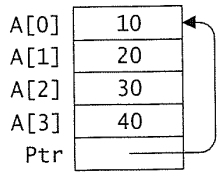


Because A[1] is 20, the insertion of *Ptr to cout in the preceding code segment displays the value 20.

The effect of the decrement operator -- on a pointer object is analogous to the behavior of the increment operator. If a pointer object is decremented, the new location to which it points is the object immediately before the object to which it previously pointed. For example, suppose Ptr is now decremented as in the following statement:

```
--Ptr;
```

The new value of Ptr would be the location of A[0].



Addition and subtraction operators are also defined for pointer objects. For example, the expression Ptr + i would be a pointer to the ith object beyond

Programmer Alert

Inadvertent representation errors

In both a decrement and an increment operation, the system does not check during execution whether the object stored at the resulting address is the same type as the base type of the pointer object. If a type mismatch occurs, then the effect on the program is undefined. For example, consider the following code segment:

```
int A[5];
float x;
int *Ptr = &A[4];  // Ptr points to last element of A
++Ptr;             // undefined: Ptr not pointing to
                    // an int location
```

The initialization of `Ptr` has it point to the last element of **`int`** array A. In most C++ implementations, **`float`** object `x` will occur immediately after A in memory. Therefore, the subsequent increment of `Ptr` has it pointing into the representation of `x`. As a result, the further use of `Ptr` produces undefined results.

the object to which `Ptr` points. In this expression, `i` is called an *offset*. The contents of that offset address are given by the lvalue expression `*(Ptr + i)`. Similarly, the expression `Ptr - i` is a pointer to the `i`th object prior to the object to which `Ptr` points. The contents of that offset address are given by the lvalue expression `*(Ptr - i)`.

The following example demonstrates the use of the dereferencing and pointer arithmetic operators to display the contents of an array A.

```
int A[4] = {10, 20, 30, 40};
int *Ptr = A;
for (int i = 0; i < 4; ++i) {
   cout << *(Ptr+i) << endl;
}
```

To complete the correspondence between pointers and arrays, we observe that in C++ the expression `*(Ptr+i)` is equivalent to the expression `Ptr[i]`. Therefore, the contents of the array A can also be displayed using `Ptr` in the following manner:

```
for (int i = 0; i < 4; ++i) {
   cout << Ptr[i] << endl;
}
```

11.5 CHARACTER STRING PROCESSING

The iostream library includes definitions for the insertion and extraction operators when the right operands are **`char*`** objects. The importance of this overloading is diminishing as programmers increasingly use the `string` class for character string representation. However, because many legacy libraries use the **`char`** array/pointer representation, it is still important to discuss the behavior of these insertion and extraction operators.

The result of inserting a **char** pointer to an output stream is equivalent to displaying a **char** array whose first element is located at the address to which the **char*** object points. An example of this use of the insertion operator is contained in the following code segment:

```
char Text[9] = "Sandrine";
for (char *Ptr = Text; *Ptr != '\0'; ++Ptr) {
   cout << Ptr << endl;
}
```

The output of the code segment is

```
Sandrine
andrine
ndrine
drine
rine
ine
ne
e
```

For each iteration of the **for** statement, the string is displayed that starts at the current location to which `Ptr` points. Once `Ptr` points to the address that contains the null character, the test expression evaluates to false and the **for** statement is done.

When the right operand of an extraction operator is a **char*** object, the behavior is the same as that of an extraction target to a **char** array—by default leading white space is skipped and the next nonwhitespace string of characters is extracted. The characters in the extracted string—including a terminating null character—are assigned to memory starting at the location to which the **char*** object points. For example, suppose the following definitions are in effect.

```
char Word[4];
char *WordPtr = Word;
```

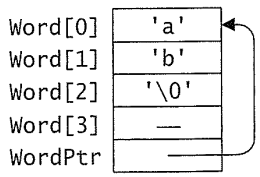
If standard input contains

```
ab xyz
```

then the extraction

```
cin >> WordPtr;
```

would have the following effect (the dash again indicates that the object is uninitialized).



Programmer Alert

Make sure the memory is there

When using a **char** pointer with an extraction the pointer must point to a **char** array of sufficient length to store the extracted characters. The pointer is automatically assumed to be pointing to valid and sufficient memory. If it is not, then the effect of the extraction on your program is undefined. What is likely to happen is that the data values stored immediately following the **char** array will be overwritten. If you are lucky, the program will fault near the extraction and the reason for the fault will be obvious. However, the fault can sometimes happen far from the point where the array was overwritten. In these cases, determining what went wrong can be challenging.

The interchangeability of pointer and array notation is most obvious in the passing of character strings to functions. The following is a possible implementation of the cstring library function `strlen()`.

```
// strlen() with s passed as a const array
int strlen(const char s[]) {
    int i;
    for (i = 0; s[i] != '\0'; ++i) {
        continue;
    }
    return i;
}
```

The function expects a character string terminated with a null character as its single parameter, and it returns the length of the string (excluding the terminating null character). The length of the string is determined by scanning the array for the terminating null character. The index of the null character is the length of the string (the characters in the string are array elements `s[0]` through `s[i-1]`).

The function `strlen()` can also be implemented in the following manner with the character string being passed via a pointer to its first character.

```
// strlen() with s passed as a pointer to constant
// characters
int strlen(const char *s) {
    int length;
    for (length = 0; *s != '\0'; ++s) {
        ++length;
    }
    return length;
}
```

The **const** declaration of the value parameter `s` indicates that the contents of the location to which `s` points are considered constant; that is, `*s` cannot be the target of an assignment. The length of the string is determined by incrementing a local **int** object `length` once for each nonnull character in the string. A separate increment operation updates `s` so that it points to the next character in the string.

We now consider another implementation of function `strlen()`. This implementation combines the character comparison with a side effect that increments the pointer to the next character.

```
// strlen() with s passed as a pointer to constant
// characters. The incrementing of the pointer is
// done as a side effect of the test expression.
int strlen(const char *s) {
    int length;
    for (length = 0; *s++ != '\0'; ++length) {
        continue;
    }
    return length;
}
```

The test expression has the following interpretation: determine whether the **char** object to which `s` points is the null character and, as a side effect of that test, increment pointer `s`. This combination of evaluation and pointer increment is quite common in string-processing code. However, whether the terseness contributes to or detracts from its understanding is debatable.

We next consider an **int** function `strcmp()`. This function expects two strings as parameters, and through its return value indicates whether the two strings are the same or whether the first occurs before or after the second lexicographically (dictionary order).

```
int strcmp(const char *s, const char *t) {
    // look for first difference or until *s is null
    while ((*s == *t) && (*s != '\0')) {
        ++s;
        ++t;
    }
    // return encoding difference
    return *s - *t;
}
```

The string parameters for `strcmp()` are `s` and `t`. Function `strcmp()` returns 0 if the strings are the same, returns a negative value if `s` occurs lexicographically before `t`, and returns a positive value if `s` occurs lexicographically after `t`.

The **while** loop in `strcmp()` iterates while two conditions are true: the current characters to which `s` and `t` point are the same, and the current character to which `s` points is not the null character. (We do not need to explicitly check whether `t` is null because the existing terms in the **while** expression catch this condition.) When the loop terminates, we examine the current characters to which `s` and `t` point to determine the return value of the function. If the characters are the same, the return value, which is the difference `*s - *t`, will be 0. If the strings are different, then `s` and `t` will point to the first difference in the two strings. In this case, the expression `*s - *t` will be negative if `s` occurs lexicographically before `t` and positive if `s` occurs lexicographically after `t`.