Big C++

Cay Horstmann

Late Objects

3/e

WILEY

Big C++

Late Objects

3/e

Cay Horstmann

San Jose State University

WILEY

PUBLISHER	Laurie Rosatone
EDITORIAL DIRECTOR	Don Fowley
DEVELOPMENTAL EDITOR	Cindy Johnson
ASSISTANT DEVELOPMENT EDITOR	Ryann Dannelly
EXECUTIVE MARKETING MANAGER	Dan Sayre
SENIOR PRODUCTION EDITOR	Laura Abrams
SENIOR CONTENT MANAGER	Valerie Zaborski
EDITORIAL ASSISTANT	Anna Pham
SENIOR DESIGNER	Tom Nery
SENIOR PHOTO EDITOR	Billy Ray
PRODUCTION MANAGEMENT	Cindy Johnson
COVER IMAGE	© 3alexd/Getty Images

This book was set in Stempel Garamond LT Std by Publishing Services, and printed and bound by Quad/Graphics, Versailles. The cover was printed by Quad/Graphics, Versailles.

This book is printed on acid-free paper. ∞

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: *www.wiley.com/go/citizenship*.

ISBN 13: 978-1-119-40297-8

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

PREFACE

This book is an introduction to C++ and computer programming that focuses on the essentials—and on effective learning. The book is designed to serve a wide range of student interests and abilities and is suitable for a first course in programming for computer scientists, engineers, and students in other disciplines. No prior programming experience is required, and only a modest amount of high school algebra is needed.

Here are the key features of this book:

Present fundamentals first.

This book uses the C++ programming language as a vehicle for introducing computer science concepts. A substantial subset of the C++ language is covered, focusing on the modern features of standard C++ that make students productive. The book takes a traditional route, first stressing control structures, procedural decomposition, and array algorithms. Objects are used when appropriate in the early chapters. Students start designing and implementing their own classes in Chapter 9.

Guidance and worked examples help students succeed.

Beginning programmers often ask "How do I start? Now what do I do?" Of course, an activity as complex as programming cannot be reduced to cookbook-style instructions. However, step-by-step guidance is immensely helpful for building confidence and providing an outline for the task at hand. "Problem Solving" sections stress the importance of design and planning. "How To" guides help students with common programming tasks. Additional Worked Examples are available in the E-Text or online.

Tip: Source files for all of the program examples in the book, including the Worked Examples, are provided with the source code for this book. Download the files to your computer for easy access as you work through the chapters.

Practice makes perfect.

Of course, programming students need to be able to implement nontrivial programs, but they first need to have the confidence that they can succeed. The Enhanced E-Text immerses students in activities designed to foster in-depth learning. Students don't just watch animations and code traces, they work on generating them. The activities provide instant feedback to show students what they did right and where they need to study more. A wealth of practice opportunities, including code completion questions and skill-oriented multiple-choice questions, appear at the end of each section, and each chapter ends with well-crafted review exercises and programming projects.

Problem solving strategies are made explicit.

Practical, step-by-step illustrations of techniques help students devise and evaluate solutions to programming problems. Introduced where they are most relevant, these strategies address barriers to success for many students. Strategies included are:

- Algorithm Design (with pseudocode)
- First Do It By Hand (doing sample calculations by hand)
- Flowcharts

- Selecting Test Cases
- Hand-Tracing
- Storyboards
- Solve a Simpler Problem First
- Reusable Functions
- Stepwise Refinement
- Adapting Algorithms
- Discovering Algorithms by Manipulating Physical Objects
- Draw a Picture (pointer diagrams)
- Tracing Objects (identifying state and behavior)
- Discovering Classes
- Thinking Recursively
- Estimating the Running Time of an Algorithm

A visual approach motivates the reader and eases navigation.

Photographs present visual analogies that explain the nature and behavior of computer concepts. Step-by-step figures illustrate complex program operations. Syntax boxes and example tables present a variety of typical and special cases in a compact format. It is easy to get the "lay of the land" by browsing the visuals, before focusing on the textual material.

Visual features help the reader with navigation.

Focus on the essentials while being technically accurate.

An encyclopedic coverage is not helpful for a beginning programmer, but neither is the opposite—reducing the material to a list of simplistic bullet points. In this book, the essentials are presented in digestible chunks, with separate notes that go deeper into good practices or language features when the reader is ready for the additional information. You will not find artificial over-simplifications that give an illusion of knowledge.

Reinforce sound engineering practices.

A multitude of useful tips on software quality and common errors encourage the development of good programming habits. The focus is on test-driven development, encouraging students to test their programs systematically.

Engage with optional engineering and business exercises.

End-of-chapter exercises are enhanced with problems from scientific and business domains. Designed to engage students, the exercises illustrate the value of programming in applied fields.

New to This Edition

Updated for Modern Versions of C++

A number of features of the C++ 2011 and C++ 2014 standards are described either as recommended "best practice" or as Special Topics.

New and Reorganized Topics

The book now supports two pathways into object-oriented programming and inheritance. Pointers and structures can be covered before introducing classes. Alternatively, pointers can be deferred until after the implementation of classes.

This edition further supports a second course in computer science by adding coverage of the implementation of common data structures and algorithms.

A sequence of Worked Examples and exercises introduces "media computation," such as generating and modifying images, sounds, and animations.

Lower-Cost, Interactive Format

This third edition is published as a lower-cost Enhanced E-Text that supports active learning through a wealth of interactive activities. These activities engage and prepare students for independent programming and the Review Exercises, Practice Exercises, and Programming Projects at the end of each E-Text chapter. The Enhanced E-Text may also be bundled with an Abridged Print Companion, which is a bound book that contains the entire text for reference, but without exercises or practice material.

Interactive learning solutions are expanding every day, so to learn more about these options or to explore other options to suit your needs, please contact your Wiley account manager (`www.wiley.com/go/whosmyrep`) or visit the product information page for this text on wiley.com (`http://wiley.com/college/sc/horstmann`).

The Enhanced E-Text is designed to enable student practice without the instructor assigning the interactivities or recording their scores. If you are interested in assigning and grading students' work on them, ask your Wiley Account Manager about the online course option implemented in the Engage Learning Management System. The Engage course supports the assignment and automatic grading of the interactivities. Engage access includes access to the Enhanced E-Text.

Features in the Enhanced E-Text

The interactive Enhanced E-Text guides students from the basics to writing complex programs. After they read a bit, they can try all of the interactive exercises for that section. Active reading is an engaging way for students to ensure that students are prepared before going to class.

There five types of interactivities:

Code Walkthrough Code Walkthrough activities ask students to trace through a segment of code, choosing which line will be executed next and entering the new values of variables changed by the code's execution. This activity simulates the hand-tracing problem solving technique taught in Chapters 3 and 4—but with immediate feedback.

Example Table Example table activities make the student the active participant in building up tables of code examples similar to those found in the book. The tables come in many different forms. Some tables ask the student to determine the output of a line of code, or the value of an expression, or to provide code for certain tasks. This activity helps students assess their understanding of the reading—while it is easy to go back and review.

Algorithm Animation An algorithm animation shows the essential steps of an algorithm. However, instead of passively watching, students get to predict each step. When finished, students can start over with a different set of inputs. This is a surprisingly effective way of learning and remembering algorithms.

Rearrange Code Rearrange code activities ask the student to arrange lines of code by dragging them from the list on the right to the area at left so that the resulting code fulfills the task described in the problem. This activity builds facility with coding structure and implementing common algorithms.

Object Diagram Object diagram activities ask the student to create a memory diagram to illustrate how variables and objects are initialized and updated as sample code executes. The activity depicts variables, objects, and references in the same way as the figures in the book. After an activity is completed, pressing "Play" replays the animation. This activity goes beyond hand-tracing to illuminate what is happening in memory as code executes.

Code Completion Code completion activities ask the student to finish a partially-completed program, then paste the solution into CodeCheck (a Wiley-based online code evaluator) to learn whether it produces the desired result. Tester classes on the CodeCheck site run and report whether the code passed the tests. This activity serves as a skill-building lab to better prepare the student for writing programs from scratch.

A Tour of the Book

This book is intended for a two-semester introduction to programming that may also include algorithms and data structures. The organization of chapters offers the same flexibility as the previous edition; dependencies among the chapters are also shown in Figure 1.

Part A: Fundamentals (Chapters 1–8)

The first six chapters follow a traditional approach to basic programming concepts. Students learn about control structures, stepwise refinement, and arrays. Objects are used only for input/output and string processing. Input/output is first covered in Chapter 2, which may be followed by an introduction to reading and writing text files in Section 8.1.

In a course for engineers with a need for systems and embedded programming, you will want to cover Chapter 7 on pointers. Sections 7.1 and 7.4 are sufficient for using pointers with polymorphism in Chapter 10.

File processing is the subject of Chapter 8. Section 8.1 can be covered sooner for an introduction to reading and writing text files. The remainder of the chapter gives additional material for practical applications.

Part B: Object-Oriented Design (Chapters 9–10)

After students have gained a solid foundation, they are ready to tackle the implementation of classes. Chapters 9 and 10 introduce the object-oriented features of C++. Chapter 9 introduces class design and implementation. Chapter 10 covers inheritance and polymorphism. By the end of these chapters, students will be able to implement programs with multiple interacting classes.

Part C: Data Structures and Algorithms (Chapters 11–17)

Chapters 11–17 cover algorithms and data structures at a level suitable for beginning students. Recursion, in Chapter 11, starts with simple examples and progresses

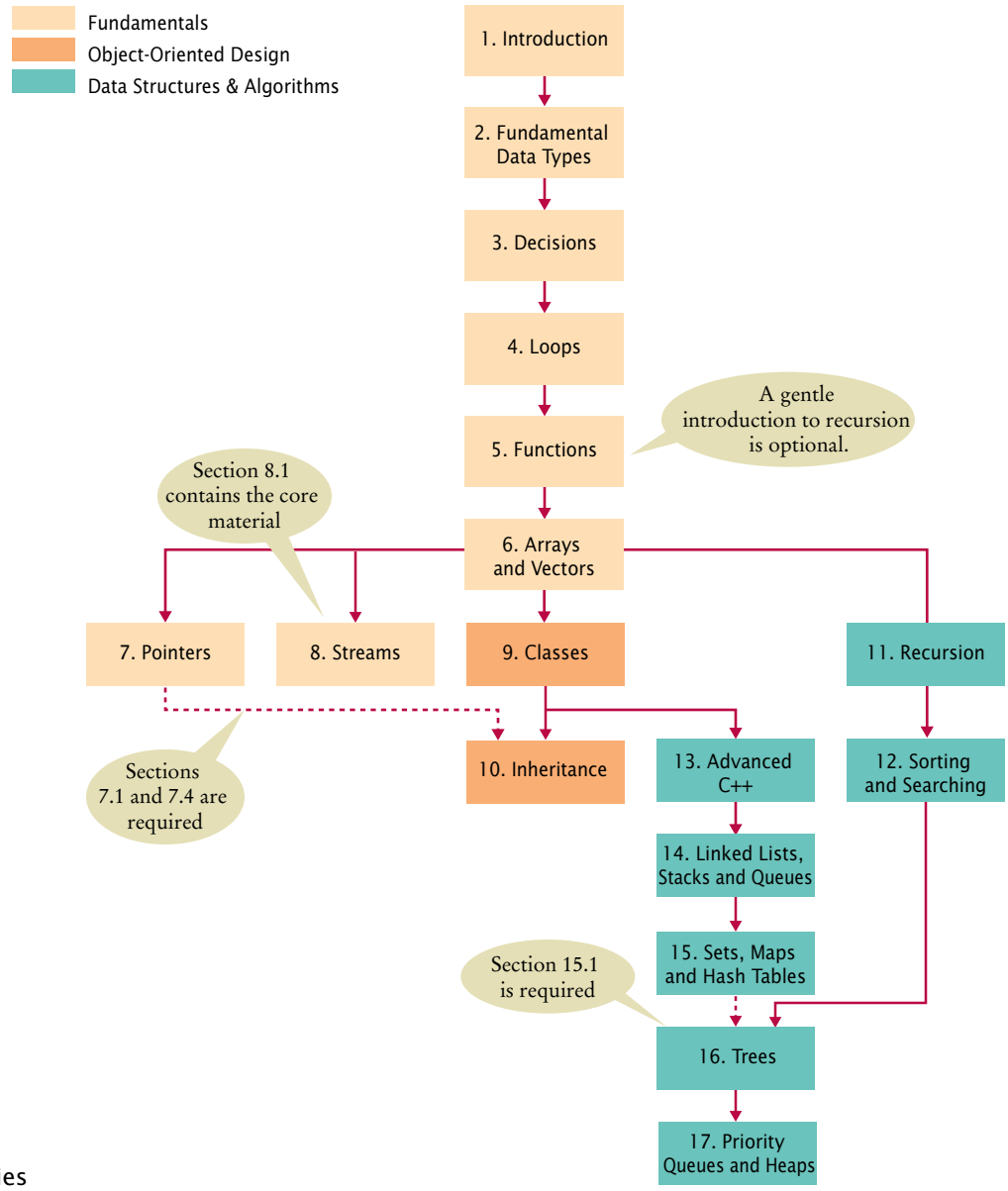


Figure 1
Chapter Dependencies

to meaningful applications that would be difficult to implement iteratively. Chapter 12 covers quadratic sorting algorithms as well as merge sort, with an informal introduction to big-Oh notation. Chapter 13 introduces advanced C++ features that are required for implementing data structures, including templates and memory management. Chapters 14–17 cover linear and tree-based data structures. Students learn how to use the standard C++ library versions. They then study the implementations of these data structures and analyze their efficiency.

Any subset of these chapters can be incorporated into a custom print version of this text; ask your Wiley sales representative for details, or visit `customselect.wiley.com` to create your custom order.

Appendices

Appendices A and B summarize C++ reserved words and operators. Appendix C lists character escape sequences and ASCII character code values. Appendix D documents all of the library functions and classes used in this book.

Appendix E contains a programming style guide. Using a style guide for programming assignments benefits students by directing them toward good habits and reducing gratuitous choice. The style guide is available in electronic form on the book's companion web site so that instructors can modify it to reflect their preferred style.

Appendix F introduces common number systems used in computing.

Web Resources

This book is complemented by a complete suite of online resources. Go to `www.wiley.com/go/bclo3` to visit the online companion sites, which include

- Source code for all example programs in the book and its Worked Examples, plus additional example programs.
- Worked Examples that apply the problem-solving steps in the book to other realistic examples.
- Lecture presentation slides (for instructors only).
- Solutions to all review and programming exercises (for instructors only).
- A test bank that focuses on skills, not just terminology (for instructors only). This extensive set of multiple-choice questions can be used with a word processor or imported into a course management system.
- "CodeCheck" assignments that allow students to work on programming problems presented in an innovative online service and receive immediate feedback. Instructors can assign exercises that have already been prepared, or easily add their own. Visit `http://codecheck.it` to learn more.

Pointers in the print companion describe what students will find in their E-Text or online.

WORKED EXAMPLE 2.1
Computing Travel Time

Learn how to develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain. See your E-Text or visit `wiley.com/go/bclo3`

Courtesy of NASA.

EXAMPLE CODE See `how_to_1/scores_vector` in your companion code for a solution using vectors instead of arrays.

A Walkthrough of the Learning Aids

The pedagogical elements in this book work together to focus on and reinforce key concepts and fundamental principles of programming, with additional tips and detail organized to support and deepen these fundamentals. In addition to traditional features, such as chapter objectives and a wealth of exercises, each chapter contains elements geared to today's visual learner.

Throughout each chapter, **margin notes** show where new concepts are introduced and provide an outline of key ideas.

4.3 The for Loop

The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
   cout << counter << endl;
   counter++; // Update the counter
}
```

Because this loop type is so common, there is a special form for it, called the `for` loop (see Syntax 4.2).

```
for (counter = 1; counter <= 10; counter++)
{
   cout << counter << endl;
}
```

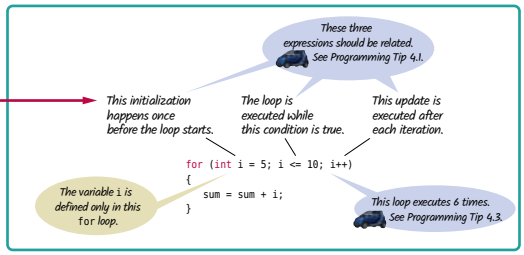
Some people call this loop *count-controlled*. In contrast, the `while` loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs (for example, when the balance reaches the target). Another commonly-used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times—ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.

© Enrico Fianchini/iStockphoto.

You can visualize the `for` *loop as an orderly sequence of steps.*

Annotated **syntax boxes** provide a quick, visual overview of new language constructs.

Syntax 4.2 for Statement



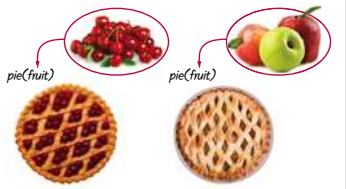
Annotations explain required components and point to more information on common errors or best practices associated with the syntax.

The `for` loop neatly groups the initialization, condition, and update expressions together. However, it is important to realize that these expressions are *not* executed together (see Figure 3).

Like a variable in a computer program, a parking space has an identifier and contents.

Analogies to everyday objects are used to explain the nature and behavior of concepts such as variables, data types, loops, and more.

Memorable photos reinforce analogies and help students remember the concepts.



A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.

Problem Solving sections teach techniques for generating ideas and evaluating proposed solutions, often using pencil and paper or other artifacts. These sections emphasize that most of the planning and problem solving that makes students successful happens away from the computer.

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as C++ programmers, we will say that we swap the coins in positions 0 and 4:

Next, we swap the coins in positions 1 and 5:

HOW TO 1.1

Describing an Algorithm with Pseudocode

This is the first of many "How To" sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in C++, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode—a sequence of precise steps formulated in English. To illustrate, we'll devise an algorithm for this problem:

Problem Statement You have the choice of buying one of two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of $4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?

© dlewis33/Getty Images.

Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- *purchase price1* and *fuel efficiency1*, the price and fuel efficiency (in mpg) of the first car
- *purchase price2* and *fuel efficiency2*, the price and fuel efficiency of the second car

How To guides give step-by-step guidance for common programming tasks, emphasizing planning and testing. They answer the beginner's question, "Now what do I do?" and integrate key concepts into a problem-solving sequence.

WORKED EXAMPLE 1.1

Writing an Algorithm for Tiling a Floor

Problem Statement Your task is to tile a rectangular bathroom floor with alternating black and white tiles measuring 4 × 4 inches. The floor dimensions, measured in inches, are multiples of 4.

Step 1 Determine the inputs and outputs.

The inputs are the floor dimensions (length × width), measured in inches. The output is a tiled floor.

Step 2 Break down the problem into smaller tasks.

A natural subtask is to lay one row of tiles. If you can solve that task, then you can solve the problem by laying one row next to the other, starting from a wall, until

© rban/iStockphoto.

Worked Examples apply the steps in the How To to a different example, showing how they can be used to plan, implement, and test a solution to another programming problem.

Table 3 Variable Names in C++

Variable Name	Comment
`can_volume1`	Variable names consist of letters, numbers, and the underscore character.
`x`	In mathematics, you use short variable names such as *x* or *y*. This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1).
⚠ `Can_volume`	**Caution:** Variable names are case sensitive. This variable name is different from `can_volume`.
⊘ `6pack`	**Error:** Variable names cannot start with a number.
⊘ `can volume`	**Error:** Variable names cannot contain spaces.
⊘ `double`	**Error:** You cannot use a reserved word as a variable name.
⊘ `ltr/fl.oz`	**Error:** You cannot use symbols such as . or /

Example tables support beginners with multiple, concrete examples. These tables point out common errors and present another quick reference to the section's topic.

Consider the function call illustrated in Figure 3:

```
double result1 = cube_volume(2);
```

- The parameter variable `side_length` of the `cube_volume` function is created. ❶
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, `side_length` is set to 2. ❷
- The function computes the expression `side_length * side_length * side_length`, which has the value 8. That value is stored in the variable `volume`. ❸
- The function returns. All of its variables are removed. The return value is transferred to the *caller*, that is, the function calling the `cube_volume` function. ❹

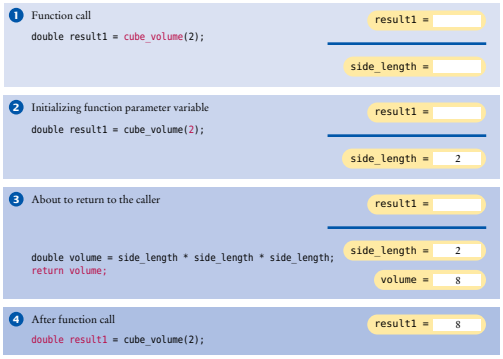


Figure 3 Parameter Passing

Progressive figures trace code segments to help students visualize the program flow. Color is used consistently to make variables and other elements easily recognizable.

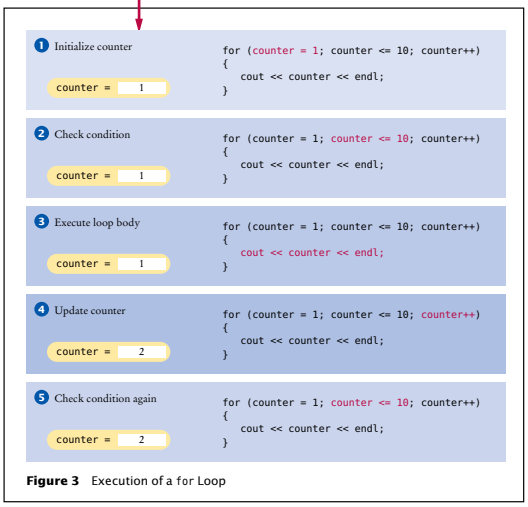


Figure 3 Execution of a `for` Loop

Optional **engineering exercises** engage students with applications from technical fields.

Engineering P7.12 Write a program that simulates the control software for a "people mover" system, a set of driverless trains that move in two concentric circular tracks. A set of switches allows trains to switch tracks.

In your program, the outer and inner tracks should each be divided into ten segments. Each track segment can contain a train that oves either clockwise or counterclockwise. train moves to an adjacent segment in its track or, if that segment is occupied, to e adjacent segment in the other track.

efine a `Segment` structure. Each segment has a pointer to the next and previous gments in its track, a pointer to the next and previous segments in the other track,

sec02/cube.cpp

```
#include <iostream>

using namespace std;

/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}

int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume " << result1 << endl;
   cout << "A cube with side length 10 has volume " << result2 << endl;

   return 0;
}
```

Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

Program listings are carefully designed for easy reading, going well beyond simple color coding. Functions are set off by a subtle outline.

EXAMPLE CODE See sec04 of your companion code for another implementation of the earthquake program that you saw in Section 3.3. Note that the `get_description` function has multiple `return` statements.

Additional example programs are provided with the companion code for students to read, run, and modify.

Common Errors describe the kinds of errors that students often make, with an explanation of why the errors occur, and what to do about them.

Common Error 2.1

Using Undefined Variables

You must define a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_per_ounce;
double liter_per_ounce = 0.0296;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `liter_per_ounce` will be defined in the next line, and it reports an error.

Programming Tips explain good programming practices, and encourage students to be more productive with tips and techniques such as hand-tracing.

Programming Tip 3.6

Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or C++ code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

For example, let's trace the tax program with the data from the program run in Section 3.4. In lines 13 and 14, `tax1` and `tax2` are initialized to 0.

© thomasd007/iStockphoto.

Hand-tracing helps you understand whether a program works correctly.

```
 6  int main()
 7  {
 8     const double RATE1 = 0.10;
 9     const double RATE2 = 0.25;
10     const double RATE1_SINGLE_LIMIT = 32000;
11     const double RATE1_MARRIED_LIMIT = 64000;
12
13     double tax1 = 0;
14     double tax2 = 0;
15
```

tax1	tax2	income	marital status
0	0		

In lines 18 and 22, `income` and `marital_status` are initialized by input statements.

```
16     double income;
17     cout << "Please enter your income: ";
18     cin >> income;
19
20     cout << "Please enter s for single, m for married: ";
21     string marital_status;
22     cin >> marital_status;
23
```

tax1	tax2	income	marital status
0	0	80000	m

Because `marital_status` is not `"s"`, we move to the `else` branch of the outer `if` statement (line 36).

```
24
25
26
27
28
29
30
31
32
```

Special Topics present optional topics and provide additional explanation of others.

Special Topic 6.5

The Range-Based `for` Loop

C++ 11 introduces a convenient syntax for visiting all elements in a "range" or sequence of elements. This loop displays all elements in a vector:

```
vector<int> values = {1, 4, 9, 16, 25, 36};
for (int v : values)
{
   cout << v << " ";
}
```

In each iteration of the loop, `v` is set to an element of the vector. Note that you do not use an index variable. The value of `v` is the element, not the index of the element.

If you want to modify elements, declare the loop variable as a reference:

```
for (int& v : values)
{
   v++;
}
```

This loop increments all elements of the vector.

You can use the reserved word `auto`, which was introduced in Special Topic 2.3, for the type of the element variable:

```
for (auto v : values) { cout << v << " "; }
```

The range-based `for` loop also works for arrays:

```
int primes[] = { 2, 3, 5, 7, 11, 13 };
for (int p : primes)
{
   cout << p << " ";
```

range-based `for` loop is a convenient shortcut for visiting or updating all elements of a or or an array. This book doesn't use it because one can achieve the same result by looping index values. But if you like the more concise form, and use C++ 11 or later, you should ainly consider using it.

special_topic_5 of your companion code for a program that demonstrates the range-based oop.

Computing & Society presents social and historical topics on computing—for interest and to fulfill the "historical and social context" requirements of the ACM/IEEE curriculum guidelines.

Computing & Society 7.1 Embedded Systems

An **embedded system** is a computer system that controls a device. The device contains a processor and other hardware and is controlled by a computer program. Unlike a personal computer, which has been designed to be flexible and run many different computer programs, the hardware and software of an embedded system are tailored to a specific device. Computer controlled devices are becoming increasingly common, ranging from washing machines to medical equipment, cell phones, automobile engines, and spacecraft.

Several challenges are specific to programming embedded systems. Most importantly, a much higher standard of quality control applies. Vendors are often unconcerned about bugs in personal computer software, because they can always make you install a patch or upgrade to the next version. But in an embedded system, that is not an option. Few consumers would feel comfortable upgrading the software in their washing machines or automobile engines. If you ever handed in a programming assignment that you believed to be correct, only to have the instructor or grader find bugs in it, then you know how hard it is to write software that can reliably do its task for many years without a chance of changing it. Quality standards are especially important in devices whose failure would destroy property or endanger human life. Many personal computer purchasers buy computers that are fast and have a lot of storage, because the investment is paid back over time when many programs are run on the same equipment. But the hardware for an embedded device is not shared—it is dedicated to one device. A separate processor, memory, and so on, are built for every copy of the device. If it is possible to shave a few pennies off the manufacturing cost of every unit, the savings can add up quickly for devices that are produced in large volumes. Thus, the programmer of an embedded system has a much larger economic incentive to conserve resources than the desktop software programmer. Unfortunately, trying to conserve resources usually makes it harder to write programs that work correctly.

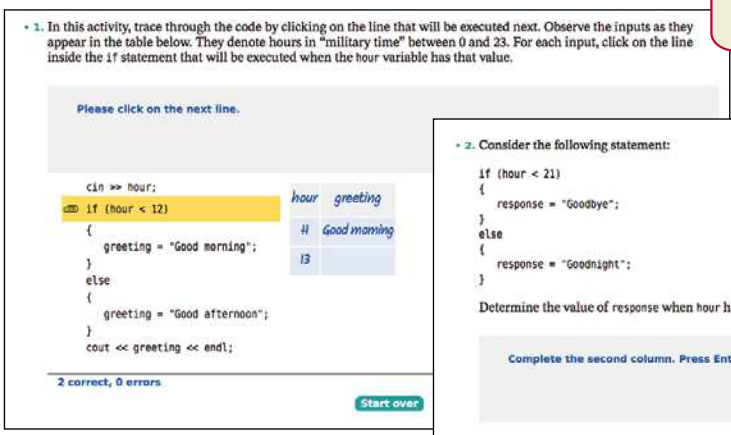
C and C++ are commonly used languages for developing embedded systems.

© Courtesy of Professor Prabal Dutta.

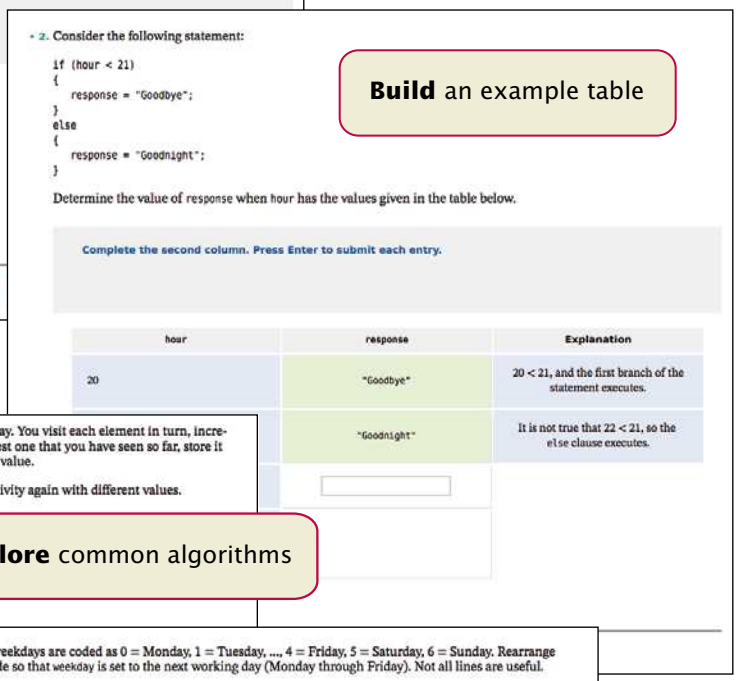
The Controller of an Embedded System

Interactive activities in the E-Text
engage students in active reading as they...

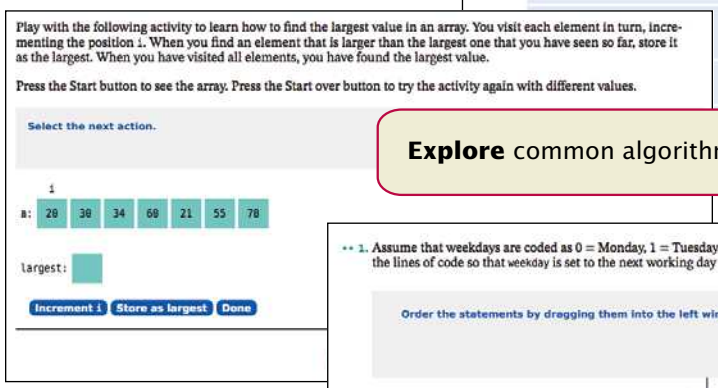
Trace through a code segment



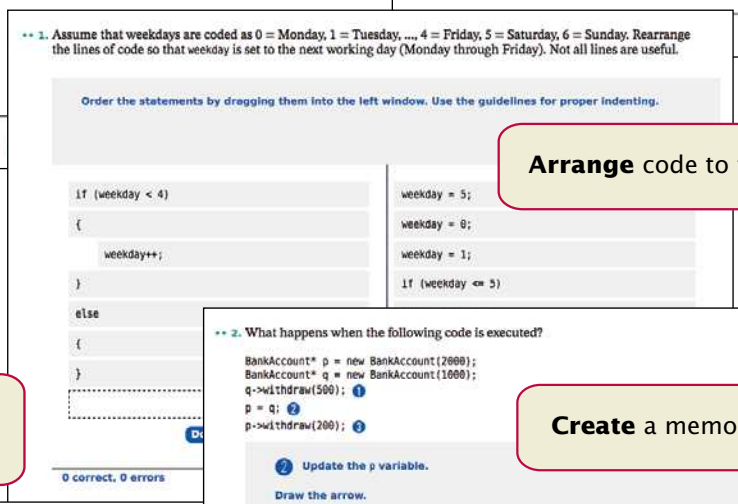
Build an example table



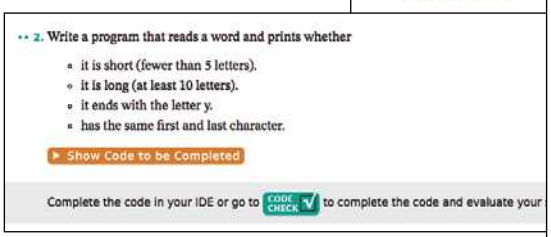
Explore common algorithms



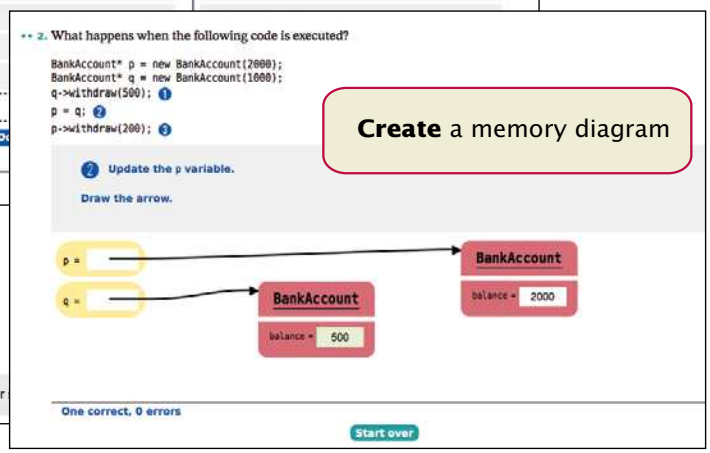
Arrange code to fulfill a task



Complete a program and get immediate feedback



Create a memory diagram



Acknowledgments

Many thanks to Don Fowley, Graig Donini, Dan Sayre, Ryann Dannelly, David Dietz, Laura Abrams, and Billy Ray at John Wiley & Sons for their help with this project. An especially deep acknowledgment and thanks goes to Cindy Johnson for her hard work, sound judgment, and amazing attention to detail.

I am grateful to Mark Atkins, *Ivy Technical College*, Katie Livsie, *Gaston College,* Larry Morell, *Arkansas Tech University*, and Rama Olson, *Gaston College,* for their contributions to the supplemental material. Special thanks to Stephen Gilbert, *Orange Coast Community College,* for his help with the interactive exercises.

Every new edition builds on the suggestions and experiences of new and prior reviewers, contributors, and users. We are very grateful to the individuals who provided feedback, reviewed the manuscript, made valuable suggestions and contributions, and brought errors and omissions to my attention. They include:

Charles D. Allison, *Utah Valley State College*
Fred Annexstein, *University of Cincinnati*
Mark Atkins, *Ivy Technical College*
Stefano Basagni, *Northeastern University*
Noah D. Barnette, *Virginia Tech*
Susan Bickford, *Tallahassee Community College*
Ronald D. Bowman, *University of Alabama, Huntsville*
Robert Burton, *Brigham Young University*
Peter Breznay, *University of Wisconsin, Green Bay*
Richard Cacace, *Pensacola Junior College, Pensacola*
Kuang-Nan Chang, *Eastern Kentucky University*
Joseph DeLibero, *Arizona State University*
Subramaniam Dharmarajan, *Arizona State University*
Mary Dorf, *University of Michigan*
Marty Dulberg, *North Carolina State University*
William E. Duncan, *Louisiana State University*
John Estell, *Ohio Northern University*
Waleed Farag, *Indiana University of Pennsylvania*
Evan Gallagher, *Polytechnic Institute of New York University*
Stephen Gilbert, *Orange Coast Community College*
Kenneth Gitlitz, *New Hampshire Technical Institute*
Daniel Grigoletti, *DeVry Institute of Technology, Tinley Park*
Barbara Guillott, *Louisiana State University*
Charles Halsey, *Richland College*
Jon Hanrath, *Illinois Institute of Technology*
Neil Harrison, *Utah Valley University*
Jurgen Hecht, *University of Ontario*
Steve Hodges, *Cabrillo College*

Jackie Jarboe, *Boise State University*
Debbie Kaneko, *Old Dominion University*
Mir Behrad Khamesee, *University of Waterloo*
Sung-Sik Kwon, *North Carolina Central University*
Lorrie Lehman, *University of North Carolina, Charlotte*
Cynthia Lester, *Tuskegee University*
Yanjun Li, *Fordham University*
W. James MacLean, *University of Toronto*
LindaLee Massoud, *Mott Community College*
Adelaida Medlock, *Drexel University*
Charles W. Mellard, *DeVry Institute of Technology, Irving*
Larry Morell, *Arkansas Tech University*
Ethan V. Munson, *University of Wisconsin, Milwaukee*
Arun Ravindran, *University of North Carolina at Charlotte*
Philip Regalbuto, *Trident Technical College*
Don Retzlaff, *University of North Texas*
Jeff Ringenberg, *University of Michigan, Ann Arbor*
John P. Russo, *Wentworth Institute of Technology*
Kurt Schmidt, *Drexel University*
Brent Seales, *University of Kentucky*
William Shay, *University of Wisconsin, Green Bay*
Michele A. Starkey, *Mount Saint Mary College*
William Stockwell, *University of Central Oklahoma*
Jonathan Tolstedt, *North Dakota State University*
Boyd Trolinger, *Butte College*
Muharrem Uyar, *City College of New York*
Mahendra Velauthapillai, *Georgetown University*
Kerstin Voigt, *California State University, San Bernardino*
David P. Voorhees, *Le Moyne College*
Salih Yurttas, *Texas A&M University*

A special thank you to all of our class testers:

Pani Chakrapani and the students of the University of Redlands
Jim Mackowiak and the students of Long Beach City College, LAC
Suresh Muknahallipatna and the students of the University of Wyoming
Murlidharan Nair and the students of the Indiana University of South Bend
Harriette Roadman and the students of New River Community College
David Topham and the students of Ohlone College
Dennie Van Tassel and the students of Gavilan College

CONTENTS

ALPHABETICAL LIST OF SYNTAX BOXES

Programming Tips

Special Topics

Computing & Society

Programming Tips

Special Topics

Computing & Society

Programming Tips

Special Topics

Computing & Society

CHAPTER 1

INTRODUCTION

© JanPietruszka/iStockphoto.

CHAPTER GOALS

- To learn about the architecture of computers
- To learn about machine languages and higher-level programming languages
- To become familiar with your compiler
- To compile and run your first C++ program
- To recognize compile-time and run-time errors
- To describe an algorithm with pseudocode
- To understand the activity of programming

CHAPTER CONTENTS

Just as you gather tools, study a project, and make a plan for tackling it, in this chapter you will gather up the basics you need to start learning to program. After a brief introduction to computer hardware, software, and programming in general, you will learn how to write and run your first C++ program. You will also learn how to diagnose and fix programming errors, and how to use pseudocode to describe an algorithm—a step-by-step description of how to solve a problem—as you plan your programs.

1.1 What Is Programming?

Computers execute very basic instructions in rapid succession.

You have probably used a computer for work or fun. Many people use computers for everyday tasks such as electronic banking or writing a term paper. Computers are good for such tasks. They can handle repetitive chores, such as totaling up numbers or placing words on a page, without getting bored or exhausted.

The flexibility of a computer is quite an amazing phenomenon. The same machine can balance your checkbook, print your term paper, and play a game. In contrast, other machines carry out a much narrower range of tasks; a car drives and a toaster toasts. Computers can carry out a wide range of tasks because they execute different programs, each of which directs the computer to work on a specific task.

A computer program is a sequence of instructions and decisions.

The computer itself is a machine that stores data (numbers, words, pictures), interacts with devices (the monitor, the sound system, the printer), and executes programs. A **computer program** tells a computer, in minute detail, the sequence of steps that are needed to fulfill a task. The physical computer and peripheral devices are collectively called the **hardware**. The programs the computer executes are called the **software**.

Today's computer programs are so sophisticated that it is hard to believe that they are composed of extremely primitive operations. A typical operation may be one of the following:

- Put a red dot at this screen position.
- Add up these two numbers.
- If this value is negative, continue the program at a certain instruction.

The computer user has the illusion of smooth interaction because a program contains a huge number of such operations, and because the computer can execute them at great speed.

Programming is the act of designing and implementing computer programs.

The act of designing and implementing computer programs is called *programming*. In this book, you will learn how to program a computer—that is, how to direct the computer to execute tasks.

To write a computer game with motion and sound effects or a word processor that supports fancy fonts and pictures is a complex task that requires a team of many highly skilled programmers. Your first programming efforts will be more mundane. The concepts and skills you learn in this book form an important foundation, and you should not be disappointed if your first programs do not rival the sophisticated software that is familiar to you. Actually, you will find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer precisely and quickly carry out a task that would take you hours of drudgery, to

make small changes in a program that lead to immediate improvements, and to see the computer become an extension of your mental powers.

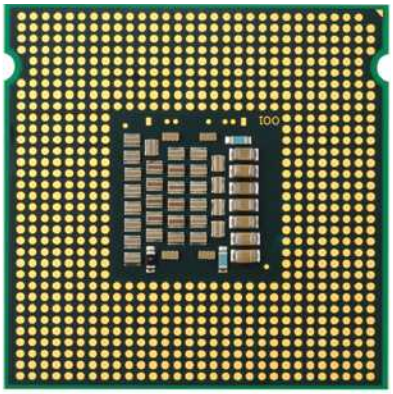
1.2 The Anatomy of a Computer

To understand the programming process, you need to have a rudimentary understanding of the building blocks that make up a computer. We will look at a personal computer. Larger computers have faster, larger, or more powerful components, but they have fundamentally the same design.

At the heart of the computer lies the **central processing unit** (CPU) (see Figure 1). It consists of a single *chip*, or a small number of chips. A computer chip (integrated circuit) is a component with a plastic or metal housing, metal connectors, and inside wiring made principally from silicon. For a CPU chip, the inside wiring is enormously complicated. For example, the Pentium chip (a popular CPU for personal computers at the time of this writing) is composed of several million structural elements, called *transistors*.

The central processing unit (CPU) performs program control and data processing.

The CPU performs program control and data processing. That is, the CPU locates and executes the program instructions; it carries out arithmetic operations such as addition, subtraction, multiplication, and division; it fetches data from external memory or devices and stores data back.



© Amorphis/iStockphoto.

Figure 1 Central Processing Unit

Storage devices include memory and secondary storage.

There are two kinds of storage. Primary storage, or memory, is made from electronic circuits that can store data, provided they are supplied with electric power. **Secondary storage**, usually a **hard disk** (see Figure 2) or a solid-state drive, provides slower and less expensive storage that persists without electricity. A hard disk consists of rotating platters, which are coated with a magnetic material. A solid-state drive uses electronic components that can retain information without power, and without moving parts.

© PhotoDisc, Inc./Getty Images.

Figure 2
A Hard Disk

Programs and data are typically stored on the hard disk and loaded into memory when the program starts. The program then updates the data in memory and writes the modified data back to the hard disk.

To interact with a human user, a computer requires peripheral devices. The computer transmits information (called *output*) to the user through a display screen, speakers, and printers. The user can enter information (called *input*) by using a keyboard or a pointing device such as a mouse.

Some computers are self-contained units, whereas others are interconnected through *networks*. Through the network cabling, the computer can read data and programs from central storage locations or send data to other computers. For the user of a networked computer it may not even be obvious which data reside on the computer itself and which are transmitted through the network.

Figure 3 gives a schematic overview of the architecture of a personal computer. Program instructions and data (such as text, numbers, audio, or video) reside in secondary storage or elsewhere on the network. When a program is started, its instructions are brought into memory, where the CPU can read them. The CPU reads and executes one instruction at a time. As directed by these instructions, the CPU reads data, modifies it, and writes it back to memory or secondary storage. Some program instructions will cause the CPU to place dots on the display screen or printer or to vibrate the speaker. As these actions happen many times over and at great speed, the human user will perceive images and sound. Some program instructions read user input from the keyboard, mouse, touch sensor, or microphone. The program analyzes the nature of these inputs and then executes the next appropriate instruction.

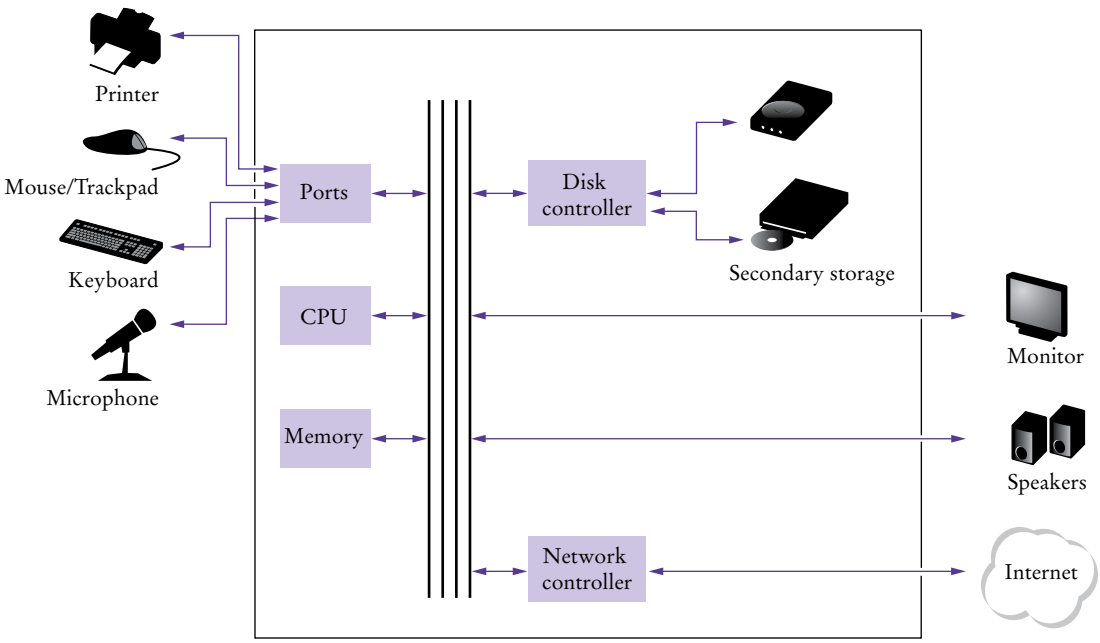


Figure 3 Schematic Design of a Personal Computer

Computing & Society 1.1 Computers Are Everywhere

When computers were first invented in the 1940s, a computer filled an entire room. Figure 4 shows the ENIAC (*e*lectronic *n*umerical *i*ntegrator *a*nd *c*omputer), completed in 1946 at the University of Pennsylvania. The ENIAC was used by the military to compute the trajectories of projectiles. Nowadays, computing facilities of search engines, Internet shops, and social networks fill huge buildings called data centers. At the other end of the spectrum, computers are all around us. Your cell phone has a computer inside, as do many credit cards and fare cards for public transit. A modern car has several computers—to control the engine, brakes, lights, and radio.

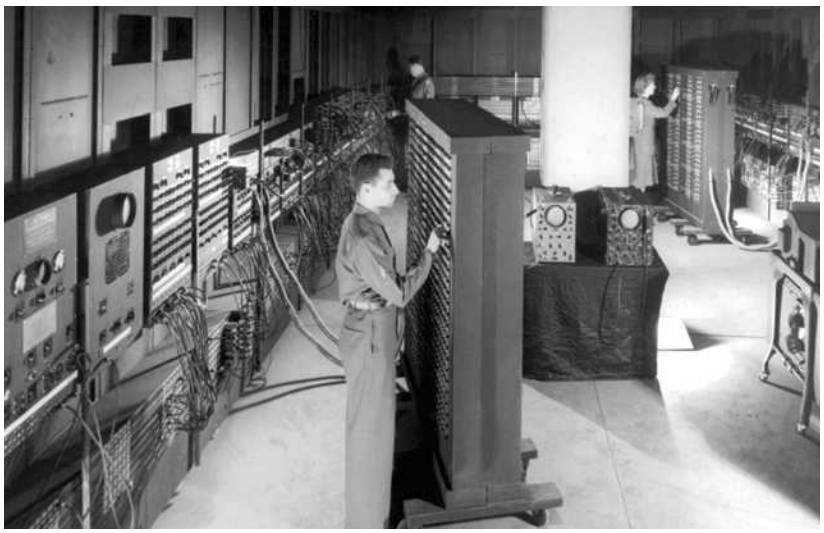
The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are today carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies nowadays are often consumed on computers, and computers are almost always involved in their production. The book that you are reading right now could not have been written without computers.

© Maurice Savage/Alamy Stock Photo.

This transit card contains a computer.

Knowing about computers and how to program them has become an essential skill in many careers. Engineers design computer-controlled cars and medical equipment that preserve lives. Computer scientists develop programs that help people come together to support social causes. For example, activists used social networks to share videos showing abuse by repressive regimes, and this information was instrumental in changing public opinion.

As computers, large and small, become ever more embedded in our everyday lives, it is increasingly important for everyone to understand how they work, and how to work with them. As you use this book to learn how to program a computer, you will develop a good understanding of computing fundamentals that will make you a more informed citizen and, perhaps, a computing professional.



© UPPA/Photoshot.

Figure 4 The ENIAC

1.3 Machine Code and Programming Languages

On the most basic level, computer instructions are extremely primitive. The processor executes *machine instructions*. A typical sequence of machine instructions is

1. Move the contents of memory location 40000 into the CPU.
2. If that value is greater than 100, continue with the instruction that is stored in memory location 11280.

Computer programs are stored as machine instructions in a code that depends on the processor type.

Actually, machine instructions are encoded as numbers so that they can be stored in memory. On a Pentium processor, this sequence of instruction is encoded as the sequence of numbers

161 40000 45 100 127 11280

On a processor from a different manufacturer, the encoding would be different. When this kind of processor fetches this sequence of numbers, it decodes them and executes the associated sequence of commands.

How can we communicate the command sequence to the computer? The simplest method is to place the actual numbers into the computer memory. This is, in fact, how the very earliest computers worked. However, a long program is composed of thousands of individual commands, and it is a tedious and error-prone affair to look up the numeric codes for all commands and place the codes manually into memory. As already mentioned, computers are really good at automating tedious and error-prone activities. It did not take long for computer scientists to realize that the computers themselves could be harnessed to help in the programming process.

Computer scientists devised **high-level programming languages** that allow programmers to describe tasks, using a **syntax** that is more closely related to the problems to be solved. In this book, we will use the C++ programming language, which was developed by Bjarne Stroustrup in the 1980s.

C++ is a general-purpose language that is in widespread use for systems and embedded programming.

Over the years, C++ has grown by the addition of many features. A standardization process culminated in the publication of the international C++ standard in 1998. A minor update to the standard was issued in 2003. A major revision came to fruition in 2011, followed by updates in 2014 and 2017. At this time, C++ is the most commonly used language for developing system software such as databases and operating systems. Just as importantly, C++ is commonly used for programming "embedded systems", computers that control devices such as automobile engines or robots.

© Courtesy of Bjarne Stroustrup.

Bjarne Stroustrup

Here is a typical statement in C++:

```
if (int_rate > 100) { cout << "Interest rate error"; }
```

High-level programming languages are independent of the processor.

This means, "If the interest rate is over 100, display an error message". A special computer program, a **compiler**, translates this high-level description into machine instructions for a particular processor.

High-level languages are independent of the underlying hardware. C++ instructions work equally well on an Intel Pentium and a processor in a cell phone. Of course, the compiler-generated machine instructions are different, but the programmer who uses the compiler need not worry about these differences.

Computing & Society 1.2 Standards Organizations

Two standards organizations, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), have jointly developed the definitive standard for the C++ language.

Why have standards? You encounter the benefits of standardization every day. When you buy a light bulb, you can be assured that it fits in the socket without having to measure the socket at home and the bulb in the store. In fact, you may have experienced how painful the lack of standards can be if you have ever purchased a flashlight with nonstandard bulbs. Replacement bulbs for such a flashlight can be difficult and expensive to obtain.

The ANSI and ISO standards organizations are associations of industry professionals who develop standards for everything from car tires and credit card shapes to programming languages. Having a standard for a programming language such as C++ means that you can take a program that you developed on one system with one manufacturer's compiler to a different system and be assured that it will continue to work.

© Denis Vorob'yev/iStockphoto.

1.4 Becoming Familiar with Your Programming Environment

Set aside some time to become familiar with the programming environment that you will use for your class work.

Many students find that the tools they need as programmers are very different from the software with which they are familiar. You should spend some time making yourself familiar with your programming environment. Because computer systems vary widely, this book can give only an outline of the steps you need to follow. It is a good idea to participate in a hands-on lab, or to ask a knowledgeable friend to give you a tour.

Step 1 Start the C++ development environment.

Computer systems differ greatly in this regard. On many computers there is an **integrated development environment** in which you can write and test your programs. On other computers you first launch an **editor**, a program that functions like a word processor, in which you can enter your C++ instructions; then open a *console window* and type commands to execute your program. Other programming environments are online. In such an environment, you write programs in a web browser. The programs are then executed on a remote machine, and the results are displayed in the web browser window. You need to find out how to get started with your environment.

Step 2 Write a simple program.

The traditional choice for the very first program in a new programming language is a program that displays a simple greeting: "Hello, World!". Let us follow that tradition. Here is the "Hello, World!" program in C++:

```
#include <iostream>

using namespace std;

int main()
{
```

```
   cout << "Hello, World!" << endl;
   return 0;
}
```

We will examine this program in the next section.

An editor is a program for entering and modifying text, such as a C++ program.

No matter which programming environment you use, you begin your activity by typing the program statements into an editor window.

Create a new file and call it `hello.cpp`, using the steps that are appropriate for your environment. (If your environment requires that you supply a project name in addition to the file name, use the name `hello` for the project.) Enter the program instructions *exactly* as they are given above. Alternatively, locate an electronic copy of the program in the source files for this book and paste it into your editor. (You can download the full set of files for this book from its companion site at `wiley.com/go/bclo3`.)

C++ is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.

As you write this program, pay careful attention to the various symbols, and keep in mind that C++ is **case sensitive**. You must enter upper- and lowercase letters exactly as they appear in the program listing. You cannot type `MAIN` or `Endl`. If you are not careful, you will run into problems—see Common Error 1.2.

Step 3 Compile and run the program.

The compiler translates C++ programs into machine code.

The process for building and running a C++ program depends greatly on your programming environment. In some integrated development environments, you simply push a button. In other environments, you may have to type commands. When you run the test program, the message

```
Hello, World!
```

will appear somewhere on the screen (see Figures 5 and 6).

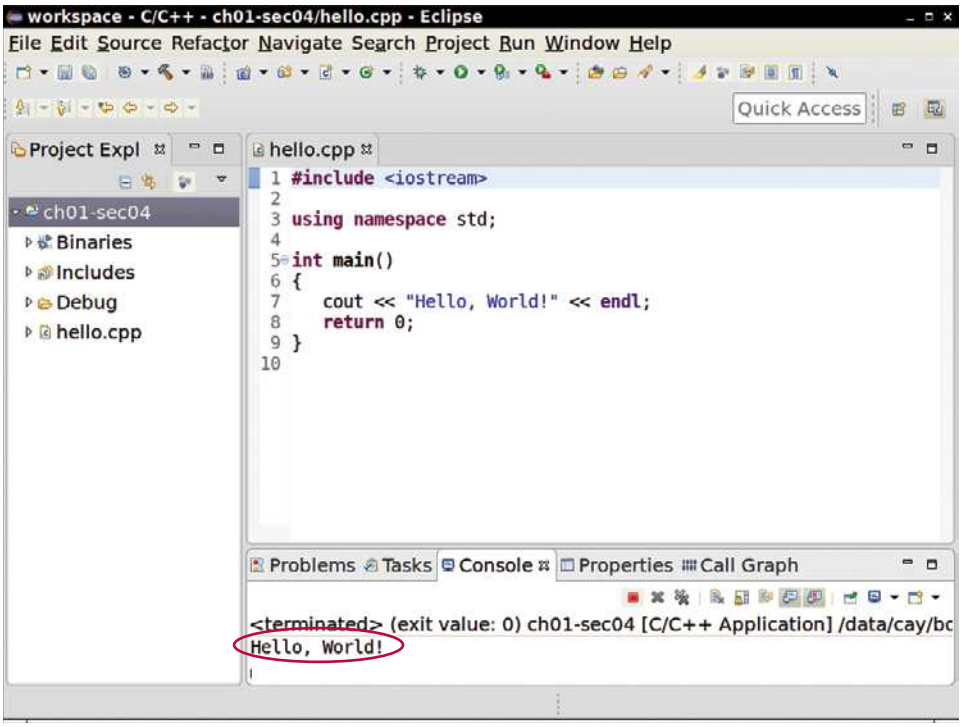


Figure 5 Running the `hello` Program in an Integrated Development Environment

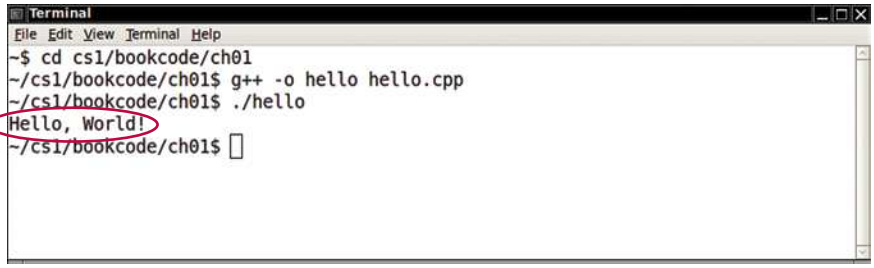


Figure 6 Compiling and Running the `hello` Program in a Console Window

The linker combines machine code with library code into an executable program.

It is useful to know what goes on behind the scenes when your program gets built. First, the compiler translates the C++ **source code** (that is, the statements that you wrote) into machine instructions. The **machine code** contains only the translation of the code that you wrote. That is not enough to actually run the program. To display a string on a window, quite a bit of low-level activity is necessary. The implementors of your C++ development environment provided a library that includes the definition of `cout` and its functionality. A **library** is a collection of code that has been programmed and translated by someone else, ready for you to use in your program. (More complicated programs are built from more than one machine code file and more than one library.) A program called the **linker** takes your machine code and the necessary parts from the C++ library and builds an **executable file**. (Figure 7 gives an overview of these steps.) The executable file is usually called `hello.exe` or `hello`, depending on your computer system. You can run the executable program even after you exit the C++ development environment.

Step 4 Organize your work.

As a programmer, you write programs, try them out, and improve them. You store your programs in *files*. Files have names, and the rules for legal names differ from one system to another. Some systems allow spaces in file names; others don't. Some distinguish between upper- and lowercase letters; others don't. Most C++ compilers require that C++ files end in an **extension** `.cpp`, `.cxx`, `.cc`, or `.C`; for example, `demo.cpp`.

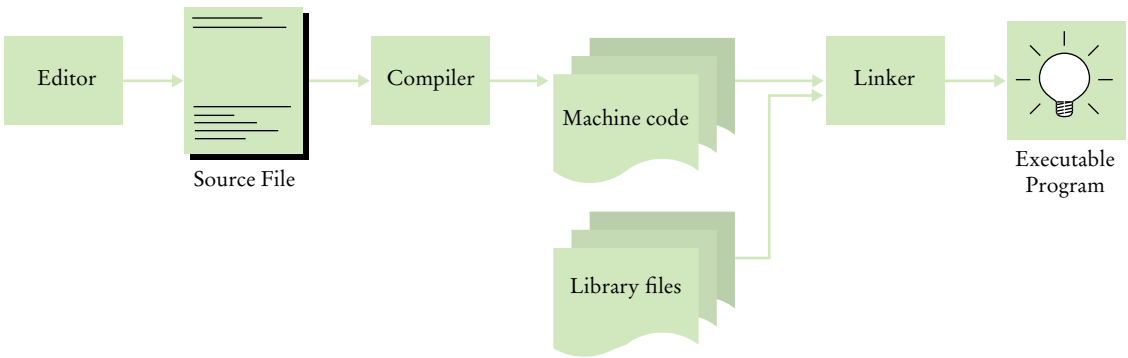


Figure 7 From Source Code to Executable Program

Files are stored in **folders** or **directories**. A folder can contain files as well as other folders, which themselves can contain more files and folders (see Figure 8). This hierarchy can be quite large, and you need not be concerned with all of its branches.

However, you should create folders for organizing your work. It is a good idea to make a separate folder for your programming class. Inside that folder, make a separate folder for each assignment.

Some programming environments place your programs into a default location if you don't specify a folder yourself. In that case, you need to find out where those files are located.

Develop a strategy for keeping backup copies of your work before disaster strikes.

Be sure that you understand where your files are located in the folder hierarchy. This information is essential when you submit files for grading, and for making *backup copies*.

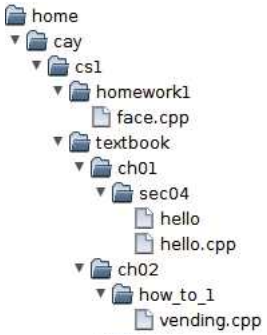


Figure 8 A Folder Hierarchy

You will spend many hours creating and improving C++ programs. It is easy to delete a file by accident, and occasionally files are lost because of a computer malfunction. To avoid the frustration of recreating lost files, get in the habit of making backup copies of your work on a memory stick or on another computer.

Programming Tip 1.1

Backup Copies

Backing up files on a memory stick is an easy and convenient storage method for many people. Another increasingly popular form of backup is Internet file storage. Here are a few pointers to keep in mind.

© Tatiana Popova/iStockphoto.

- *Back up often.* Backing up a file takes only a few seconds, and you will hate yourself if you have to spend many hours recreating work that you could have saved easily. I recommend that you back up your work once every thirty minutes.
- *Rotate backups.* Use more than one directory for backups, and rotate them. That is, first back up to the first directory. Then back up to the second directory. Then use the third, and then go back to the first. That way you always have three recent backups. If your recent changes made matters worse, you can then go back to the older version.
- *Pay attention to the backup direction.* Backing up involves copying files from one place to another. It is important that you do this right—that is, copy from your work location to the backup location. If you do it the wrong way, you will overwrite a newer file with an older version.
- *Check your backups once in a while*. Double-check that your backups are where you think they are. There is nothing more frustrating than to find out that the backups are not there when you need them.
- *Relax, then restore.* When you lose a file and need to restore it from backup, you are likely to be in an unhappy, nervous state. Take a deep breath and think through the recovery process before you start. It is not uncommon for an agitated computer user to wipe out the last backup when trying to restore a damaged file.

1.5 Analyzing Your First Program

In this section, we will analyze the first C++ program in detail. Here again is the source code:

sec05/hello.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     cout << "Hello, World!" << endl;
8     return 0;
9  }
```

The first line,

```
#include <iostream>
```

tells the compiler to include a service for "stream input/output". You will learn in Chapter 8 what a stream is. For now, you should simply remember to add this line into all programs that perform input or output.

The next line,

```
using namespace std;
```

tells the compiler to use the "standard namespace". Namespaces are a mechanism for avoiding naming conflicts in large programs. You need not be concerned about namespaces. For the programs that you will be writing in this book, you will always use the standard namespace. Simply add `using namespace std;` at the top of every program that you write, just below the `#include` directives.

The construction

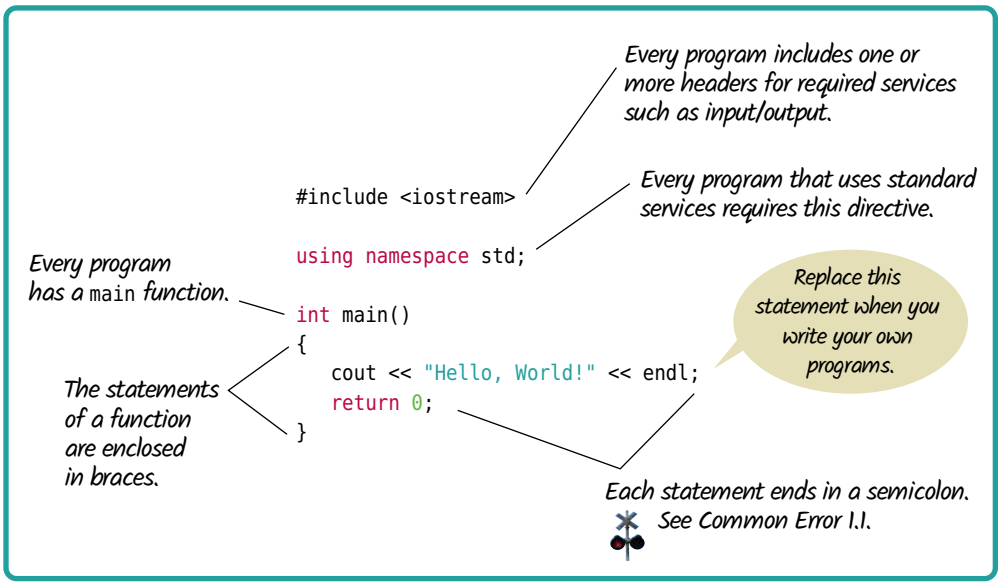
```
int main()
{
   . . .
   return 0;
}
```

defines a *function* called `main` that "returns" an "integer" (that is, a whole number without a fractional part, called `int` in C++) with value `0`. This value indicates that the program finished successfully. A **function** is a collection of programming instructions that carry out a particular task. Every C++ program must have a `main` function. Most C++ programs contain other functions besides `main`, but it will take us until Chapter 5 to discuss functions and return values.

Every C++ program contains a function called `main`.

For now, it is a good idea to consider all these parts as the "plumbing" that is necessary to write a simple program. Simply place the code that you want to execute inside the braces of the `main` function. (The basic structure of a C++ program is shown in Syntax 1.1.)

Syntax 1.1 C++ Program



Use `cout` and the `<<` operator to display values on the screen.

To display values on the screen, you use an entity called `cout` and the `<<` operator (sometimes called the *insertion* operator). For example, the statement

```
cout << 39 + 3;
```

displays the number `42`.

Enclose text strings in quotation marks.

The statement

```
cout << "Hello";
```

displays the **string** `Hello`. A string is a sequence of characters. You must enclose the contents of a string inside quotation marks so that the compiler knows you literally mean the text `"Hello"` and not a function with the same name.

Use `+` to add two numbers and `*` to multiply two numbers.

You can send more than one item to `cout`. Use a `<<` before each one of them. For example,

```
cout << "The answer is " << 6 * 7;
```

displays `The answer is 42` (in C++, the `*` denotes multiplication).

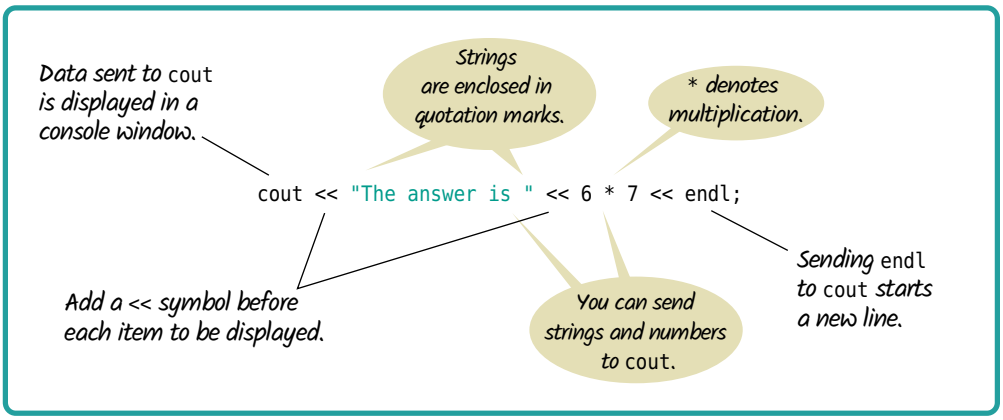
Send `endl` to `cout` to end a line of displayed output.

The `endl` symbol denotes an *end of line* marker. When this marker is sent to `cout`, the cursor is moved to the first column in the next screen row. If you don't use an end of line marker, then the next displayed item will simply follow the current string on the same line. In this program we only printed one item, but in general we will want to print multiple items, and it is a good habit to end all lines of output with an end of line marker.

End each statement with a semicolon.

Finally, note that the output and return statements end in a semicolon, just as every English sentence ends in a period.

Syntax 1.2 Output Statement



Common Error 1.1

Omitting Semicolons

In C++, statements such as output or return statements end in a semicolon. Forgetting to type a semicolon is a common error. It confuses the compiler because the compiler uses the semicolon to find where one statement ends and the next one starts. The compiler does not use line ends or closing braces to recognize the ends of statements. For example, the compiler considers

```
cout << "Hello, World!" << endl
return 0;
```

a single statement, as if you had written

```
cout << "Hello, World!" << endl return 0;
```

and then it doesn't understand that statement, because it does not expect the word `return` in the middle of an output command. The remedy is simple. Just scan every statement for a terminating semicolon, just as you would check that every English sentence ends in a period.

Sometimes, the error message that flags the missing semicolon is displayed in the following line. When you see such a message, be sure to check the preceding line as well.

Special Topic 1.1

Escape Sequences

How can you display a string containing quotation marks, such as

```
Hello, "World"
```

You can't use

```
cout << "Hello, "World"";
```

As soon as the compiler reads `"Hello, "`, it thinks the string is finished, and then it gets all confused about `World`. Compilers have a one-track mind, and if a simple analysis of the input doesn't make sense to them, they just refuse to go on, and they report an error. In contrast, a human would probably realize that the second and third quotation marks were supposed to be part of the string.

Well, how do we then display quotation marks on the screen? The designers of C++ provided an escape hatch. Mark each quotation mark with a backslash character (\), like this:

```
cout << "Hello, \"World\"";
```

The sequence \" denotes a literal quote, not the end of a string. Such a sequence is called an **escape sequence**.

There are a few other escape sequences. If you actually want to show a backslash on the display, you use the escape sequence \\. The statement

```
cout << "Hello\\World";
```

prints

```
Hello\World
```

Finally, the escape sequence \n denotes a **newline** character that starts a new line on the screen. The command

```
cout << "Hello, World!\n";
```

has the same effect as

```
cout << "Hello, World!" << endl;
```

1.6 Errors

Programming languages follow very strict conventions. When you talk to another person, and you scramble or omit a word or two, your conversation partner will usually still understand what you have to say. But when you make an error in a C++ program, the compiler will not try to guess what you meant. (This is actually a good thing. If the compiler were to guess wrongly, the resulting program would do the wrong thing—quite possibly with disastrous effects.) In this section, you will learn how to cope with errors in your program.

© Martin Carlsson/iStockphoto.

Programmers spend a fair amount of time fixing compile-time and run-time errors.

Experiment a little with the `hello.cpp` program. What happens if you make a typing error such as

```
cot << "Hello, World!" << endl;
cout << "Hello, World! << endl;
cout << "Hollo, World!" << endl;
```

A compile-time error is a violation of the programming language rules that is detected by the compiler.

In the first case, the compiler will complain that it has no clue what you mean by `cot`. The exact wording of the error message is dependent on the compiler, but it might be something like "`Undefined symbol cot`". This is a **compile-time error** or **syntax error**. Something is wrong according to the language rules, and the compiler finds it. When the compiler finds one or more errors, it will not translate the program to machine code, and as a consequence there is no program to run. You must fix the error and compile again. It is common to go through several rounds of fixing compile-time errors before compilation succeeds for the first time.

If the compiler finds an error, it will not simply stop and give up. It will try to report as many errors as it can find, so you can fix them all at once. Sometimes,

however, one error throws it off track. This is likely to happen with the error in the second line. Because the programmer forgot the closing quote, the compiler will keep looking for the end of the string. In such cases, it is common for the compiler to emit bogus error reports for neighboring lines. You should fix only those error messages that make sense to you and then recompile.

The error in the third line is of a different kind. The program will compile and run, but its output will be wrong. It will print

```
Hollo, World!
```

A run-time error causes a program to take an action that the programmer did not intend.

This is a **run-time error**. The program is syntactically correct and does something, but it doesn't do what it is supposed to do. The compiler cannot find the error, and it must be flushed out when the program runs, by testing it and carefully looking at its output. Because run-time errors are caused by logical flaws in the program, they are often called **logic errors**. Some kinds of run-time errors are so severe that they generate an **exception**: a signal from the processor that aborts the program with an error message. For example, if your program includes the statement `cout << 1 / 0;` your program may terminate with a "divide by zero" exception.

During program development, errors are unavoidable. Once a program is longer than a few lines, it requires superhuman concentration to enter it correctly without slipping up once. You will find yourself omitting semicolons or quotes more often than you would like, but the compiler will track down these problems for you.

The programmer is responsible for inspecting and testing the program to guard against run-time errors.

Run-time errors are more troublesome. The compiler will not find them—in fact, the compiler will cheerfully translate any program as long as its syntax is correct—but the resulting program will do something wrong. It is the responsibility of the program author to test the program and find any run-time errors. Program testing is an important topic that you will encounter many times in this book.

Common Error 1.2

Misspelling Words

If you accidentally misspell a word, strange things may happen, and it may not always be completely obvious from the error messages what went wrong. Here is a good example of how simple spelling errors can cause trouble:

```
#include <iostream>

using namespace std;

int Main()
{
   cout << "Hello, World!" << endl;
   return 0;
}
```

This code defines a function called `Main`. The compiler will not consider this to be the same as the `main` function, because `Main` starts with an uppercase letter and the C++ language is **case sensitive**. Upper- and lowercase letters are considered to be completely different from each other, and to the compiler `Main` is no better match for `main` than `rain`. The compiler will compile your `Main` function, but when the linker is ready to build the executable file, it will complain about the missing `main` function and refuse to link the program. Of course, the message "missing `main` function" should give you a clue where to look for the error.

If you get an error message that seems to indicate that the compiler is on the wrong track, it is a good idea to check for spelling and capitalization. In C++, most names use only lowercase letters. If you misspell the name of a symbol (for example `out` instead of `cout`), the compiler will complain about an "undefined symbol" (or, on some systems, an "undeclared identifier" or "not declared in this scope"). This error message is usually a good clue that you made a spelling error.

1.7 Problem Solving: Algorithm Design

You will soon learn how to program calculations and decision making in C++. But before we look at the mechanics of implementing computations in the next chapter, let's consider how you can describe the steps that are necessary for finding the solution to a problem.

1.7.1 The Algorithm Concept

© mammamaart/iStockphoto.

Finding the perfect partner is not a problem that a computer can solve.

You may have run across advertisements that encourage you to pay for a computerized service that matches you up with a love partner. Think how this might work. You fill out a form and send it in. Others do the same. The data are processed by a computer program. Is it reasonable to assume that the computer can perform the task of finding the best match for you? Suppose your younger brother, not the computer, had all the forms on his desk. What instructions could you give him? You can't say, "Find the best-looking person who likes inline skating and browsing the Internet". There is no objective standard for good looks, and your brother's opinion (or that of a computer program analyzing the photos of prospective partners) will likely be different from yours. If you can't give written instructions for someone to solve the problem, there is no way the computer can magically find the right solution. The computer can only do what you tell it to do. It just does it faster, without getting bored or exhausted.

For that reason, a computerized match-making service cannot guarantee to find the optimal match for you. Instead, you may be presented with a set of potential partners who share common interests with you. That is a task that a computer program can solve.

In order for a computer program to provide an answer to a problem that computes an answer, it must follow a sequence of steps that is

- Unambiguous
- Executable
- Terminating

An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.

The step sequence is *unambiguous* when there are precise instructions for what to do at each step and where to go next. There is no room for guesswork or personal opinion. A step is *executable* when it can be carried out in practice. For example, a computer can list all people that share your hobbies, but it can't predict who will be your life-long partner. Finally, a sequence of steps is *terminating* if it will eventually come to an end. A program that keeps working without delivering an answer is clearly not useful.

A sequence of steps that is unambiguous, executable, and terminating is called an **algorithm**. Although there is no algorithm for finding a partner, many problems do have algorithms for solving them. The next section gives an example.

An algorithm is a recipe for finding a solution.

1.7.2 An Algorithm for Solving an Investment Problem

Consider the following investment problem:

> You put $10,000 into an account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?

Could you solve this problem by hand? Sure, you could. You figure out the balance as follows:

year	interest	balance
0		10000
1	10000.00 x 0.05 = 500.00	10000.00 + 500.00 = 10500.00
2	10500.00 x 0.05 = 525.00	10500.00 + 525.00 = 11025.00
3	11025.00 x 0.05 = 551.25	11025.00 + 551.25 = 11576.25
4	11576.25 x 0.05 = 578.81	11576.25 + 578.81 = 12155.06

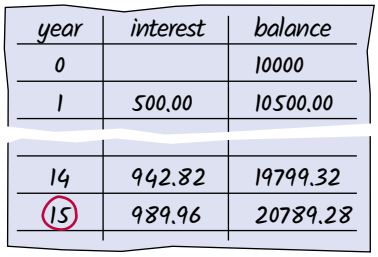
You keep going until the balance is at least $20,000. Then the last number in the year column is the answer.

Of course, carrying out this computation is intensely boring to you or your younger brother. But computers are very good at carrying out repetitive calculations quickly and flawlessly. What is important to the computer is a description of the steps for finding the solution. Each step must be clear and unambiguous, requiring no guesswork. Here is such a description:

Set year to 0, balance to 10000.

year	interest	balance
0		10000

```
While the balance is less than $20,000
   Add 1 to the year.
   Set the interest to balance x 0.05 (i.e., 5 percent interest).
   Add the interest to the balance.
```



year	interest	balance
0		10000
1	500.00	10500.00
14	942.82	19799.32
15	989.96	20789.28

```
   Report year as the answer.
```

These steps are not yet in a language that a computer can understand, but you will soon learn how to formulate them in C++. This informal description is called **pseudocode**. We examine the rules for writing pseudocode in the next section.

1.7.3 Pseudocode

Pseudocode is an informal description of a sequence of steps for solving a problem.

There are no strict requirements for pseudocode because it is read by human readers, not a computer program. Here are the kinds of pseudocode statements and how we will use them in this book:

- Use statements such as the following to describe how a value is set or changed:

  ```
  total cost = purchase price + operating cost
  Multiply the balance value by 1.05.
  Remove the first and last character from the word.
  ```

- Describe decisions and repetitions as follows:

  ```
  If total cost 1 < total cost 2
  While the balance is less than $20,000
  For each picture in the sequence
  ```

  Use indentation to indicate which statements should be selected or repeated:

  ```
  For each car
     operating cost = 10 x annual fuel cost
     total cost = purchase price + operating cost
  ```

  Here, the indentation indicates that both statements should be executed for each car.

- Indicate results with statements such as:

  ```
  Choose car2.
  Report year as the answer.
  ```

1.7.4 From Algorithms to Programs

In Section 1.7.2, we developed pseudocode for finding how long it takes to double an investment. Let's double-check that the pseudocode represents an algorithm; that is, that it is unambiguous, executable, and terminating.

Our pseudocode is unambiguous. It simply tells how to update values in each step. The pseudocode is executable because we use a fixed interest rate. Had we said to use the actual interest rate that will be charged in years to come, and not a fixed rate of 5 percent per year, the instructions would not have been executable. There is no way for anyone to know what the interest rate will be in the future. It requires a bit of thought to see that the steps are terminating: With every step, the balance goes up by at least $500, so eventually it must reach $20,000.

Therefore, we have found an algorithm to solve our investment problem, and we know we can find the solution by programming a computer. The existence of an algorithm is an essential prerequisite for programming a task. You need to first discover and describe an algorithm for the task before you start programming (see Figure 9). In the chapters that follow, you will learn how to express algorithms in the C++ language.

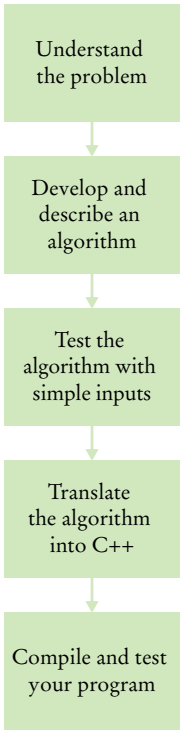


Figure 9 The Software Development Process

HOW TO 1.1

Describing an Algorithm with Pseudocode

This is the first of many "How To" sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in C++, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode—a sequence of precise steps formulated in English. To illustrate, we'll devise an algorithm for this problem:

Problem Statement You have the choice of buying one of two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of $4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?

© dlewis33/Getty Images.

Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- *purchase price1* and *fuel efficiency1*, the price and fuel efficiency (in mpg) of the first car
- *purchase price2* and *fuel efficiency2*, the price and fuel efficiency of the second car

We simply want to know which car is the better buy. That is the desired output.

Step 2 Break down the problem into smaller tasks.

For each car, we need to know the total cost of driving it. Let's do this computation separately for each car. Once we have the total cost for each car, we can decide which car is the better deal.

The total cost for each car is *purchase price + operating cost*.

We assume a constant usage and gas price for ten years, so the operating cost depends on the cost of driving the car for one year.

The operating cost is *10 x annual fuel cost*.

The annual fuel cost is *price per gallon x annual fuel consumed*.

The annual fuel consumed is *annual miles driven / fuel efficiency*. For example, if you drive the car for 15,000 miles and the fuel efficiency is 15 miles/gallon, the car consumes 1,000 gallons.

Step 3 Describe each subtask in pseudocode.

In your description, arrange the steps so that any intermediate values are computed before they are needed in other computations. For example, list the step

total cost = purchase price + operating cost

after you have computed *operating cost*.

Here is the algorithm for deciding which car to buy:

```
For each car, compute the total cost as follows:
   annual fuel consumed = annual miles driven / fuel efficiency
   annual fuel cost = price per gallon x annual fuel consumed
   operating cost = 10 x annual fuel cost
   total cost = purchase price + operating cost
If total cost of car1 < total cost of car2
   Choose car1.
Else
   Choose car2.
```

Step 4 Test your pseudocode by working a problem.

We will use these sample values:

Car 1: $25,000, 50 miles/gallon

Car 2: $20,000, 30 miles/gallon

Here is the calculation for the cost of the first car:

```
annual fuel consumed = annual miles driven / fuel efficiency = 15000 / 50 = 300
annual fuel cost = price per gallon x annual fuel consumed = 4 x 300 = 1200
operating cost = 10 x annual fuel cost = 10 x 1200 = 12000
total cost = purchase price + operating cost = 25000 + 12000 = 37000
```

Similarly, the total cost for the second car is $40,000. Therefore, the output of the algorithm is to choose car 1.

The following Worked Example demonstrates how to use the concepts in this chapter and the steps in the How To to solve another problem. In this case, you will see how to develop an algorithm for laying tile in an alternating pattern of colors. You should read the Worked Example to review what you have learned, and for help in tackling another problem.

In future chapters, Worked Examples are indicated by a brief description of the problem tackled in the example, plus a reminder to view it in your eText or download

it from the book's companion Web site at `wiley.com/go/bclo3`. You will find any code related to the Worked Example included with the book's companion code for the chapter. When you see the Worked Example description, go to the example and view the code to learn how the problem was solved.

WORKED EXAMPLE 1.1

Writing an Algorithm for Tiling a Floor

Problem Statement Your task is to tile a rectangular bathroom floor with alternating black and white tiles measuring 4 × 4 inches. The floor dimensions, measured in inches, are multiples of 4.

Step 1 Determine the inputs and outputs.

The inputs are the floor dimensions (length × width), measured in inches. The output is a tiled floor.

© rban/iStockphoto.

Step 2 Break down the problem into smaller tasks.

A natural subtask is to lay one row of tiles. If you can solve that task, then you can solve the problem by laying one row next to the other, starting from a wall, until you reach the opposite wall.

How do you lay a row? Start with a tile at one wall. If it is white, put a black one next to it. If it is black, put a white one next to it. Keep going until you reach the opposite wall. The row will contain *width* / 4 tiles.

Step 3 Describe each subtask in pseudocode.

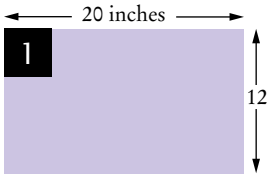
In the pseudocode, you want to be more precise about exactly where the tiles are placed.

```
Place a black tile in the northwest corner.
While the floor is not yet filled, repeat the following steps:
   Repeat this step width / 4 – 1 times:
      If the previously placed tile was white
         Pick a black tile.
      Else
         Pick a white tile.
      Place the picked tile east of the previously placed tile.
   Locate the tile at the beginning of the row that you just placed. If there is space to the south,
   place a tile of the opposite color below it.
```

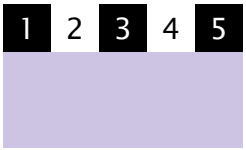
Step 4 Test your pseudocode by working a problem.

Suppose you want to tile an area measuring 20 × 12 inches.

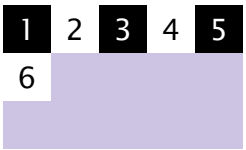
The first step is to place a black tile in the northwest corner.



Next, alternate four tiles until reaching the east wall. (*width / 4 − 1 = 20 / 4 − 1 = 4*)

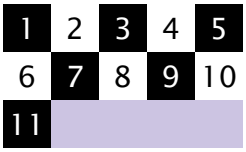


There is room to the south. Locate the tile at the beginning of the completed row. It is black. Place a white tile south of it.

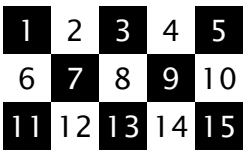


Complete the row.

There is still room to the south. Locate the tile at the beginning of the completed row. It is white. Place a black tile south of it.



Complete the row.



Now the entire floor is filled, and you are done.

CHAPTER SUMMARY

Define "computer program" and programming.

- Computers execute very basic instructions in rapid succession.
- A computer program is a sequence of instructions and decisions.
- Programming is the act of designing and implementing computer programs.

Describe the components of a computer.

- The central processing unit (CPU) performs program control and data processing.
- Storage devices include memory and secondary storage.

Describe the process of translating high-level languages to machine code.

- Computer programs are stored as machine instructions in a code that depends on the processor type.
- C++ is a general-purpose language that is in widespread use for systems and embedded programming.
- High-level programming languages are independent of the processor.

Become familiar with your C++ programming environment.

- Set aside some time to become familiar with the programming environment that you will use for your class work.
- An editor is a program for entering and modifying text, such as a C++ program.
- C++ is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.
- The compiler translates C++ programs into machine code.
- The linker combines machine code with library code into an executable program.
- Develop a strategy for keeping backup copies of your work before disaster strikes.

Describe the building blocks of a simple program.

- Every C++ program contains a function called `main`.
- Use `cout` and the `<<` operator to display values on the screen.
- Enclose text strings in quotation marks.
- Use `+` to add two numbers and `*` to multiply two numbers.
- Send `endl` to `cout` to end a line of displayed output.
- End each statement with a semicolon.

Classify program errors as compile-time and run-time errors.

- A compile-time error is a violation of the programming language rules that is detected by the compiler.
- A run-time error causes a program to take an action that the programmer did not intend.
- The programmer is responsible for inspecting and testing the program to guard against run-time errors.

Write pseudocode for simple algorithms.

- An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.
- Pseudocode is an informal description of a sequence of steps for solving a problem.

REVIEW EXERCISES

■ **R1.1** Explain the difference between using a computer program and programming a computer.

■ **R1.2** Which parts of a computer can store program code? Which can store user data?

■ **R1.3** Which parts of a computer serve to give information to the user? Which parts take user input?

■■ **R1.4** A toaster is a single-function device, but a computer can be programmed to carry out different tasks. Is your cell phone a single-function device, or is it a programmable computer?

■■ **R1.5** Explain two benefits of using C++ over machine code.

■■ **R1.6** On your own computer or on your lab computer, find the exact location (folder or directory name) of

a. The sample file `hello.cpp` (after you saved it in your development environment).

b. The standard header file `<iostream>`.

■■ **R1.7** What does this program print?

```
#include <iostream>
using namespace std;
int main()
{
   cout << "6 * 7 = " << 6 * 7 << endl;
   return 0;
}
```

■■ **R1.8** What does this program print? Pay close attention to spaces.

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello" << "World" << endl;
   return 0;
}
```

■■ **R1.9** What does this program print?

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello" << endl << "World" << endl;
   return 0;
}
```

■■ **R1.10** Write three versions of the `hello.cpp` program that have different compile-time errors. Write a version that has a run-time error.

■ **R1.11** How do you discover compile-time errors? How do you discover run-time errors?

■■ **R1.12** Write an algorithm to settle the following question: A bank account starts out with $10,000. Interest is compounded monthly at 6 percent per year (0.5 percent per month). Every month, $500 is withdrawn to meet college expenses. After how many years is the account depleted?

■■■ **R1.13** Consider the question in Exercise R1.12. Suppose the numbers ($10,000, 6 percent, $500) were user selectable. Are there values for which the algorithm you developed would not terminate? If so, change the algorithm to make sure it always terminates.

■■■ **R1.14** In order to estimate the cost of painting a house, a painter needs to know the surface area of the exterior. Develop an algorithm for computing that value. Your inputs are the width, length, and height of the house, the number of windows and doors, and their dimensions. (Assume the windows and doors have a uniform size.)

■■ **R1.15** You want to decide whether you should drive your car to work or take the train. You know the one-way distance from your home to your place of work, and the fuel efficiency of your car (in miles per gallon). You also know the one-way price of a train ticket. You assume the cost of gas at $4 per gallon, and car maintenance at 5 cents per mile. Write an algorithm to decide which commute is cheaper.

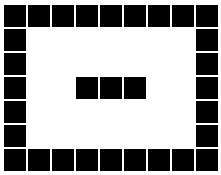
■■ **R1.16** Suppose you put your younger brother in charge of backing up your work. Write a set of detailed instructions for carrying out his task. Explain how often he should do it, and what files he needs to copy from which folder to which location. Explain how he should verify that the backup was carried out correctly.

■■■ **R1.17** The cafeteria offers a discount card for sale that entitles you, during a certain period, to a free meal whenever you have bought a given number of meals at the regular price. The exact details of the offer change from time to time. Describe an algorithm that lets you determine whether a particular offer is a good buy. What other inputs do you need?

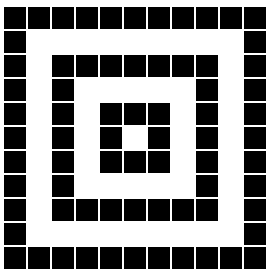
■ **R1.18** Write pseudocode for an algorithm that describes how to prepare Sunday breakfast in your household.

■■ **R1.19** The ancient Babylonians had an algorithm for determining the square root of a number a. Start with an initial guess of $a / 2$. Then find the average of your guess g and a / g. That's your next guess. Repeat until two consecutive guesses are close enough. Write pseudocode for this algorithm.

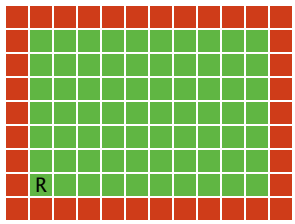
■■ **R1.20** Write an algorithm to create a tile pattern composed of black and white tiles, with a fringe of black tiles all around and two or three black tiles in the center, equally spaced from the boundary. The inputs to your algorithm are the total number of rows and columns in the pattern.



■■■ **R1.21** Write an algorithm to create a tile pattern composed of alternating black and white squares, like this:



■■■ **R1.22** Write an algorithm that allows a robot to mow a rectangular lawn, provided it has been placed in a corner, like this:

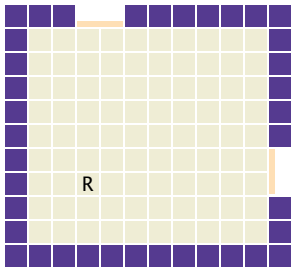


The robot (marked as R) can:

- Move forward by one unit.
- Turn left or right.
- Sense the color of the ground one unit in front of it.

■■■ **R1.23** Consider a robot that is placed in a room. The robot can:

- Move forward by one unit.
- Turn left or right.
- Sense what is in front of it: a wall, a window, or neither.



Write an algorithm that enables the robot, placed anywhere in the room, to count the number of windows. For example, in the room at right, the robot (marked as R) should find that it has two windows.

■■■ **R1.24** Consider a robot that has been placed in a maze. The right-hand rule tells you how to escape from a maze: Always have the right hand next to a wall, and eventually you will find an exit. The robot can:

- Move forward by one unit.
- Turn left or right.
- Sense what is in front of it: a wall, an exit, or neither.

© Skip ODonnell/iStockphoto.

Write an algorithm that lets the robot escape the maze. You may assume that there is an exit that is reachable by the right-hand rule. Your challenge is to deal with situations in which the path turns. The robot can't see turns. It can only see what is directly in front of it.

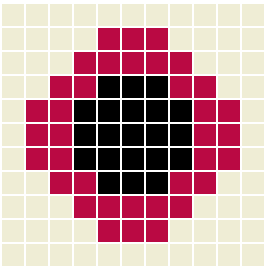
■■■ **Business R1.25** Suppose you received a loyalty promotion that lets you purchase one item, valued up to $100, from an online catalog. You want to make the best of the offer. You have a list of all items for sale, some of which are less than $100, some more. Write an algorithm to produce the item that is closest to $100. If there is more than one such item, list them all. Remember that a computer will inspect one item at a time—it can't just glance at a list and find the best one.

■■ **Engineering R1.26** A television manufacturer advertises that a television set has a certain size, measured diagonally. You wonder how the set will fit into your living room. Write an algorithm that yields the horizontal and vertical size of the television. Your inputs are the diagonal size and the aspect ratio (the ratio of width to height, usually 16 : 9 for television sets).

© Don Bayley/iStockPhoto.

■■■ **Engineering R1.27** Cameras today can correct "red eye" problems caused when the photo flash makes eyes look red. Write pseudocode for an algorithm that can detect red eyes. Your input is a pattern of colors, such as that below.

You are given the number of rows and columns. For any row or column number, you can query the color, which will be red, black, or something else. If you find that the center of the black pixels coincides with the center of the red pixels, you have found a red eye, and your output should be "yes". Otherwise, your output is "no".



▪▪ **Engineering R1.28** The San Francisco taxi commission set the following rates for 2017:

- First 1/5th of a mile: $3.50
- Each additional 1/5th of a mile or fraction thereof: $0.55
- Each minute of waiting or traffic delay: $0.55

The charge for "waiting or traffic delay" applies instead of the mileage charge for each minute in which the speed is slower than the break-even point. The break-even point is the speed at which 1/5th of a mile is traversed in one minute.

Develop an algorithm that yields the fare for traveling a given distance in a given amount of time, assuming that the taxi moves at a constant speed.

© Mark Massel/iStockphoto.

▪▪ **Engineering R1.29** Suppose you know how long it takes a car to accelerate from 0 to 60 miles per hour. Develop an algorithm for computing the time required to travel a given distance (for example 5 miles), assuming that the car is initially at rest, accelerates to a given speed (for example 25 miles per hour), and drives at that speed until the distance is covered. *Hint:* An object that starts at rest and accelerates at a constant rate a for t seconds travels a distance of $s = 1/2at^2$.

▪▪ **Engineering R1.30** Taxi meters measure the distance traveled by counting the number of revolutions of an axle. The meter is initialized by driving a known distance. A disreputable taxi driver tricks his customers into paying more per trip. He does this by slightly deflating his taxi's tires after the meter has been initialized, so that they have a smaller radius. This way, the tire will have a greater number of revolutions per trip, making each trip appear to be a greater number of miles.

Develop an algorithm for computing what the deflated tire radius must be if the taxi driver would like a short x-mile taxi ride to appear to be a longer y-mile taxi ride. Inputs are the proper tire radius R and the distances x and y.

PRACTICE EXERCISES

▪ **E1.1** Write a program that prints a greeting of your choice, perhaps in a language other than English.

▪▪ **E1.2** Write a program that prints the sum of the first ten positive integers, $1 + 2 + \cdots + 10$.

▪▪ **E1.3** Write a program that prints the product of the first ten positive integers, $1 \times 2 \times \cdots \times 10$. (Use * to indicate multiplication in C++.)

▪▪ **E1.4** Write a program that prints the balance of an account after the first, second, and third year. The account has an initial balance of $1,000 and earns 5 percent interest per year.

■ **E1.5** Write a program that displays your name inside a box on the screen, like this: Dave
Do your best to approximate lines with characters such as | - +.

■■ **E1.6** Write a program that prints your name in Morse code, like this:

```
.... .- .-. .-. -.--
```

Use a separate output statement for each letter.

■ **E1.7** Write a program that prints three items, such as the names of your three best friends or favorite movies, on three separate lines.

■ **E1.8** Write a program that prints a poem of your choice. If you don't have a favorite poem, search the Internet for "Emily Dickinson" or "e e cummings".

■ **Business E1.9** Write a program that prints a two-column list of your friends' birthdays. In the first column, print the names of your best friends; in the second, print their birthdays.

■ **Business E1.10** In the United States there is no federal sales tax, so every state may impose its own sales taxes. Look on the Internet for the sales tax charged in five U.S. states, then write a program that prints the tax rate for five states of your choice.

```
Sales Tax Rates
--------------
Alaska:      0%
Hawaii:      4%
. . .
```

■ **Business E1.11** To speak more than one language is a valuable skill in the labor market today. One of the basic skills is learning to greet people. Write a program that prints a two-column list with the greeting phrases shown in the table. In the first column, print the phrase in English, in the second column, print the phrase in a language of your choice. If you don't speak a language other than English, use an online translator or ask a friend.

List of Phrases to Translate
Good morning.
It is a pleasure to meet you.
Please call me tomorrow.
Have a nice day!

■■ **Engineering E1.12** Write a program that prints out the following English/metric conversions:

1 kilogram = 2.21 pounds
1 pound = 0.454 kilograms
1 foot = 0.305 meters
1 meter = 3.28 feet
1 mile = 1.61 kilometers
1 kilometer = 0.621 miles

PROGRAMMING PROJECTS

■ **P1.1** Write a program that prints the message, "Hello, my name is Hal!" Then, on a new line, the program should print the message "What would you like me to do?" Then it's the user's turn to type in an input. You haven't yet learned how to do it—just use the following lines of code:

```
string user_input;
getline(cin, user_input);
```

Finally, the program should ignore the user input and print the message "I am sorry, I cannot do that."

This program uses the `string` data type. To access this feature, you must place the line

```
#include <string>
```

before the `main` function.

Here is a typical program run. The user input is printed in color.

```
Hello, my name is Hal!
What would you like me to do?
Clean up my room
I am sorry, I cannot do that.
```

When running the program, remember to press the Enter key after typing the last word of the input line.

■■ **P1.2** Write a program that prints out a message "Hello, my name is Hal!" Then, on a new line, the program should print the message "What is your name?" As in Exercise P1.1, just use the following lines of code:

```
string user_name;
getline(cin, user_name);
```

Finally, the program should print the message "Hello, *user name*. I am glad to meet you!" To print the user name, simply use

```
cout << user_name;
```

As in Exercise P1.1, you must place the line

```
#include <string>
```

before the `main` function.

Here is a typical program run. The user input is printed in color.

```
Hello, my name is Hal!
What is your name?
Dave
Hello, Dave. I am glad to meet you!
```

■■■ **P1.3** Write a program that prints your name in large letters, such as

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

■■ **P1.4** Write a program that prints a face similar to (but different from) the following:

```
 /////
 +-----+
(| o o |)
 |  ^  |
 | '-' |
 +-----+
```

■■ **P1.5** Write a program that prints a house that looks exactly like the following:

```
   /\
  /  \
 +----+
 | .-.|
 | | ||
 +-+-++
```

■■■ **P1.6** Write a program that prints an animal speaking a greeting, similar to (but different from) the following:

```
 /\_/\     -----
( ' ' )  / Hello \
(  -  ) <  Junior |
 | | |   \ Coder!/
(__|__)    -----
```

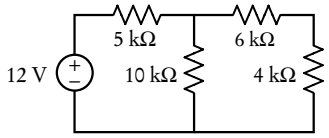
■■ **P1.7** Write a program that prints an imitation of a Piet Mondrian painting. (Search the Internet if you are not familiar with his paintings.) Use character sequences such as `@@@` or `:::` to indicate different colors, and use `-` and `|` to form lines.

■■ **P1.8** Write a program that prints the United States flag, using `*` and `=` characters.

■■ **Engineering P1.9** The atmospheres of the gas giant planets (Jupiter, Saturn, Uranus, and Neptune) are mostly comprised of hydrogen (H_2) followed by helium (He). The atmospheres of the terrestrial planets are mostly comprised of carbon dioxide (CO_2) followed by nitrogen (N_2) for Venus and Mars, and for Earth, mostly Nitrogen (N_2) followed by Oxygen (O_2). Write a program that outputs this information in a chart with four columns for the type of planet, the name of the planet, its primary atmospheric gas, and its secondary atmospheric gas.

© Courtesy NASA/JPL-Caltech.

■■■ **Engineering P1.10** Write a program that displays the following image, using characters such as `/ \ - | +` for the lines. Write Ω as "Ohm".



CHAPTER 2

FUNDAMENTAL DATA TYPES

© samxmeg/iStockphoto.

CHAPTER GOALS

- To be able to define and initialize variables and constants
- To understand the properties and limitations of integer and floating-point numbers
- To write arithmetic expressions and assignment statements in C++
- To appreciate the importance of comments and good code layout
- To create programs that read and process input, and display the results
- To process strings, using the standard C++ `string` type

CHAPTER CONTENTS

Numbers and character strings (such as the ones on this display board) are important data types in any C++ program. In this chapter, you will learn how to work with numbers and text, and how to write simple programs that perform useful tasks with them.

2.1 Variables

When your program carries out computations, you will want to store values so that you can use them later. In a C++ program, you use *variables* to store values. In this section, you will learn how to define and use variables.

To illustrate the use of variables, we will develop a program that solves the following problem. Soft drinks are sold in cans and bottles. A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle. Which should you buy? (Twelve fluid ounces equal approximately 0.355 liters.)

In our program, we will define variables for the number of cans per pack and for the volume of each can. Then we will compute the volume of a six-pack in liters and print out the answer.

cans: © blackred/iStockphoto. bottle: © travismanley/iStockphoto.

What contains more soda? A six-pack of 12-ounce cans or a two-liter bottle?

2.1.1 Variable Definitions

The following statement defines a variable named `cans_per_pack`:

```
int cans_per_pack = 6;
```

A variable is a storage location with a name.

A **variable** is a storage location in a computer program. Each variable has a name and holds a value.

A variable is similar to a parking space in a parking garage. The parking space has an identifier (such as "J053"), and it can hold a vehicle. A variable has a name (such as `cans_per_pack`), and it can hold a value (such as 6).

Javier Larrea/Age Fotostock.

Like a variable in a computer program, a parking space has an identifier and contents.

When defining a variable, you usually specify an initial value.

When defining a variable, you also specify the type of its values.

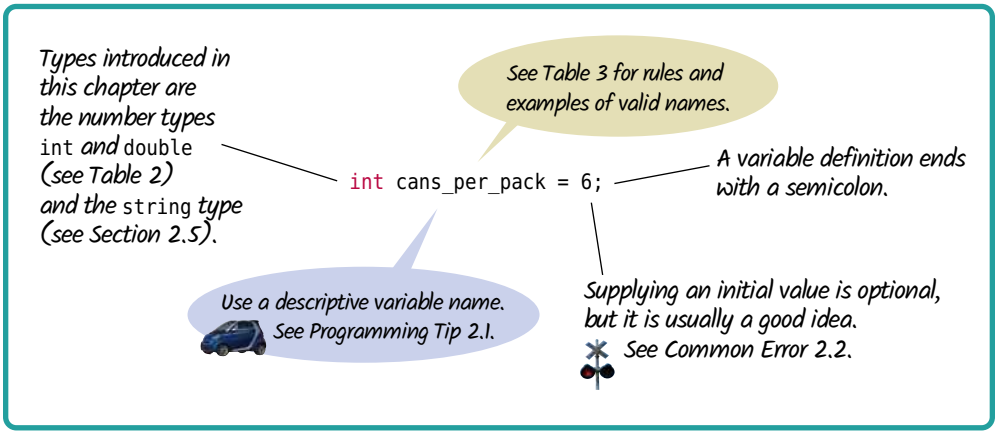
When defining a variable, you usually want to **initialize** it. That is, you specify the value that should be stored in the variable. Consider again this variable definition:

```
int cans_per_pack = 6;
```

The variable `cans_per_pack` is initialized with the value 6.

Like a parking space that is restricted to a certain type of vehicle (such as a compact car, motorcycle, or electric vehicle), a variable in C++ stores data of a specific **type**. C++ supports quite a few data types: numbers, text strings, files, dates, and many others. You must specify the type whenever you define a variable (see Syntax 2.1).

Syntax 2.1 Variable Definition



The `cans_per_pack` variable is an **integer**, a whole number without a fractional part. In C++, this type is called `int`. (See the next section for more information about number types in C++.)

Note that the type comes *before* the variable name:

```
int cans_per_pack = 6;
```

Each parking space is suitable for a particular type of vehicle, just as each variable holds a value of a particular type.

Table 1 shows variations of variable definitions.

Table 1 Variable Definitions in C++

Variable Name	Comment
`int cans = 6;`	Defines an integer variable and initializes it with 6.
`int total = cans + bottles;`	The initial value need not be a **constant**. (Of course, `cans` and `bottles` must have been previously defined.)
`int bottles = "10";`	**Error:** You cannot initialize a number with a string.
`int bottles;`	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2.
`int cans, bottles;`	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
`bottles = 1;`	**Caution:** The type is missing. This statement is not a definition but an assignment of a new value to an existing variable—see Section 2.1.4.

2.1.2 Number Types

Use the `int` type for numbers that cannot have a fractional part.

Use the `double` type for floating-point numbers.

In C++, there are several different types of numbers. You use the *integer* number type, called `int` in C++, to denote a whole number without a fractional part. For example, there must be an integer number of cans in any pack of cans—you cannot have a fraction of a can.

When a fractional part is required (such as in the number 0.355), we use **floating-point numbers**. The most commonly used type for floating-point numbers in C++ is called `double`. (If you want to know the reason, read Special Topic 2.1.) Here is the definition of a floating-point variable:

```
double can_volume = 0.355;
```

When a value such as `6` or `0.355` occurs in a C++ program, it is called a number **literal**. Table 2 shows how to write integer and floating-point literals in C++.

Table 2 Number Literals in C++

Number	Type	Comment
`6`	`int`	An integer has no fractional part.
`-6`	`int`	Integers can be negative.
`0`	`int`	Zero is an integer.
`0.5`	`double`	A number with a fractional part has type `double`.
`1.0`	`double`	An integer with a fractional part .0 has type `double`.
`1E6` or `1e6`	`double`	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type `double`.

Table 2 Number Literals in C++		
Number	Type	Comment
`2.96E-2`	`double`	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
`100,000`		**Error:** Do not use a comma as a decimal separator.
`3 1/2`		**Error:** Do not use fractions; use decimal notation: 3.5.

2.1.3 Variable Names

By convention, variable names should start with a lowercase letter.

When you define a variable, you should pick a name that explains its purpose. For example, it is better to use a descriptive name, such as `can_volume`, than a terse name, such as `cv`.

In C++, there are a few simple rules for variable names:

1. Variable names must start with a letter or the underscore (_) character, and the remaining characters must be letters, numbers, or underscores.
2. You cannot use other symbols such as `$` or `%`. Spaces are not permitted inside names either. You can use an underscore instead, as in `can_volume`.
3. Variable names are **case sensitive**, that is, `Can_volume` and `can_volume` are different names. For that reason, it is a good idea to use only lowercase letters in variable names. It is also a convention among many C++ programmers that variable names should start with a lowercase letter.
4. You cannot use **reserved words** such as `double` or `return` as names; these words are reserved exclusively for their special C++ meanings. (See Appendix A.)

Table 3 shows examples of legal and illegal variable names in C++.

Table 3 Variable Names in C++	
Variable Name	Comment
`can_volume1`	Variable names consist of letters, numbers, and the underscore character.
`x`	In mathematics, you use short variable names such as x or y. This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1).
`Can_volume`	**Caution:** Variable names are case sensitive. This variable name is different from `can_volume`.
`6pack`	**Error:** Variable names cannot start with a number.
`can volume`	**Error:** Variable names cannot contain spaces.
`double`	**Error:** You cannot use a reserved word as a variable name.
`ltr/fl.oz`	**Error:** You cannot use symbols such as . or /

2.1.4 The Assignment Statement

An assignment statement stores a new value in a variable, replacing the previously stored value.

You use the **assignment statement** to place a new value into a variable. Here is an example:

```
cans_per_pack = 8;
```

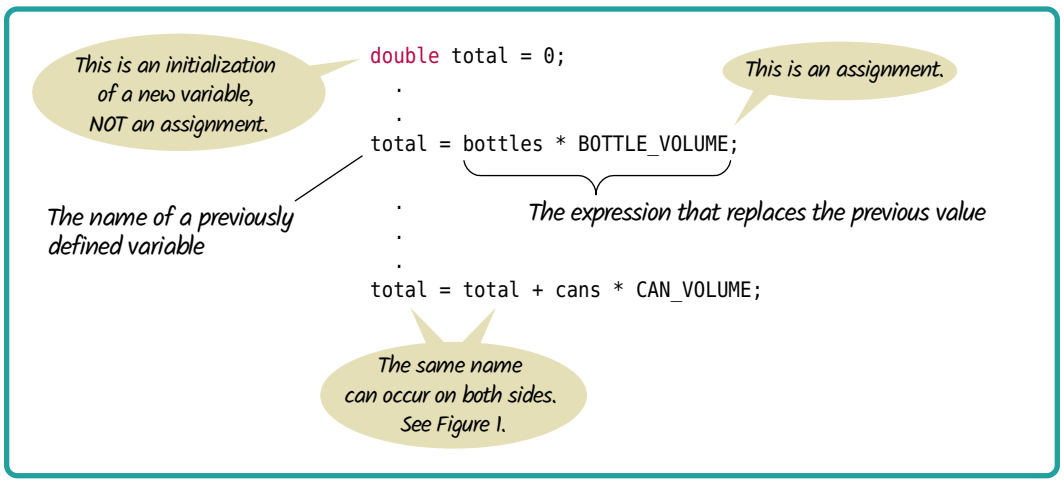
The left-hand side of an assignment statement consists of a variable. The right-hand side is an expression that has a value. That value is stored in the variable, overwriting its previous contents.

There is an important difference between a variable definition and an assignment statement:

```
int cans_per_pack = 6; // Variable definition
. . .
cans_per_pack = 8; // Assignment statement
```

The first statement is the *definition* of `cans_per_pack`. It is an instruction to create a new variable of type `int`, to give it the name `cans_per_pack`, and to initialize it with `6`. The second statement is an *assignment statement:* an instruction to replace the contents of the *existing* variable `cans_per_pack` with another value.

Syntax 2.2 Assignment



The assignment operator = does *not* denote mathematical equality.

The = sign doesn't mean that the left-hand side is *equal* to the right-hand side. The expression on the right is evaluated, and its value is placed into the variable on the left.

Do not confuse this *assignment operation* with the = used in algebra to denote *equality*. The assignment operator is an instruction to do something, namely place a value into a variable. The mathematical equality states the fact that two values are equal.

For example, in C++, it is perfectly legal to write

```
total_volume = total_volume + 2;
```

It means to look up the value stored in the variable `total_volume`, add 2 to it, and place the result back into `total_volume`. (See Figure 1.) The net effect of executing this statement is to increment `total_volume` by 2. For example, if `total_volume` was 2.13 before execution of the statement, it is set to 4.13 afterwards. Of course, in mathematics it would make no sense to write that $x = x + 2$. No value can equal itself plus 2.

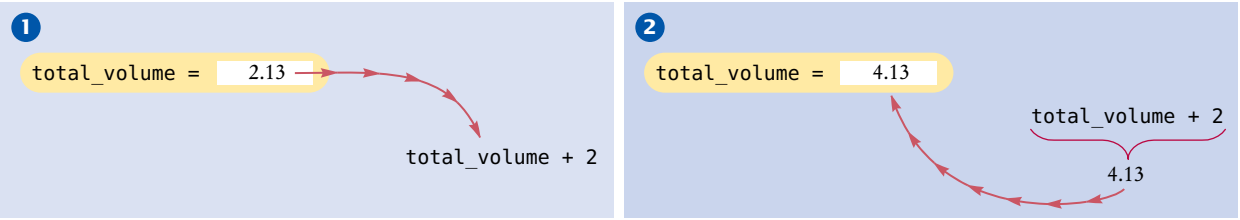


Figure 1 Executing the Assignment `total_volume = total_volume + 2`

2.1.5 Constants

You cannot change the value of a variable that is defined as `const`.

When a variable is defined with the reserved word `const`, its value can never change. Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
const double BOTTLE_VOLUME = 2;
```

It is good programming style to use named constants in your program to explain the meanings of numeric values. For example, compare the statements

```
double total_volume = bottles * 2;
```

and

```
double total_volume = bottles * BOTTLE_VOLUME;
```

A programmer reading the first statement may not understand the significance of the number 2. The second statement, with a named constant, makes the computation much clearer.

2.1.6 Comments

Use comments to add explanations for humans who read your code. The compiler ignores comments.

As your programs get more complex, you should add **comments**, explanations for human readers of your code. Here is an example:

```
const double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
```

This comment explains the significance of the value 0.355 to a human reader. The compiler does not process comments at all. It ignores everything from a `//` **delimiter** to the end of the line.

Just as a television commentator explains the news, you use comments in your program to explain its behavior.

You use the // syntax for single-line comments. If you have a comment that spans multiple lines, enclose it between /* and */ delimiters. The compiler ignores these delimiters and everything in between.

Here is a typical example, a long comment at the beginning of a program, to explain the program's purpose:

```
/*
   This program computes the volume (in liters) of a six-pack of soda cans
   and the total volume of a six-pack and a two-liter bottle.
*/
```

We are now ready to finish our program. The following program shows the use of variables, constants, and the assignment statement. The program displays the volume of a six-pack of cans and the total volume of the six-pack and a two-liter bottle. We use constants for the can and bottle volumes. The total_volume variable is initialized with the volume of the cans. Using an assignment statement, we add the bottle volume.

sec01/volume1.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /*
6     This program computes the volume (in liters) of a six-pack of soda
7     cans and the total volume of a six-pack and a two-liter bottle.
8  */
9  int main()
10 {
11    int cans_per_pack = 6;
12    const double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
13    double total_volume = cans_per_pack * CAN_VOLUME;
14
15    cout << "A six-pack of 12-ounce cans contains "
16       << total_volume << " liters." << endl;
17
18    const double BOTTLE_VOLUME = 2; // Two-liter bottle
19
20    total_volume = total_volume + BOTTLE_VOLUME;
21
22    cout << "A six-pack and a two-liter bottle contain "
23       << total_volume << " liters." << endl;
24
25    return 0;
26 }
```

Program Run

```
A six-pack of 12-ounce cans contains 2.13 liters.
A six-pack and a two-liter bottle contain 4.13 liters.
```

Common Error 2.1

Using Undefined Variables

You must define a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_per_ounce;
double liter_per_ounce = 0.0296;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `liter_per_ounce` will be defined in the next line, and it reports an error.

Common Error 2.2

Using Uninitialized Variables

If you define a variable but leave it uninitialized, then your program can act unpredictably. To understand why, consider what happens when you define a variable. Just enough space is set aside in memory to hold values of the type you specify. For example, with the definition

```
int bottles;
```

a block of memory big enough to hold integers is reserved. There is already *some* value in that memory. After all, you don't get freshly minted transistors—just an area of memory that has previously been used, filled with flotsam left over from prior computations. (In this regard, a variable differs from a parking space. A parking space can be empty, containing no vehicle. But a variable always holds some value.)

If you use the variable without initializing it, then that prior value will be used, yielding unpredictable results. For example, consider the program segment

```
int bottles; // Forgot to initialize
int bottle_volume = bottles * 2; // Result is unpredictable
```

There is no way of knowing what value will be computed. If you are unlucky, a plausible value will happen to appear when you run the program at home, and an entirely different result will occur when the program is graded.

Programming Tip 2.1

Choose Descriptive Variable Names

We could have saved ourselves a lot of typing by using shorter variable names, as in

```
double cv = 0.355;
```

Compare this definition with the one that we actually used, though. Which one is easier to read? There is no comparison. Just reading `can_volume` is a lot less trouble than reading `cv` and then *figuring out* it must mean "can volume".

In practical programming, this is particularly important when programs are written by more than one person. It may be obvious to *you* that `cv` stands for can volume and not current velocity, but will it be obvious to the person who needs to update your code years later? For that matter, will you remember yourself what `cv` means when you look at the code three months from now?

Programming Tip 2.2

Do Not Use Magic Numbers

A **magic number** is a numeric constant that appears in your code without explanation. For example,

```
total_volume = bottles * 2;
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```
const double BOTTLE_VOLUME = 2;
total_volume = bottles * BOTTLE_VOLUME;
```

© FinnBrandt/iStockphoto.

There is another reason for using named constants. Suppose circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it means a bottle volume, or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone.

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
const int DAYS_PER_YEAR = 365;
```

Special Topic 2.1

Numeric Types in C++

In addition to the `int` and `double` types, C++ has several other numeric types.

C++ has two floating-point types. The `float` type uses half the storage of the `double` type that we use in this book, but it can only store 6–7 digits. Many years ago, when computers had far less memory than they have today, `float` was the standard type for floating-point computations, and programmers would indulge in the luxury of "double precision" only when they needed the additional digits. Today, the `float` type is rarely used.

By the way, these numbers are called "floating-point" because of their internal representation in the computer. Consider numbers 29600, 2.96, and 0.0296. They can be represented in a very similar way: namely, as a sequence of the significant digits—296—and an indication of the position of the decimal point. When the values are multiplied or divided by 10, only the position of the decimal point changes; it "floats". Computers use base 2, not base 10, but the principle is the same.

In addition to the `int` type, C++ has integer types `short`, `long`, and `long long`. For each integer type, there is an `unsigned` equivalent. For example, the `short` type typically has a range from –32,768 to 32,767, whereas `unsigned short` has a range from 0 to 65,535. These strange-looking limits are the result of the use of binary numbers in computers. A `short` value uses 16 binary digits, which can encode $2^{16} = 65{,}536$ values. Keep in mind that the ranges for integer types are not standardized, and they differ among compilers. Table 4 contains typical values.

Table 4 Number Types

Type	Typical Range	Typical Size
`int`	–2,147,483,648 … 2,147,483,647 (about 2 billion)	4 bytes
`unsigned`	0 … 4,294,967,295	4 bytes

Table 4 Number Types

Type	Typical Range	Typical Size
`short`	–32,768 … 32,767	2 bytes
`unsigned short`	0 … 65,535	2 bytes
`long long`	–9,223,372,036,854,775,808 … 9,223,372,036,854,775,807	8 bytes
`long`	Depending on the compiler, the same as `int` or `long long`	4 or 8 bytes
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes

Special Topic 2.2

Numeric Ranges and Precisions

Because numbers are represented in the computer with a limited number of digits, they cannot represent arbitrary integer or floating-point numbers.

The `int` type has a *limited range:* On most platforms, it can represent numbers up to a little more than two billion. For many applications, this is not a problem, but you cannot use an `int` to represent the world population.

If a computation yields a value that is outside the `int` range, the result *overflows*. No error is displayed. Instead, the result is truncated to fit into an `int`, yielding a useless value. For example,

```
int one_billion = 1000000000;
cout << 3 * one_billion << endl;
```

displays –1294967296.

In situations such as this, you can switch to `double` values. However, read Common Error 2.6 for more information about a related issue: roundoff errors.

Special Topic 2.3

Defining Variables with `auto`

Instead of providing a type for a variable, you can use the reserved word `auto`. Then the type is automatically deduced from the type of the initial value. For example,

```
auto cans = 6; // This variable has type int
const auto CAN_VOLUME = 0.355; // This constant has type double
```

For simple types such as `int` or `double`, it is better to use the explicit type in the variable definition. The `auto` reserved word is useful to avoid complex types that can be automatically determined.

2.2 Arithmetic

In the following sections, you will learn how to carry out arithmetic and mathematical calculations in C++.

2.2.1 Arithmetic Operators

© hocus-focus/iStockphoto.

C++ supports the same four basic arithmetic operations as a calculator—addition, subtraction, multiplication, and division—but it uses different symbols for multiplication and division.

You must write `a * b` to denote multiplication. Unlike in mathematics, you cannot write `a b`, a · `b`, or `a` × `b`. Similarly, division is always indicated with `/`, never ÷ or a fraction bar.

For example, $\frac{a + b}{2}$ becomes `(a + b) / 2`.

Use * for multiplication and / for division.

Parentheses are used just as in algebra: to indicate in which order the subexpressions should be computed. For example, in the expression `(a + b) / 2`, the sum `a + b` is computed first, and then the sum is divided by 2. In contrast, in the expression

```
a + b / 2
```

only `b` is divided by 2, and then the sum of `a` and `b / 2` is formed. Just as in regular algebraic notation, multiplication and division have a *higher precedence* than addition and subtraction. For example, in the expression `a + b / 2`, the `/` is carried out first, even though the `+` operation occurs further to the left. If both arguments of an arithmetic operation are integers, the result is an integer. If one or both arguments are floating-point numbers, the result is a floating-point number. For example, `4 * 0.5` is `2.0`.

2.2.2 Increment and Decrement

The ++ operator adds 1 to a variable; the -- operator subtracts 1.

Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for it, namely

```
counter++;
counter--;
```

The ++ increment operator gave the C++ programming language its name. C++ is the incremental improvement of the C language.

2.2.3 Integer Division and Remainder

If both arguments of / are integers, the remainder is discarded.

Division works as you would expect, as long as at least one of the numbers involved is a floating-point number. That is, `7.0 / 4.0`, `7 / 4.0`, and `7.0 / 4` all yield 1.75. However, if *both* numbers are integers, then the result of the division is always an integer, with the remainder discarded. That is,

```
7 / 4
```

evaluates to 1 because 7 divided by 4 is 1 with a remainder of 3 (which is discarded). This can be a source of subtle programming errors—see Common Error 2.3.

The % operator computes the remainder of an integer division.

If you are interested in the remainder only, use the % operator:

```
7 % 4
```

is 3, the remainder of the **integer division** of 7 by 4. The % symbol has no analog in algebra. It was chosen because it looks similar to /, and the remainder operation is related to division. The operator is called **modulus**. (Some people call it *modulo* or *mod*.) It has no relationship with the percent operation that you find on some calculators.

Here is a typical use for the integer / and % operations. Suppose you have an amount of pennies in a piggybank:

```
int pennies = 1729;
```

You want to determine the value in dollars and cents. You obtain the dollars through an integer division by 100.

```
int dollars = pennies / 100;  // Sets dollars to 17
```

The integer division discards the remainder. To obtain the remainder, use the % operator:

```
int cents = pennies % 100;  // Sets cents to 29
```

A common use of the % operator is to check whether a number is even or odd.

Another common use of the % operator is to check whether a number is even or odd. If a number n is even, then n % 2 is zero.

© Michael Flippo/iStockphoto.

Integer division and the % operator yield the dollar and cent values of a piggybank full of pennies.

Table 5 Integer Division and Remainder

Expression (where n = 1729)	Value	Comment
`n % 10`	9	`n % 10` is always the last digit of `n`.
`n / 10`	172	This is always `n` without the last digit.
`n % 100`	29	The last two digits of `n`.
`n / 10.0`	172.9	Because 10.0 is a floating-point number, the fractional part is not discarded.
`-n % 10`	-9	Because the first argument is negative, the remainder is also negative.
`n % 2`	1	`n % 2` is 0 if `n` is even, 1 or –1 if `n` is odd.

2.2.4 Converting Floating-Point Numbers to Integers

When a floating-point value is assigned to an integer variable, the fractional part is discarded:

```
double price = 2.55;
int dollars = price; // Sets dollars to 2
```

Assigning a floating-point variable to an integer drops the fractional part.

Discarding the fractional part is not always what you want. Often, you want to round to the *nearest* integer. To round a positive floating-point value to the nearest integer, add 0.5 and then convert to an integer:

```
int dollars = price + 0.5; // Rounds to the nearest integer
```

In our example, adding 0.5 turns all values above 2.5 into values above 3. In particular, 2.55 is turned into 3.05, which is then truncated to 3. (For a negative floating-point value, you subtract 0.5.)

Because truncation is a potential cause for errors, your compiler may issue a warning that assigning a floating-point value to an integer variable is unsafe. See Special Topic 2.4 on how to avoid this warning.

2.2.5 Powers and Roots

The C++ library defines many mathematical functions such as `sqrt` (square root) and `pow` (raising to a power).

In C++, there are no symbols for powers and roots. To compute them, you must call **functions**. To take the square root of a number, you use the `sqrt` function. For example, $\sqrt{x}$ is written as `sqrt(x)`. To compute x^n, you write `pow(x, n)`.

To use the `sqrt` and `pow` functions, you must place the line `#include <cmath>` at the top of your program file. The header file `<cmath>` is a standard C++ header that is available with all C++ systems, as is `<iostream>`.

As you can see, the effect of the `/`, `sqrt`, and `pow` operations is to flatten out mathematical terms. In algebra, you use fractions, exponents, and roots to arrange expressions in a compact two-dimensional form. In C++, you have to write all expressions in a linear arrangement. For example, the mathematical expression

$$b \times \left(1 + \frac{r}{100}\right)^n$$

becomes

```
b * pow(1 + r / 100, n)
```

Figure 2 shows how to analyze such an expression.

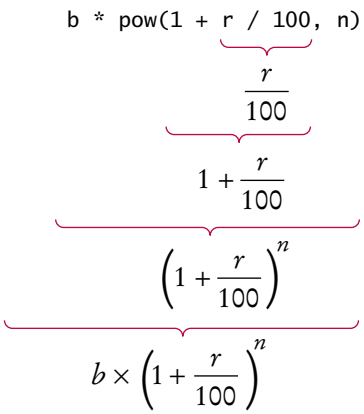


Figure 2 Analyzing an Expression

Table 6 Arithmetic Expressions

Mathematical Expression	C++ Expression	Comments
$\frac{x + y}{2}$	`(x + y) / 2`	The parentheses are required; `x + y / 2` computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	`x * y / 2`	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	`pow(1 + r / 100, n)`	Remember to add `#include <cmath>` to the top of your program.
$\sqrt{a^2 + b^2}$	`sqrt(a * a + b * b)`	`a * a` is simpler than `pow(a, 2)`.
$\frac{i + j + k}{3}$	`(i + j + k) / 3.0`	If i, j, and k are integers, using a denominator of 3.0 forces floating-point division.

Table 7 shows additional functions that are declared in the `<cmath>` header. Inputs and outputs are floating-point numbers.

Table 7 Other Mathematical Functions

Function	Description
`sin(x)`	sine of x (x in radians)
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`log(x)`	(natural log) $\ln(x), x > 0$
`log10(x)`	(decimal log) $\log_{10}(x), x > 0$
`abs(x)`	absolute value $\lvert x \rvert$

EXAMPLE CODE See sec02 of your companion code for a program that gives examples of working with numbers in C++.

Common Error 2.3

Unintended Integer Division

It is unfortunate that C++ uses the same symbol, namely `/`, for both integer and floating-point division. These are really quite different operations. It is a common error to use integer division by accident. Consider this segment that computes the average of three integers:

```
cout << "Please enter your last three test scores: ";
int s1;
int s2;
int s3;
cin >> s1 >> s2 >> s3;
```

```
double average = (s1 + s2 + s3) / 3; // Error
cout << "Your average score is " << average << endl;
```

What could be wrong with that? Of course, the average of `s1`, `s2`, and `s3` is

$$\frac{s1+s2+s3}{3}$$

Here, however, the `/` does not mean division in the mathematical sense. It denotes integer division because both `s1 + s2 + s3` and `3` are integers. For example, if the scores add up to 14, the average is computed to be 4, the result of the integer division of 14 by 3. That integer 4 is then moved into the floating-point variable `average`. The remedy is to make the numerator or denominator into a floating-point number:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

or

```
double average = (s1 + s2 + s3) / 3.0;
```

Common Error 2.4

Unbalanced Parentheses

Consider the expression

```
(-(b * b - 4 * a * c) / (2 * a)
```

© Croko/iStockphoto.

What is wrong with it? Count the parentheses. There are three `(` and two `)`. The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

```
-(b * b - (4 * a * c))) / (2 * a
```

This expression has three `(` and three `)`, but it still is not correct. In the middle of the expression,

```
-(b * b - (4 * a * c))) / (2 * a
                        ↑
```

there are only two `(` but three `)`, which is an error. In the middle of an expression, the count of `(` must be greater than or equal to the count of `)`, and at the end of the expression the two counts must be the same.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

```
-(b * b - (4 * a * c ) ) ) / (2 * a
 1        2           1 0 –1
```

and you would find the error.

Common Error 2.5

Forgetting Header Files

Every program that carries out input or output needs the `<iostream>` header. If you use mathematical functions such as `sqrt`, you need to include `<cmath>`. If you forget to include the

appropriate header file, the compiler will not know symbols such as `cout` or `sqrt`. If the compiler complains about an undefined function or symbol, check your header files.

Sometimes you may not know which header file to include. Suppose you want to compute the absolute value of an integer using the `abs` function. As it happens, this version of `abs` is not defined in the `<cmath>` header but in `<cstdlib>`. How can you find the correct header file? You need to locate the documentation of the `abs` function, preferably using the online help of your development environment or a reference site on the Internet such as `http://cplusplus.com` (see Figure 3). The documentation includes a short description of the function and the name of the header file that you must include.

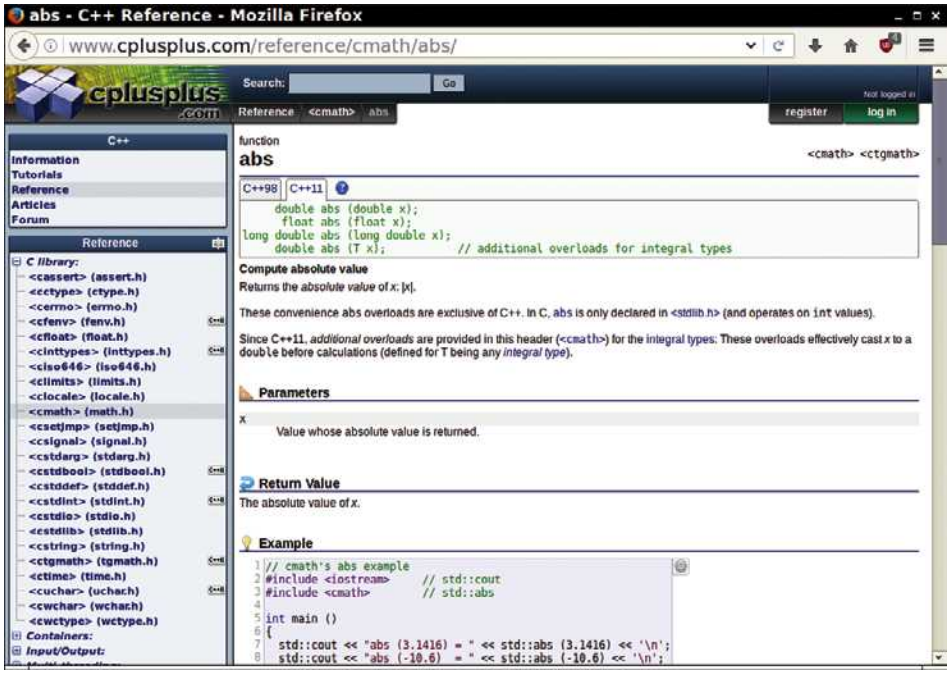


Figure 3 Online Documentation

Common Error 2.6

Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that phenomenon yourself with manual calculations. If you calculate 1/3 to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, not in decimal. You still get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect. Here is an example.

```
#include <iostream>

using namespace std;

int main()
{
   double price = 4.35;
   int cents = 100 * price; // Should be 100 * 4.35 = 435
   cout << cents << endl; // Prints 434!
```

```
   return 0;
}
```

Of course, one hundred times 4.35 is 435, but the program prints 434.

Most computers represent numbers in the binary system. In the binary system, there is no exact representation for 4.35, just as there is no exact representation for 1/3 in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435. When a floating-point value is converted to an integer, the entire fractional part, which is almost 1, is thrown away, and the integer 434 is stored in `cents`. The remedy is to add 0.5 in order to round to the nearest integer:

```
int cents = 100 * price + 0.5;
```

Programming Tip 2.3

Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a);
```

Simply put spaces around all operators `+ - * / % =`. However, don't put a space after a *unary* minus: a – used to negate a single quantity, such as `-b`. That way, it can be easily distinguished from a *binary* minus, as in `a - b`.

It is customary not to put a space after a function name. That is, write `sqrt(x)` and not `sqrt (x)`.

Special Topic 2.4

Casts

Occasionally, you need to store a value into a variable of a different type. Whenever there is the risk of *information loss*, the compiler issues a warning. For example, if you store a `double` value into an `int` variable, you can lose information in two ways:

- The fractional part is lost.
- The magnitude may be too large.

For example,

```
int n = 1.0E100; // NO
```

is not likely to work, because 10^{100} is larger than the largest representable integer.

Nevertheless, sometimes you do want to convert a floating-point value into an integer value. If you are prepared to lose the fractional part and you know that this particular floating-point number is not larger than the largest possible integer, then you can turn off the warning by using a **cast**. A cast is a conversion from one type (such as `double`) to another type (such as `int`) that is not safe in general, but that you know to be safe in a particular circumstance. You express a cast in C++ as follows:

```
int cents = static_cast<int>(100 * price + 0.5);
```

Special Topic 2.5

Combining Assignment and Arithmetic

In C++, you can combine arithmetic and assignment. For example, the instruction

```
total += cans * CAN_VOLUME;
```

is a shortcut for

```
total = total + cans * CAN_VOLUME;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

Many programmers find this a convenient shortcut. If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book, though.

Computing & Society 2.1 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

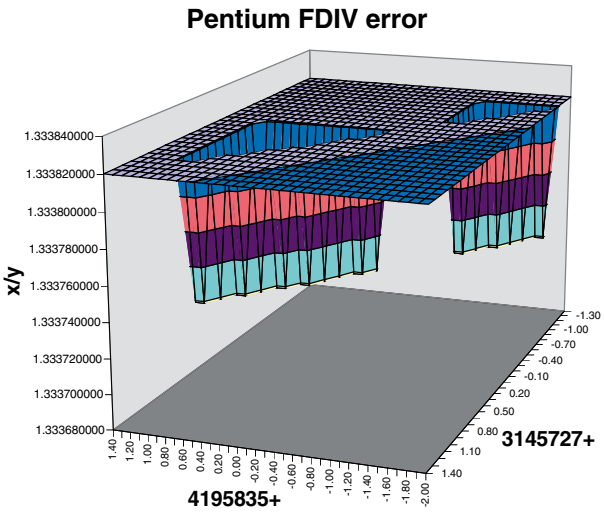
In the summer of 1994, Dr. Thomas Nicely of Lynchburg College, Virginia, ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal roundoff behavior of floating-point calculations were standardized by the Institute for Electrical and Electronic Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

$$4{,}195{,}835 - \left((4{,}195{,}835/3{,}145{,}727) \times 3{,}145{,}727\right)$$

is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the processor's floating-point multiplication algorithm. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of $700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return. Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.



Courtesy of Larry Hoyle, Institute for Policy & Social Research, University of Kansas.

This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.

2.3 Input and Output

In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

2.3.1 Input

In this section, you will see how to place user input into a variable. Consider for example the `volume1.cpp` program in Section 2.1.6. Rather than assuming that the price for the two-liter bottle and the six-pack of cans are identical, we can ask the program user for the prices.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**.

```
cout << "Please enter the number of bottles: "; // Display prompt
```

Do not add an `endl` after the prompt. You want the input to appear after the colon, not on the following line.

Use the >> operator to read a value and place it in a variable.

Next, the program issues a command to read the input. The `cin` object reads input from the console window. You use the >> operator (sometimes called the *extraction* operator) to place an input value into a variable, like this:

```
int bottles;
cin >> bottles;
```

When the program executes the input statement, it waits for the user to provide input. The user also needs to press the Enter key so that the program accepts the input. After the user supplies the input, the number is placed into the `bottles` variable, and the program continues.

Note that in this code segment, there was no need to initialize the `bottles` variable because it is being filled by the very next statement. As a rule of thumb, you should initialize a variable when you declare it *unless* it is filled in an input statement that follows immediately.

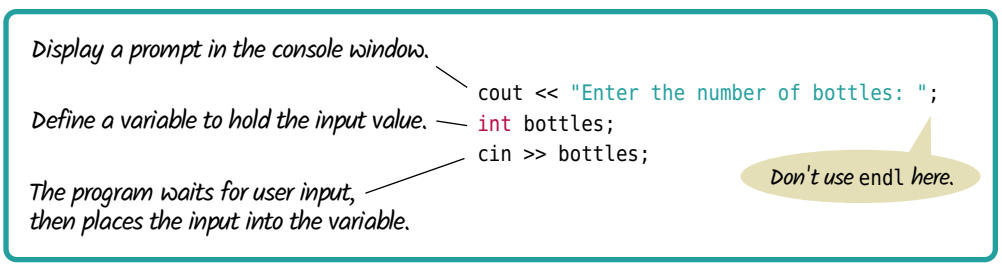
You can read more than one value in a single input statement:

```
cout << "Please enter the number of bottles and cans: ";
cin >> bottles >> cans;
```

The user can supply both inputs on the same line:

```
Please enter the number of bottles and cans: 2 6
```

Syntax 2.3 Input Statement



Alternatively, the user can press the Enter key after each input:

```
Please enter the number of bottles and cans: 2
6
```

2.3.2 Formatted Output

When you print the result of a computation, you often want some control over its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

```
Price per ounce: 0.04
```

instead of

```
Price per ounce: 0.0409722
```

You use manipulators to specify how values should be formatted.

The following command instructs `cout` to use two digits after the decimal point for all floating-point numbers:

```
cout << fixed << setprecision(2);
```

This command does not produce any output; it just manipulates `cout` so that it will change the output format. The values `fixed` and `setprecision` are called *manipulators*. We will discuss manipulators in detail in Chapter 8. For now, just remember to include the statement given above whenever you want currency values displayed neatly.

To use manipulators, you must include the `<iomanip>` header in your program:

```
#include <iomanip>
```

You can combine the manipulators and the values to be displayed into a single statement:

```
cout << fixed << setprecision(2)
   << "Price per ounce: "
   << price_per_ounce << endl;
```

There is another manipulator that is sometimes handy. When you display several rows of data, you usually want the columns to line up.

You use the `setw` manipulator to set the *width* of the next output field. The width is the total number of characters used for showing the value, including digits, the

You use manipulators to line up your output in neat columns.

decimal point, and spaces. Controlling the width is important when you want columns of numbers to line up.

For example, if you want a number to be printed in a column that is eight characters wide, you use

```
cout << setw(8) << price_per_ounce;
```

This command prints the value `price_per_ounce` in a field of width 8, for example

␣ ␣ ␣ ␣ 0 . 0 4

(where each ␣ represents a space).

There is a notable difference between the `setprecision` and `setw` manipulators. Once you set the precision, that value is used for all floating-point numbers in that statement. But the width affects only the *next* value. Subsequent values are formatted without added spaces.

Our next example program will prompt for the price of a six-pack and the volume of each can, then print out the price per ounce. The program puts to work what you just learned about reading input and formatting output.

sec03/volume2.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   // Read price per pack

   cout << "Please enter the price for a six-pack: ";
   double pack_price;
   cin >> pack_price;

   // Read can volume

   cout << "Please enter the volume for each can (in ounces): ";
   double can_volume;
   cin >> can_volume;

   // Compute pack volume

   const double CANS_PER_PACK = 6;
   double pack_volume = can_volume * CANS_PER_PACK;

   // Compute and print price per ounce

   double price_per_ounce = pack_price / pack_volume;

   cout << fixed << setprecision(2);
   cout << "Price per ounce: " << price_per_ounce << endl;

   return 0;
}
```

Program Run

```
Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce: 0.04
```

Table 8 Formatting Output

Output Statement	Output	Comment
`cout << 12.345678;`	`12.3457`	By default, a number is printed with 6 significant digits.
`cout << fixed << setprecision(2) << 12.3;`	`12.30`	Use the `fixed` and `setprecision` manipulators to control the number of digits after the decimal point.
`cout << ":" << setw(6) << 12;`	`:  12`	Four spaces are printed before the number, for a total width of 6 characters.
`cout << ":" << setw(2) << 123;`	`:123`	If the width not sufficient, it is ignored.
`cout << setw(6) << ":" << 12;`	`  :12`	The width only refers to the next item. Here, the : is preceded by five spaces.

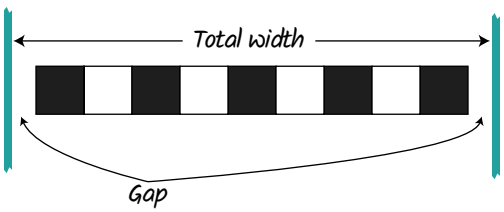
2.4 Problem Solving: First Do It By Hand

A very important step for developing an algorithm is to first carry out the computations *by hand*. If you can't compute a solution yourself, it's unlikely that you'll be able to write a program that automates the computation.

To illustrate the use of hand calculations, consider the following problem.

A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.

Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.



Pick concrete values for a typical situation to use in a hand calculation.

To make the problem more concrete, let's assume the following dimensions:

- Total width: 100 inches
- Tile width: 5 inches

The obvious solution would be to fill the space with 20 tiles, but that would not work—the last tile would be white.

Instead, look at the problem this way: The first tile must always be black, and then we add some number of white/black pairs:

The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs is 95 / 10 = 9.5. However, we need to discard the fractional part because we can't have fractions of tile pairs.

Therefore, we will use 9 tile pairs or 18 tiles, together with the initial black tile. Altogether, we require 19 tiles.

The tiles span 19 × 5 = 95 inches, leaving a total gap of 100 – 19 × 5 = 5 inches. The gap should be evenly distributed at both ends. At each end, the gap is (100 – 19 × 5) / 2 = 2.5 inches.

This computation gives us enough information to devise an algorithm with arbitrary values for the total width and tile width.

number of pairs = integer part of (total width - tile width) / (2 x tile width)
number of tiles = 1 + 2 x number of pairs
gap at each end = (total width - number of tiles x tile width) / 2

As you can see, doing a hand calculation gives enough insight into the problem that it becomes easy to develop an algorithm.

EXAMPLE CODE See sec04 of your companion code for a program that implements this algorithm.

WORKED EXAMPLE 2.1

Computing Travel Time

Learn how to develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain. See your E-Text or visit `wiley.com/go/bclo3`.

Courtesy of NASA.

HOW TO 2.1

Carrying out Computations

Many programming problems require that you carry out arithmetic computations. This How To shows you how to turn a problem statement into pseudocode and, ultimately, a C++ program.

Problem Statement Suppose you are asked to write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. We will assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters. Your task is to compute how many coins of each type to return.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

Step 2 Work out examples by hand.

Let's assume that a customer purchased an item that cost $2.25 and inserted a $5 bill. The customer is due $2.75, or two dollar coins and three quarters.

That is easy for you to see, but how can a C++ program come to the same conclusion? The computation is simpler if you work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the change due in coins? First, compute the change due in pennies:

change due = 100 x bill value - item price in pennies

To get the dollars, divide by 100 and discard the remainder:

dollar coins = change due / 100 (without remainder)

The remaining amount due can be computed in two ways. If you are familiar with the modulus operator, you can simply compute

change due = change due % 100

Alternatively, subtract the penny value of the dollar coins from the change due:

change due = change due - 100 x dollar coins

To get the quarters due, divide by 25:

quarters = change due / 25

Step 4 Define the variables and constants that you need, and specify their types.

Here, we have five variables:

- `bill_value`
- `item_price`
- `change_due`
- `dollar_coins`
- `quarters`

A vending machine takes bills and gives change in coins.

Jupiter Images/Getty Images.

Should we introduce constants to explain 100 and 25 as PENNIES_PER_DOLLAR and PENNIES_PER_QUARTER? Doing so will make it easier to convert the program to international markets, so we will take this step.

It is very important that change_due and PENNIES_PER_DOLLAR are of type int because the computation of dollar_coins uses integer division. Similarly, the other variables are integers.

Step 5 Turn the pseudocode into C++ statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as powers or integer division) in C++.

```
change_due = PENNIES_PER_DOLLAR * bill_value - item_price;
dollar_coins = change_due / PENNIES_PER_DOLLAR;
change_due = change_due % PENNIES_PER_DOLLAR;
quarters = change_due / PENNIES_PER_QUARTER;
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
cout << "Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ";
cin >> bill_value;
cout << "Enter item price in pennies: ";
cin >> item_price;
```

When the computation is finished, we display the result. For extra credit, we use the setw manipulator to make sure that the output lines up neatly.

```
cout << "Dollar coins: " << setw(6) << dollar_coins << endl
     << "Quarters:     " << setw(6) << quarters << endl;
```

Step 7 Include the required headers and provide a main function.

We need the <iostream> header for all input and output. Because we use the setw manipulator, we also require <iomanip>. This program does not use any special mathematical functions. Therefore, we do not include the <cmath> header.

In the main function, you need to define constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Define the constants at the beginning of the function, and define each variable just before it is needed.

Here is the complete program:

how_to_1/vending.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   const int PENNIES_PER_DOLLAR = 100;
   const int PENNIES_PER_QUARTER = 25;

   cout << "Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ";
   int bill_value;
   cin >> bill_value;
   cout << "Enter item price in pennies: ";
   int item_price;
   cin >> item_price;

   int change_due = PENNIES_PER_DOLLAR * bill_value - item_price;
   int dollar_coins = change_due / PENNIES_PER_DOLLAR;
```

```
20    change_due = change_due % PENNIES_PER_DOLLAR;
21    int quarters = change_due / PENNIES_PER_QUARTER;
22
23    cout << "Dollar coins: " << setw(6) << dollar_coins << endl
24       << "Quarters:     " << setw(6) << quarters << endl;
25
26    return 0;
27 }
```

Program Run

```
Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:      2
Quarters:          3
```

WORKED EXAMPLE 2.2

Computing the Cost of Stamps

Learn how to use arithmetic functions to simulate a stamp vending machine. See your E-Text or visit wiley.com/go.bclo3.

2.5 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string "Harry" is a sequence of five characters.

© essxboy/iStockphoto.

2.5.1 The string Type

You can define variables that hold strings.

```
string name = "Harry";
```

The string type is a part of the C++ standard. To use it, simply include the header file, <string>:

```
#include <string>
```

We distinguish between string *variables* (such as the variable name defined above) and string *literals* (character sequences enclosed in quotes, such as "Harry"). The string stored in a string variable can change. A string literal denotes a particular string, just as a number literal (such as 2) denotes a particular number.

Unlike number variables, string variables are guaranteed to be initialized even if you do not supply an initial value. By default, a string variable is set to an empty

string: a string containing no characters. An empty string literal is written as `""`. The definition

```
string response;
```

has the same effect as

```
string response = "";
```

2.5.2 Concatenation

Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.

Given two strings, such as `"Harry"` and `"Morgan"`, you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In C++, you use the `+` operator to concatenate two strings. For example,

```
string fname = "Harry";
string lname = "Morgan";
string name = fname + lname;
```

results in the string

```
"HarryMorgan"
```

What if you'd like the first and last name separated by a space? No problem:

```
string name = fname + " " + lname;
```

This statement concatenates three strings: `fname`, the string literal `" "`, and `lname`. The result is

```
"Harry Morgan"
```

2.5.3 String Input

You can read a string from the console:

```
cout << "Please enter your name: ";
string name;
cin >> name;
```

When a string is read with the `>>` operator, only one word is placed into the string variable. For example, suppose the user types

```
Harry Morgan
```

as the response to the prompt. This input consists of two words. After the call `cin >> name`, the string `"Harry"` is placed into the variable `name`. Use another input statement to read the second word.

2.5.4 String Functions

The `length` member function yields the number of characters in a string.

The number of characters in a string is called the *length* of the string. For example, the length of `"Harry"` is 5. You can compute the length of a string with the `length` function. Unlike the `sqrt` or `pow` function, the `length` function is invoked with the **dot notation**.

That is, you write the string whose length you want, then a period, then the name of the function, followed by parentheses:

```
int n = name.length();
```

A member function is invoked using the dot notation.

Many C++ functions require you to use this dot notation, and you must memorize (or look up) which do and which don't. These functions are called **member functions**. We say that the member function `length` is *invoked on* the variable `name`.

Once you have a string, you can extract substrings by using the `substr` member function. The member function call

```
s.substr(start, length)
```

returns a string that is made from the characters in the string `s`, starting at character `start`, and containing `length` characters. Here is an example:

```
string greeting = "Hello, World!";
string sub = greeting.substr(0, 5);
// sub is "Hello"
```

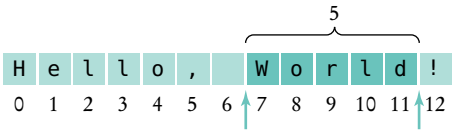
Use the `substr` member function to extract a substring of a string.

The `substr` operation makes a string that consists of five characters taken from the string `greeting`. Indeed, `"Hello"` is a string of length 5 that occurs inside `greeting`. A curious aspect of the `substr` operation is the starting position. Starting position `0` means "start at the beginning of the string". The first position in a string is labeled `0`, the second one 1, and so on. For example, here are the position numbers in the `greeting` string:

The position number of the last character (12) is always one less than the length of the string.

Let's figure out how to extract the substring `"World"`. Count characters starting at `0`, not 1. You find that `W`, the 8th character, has position number 7. The string you want is 5 characters long. Therefore, the appropriate substring command is

```
string w = greeting.substr(7, 5);
```



If you omit the length, you get all characters from the given position to the end of the string. For example,

```
greeting.substr(7)
```

is the string `"World!"` (including the exclamation point).

In a string that contains characters other than the English alphabet, digits, and punctuation marks, a character may take up more than a single `char` value. We will discuss this issue further in Chapter 8.

Here is a simple program that puts these concepts to work. The program asks for your name and that of your significant other. It then prints out your initials.

The operation `first.substr(0, 1)` makes a string consisting of one character, taken from the start of `first`. The program does the same for `second`. Then it concatenates the resulting one-character strings with the string literal `"&"` to get a string of length 3, the `initials` string. (See Figure 4.)

© Rich Legg/iStockphoto.

Initials are formed from the first letter of each name.

Figure 4 Building the `initials` String

sec05/initials.cpp

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   cout << "Enter your first name: ";
   string first;
   cin >> first;
   cout << "Enter your significant other's first name: ";
   string second;
   cin >> second;
   string initials = first.substr(0, 1)
      + "&" + second.substr(0, 1);
   cout << initials << endl;

   return 0;
}
```

Program Run

```
Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```

Table 9 String Operations

Statement	Result	Comment
`string str = "C";` `str = str + "++";`	`str` is set to `"C++"`	When applied to strings, `+` denotes concatenation.
`string str = "C" + "++";`	**Error**	**Error:** You cannot concatenate two string literals.

`cout << "Enter name: ";`
`cin >> name;`
(User input: Harry Morgan)
	`name` contains `"Harry"`	The `>>` operator places the next word into the `string` variable.

`cout << "Enter name: ";`
`cin >> name >> last_name;`
(User input: Harry Morgan)
	`name` contains `"Harry"`, `last_name` contains `"Morgan"`	Use multiple `>>` operators to read more than one word.

`string greeting = "H & S";` `int n = greeting.length();`	`n` is set to 5	Each space counts as one character.
`string str = "Sally";` `string str2 = str.substr(1, 3);`	`str2` is set to `"all"`	Extracts the substring of length 3 starting at position 1. (The initial position is 0.)
`string str = "Sally";` `string str2 = str.substr(1);`	`str2` is set to `"ally"`	If you omit the length, all characters from the position until the end are included.
`string a = str.substr(0, 1);`	`a` is set to the initial letter in `str`	Extracts the substring of length 1 starting at position 0.
`string b = str.substr(str.length() - 1);`	`b` is set to the last letter in `str`	The last letter has position `str.length() - 1`. We need not specify the length.

Computing & Society 2.2 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, ä, ö, ü, and a *double-s* character ß. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.

This poses a problem for computer users and designers. The American standard character encoding (called ASCII, for American Standard Code for Information Interchange) specifies 128 codes: 52 upper- and lowercase characters, 10 digits, 32 typographical symbols, and 34 control characters (such as space, newline, and 32 others for controlling printers and other devices). The umlaut and double-s are

© pvachier/iStockphoto.

The German Keyboard Layout

not among them. Some German data processing systems replace seldom-used ASCII characters with German letters: [\ ] { | } ~ are replaced with Ä Ö Ü ä ö ü ß. While most people can live without these characters, C++ programmers definitely cannot. Other encoding schemes take advantage of the fact that one byte can encode 256 different characters, of which only 128 are standardized by ASCII. Unfortunately, there are multiple incompatible standards for such encodings, resulting in a certain amount of aggravation among European computer users.

Many countries don't use the Roman script at all. Russian, Greek, Hebrew, Arabic, and Thai letters, to name just a few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has between 30 and 100 letters, and the countries using them have established encoding standards for them.

The situation is much more dramatic in languages that use the Chinese script: the Chinese dialects, Japanese, and Korean. The Chinese script is not alphabetic but *ideographic*. A character represents an idea or thing. Most words are made up of one, two, or three of these ideographic characters. (Over 50,000 ideographs are known, of which about 20,000 are in active use.) Therefore, two bytes are needed to encode them. China, Taiwan, Japan, and Korea have incompatible encoding standards for them. (Japanese and Korean writing uses a mixture of native syllabic and Chinese ideographic characters.)

The inconsistencies among character encodings have been a major nuisance for international electronic communication and for software manufacturers vying for a global market. Starting in 1988, a consortium of hardware and software manufacturers developed a uniform 21-**bit** encoding scheme called **Unicode** that is capable of encoding text in essentially all written languages of the world. About 100,000 characters have been given codes, including more than 70,000 Chinese, Japanese, and Korean ideographs. Even extinct languages, such as Egyptian hieroglyphs, have been included in Unicode.

© Joel Carillet/iStockphoto.

Hebrew, Arabic, and English

© Saipg/iStockphoto.

The Chinese Script

CHAPTER SUMMARY

Write variable definitions in C++.

- A variable is a storage location with a name.
- When defining a variable, you usually specify an initial value.
- When defining a variable, you also specify the type of its values.
- Use the `int` type for numbers that cannot have a fractional part.
- Use the `double` type for floating-point numbers.
- By convention, variable names should start with a lowercase letter.
- An assignment statement stores a new value in a variable, replacing the previously stored value.
- The assignment operator = does *not* denote mathematical equality.

- You cannot change the value of a variable that is defined as `const`.
- Use comments to add explanations for humans who read your code. The compiler ignores comments.

Use the arithmetic operations in C++.

- Use `*` for multiplication and `/` for division.
- The `++` operator adds 1 to a variable; the `--` operator subtracts 1.
- If both arguments of `/` are integers, the remainder is discarded.
- The `%` operator computes the remainder of an integer division.
- A common use of the `%` operator is to check whether a number is even or odd.
- Assigning a floating-point variable to an integer drops the fractional part.
- The C++ library defines many mathematical functions such as `sqrt` (square root) and `pow` (raising to a power).

Write programs that read user input and write formatted output.

- Use the `>>` operator to read a value and place it in a variable.
- You use manipulators to specify how values should be formatted.

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

Write programs that process strings.

- Strings are sequences of characters.
- Use the `+` operator to *concatenate* strings; that is, put them together to yield a longer string.
- The `length` member function yields the number of characters in a string.
- A member function is invoked using the dot notation.
- Use the `substr` member function to extract a substring of a string.

REVIEW EXERCISES

■ **R2.1** Write declarations for storing the following quantities. Choose between integers and floating-point numbers. Declare constants when appropriate.

a. The number of days per week

b. The number of days until the end of the semester

c. The number of centimeters in an inch

d. The height of the tallest person in your class, in centimeters

■ **R2.2** What is the value of `mystery` after this sequence of statements?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

■ **R2.3** What is wrong with the following sequence of statements?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

■■ **R2.4** Write the following mathematical expressions in C++.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2 \qquad \text{FV} = \text{PV} \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)} \qquad c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$$

■■ **R2.5** Write the following C++ expressions in mathematical notation.

a. `dm = m * (sqrt(1 + v / c) / sqrt(1 - v / c) - 1);`

b. `volume = PI * r * r * h;`

c. `volume = 4 * PI * pow(r, 3) / 3;`

d. `z = sqrt(x * x + y * y);`

■■ **R2.6** What are the values of the following expressions? In each line, assume that

```
double x = 2.5;
double y = -1.5;
int m = 18;
int n = 4;
```

a. `x + n * y - (x + n) * y`

b. `m / n + m % n`

c. `5 * x - n / 5`

d. `1 - (1 - (1 - (1 - (1 - n))))`

e. `sqrt(sqrt(n))`

■ **R2.7** What are the values of the following expressions? In each line, assume that

```
string s = "Hello";
string t = "World";
```

a. `s.length() + t.length()`

b. `s.substr(1, 2)`

c. `s.substr(s.length() / 2, 1)`

d. `s + t`

e. `t + s`

■ **R2.8** Find at least five *compile-time* errors in the following program.

```
#include iostream

int main();
{
   cout << "Please enter two numbers:"
   cin << x, y;
   cout << "The sum of << x << "and" << y
      << "is: " x + y << endl;
   return;
}
```

■■ **R2.9** Find at least four *run-time* errors in the following program.

```
#include <iostream>

using namespace std;

int main()
{
   int total;
   int x1;
   cout << "Please enter a number: ";
   cin >> x1;
   total = total + x1;
   cout << "Please enter another number: ";
   int x2;
   cin >> x2;
   total = total + x1;
   double average = total / 2;
   cout << "The average of the two numbers is "
      << average << "endl";
   return 0;
}
```

■ **R2.10** Explain the differences between `2`, `2.0`, `"2"`, and `"2.0"`.

■ **R2.11** Explain what each of the following program segments computes.

a.
```
int x = 2;
int y = x + x;
```

b.
```
string s = "2";
string t = s + s;
```

■■ **R2.12** Write pseudocode for a program that reads a word and then prints the first character, the last character, and the characters in the middle. For example, if the input is `Harry`, the program prints `H y arr`.

■■ **R2.13** Write pseudocode for a program that reads a name (such as `Harold James Morgan`) and then prints a monogram consisting of the initial letters of the first, middle, and last names (such as `HJM`).

■■■ **R2.14** Write pseudocode for a program that computes the first and last digits of a number. For example, if the input is `23456`, the program should print out `2` and `6`. *Hint:* `%`, `log10`.

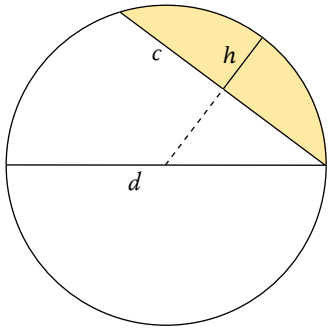
■ **R2.15** Modify the pseudocode for the program in How To 2.1 so that the program gives change in quarters, dimes, and nickels. You can assume that the price is a multiple of 5 cents. To develop your pseudocode, first work with a couple of specific values.

■■ **R2.16** A cocktail shaker is composed of three cone sections.

© Media Bakery.

The volume of a cone section with height h and top and bottom radius r_1 and r_2 is $V = \pi \dfrac{\left(r_1^2 + r_1 r_2 + r_2^2\right) h}{3}$. Compute the total volume by hand for one set of realistic values for the radii and heights. Then develop an algorithm that works for arbitrary dimensions.

■■■ **R2.17** You are cutting off a piece of pie like this, where c is the length of the straight part (called the chord length) and h is the height of the piece.



There is an approximate formula for the area: $A \approx \dfrac{2}{3}ch + \dfrac{h^3}{2c}$

However, h is not so easy to measure, whereas the diameter d of a pie is usually well-known. Calculate the area where the diameter of the pie is 12 inches and the chord length of the segment is 10 inches. Generalize to an algorithm that yields the area for any diameter and chord length.

■■ **R2.18** The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

```
Define a string called names containing "SunMonTueWedThuFriSat".
Compute the starting position as 3 x the day number.
Extract the substring of names at the starting position with length 3.
```

Check this pseudocode, using the day number 4. Draw a diagram of the string that is being computed, similar to Figure 4.

■■■ **R2.19** The following pseudocode describes how to swap two letters in a word.

```
We are given a string str and two positions i and j. (i comes before j)
Set first to the substring from the start of the string to the last position before i.
Set middle to the substring from positions i + 1 to j - 1.
Set last to the substring from position j + 1 to the end of the string.
Concatenate the following five strings: first, the string containing just the character at position j,
    middle, the string containing just the character at position i, and last.
```

Check this pseudocode, using the string `"Gateway"` and positions 2 and 4. Draw a diagram of the string that is being computed, similar to Figure 4.

■■ **R2.20** Run the following program, and explain the output you get.

```
#include <iostream>

using namespace std;

int main()
{
   int total;
   cout << "Please enter a number: ";
   double x1;
   cin >> x1;
   total = total + x1;
   cout << "total: " << total << endl;
   cout << "Please enter a number: ";
   double x2;
   cin >> x2;
   total = total + x2;
   cout << "total: " << total << endl;
   total = total / 2;
   cout << "total: " << total << endl;
   cout << "The average is " << total << endl;
   return 0;
}
```

Note the **trace messages** (in blue) that are inserted to show the current contents of the `total` variable. How do you fix the program? (The program has two separate errors.)

■■ **R2.21** How do you get the first character of a string? The last character? How do you remove the first character? The last character?

■■ **R2.22** For each of the following computations in C++, determine whether the result is exact, an overflow, or a roundoff error.

a. `2.0 - 1.1`

b. `1.0E6 * 1.0E6`

c. `65536 * 65536`

d. `1000000 * 1000000`

■■■ **R2.23** Write a program that prints the values

```
3 * 1000 * 1000 * 1000
3.0 * 1000 * 1000 * 1000
```

Explain the results.

■ **R2.24** This chapter contains a number of recommendations regarding variables and constants that make programs easier to read and maintain. Briefly summarize these recommendations.

PRACTICE EXERCISES

■ **E2.1** Write a program that displays the dimensions of a letter-size (8.5 × 11 inches) sheet of paper in millimeters. There are 25.4 millimeters per inch. Use constants and comments in your program.

■ **E2.2** Write a program that computes and displays the circumference of a letter-size (8.5 × 11 inches) sheet of paper and the length of its diagonal.

■ **E2.3** Write a program that reads a number and displays the square, cube, and fourth power. Use the `pow` function only for the fourth power.

■ **E2.4** Write a program that prompts the user for two integers and then prints

- The sum
- The difference
- The product
- The average

■■ **E2.5** Write a program that prompts the user for two integers and then prints

- The distance (absolute value of the difference)
- The maximum (the larger of the two)
- The minimum (the smaller of the two)

Hint: The `max` and `min` functions are defined in the `<algorithm>` header.

■■ **E2.6** Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.

■ **E2.7** Write a program that prompts the user for a radius and then prints

- The area and circumference of a circle with that radius
- The volume and surface area of a sphere with that radius

■■ **E2.8** Write a program that asks the user for the lengths of the sides of a rectangle and then prints

- The area and perimeter of the rectangle
- The length of the diagonal (use the Pythagorean theorem)

■ **E2.9** Improve the program discussed in How To 2.1 to allow input of quarters in addition to bills.

■■ **E2.10** Write a program that asks the user to input

- The number of gallons of gas in the tank
- The fuel efficiency in miles per gallon
- The price of gas per gallon

Then print the cost per 100 miles and how far the car can go with the gas in the tank.

■ **E2.11** *File names and extensions.* Write a program that prompts the user for the drive letter (`C`), the path (`\Windows\System`), the file name (`Readme`), and the extension (`txt`). Then print the complete file name `C:\Windows\System\Readme.txt`. (If you use UNIX or a Macintosh, skip the drive name and use `/` instead of `\` to separate directories.)

■■ **E2.12** Write a program that reads a number between 1,000 and 999,999 from the user and prints it *with a comma separating the thousands*. Here is a sample dialog; the user input is in color:

```
Please enter an integer between 1000 and 999999: 23456
23,456
```

■■■ **E2.13** Write a program that reads a number between 1,000 and 999,999 from the user, where the user enters a comma in the input. Then print the number without a comma. Here is a sample dialog; the user input is in color:

```
Please enter an integer between 1,000 and 999,999: 23,456
23456
```

Hint: Read the input as a string. Measure the length of the string. Suppose it contains n characters. Then extract substrings consisting of the first $n - 4$ characters and the last three characters.

■ **E2.14** *Printing a grid.* Write a program that prints the following grid to play tic-tac-toe.

```
+--+--+--+
|  |  |  |
+--+--+--+
|  |  |  |
+--+--+--+
|  |  |  |
+--+--+--+
```

Of course, you could simply write seven statements of the form

```
cout << "+--+--+--+";
```

You should do it the smart way, though. Define string variables to hold two kinds of patterns: a comb-shaped pattern

```
+--+--+--+
|  |  |  |
```

and the bottom line. Print the comb three times and the bottom line once.

■■ **E2.15** Write a program that reads an integer and breaks it into a sequence of individual digits. For example, the input 16384 is displayed as

```
1 6 3 8 4
```

You may assume that the input has no more than five digits and is not negative.

■■ **E2.16** Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in color.

```
Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes
```

Extra credit if you can deal with the case where the first time is later than the second:

```
Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes
```

■■■ **E2.17** *Writing large letters.* A large letter H can be produced like this:

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

It can be defined as a string constant like this:

```
const string LETTER_H =
   "*   *\n*   *\n*****\n*   *\n*   *\n";
```

(The `\n` character is explained in Special Topic 1.1.) Do the same for the letters `E`, `L`, and `O`. Then write the message

```
H
E
L
L
O
```

in large letters.

© José Luis Gutiérrez/iStockphoto.

■■ **E2.18** Write a program that transforms numbers `1`, `2`, `3`, …, `12` into the corresponding month names `January`, `February`, `March`, …, `December`. *Hint:* Make a very long string `"January February March ..."`, in which you add spaces such that each month name has *the same length*. Then use `substr` to extract the month you want.

PROGRAMMING PROJECTS

■■ **P2.1** Easter Sunday is the first Sunday after the first full moon of spring. To compute the date, you can use this algorithm, invented by the mathematician Carl Friedrich Gauss in 1800:

1. Let `y` be the year (such as 1800 or 2001).
2. Divide `y` by `19` and call the remainder `a`. Ignore the quotient.
3. Divide `y` by `100` to get a quotient `b` and a remainder `c`.
4. Divide `b` by `4` to get a quotient `d` and a remainder `e`.
5. Divide `8 * b + 13` by `25` to get a quotient `g`. Ignore the remainder.
6. Divide `19 * a + b - d - g + 15` by `30` to get a remainder `h`. Ignore the quotient.
7. Divide `c` by `4` to get a quotient `j` and a remainder `k`.
8. Divide `a + 11 * h` by `319` to get a quotient `m`. Ignore the remainder.
9. Divide `2 * e + 2 * j - k - h + m + 32` by `7` to get a remainder `r`. Ignore the quotient.
10. Divide `h - m + r + 90` by `25` to get a quotient `n`. Ignore the remainder.
11. Divide `h - m + r + n + 19` by `32` to get a remainder `p`. Ignore the quotient.

Then Easter falls on day `p` of month `n`. For example, if `y` is `2001`:

```
a = 6            g = 6           m = 0    n = 4
b = 20, c = 1    h = 18          r = 6    p = 15
d = 5, e = 0     j = 0, k = 1
```

Therefore, in 2001, Easter Sunday fell on April 15. Write a program that prompts the user for a year and prints out the month and day of Easter Sunday.

■■■ **P2.2** In this project, you will perform calculations with triangles. A triangle is defined by the x- and y-coordinates of its three corner points.

Your job is to compute the following properties of a given triangle:

- the lengths of all sides
- the angles at all corners
- the perimeter
- the area

Supply a program that prompts a user for the corner point coordinates and produces a nicely formatted table of the triangle properties.

■ Business P2.3 A video club wants to reward its best members with a discount based on the member's number of movie rentals and the number of new members referred by the member. The discount is in percent and is equal to the sum of the rentals and the referrals, but it cannot exceed 75 percent. (*Hint:* The `min` function in the `<algorithm>` header.) Write a program to calculate the value of the discount.

Here is a sample run:

```
Enter the number of movie rentals: 56
Enter the number of members referred to the video club: 3
The discount is equal to:     59.00 percent.
```

■■■ P2.4 Write a program that helps a person decide whether to buy a hybrid car. Your program's inputs should be:

- The cost of a new car
- The estimated miles driven per year
- The estimated gas price
- The estimated resale value after 5 years

Compute the total cost of owning the car for 5 years. (For simplicity, we will not take the cost of financing into account.) Obtain realistic prices for a new and used hybrid and a comparable car from the Web. Run your program twice, using today's gas price and 15,000 miles per year. Include pseudocode and the program runs with your assignment.

© asiseeit/iStockphoto.

■■ Business P2.5 The following pseudocode describes how a bookstore computes the price of an order from the total price and the number of the books that were ordered.

Read the total book price and the number of books.
Compute the tax (7.5% of the total book price).
Compute the shipping charge ($2 per book).
The price of the order is the sum of the total book price, the tax, and the shipping charge.

Print the price of the order. Translate this pseudocode into a C++ program.

■■ Business P2.6 The following pseudocode describes how to turn a string containing a ten-digit phone number (such as `"4155551212"`) into a more readable string with parentheses and dashes, like this: `"(415) 555-1212"`.

Take the substring consisting of the first three characters and surround it with "(" and ")". This is the area code.
Concatenate the area code, the substring consisting of the next three characters, a hyphen, and the substring consisting of the last four characters. This is the formatted number.

Translate this pseudocode into a C++ program that reads a telephone number into a string variable, computes the formatted number, and prints it.

■■ Business P2.7 The following pseudocode describes how to extract the dollars and cents from a price given as a floating-point value. For example, a price of 2.95 yields values 2 and 95 for the dollars and cents.

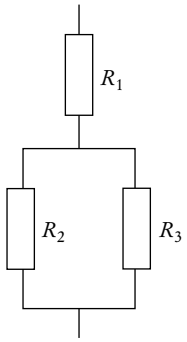
```
Assign the price to an integer variable dollars.
Multiply the difference price - dollars by 100 and add 0.5.
Assign the result to an integer variable cents.
```

Translate this pseudocode into a C++ program. Read a price and print the dollars and cents. Test your program with inputs 2.95 and 4.35.

■■ Business P2.8 *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Display the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return.

© Captainflash/iStockphoto.

■ Engineering P2.9 Consider the following circuit.



Write a program that reads the resistances of the three resistors and computes the total resistance, using the rules for series and parallel resistors that are derived from Ohm's law.

■■ Engineering P2.10 The dew point temperature T_d can be calculated (approximately) from the relative humidity RH and the actual temperature T by

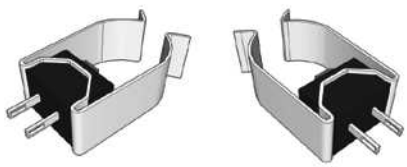
$$T_d = \frac{b \cdot f(T, RH)}{a - f(T, RH)}$$

$$f(T, RH) = \frac{a \cdot T}{b + T} + \ln(RH)$$

where $a = 17.27$ and $b = 237.7°$ C.

Write a program that reads the relative humidity (between 0 and 1) and the temperature (in degrees C) and prints the dew point value. Use the C++ `log` function to compute the natural logarithm.

■■■ Engineering P2.11 The pipe clip temperature sensors shown here are robust sensors that can be clipped directly onto copper pipes to measure the temperature of the liquids in the pipes.



Each sensor contains a device called a *thermistor*. Thermistors are semiconductor devices that exhibit a temperature-dependent resistance described by:

$$R = R_0 \, e^{\beta\left(\frac{1}{T}-\frac{1}{T_0}\right)}$$

where R is the resistance (in Ω) at the temperature T (in °K), and R_0 is the resistance (in Ω) at the temperature T_0 (in °K). β is a constant that depends on the material used to make the thermistor. Thermistors are specified by providing values for R_0, T_0, and β.

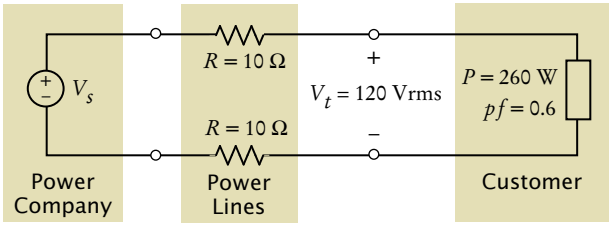
The thermistors used to make the pipe clip temperature sensors have $R_0 = 1075\ \Omega$ at $T_0 = 85$ °C, and $\beta = 3969$ °K. (Notice that β has units of °K. Recall that the temperature in °K is obtained by adding 273 to the temperature in °C.) The liquid temperature, in °C, is determined from the resistance R, in Ω, using

$$T = \frac{\beta T_0}{T_0 \ln\left(\frac{R}{R_0}\right) + \beta} - 273$$

Write a C++ program that prompts the user for the thermistor resistance R and prints a message giving the liquid temperature in °C.

■■■ Engineering P2.12 The circuit shown below illustrates some important aspects of the connection between a power company and one of its customers. The customer is represented by three parameters, V_t, P, and pf. V_t is the voltage accessed by plugging into a wall outlet. Customers depend on having a dependable value of V_t in order for their appliances to work properly. Accordingly, the power company regulates the value of V_t carefully. P describes the amount of power used by the customer and is the primary factor in determining the customer's electric bill. The power factor, pf, is less familiar. (The power factor is calculated as the cosine of an angle so that its value will always be between zero and one.) In this problem you will be asked to write a C++ program to investigate the significance of the power factor.

© TebNad/iStockphoto.

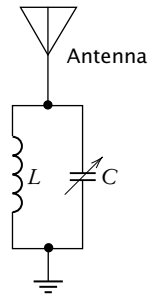


In the figure, the power lines are represented, somewhat simplistically, as resistances in Ohms. The power company is represented as an AC voltage source. The source voltage, V_s, required to provide the customer with power P at voltage V_t can be determined using the formula

$$V_s = \sqrt{\left(V_t + \frac{2RP}{V_t}\right)^2 + \left(\frac{2RP}{pf\,V_t}\right)^2 \left(1 - pf^2\right)}$$

(V_s has units of Vrms.) This formula indicates that the value of V_s depends on the value of *pf*. Write a C++ program that prompts the user for a power factor value and then prints a message giving the corresponding value of V_s, using the values for P, R, and V_t shown in the figure above.

■■■ Engineering P2.13 Consider the following tuning circuit connected to an antenna, where C is a variable capacitor whose capacitance ranges from C_{min} to C_{max}.



The tuning circuit selects the frequency $f = \dfrac{1}{2\pi\sqrt{LC}}$. To design this circuit for a given frequency, take $C = \sqrt{C_{\text{min}} C_{\text{max}}}$ and calculate the required inductance L from f and C. Now the circuit can be tuned to any frequency in the range

$$f_{\text{min}} = \frac{1}{2\pi\sqrt{LC_{\text{max}}}} \text{ to } f_{\text{max}} = \frac{1}{2\pi\sqrt{LC_{\text{min}}}}.$$

Write a C++ program to design a tuning circuit for a given frequency, using a variable capacitor with given values for C_{min} and C_{max}. (A typical input is $f = 16.7$ MHz, $C_{\text{min}} = 14$ pF, and $C_{\text{max}} = 365$ pF.) The program should read in f (in Hz), C_{min} and C_{max} (in F), and print the required inductance value and the range of frequencies to which the circuit can be tuned by varying the capacitance.

■ Engineering P2.14 According to the Coulomb force law, the electric force between two charged particles of charge Q_1 and Q_2 Coulombs, that are a distance r meters apart, is

$F = \dfrac{Q_1 Q_2}{4\pi\varepsilon r^2}$ Newtons, where $\varepsilon = 8.854 \times 10^{-12}$ Farads/meter. Write a program that calculates the force on a pair of charged particles, based on the user input of Q_1 Coulombs, Q_2 Coulombs, and r meters, and then computes and displays the electric force.

■■ Engineering P2.15 According to Newton's law of gravitation, the gravitational force between two masses M_1 and M_2 (in kilograms), that are a distance r meters apart, is $F = GM_1M_2 / r^2$, where $G = 6.67 \times 10^{-11} N(\text{m/kg})^2$. Write a program that calculates the force on a pair of masses, based on the user input of M_1, M_2, and r.

■■ **Engineering P2.16** The relationship between angles measured in units of degrees and units of radians is π radians = 180°. Write a program that converts angles in degrees to angles in radians. The input is a positive integer.

Note: Make sure that numerical output is not greater than 2π radians or 360°. For example, 370° = 0.174533 radian.

■■ **Engineering P2.17** The equation for the distance traveled by a freely falling object is $y = 1/2gt^2$, where t is time in seconds and g is the acceleration of gravity near the surface of the Earth, $g = 9.8\ m/s^2$. Notice that this equation is independent of the mass of the object.

Create a program that outputs a sentence expressing the distance traveled in the given time. For example, an input of 10 seconds should have the output sentence "After the first 10 seconds, the object has fallen 490 meters."

WORKED EXAMPLE 2.1

Computing Travel Time

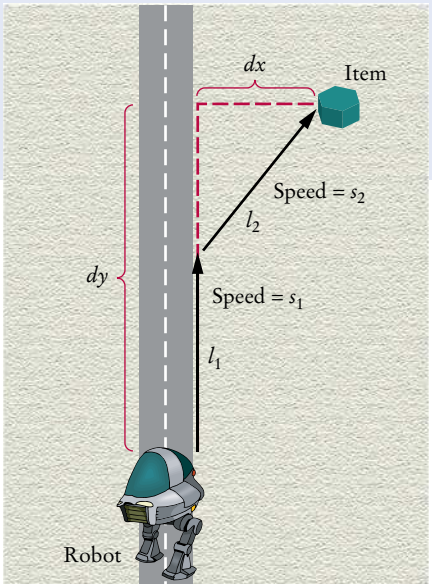
In this example, we develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain.

Courtesy of NASA.

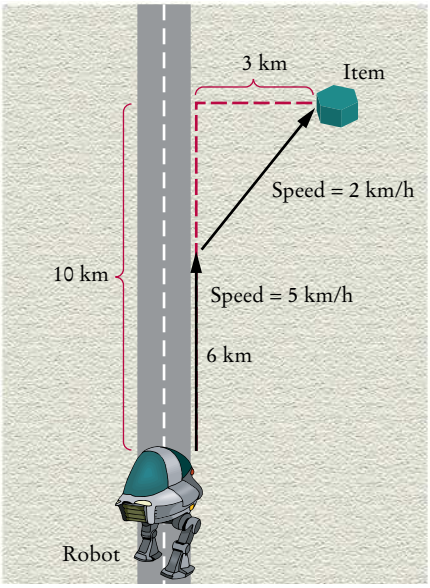
Problem Statement A robot needs to retrieve an item that is located in rocky terrain adjacent to a road. The robot can travel at a faster speed on the road than on the rocky terrain, so it will want to do so for a certain distance before moving on a straight line to the item.

Your task is to compute the total time taken by the robot to reach its goal, given the following inputs:

- The distance between the robot and the item in the x- and y-direction (dx and dy)
- The speed of the robot on the road and the rocky terrain (s_1 and s_2)
- The length l_1 of the first segment (on the road)

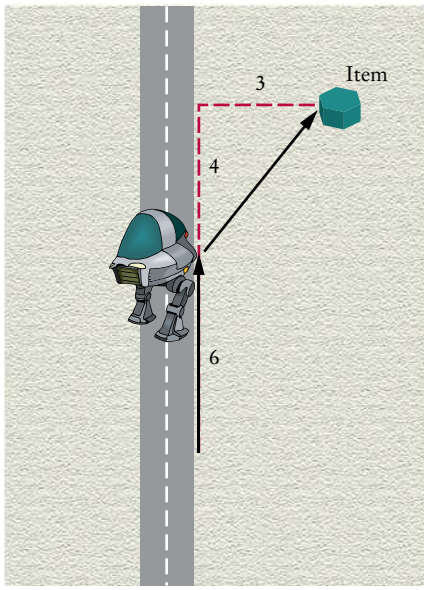


To make the problem more concrete, let's assume the following dimensions:



The total time is the time for traversing both segments. The time to traverse the first segment is simply the length of the segment divided by the speed: 6 km divided by 5 km/h, or 1.2 hours.

To compute the time for the second segment, we first need to know its length. It is the hypotenuse of a right triangle with side lengths 3 and 4.



Therefore, its length is $\sqrt{3^2 + 4^2} = 5$. At 2 km/h, it takes 2.5 hours to traverse it. That makes the total travel time 3.7 hours.

This computation gives us enough information to devise an algorithm for the total travel time with arbitrary parameters.

Time for segment 1 = l_1 / s_1
Length of segment 2 = square root of $dx^2 + (dy - l_1)^2$
Time for segment 2 = l_2 / s_2
Total time = time for segment 1 + time for segment 2

Translated into C++, the computations are

```
double segment1_time = segment1_length / segment1_speed;
double segment2_length = sqrt(pow(x_distance, 2)
   + pow(y_distance - distance_on_road, 2));
double segment2_time = segment2_length / segment2_speed;
double total_time = segment1_time + segment2_time;
```

Note that we use variable names that are longer and more descriptive than dx or s_1. When you do hand calculations, it is convenient to use the shorter names, but you should change them to descriptive names in your program.

WORKED EXAMPLE 2.2

Computing the Cost of Stamps

Problem Statement You are asked to simulate a postage stamp vending machine. A customer inserts dollar bills into the vending machine and then pushes a "purchase" button. The machine gives out as many first-class stamps as the customer paid for, and returns the change in penny (one-cent) stamps. A first-class stamp cost 47 cents at the time this book was written.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there is one input:

- The amount of money the customer inserts

There are two desired outputs:

- The number of first-class stamps the machine returns
- The number of penny stamps the machine returns

Step 2 Work out examples by hand.

Let's assume that a first-class stamp costs 47 cents and the customer inserts $1.00. That's enough for two stamps (94 cents) but not enough for three stamps ($1.41). Therefore, the machine returns two first-class stamps and 6 penny stamps.

Step 3 Write pseudocode for computing the answers.

Given an amount of money and the price of a first-class stamp, how can you compute how many first-class stamps can be purchased with the money? Clearly, the answer is related to the quotient

$$\frac{\text{amount of money}}{\text{price of first-class stamp}}$$

For example, suppose the customer paid $1.00. Use a pocket calculator to compute the quotient: $1.00 / $0.47 ≈ 2.13.

How do you get "2 stamps" out of 2.13? It's the quotient without the remainder. In C++, this is easy to compute if both arguments are integers. Therefore, let's switch our computation to pennies. Then we have

number of first-class stamps = 100 / 47 (integer division, without remainder)

What if the user inputs two dollars? Then the numerator becomes 200. What if the price of a stamp goes up? A more general equation is

number of first-class stamps = 100 x dollars / price of first-class stamps in cents

How about the penny stamps that are returned as change? Look at it this way. The change is the customer payment, reduced by the value of the first-class stamps purchased. In our example, the customer is due 6 cents worth of penny stamps, the difference between 100 and 2 × 47. Here is the general formula:

penny_stamps = 100 x dollars – number of first-class stamps x price of first-class stamp

Step 4 Declare the variables and constants that you need, and specify their types.

Here, we have three variables:

- `dollars`
- `first_class_stamps`
- `penny_stamps`

There is one constant, `FIRST_CLASS_STAMP_PRICE`.

The variable `dollars` and constant `FIRST_CLASS_STAMP_PRICE` must be of type `int` because the computation of `first_class_stamps` uses integer division. The remaining variables are also integers, counting the number of first-class and penny stamps. Thus, we have

```
const int FIRST_CLASS_STAMP_PRICE = 47; // Price in pennies
int dollars; // Filled through input statement

int first_class_stamps = 100 * dollars / FIRST_CLASS_STAMP_PRICE;
int penny_stamps = 100 * dollars - first_class_stamps * FIRST_CLASS_STAMP_PRICE;
```

Step 5 Turn the pseudocode into C++ statements.

Our computation depends on the number of dollars that the user provides. Translating the math into C++ yields the following statements:

```
first_class_stamps = 100 * dollars / FIRST_CLASS_STAMP_PRICE;
penny_stamps = 100 * dollars - first_class_stamps * FIRST_CLASS_STAMP_PRICE;
```

Step 6 Provide input and output.

```
cout << "Enter number of dollars: ";
cin >> dollars;
```

When the computation is finished, we display the result.

```
cout << "First class stamps: " << setw(6) << first_class_stamps << endl
     << "Penny stamps:       " << setw(6) << penny_stamps << endl;
```

Step 7 Include the required headers and provide a `main` function.

We need the `<iostream>` header for all input and output. Because we use the `setw` manipulator, we also require the `<iomanip>` header.

Here is the complete program:

worked_example_2/stamps.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   const int FIRST_CLASS_STAMP_PRICE = 47; // Price in pennies

   cout << "Enter number of dollars: ";
   int dollars;
   cin >> dollars;

   int first_class_stamps = 100 * dollars / FIRST_CLASS_STAMP_PRICE;
   int penny_stamps = 100 * dollars - first_class_stamps
      * FIRST_CLASS_STAMP_ PRICE;
   cout << "First class stamps: " << setw(6) << first_class_stamps << endl
        << "Penny stamps:       " << setw(6) << penny_stamps << endl;

   return 0;
}
```

Program Run

```
Enter number of dollars: 2
First class stamps:      4
Penny stamps:           12
```

CHAPTER 3

DECISIONS

© zennie/iStockphoto.

CHAPTER GOALS

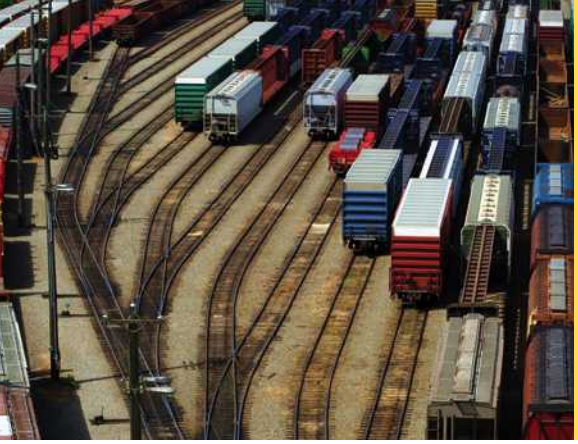
To be able to implement decisions using `if` statements

To learn how to compare integers, floating-point numbers, and strings

To understand the Boolean data type

To develop strategies for validating user input

CHAPTER CONTENTS



One of the essential features of computer programs is their ability to make decisions. Like a train that changes tracks depending on how the switches are set, a program can take different actions, depending on inputs and other circumstances.

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.

3.1 The `if` Statement

The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

The `if` statement is used to implement a decision. When a condition is fulfilled, one set of statements is executed. Otherwise, another set of statements is executed (see Syntax 3.1).

Here is an example using the `if` statement. In many countries, the number 13 is considered unlucky. Rather than offending superstitious tenants, building owners sometimes skip the thirteenth floor; floor 12 is immediately followed by floor 14. Of course, floor 13 is not usually left empty or, as some conspiracy theorists believe, filled with secret offices and research labs. It is simply called floor 14. The computer that controls the building elevators needs to compensate for this foible and adjust all floor numbers above 13.

Let's simulate this process in C++. We will ask the user to type in the desired floor number and then compute the actual floor. When the input is above 13, then we need to decrement the input to obtain the actual floor.

This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

An `if` statement is like a fork in the road. Depending upon a decision, different parts of the program are executed.

For example, if the user provides an input of 20, the program determines the actual floor as 19. Otherwise, we simply use the supplied floor number.

```
int actual_floor;

if (floor > 13)
{
   actual_floor = floor - 1;
}
else
{
   actual_floor = floor;
}
```

The flowchart in Figure 1 shows the branching behavior.

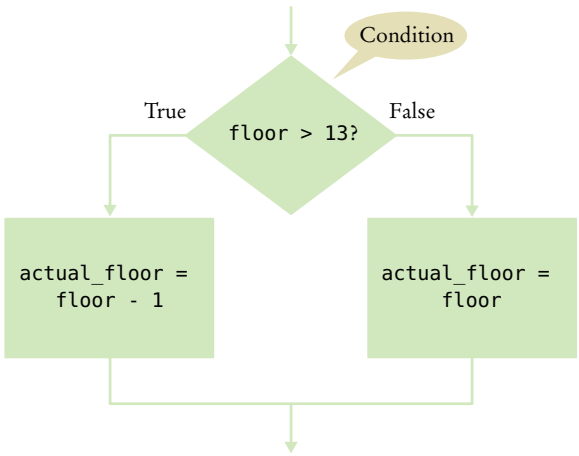
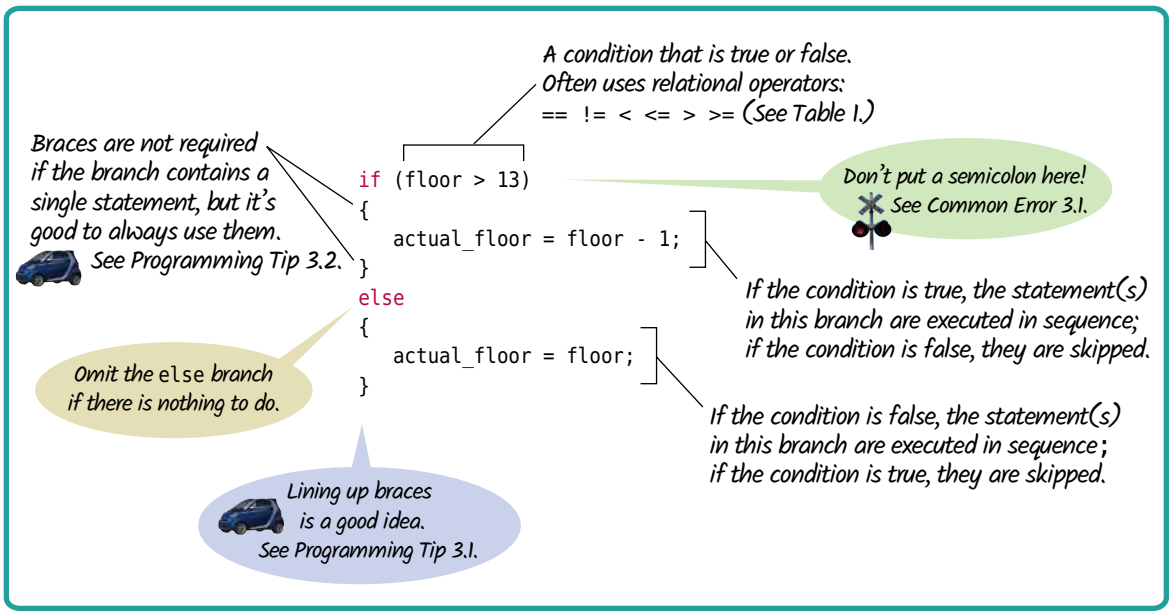


Figure 1 Flowchart for `if` Statement

Syntax 3.1 `if` Statement



In our example, each branch of the `if` statement contains a single statement. You can include as many statements in each branch as you like. Sometimes, it happens that there is nothing to do in the `else` branch of the statement. In that case, you can omit it entirely, such as in this example:

```
int actual_floor = floor;

if (floor > 13)
{
   actual_floor--;
} // No else needed
```

See Figure 2 for the flowchart.

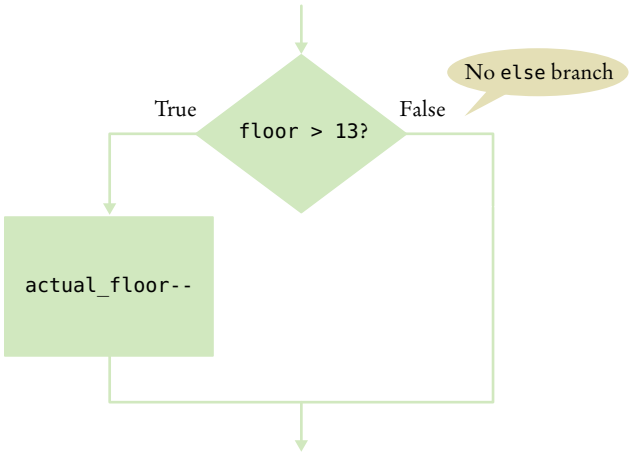


Figure 2 Flowchart for `if` Statement with No `else` Branch

The following program puts the `if` statement to work. This program asks for the desired floor and then prints out the actual floor.

sec01/elevator1.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     int floor;
8     cout << "Floor: ";
9     cin >> floor;
10    int actual_floor;
11    if (floor > 13)
12    {
13       actual_floor = floor - 1;
14    }
15    else
16    {
17       actual_floor = floor;
18    }
19
20    cout << "The elevator will travel to the actual floor "
21       << actual_floor << endl;
```

```
22
23     return 0;
24  }
```

Program Run

```
Floor: 20
The elevator will travel to the actual floor 19
```

Common Error 3.1

A Semicolon After the `if` Condition

The following code fragment has an unfortunate error:

```
if (floor > 13) ; // ERROR
{
   floor--;
}
```

There should be no semicolon after the `if` condition. The compiler interprets this statement as follows: If `floor` is greater than 13, execute the statement that is denoted by a single semicolon, that is, the do-nothing statement. The statement enclosed in braces is no longer a part of the `if` statement. It is always executed. Even if the value of `floor` is not above 13, it is decremented.

Placing a semicolon after the `else` reserved word is also wrong:

```
if (floor > 13)
{
   actual_floor = floor - 1;
}
else ;
{
   actual_floor = floor;
}
```

In this case, the do-nothing statement is executed if `floor > 13` is not fulfilled. This is the end of the `if` statement. The next statement, enclosed in braces, is executed in both cases; that is, `actual_floor` is always set to `floor`.

Programming Tip 3.1

Brace Layout

Programmers vary in how they align braces in their code. In this book, we follow the simple rule of making { and } line up.

```
if (floor > 13)
{
   floor--;
}
```

This style makes it easy to spot matching braces.

Some programmers put the opening brace on the same line as the `if`:

```
if (floor > 13) {
   floor--;
}
```

This style makes it harder to match the braces, but it saves a line of code, allowing you to view more code on the screen without scrolling. There are passionate advocates of both styles.

© Timothy Large/iStockphoto.

Properly lining up your code makes your programs easier to read.

It is important that you pick a layout style and stick with it consistently within a given programming project. Which style you choose may depend on your personal preference or a coding style guide that you need to follow.

Programming Tip 3.2

Always Use Braces

When a branch of an `if` statement consists of a single statement, you need not use braces. For example, the following is legal:

```
if (floor > 13)
   floor--;
```

However, it is a good idea to always include the braces:

```
if (floor > 13)
{
   floor--;
}
```

The braces makes your code easier to read, and you are less likely to make errors such as the one described in Common Error 3.1.

Programming Tip 3.3

Tabs

Block-structured code has the property that nested statements are indented by one or more levels:

```
int main()
{
|  int floor;
|  . . .
|  if (floor > 13)
|  {
|  |  floor--;
|  }  |
|  . . .
|  return 0;
}  |  |
0  1  2     Indentation level
```

How do you move the cursor from the leftmost column to the appropriate indentation level? A perfectly reasonable strategy is to hit the space bar a sufficient number of times. However, many programmers use the Tab key instead. A tab moves the cursor to the next indentation level.

While the Tab *key* is nice, some editors use *tab characters* for alignment, which is not so nice. Tab characters can lead to problems when you send your file to another person or a printer. There is no universal agreement on the width of a tab character, and some software will ignore tabs altogether. It is therefore best to save your files with spaces instead of tabs. Most editors have a setting to automatically convert all tabs to spaces. Look at the documentation of your development environment to find out how to activate this useful setting.

You use the Tab key to move the cursor to the next indentation level.

Programming Tip 3.4

Avoid Duplication in Branches

Look to see whether you *duplicate code* in each branch. If so, move it out of the `if` statement. Here is an example of such duplication:

```
if (floor > 13)
{
   actual_floor = floor - 1;
   cout << "Actual floor: " << actual_floor << endl;
}
else
{
   actual_floor = floor;
   cout << "Actual floor: " << actual_floor << endl;
}
```

The output statement is exactly the same in both branches. This is not an error—the program will run correctly. However, you can simplify the program by moving the duplicated statement, like this:

```
if (floor > 13)
{
   actual_floor = floor - 1;
}
else
{
   actual_floor = floor;
}
cout << "Actual floor: " << actual_floor << endl;
```

Removing duplication is particularly important when programs are maintained for a long time. When there are two sets of statements with the same effect, it can easily happen that a programmer modifies one set but not the other.

Special Topic 3.1

The Conditional Operator

C++ has a *conditional operator* of the form

condition `?` $value_1$ `:` $value_2$

The value of that expression is either $value_1$ if the test passes or $value_2$ if it fails. For example, we can compute the actual floor number as

```
actual_floor = floor > 13 ? floor - 1 : floor;
```

which is equivalent to

```
if (floor > 13)
{
   actual_floor =  floor - 1;
}
else
{
   actual_floor = floor;
}
```

You can use the conditional operator anywhere that a value is expected, for example:

```
cout << "Actual floor: " << (floor > 13 ? floor - 1 : floor);
```

We don't use the conditional operator in this book, but it is a convenient construct that you will find in many C++ programs.

3.2 Comparing Numbers and Strings

Relational operators (< <= > >= == !=) are used to compare numbers and strings.

Every `if` statement contains a *condition*. In many cases, the condition involves comparing two values. For example, in the previous examples we tested `floor > 13`. The comparison > is called a **relational operator**. C++ has six relational operators (see Table 1).

© arturbo/iStockphoto.

In C++, you use a relational operator to check whether one value is greater than another.

As you can see, only two C++ relational operators (> and <) look as you would expect from the mathematical notation. Computer keyboards do not have keys for ≥, ≤, or ≠, but the >=, <=, and != operators are easy to remember because they look similar. The == operator is initially confusing to most newcomers to C++. In C++, = already has a meaning, namely assignment.

The == operator denotes equality testing:

```
floor = 13; // Assign 13 to floor
if (floor == 13)  // Test whether floor equals 13
```

You must remember to use == inside tests and to use = outside tests. (See Common Error 3.2 for more information.)

Table 1 Relational Operators

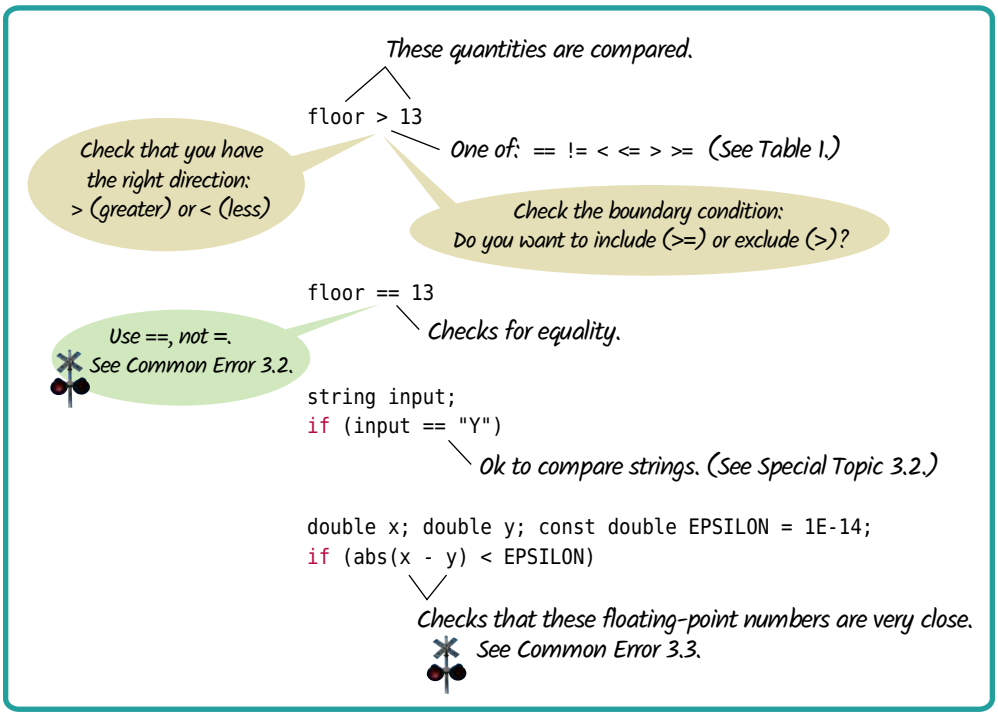
C++	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

You can compare strings as well:

```
if (input == "Quit") . . .
```

Use != to check whether two strings are different. In C++, letter case matters. For example, `"Quit"` and `"quit"` are not the same string.

Syntax 3.2 Comparisons



EXAMPLE CODE See sec02 of your companion code for a program that compares numbers and strings.

Table 2 summarizes how to use relational operators in C++.

Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
`3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Common Error 3.3.
`"10" > 5`	**Error**	You cannot compare a string to a number.

Common Error 3.2

Confusing = and ==

The rule for the correct usage of = and == is very simple: In tests, always use == and never use =. If it is so simple, why can't the compiler be helpful and flag any errors?

Actually, the C++ language allows the use of = inside tests. To understand this, we have to go back in time. The creators of C, the predecessor to C++, were very frugal. They did not want to have special values `true` and `false`. Instead, they allowed any numeric value inside a condition, with the convention that `0` denotes false and any non-`0` value denotes true. Furthermore, in C and C++ assignments have values. For example, the value of the assignment expression `floor = 13` is 13.

These two features—namely that numbers can be used as truth values and that assignments are expressions with values—conspire to make a horrible pitfall. The test

```
if (floor = 13) // ERROR
```

is legal C++, but it does not test whether `floor` and `13` are equal. Instead, the code sets `floor` to `13`, and because that value is not zero, the condition of the `if` statement is always fulfilled.

Fortunately, most compilers issue a warning when they encounter such a statement. You should take such warnings seriously. (See Programming Tip 3.5 for more advice about compiler warnings.)

Some shell-shocked programmers are so nervous about using = that they use == even when they want to make an assignment:

```
floor == floor - 1; // ERROR
```

This statement tests whether `floor` equals `floor - 1`. It doesn't do anything with the outcome of the test, but that is not an error. Some compilers will warn that "the code has no effect", but others will quietly accept the code.

Common Error 3.3

Exact Comparison of Floating-Point Numbers

Floating-point numbers have only a limited precision, and calculations can introduce roundoff errors. You must take these inevitable roundoffs into account when comparing floating-point numbers. For example, the following code multiplies the square root of 2 by itself. Ideally, we expect to get the answer 2:

© caracterdesign/iStockphoto.

Take limited precision into account when comparing floating-point numbers.

```
double r = sqrt(2.0);
if (r * r == 2)
{
   cout << "sqrt(2) squared is 2" << endl;
}
else
{
   cout << "sqrt(2) squared is not 2 but "
      << setprecision(18) << r * r << endl;
}
```

This program displays

```
sqrt(2) squared is not 2 but 2.00000000000000044
```

It does not make sense in most circumstances to compare floating-point numbers exactly. Instead, we should test whether they are *close enough*. That is, the magnitude of their

difference should be less than some threshold. Mathematically, we would write that x and y are close enough if

$$|x - y| < \varepsilon$$

for a very small number, ε. ε is the Greek letter epsilon, a letter used to denote a very small quantity. It is common to set ε to 10^{-14} when comparing `double` numbers:

```
const double EPSILON = 1E-14;
double r = sqrt(2.0);
if (abs(r * r - 2) < EPSILON)
{
   cout << "sqrt(2) squared is approximately 2";
}
```

Include the `<cmath>` header when you use the `abs` function.

Programming Tip 3.5

Compile with Zero Warnings

There are two kinds of messages that the compiler gives you: *errors* and *warnings*. Error messages are fatal; the compiler will not translate a program with one or more errors. Warning messages are advisory; the compiler will translate the program, but there is a good chance that the program will not do what you expect it to do.

It is a good idea to learn how to activate warnings with your compiler, and to write code that emits no warnings at all. For example, consider the test

```
if (floor = 13)
```

One C++ compiler emits a curious warning message: "Suggest parentheses around assignment used as truth value". Sadly, the message is misleading because it was not written for students. Nevertheless, such a warning gives you another chance to look at the offending statement and fix it, in this case, by replacing the = with an ==.

In order to make warnings more visible, many compilers require you to take some special action. This might involve clicking a checkbox in an integrated environment or supplying a special option on the command line. Ask your instructor or lab assistant how to turn on warnings for your compiler.

Special Topic 3.2

Lexicographic Ordering of Strings

If you compare strings using < <= > >=, they are compared in "lexicographic" order. This ordering is very similar to the way in which words are sorted in a dictionary.

For example, consider this code fragment.

```
string name = "Tom";
if (name < "Dick") . . .
```

The condition is not fulfilled, because in the dictionary Dick comes before Tom. There are a few differences between the ordering in a dictionary and in C++. In C++:

- All uppercase letters come before the lowercase letters. For example, `"Z"` comes before `"a"`.
- The space character comes before all printable characters.

Corbis Digital Stock.

To see which of two terms comes first in the dictionary, consider the first letter in which they differ.

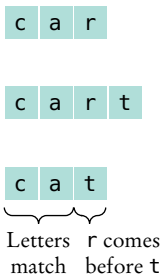
- Numbers come before letters.
- For the ordering of punctuation marks, see Appendix C.

Lexicographic order is used to compare strings.

When comparing two strings, the first letters of each word are compared, then the second letters, and so on, until one of the strings ends or a letter pair doesn't match.

If one of the strings ends, the longer string is considered the "larger" one. For example, compare `"car"` with `"cart"`. The first three letters match, and we reach the end of the first string. Therefore `"car"` comes before `"cart"` in lexicographic ordering.

When you reach a mismatch, the string containing the "larger" character is considered "larger". For example, let's compare `"cat"` with `"cart"`. The first two letters match. Since `t` comes after `r`, the string `"cat"` comes after `"cart"` in the lexicographic ordering.



Lexicographic Ordering

HOW TO 3.1

Implementing an `if` Statement

This How To walks you through the process of implementing an `if` statement. We will illustrate the steps with the following example problem:

Problem Statement The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than $128, and a 16 percent discount if the price is at least $128. Write a program that asks the cashier for the original price and then prints the discounted price.

Step 1 Decide upon the branching condition.

In our sample problem, the obvious choice for the condition is:

original price < 128?

That is just fine, and we will use that condition in our solution.

But you could equally well come up with a correct solution if you chose the opposite condition: Is the original price at least (≥) $128? You might choose this condition if you put yourself into the position of a shopper who wants to know when the bigger discount applies.

© MikePanic/iStockphoto.

Sales discounts are often higher for expensive products. Use the `if` statement to implement such a decision.

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

In this step, you list the action or actions that are taken in the "positive" branch. The details depend on your problem. You may want to print a message, compute values, or even exit the program.

In our example, we need to apply an 8 percent discount:

discounted price = 0.92 x original price

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

What do you want to do in the case that the condition of Step 1 is not fulfilled? Sometimes, you want to do nothing at all. In that case, use an `if` statement without an `else` branch.

In our example, the condition tested whether the price was less than $128. If that condition is *not* true, the price is at least $128, so the higher discount of 16 percent applies to the sale:

```
discounted price = 0.84 x original price
```

Step 4 Double-check relational operators.

First, be sure that the test goes in the right *direction*. It is a common error to confuse > and <. Next, consider whether you should use the < operator or its close cousin, the <= operator.

What should happen if the original price is exactly $128? Reading the problem carefully, we find that the lower discount applies if the original price is *less than* $128, and the higher discount applies when it is *at least* $128. A price of $128 should therefore *not* fulfill our condition, and we must use <, not <=.

Step 5 Remove duplication.

Check which actions are common to both branches, and move them outside. (See Programming Tip 3.4.)

In our example, we have two statements of the form

```
discounted price = ___ x original price
```

They only differ in the discount rate. It is best to just set the rate in the branches, and to do the computation afterwards:

```
If original price < 128
   discount rate = 0.92
Else
   discount rate = 0.84
discounted price = discount rate x original price
```

Step 6 Test both branches.

Formulate two test cases, one that fulfills the condition of the `if` statement, and one that does not. Ask yourself what should happen in each case. Then follow the pseudocode and act each of them out.

In our example, let us consider two scenarios for the original price: $100 and $200. We expect that the first price is discounted by $8, the second by $32.

When the original price is 100, then the condition 100 < 128 is true, and we get

```
discount rate = 0.92
discounted price = 0.92 x 100 = 92
```

When the original price is 200, then the condition 200 < 128 is false, and

```
discount rate = 0.84
discounted price = 0.84 x 200 = 168
```

In both cases, we get the expected answer.

Step 7 Assemble the `if` statement in C++.

Type the skeleton

```
if ()
{
}
else
{
}
```

and fill it in, as shown in Syntax 3.1. Omit the `else` branch if it is not needed.

In our example, the completed statement is

```
if (original_price < 128)
{
   discount_rate = 0.92;
}
else
{
   discount_rate = 0.84;
}
discounted_price = discount_rate * original_price;
```

EXAMPLE CODE See how_to_1 of your companion code for a program that calculates a discounted price.

WORKED EXAMPLE 3.1

Extracting the Middle

Learn how to extract the middle character from a string, or the two middle characters if the length of the string is even. See your E-Text or visit wiley.com/go/bclo3.

Computing & Society 3.1 Dysfunctional Computerized Systems

Making decisions is an essential part of any computer program. Nowhere is this more obvious than in a computer system that helps sort luggage at an airport. After scanning the luggage identification codes, the system sorts the items and routes them to different conveyor belts. Human operators then place the items onto trucks. When the city of Denver built a huge airport to replace an outdated and congested facility, the luggage system contractor went a step further. The new system was designed to replace the human operators with robotic carts. Unfortunately, the system plainly did not work. It was plagued by mechanical problems, such as luggage falling onto the tracks and jamming carts. Equally frustrating were the software glitches. Carts would uselessly accumulate at some locations when they were needed elsewhere.

The airport had been scheduled to open in 1993, but without a functioning luggage system, the opening was delayed for over a year while the contractor tried to fix the problems. The contractor never succeeded, and ultimately a manual system was installed. The delay cost the city and airlines close to a billion dollars, and the contractor, once the leading luggage systems vendor in the United States, went bankrupt.

Clearly, it is very risky to build a large system based on a technology that has never been tried on a smaller scale. In 2013, the rollout of universal healthcare in the United States was put in jeopardy by a dysfunctional web site for selecting insurance plans. The system promised an insurance shopping experience similar to booking airline flights. But, the HealthCare.gov site didn't simply present the available insurance plans. It also had to check the income level of each applicant and use that information to determine the subsidy level. That task turned out to be quite a bit harder than checking whether a credit card had sufficient credit to pay for an airline ticket. The Obama administration would have been well advised to design a signup process that did not rely on an untested computer program.

Lyn Alweis/Contributor/Getty Images.

The Denver airport originally had a fully automatic system for moving luggage, replacing human operators with robotic carts. Unfortunately, the system never worked and was dismantled before the airport was opened.

3.3 Multiple Alternatives

Multiple alternatives are required for decisions that have more than two cases.

In Section 3.1, you saw how to program a two-way branch with an `if` statement. In many situations, there are more than two cases. In this section, you will see how to implement a decision with multiple alternatives. For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale (see Table 3).

Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

The Richter scale is a measurement of the strength of an earthquake. Every step in the scale, for example from 6.0 to 7.0, signifies a tenfold increase in the strength of the quake.

In this case, there are five branches: one each for the four descriptions of damage, and one for no destruction.

You use multiple `if` statements to implement multiple alternatives, like this:

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, "
   << "some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
```

© kevinruss/iStockphoto.

The 1989 Loma Prieta earthquake that damaged the Bay Bridge in San Francisco and destroyed many buildings measured 7.1 on the Richter scale.

As soon as one of the four tests succeeds, the effect is displayed, and no further tests are attempted. If none of the four cases applies, the final `else` clause applies, and a

default message is printed. Figure 3 shows the flowchart for this multiple-branch statement.

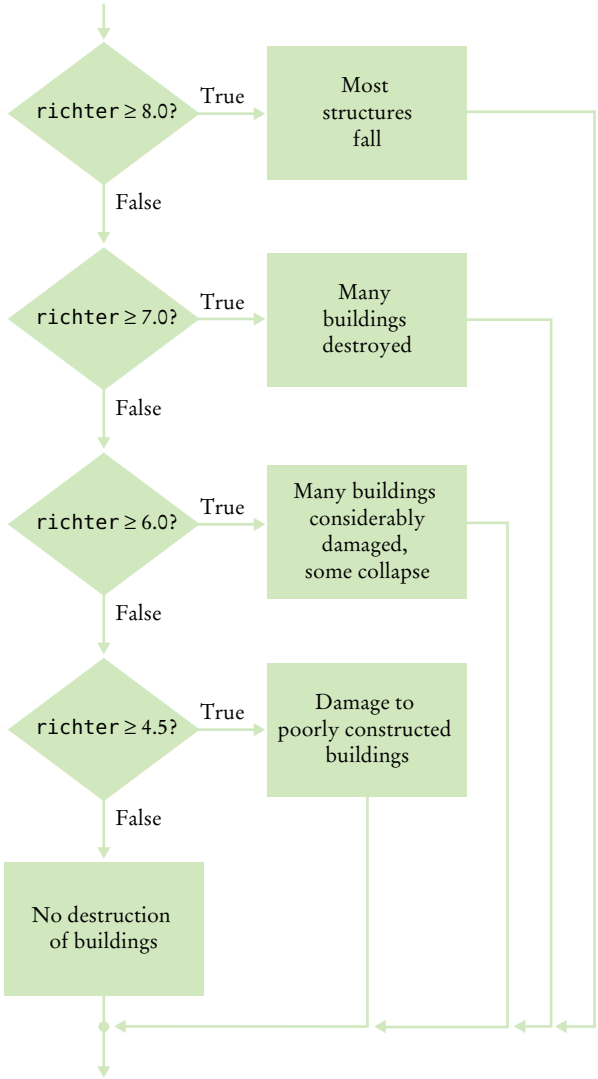


Figure 3 Multiple Alternatives

Here you must sort the conditions and test against the largest cutoff first. Suppose we reverse the order of tests:

```
if (richter >= 4.5) // Tests in wrong order
{
   cout << "Damage to poorly constructed buildings";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 7.0)
{
```

```
   cout << "Many buildings destroyed";
}
else if (richter >= 8.0)
{
   cout << "Most structures fall";
}
```

When using multiple `if` statements, pay attention to the order of the conditions.

This does not work. Suppose the value of `richter` is 7.1. That value is at least 4.5, matching the first case. The other tests will never be attempted.

In this example, it is also important that we use a sequence of `else if` clauses, not just multiple independent `if` statements. Consider this sequence of independent tests:

```
if (richter >= 8.0) // Didn't use else
{
   cout << "Most structures fall";
}
if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
```

Now the alternatives are no longer exclusive. If `richter` is 7.1, then the last *three* tests all match, and three messages are printed.

EXAMPLE CODE See sec03 of your companion code for the full program that prints earthquake descriptions.

Special Topic 3.3

The `switch` Statement

A sequence of `if` statements that compares a *single integer value* against several *constant* alternatives can be implemented as a `switch` statement. For example,

```
int digit;
. . .
switch (digit)
{
   case 1: digit_name = "one"; break;
   case 2: digit_name = "two"; break;
   case 3: digit_name = "three"; break;
   case 4: digit_name = "four"; break;
   case 5: digit_name = "five"; break;
   case 6: digit_name = "six"; break;
   case 7: digit_name = "seven"; break;
   case 8: digit_name = "eight"; break;
   case 9: digit_name = "nine"; break;
   default: digit_name = ""; break;
}
```

© travelpixpro/iStockphoto.

The `switch` statement lets you choose from a fixed set of alternatives.

This is a shortcut for

```
int digit;
if (digit == 1) { digit_name = "one"; }
else if (digit == 2) { digit_name = "two"; }
else if (digit == 3) { digit_name = "three"; }
else if (digit == 4) { digit_name = "four"; }
else if (digit == 5) { digit_name = "five"; }
else if (digit == 6) { digit_name = "six"; }
else if (digit == 7) { digit_name = "seven"; }
else if (digit == 8) { digit_name = "eight"; }
else if (digit == 9) { digit_name = "nine"; }
else { digit_name = ""; }
```

Well, it isn't much of a shortcut, but it has one advantage—it is obvious that all branches test the *same* value, namely `digit`.

It is possible to have multiple `case` clauses for a branch, such as

```
case 1: case 3: case 5: case 7: case 9:
   odd = true; break;
```

The `default` branch is chosen if none of the `case` clauses match.

Every branch of the switch must be terminated by a `break` instruction. If the `break` is missing, execution *falls through* to the next branch, and so on, until finally a `break` or the end of the `switch` is reached. In practice, this fall-through behavior is rarely useful, but it is a common cause of errors. If you accidentally forget the `break` statement, your program compiles but executes unwanted code. Many programmers consider the `switch` statement somewhat dangerous and prefer the `if` statement.

We leave it to you to use the `switch` statement for your own code or not. At any rate, you need to have a reading knowledge of `switch` in case you find it in other programmers' code.

3.4 Nested Branches

When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.

It is often necessary to include an `if` statement inside another. Such an arrangement is called a *nested* set of statements. Here is a typical example.

In the United States, different tax rates are used depending on the taxpayer's marital status. There are different tax schedules for single and for married taxpayers. Married taxpayers add their income together and pay taxes on the total. Table 4 gives the tax rate computations, using a simplified version of the tax rate schedule. A different tax rate applies to each "bracket". In this schedule, the income at the first bracket is

Computing income taxes requires multiple levels of decisions.

taxed at 10 percent, and the income at the second bracket is taxed at 25 percent. The income limits for each bracket depend on the marital status.

Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	**the tax is**	**of the amount over**
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

Nested decisions are required for problems that have two levels of decision making.

Now compute the taxes due, given a filing status and an income figure. The key point is that there are two *levels* of decision making. First, you must branch on the marital status. Then, for each filing status, you must have another branch on income level. The two-level decision process is reflected in two levels of `if` statements in the program at the end of this section. (See Figure 4 for a flowchart.)

In theory, nesting can go deeper than two levels. A three-level decision process (first by state, then by filing status, then by income level) requires three nesting levels.

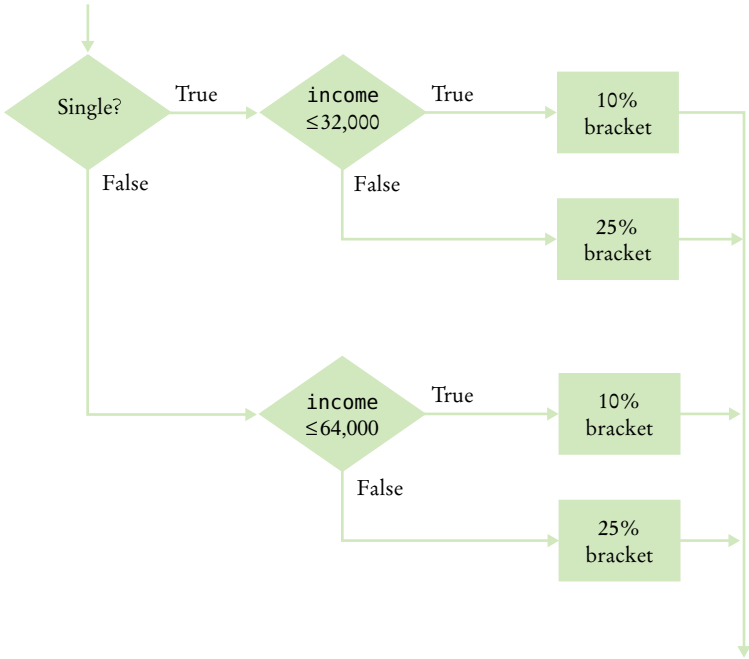


Figure 4 Income Tax Computation

sec04/tax.cpp

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   const double RATE1 = 0.10;
   const double RATE2 = 0.25;
   const double RATE1_SINGLE_LIMIT = 32000;
   const double RATE1_MARRIED_LIMIT = 64000;

   double tax1 = 0;
   double tax2 = 0;

   double income;
   cout << "Please enter your income: ";
   cin >> income;

   cout << "Please enter s for single, m for married: ";
   string marital_status;
   cin >> marital_status;

   if (marital_status == "s")
   {
      if (income <= RATE1_SINGLE_LIMIT)
      {
         tax1 = RATE1 * income;
      }
      else
      {
         tax1 = RATE1 * RATE1_SINGLE_LIMIT;
         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
      }
   }
   else
   {
      if (income <= RATE1_MARRIED_LIMIT)
      {
         tax1 = RATE1 * income;
      }
      else
      {
         tax1 = RATE1 * RATE1_MARRIED_LIMIT;
         tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
      }
   }

   double total_tax = tax1 + tax2;

   cout << "The tax is $" << total_tax << endl;
   return 0;
}
```

Program Run

```
Please enter your income: 80000
Please enter s for single, m for married: m
The tax is $10400
```

Common Error 3.4

The Dangling else Problem

When an `if` statement is nested inside another `if` statement, the following error may occur.

```
double shipping_charge = 5.00; // $5 inside continental U.S.
if (country == "USA")
   if (state == "HI")
      shipping_charge = 10.00; // Hawaii is more expensive
else // Pitfall!
   shipping_charge = 20.00; // As are foreign shipments
```

The indentation level seems to suggest that the `else` is grouped with the test `country == "USA"`. Unfortunately, that is not the case. The compiler ignores all indentation and matches the `else` with the preceding `if`. That is, the code is actually

```
double shipping_charge = 5.00; // $5 inside continental U.S.
if (country == "USA")
   if (state == "HI")
      shipping_charge = 10.00; // Hawaii is more expensive
   else // Pitfall!
      shipping_charge = 20.00; // As are foreign shipments
```

That isn't what you want. You want to group the else with the first `if`.

The ambiguous `else` is called a *dangling* `else`. You can avoid this pitfall if you *always use braces*, as recommended in Programming Tip 3.2:

```
double shipping_charge = 5.00; // $5 inside continental U.S.
if (country == "USA")
{
   if (state == "HI")
   {
      shipping_charge = 10.00; // Hawaii is more expensive
   }
}
else
{
   shipping_charge = 20.00; // As are foreign shipments
}
```

Programming Tip 3.6

Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or C++ code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

For example, let's trace the tax program with the data from the program run in Section 3.4.

© thomasd007/iStockphoto.

Hand-tracing helps you understand whether a program works correctly.

In lines 13 and 14, `tax1` and `tax2` are initialized to `0`.

```
 6 int main()
 7 {
 8    const double RATE1 = 0.10;
 9    const double RATE2 = 0.25;
10    const double RATE1_SINGLE_LIMIT = 32000;
11    const double RATE1_MARRIED_LIMIT = 64000;
12
13    double tax1 = 0;
14    double tax2 = 0;
15
```

tax1	tax2	income	marital status
0	0		

In lines 18 and 22, `income` and `marital_status` are initialized by input statements.

```
16    double income;
17    cout << "Please enter your income: ";
18    cin >> income;
19
20    cout << "Please enter s for single, m for married: ";
21    string marital_status;
22    cin >> marital_status;
23
```

tax1	tax2	income	marital status
0	0	80000	m

Because `marital_status` is not `"s"`, we move to the `else` branch of the outer `if` statement (line 36).

```
24    if (marital_status == "s")
25    {
26       if (income <= RATE1_SINGLE_LIMIT)
27       {
28          tax1 = RATE1 * income;
29       }
30       else
31       {
32          tax1 = RATE1 * RATE1_SINGLE_LIMIT;
33          tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
34       }
35    }
36    else
37    {
```

Because `income` is not `<= 64000`, we move to the `else` branch of the inner `if` statement (line 42).

```
38       if (income <= RATE1_MARRIED_LIMIT)
39       {
40          tax1 = RATE1 * income;
41       }
42       else
43       {
44          tax1 = RATE1 * RATE1_MARRIED_LIMIT;
45          tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
46       }
```

The values of `tax1` and `tax2` are updated.

```
43       {
44          tax1 = RATE1 * RATE1_MARRIED_LIMIT;
45          tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
46       }
47    }
```

tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

Their sum `total_tax` is computed and printed. Then the program ends.

```
48
49    double total_tax = tax1 + tax2;
50
51    cout << "The tax is $" << total_tax << endl;
52    return 0;
53 }
```

tax1	tax2	income	marital status	total tax
~~0~~	~~0~~	80000	m	
6400	4000			10400

Because the program trace shows the expected output ($10,400), it successfully demonstrated that this test case works correctly.

3.5 Problem Solving: Flowcharts

Flowcharts are made up of elements for tasks, input/outputs, and decisions.

You have seen examples of flowcharts earlier in this chapter. A flowchart shows the structure of decisions and tasks that are required to solve a problem. When you have to solve a complex problem, it is a good idea to draw a flowchart to visualize the flow of control.

The basic flowchart elements are shown in Figure 5.

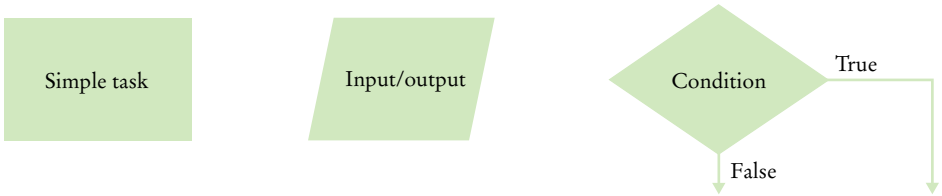


Figure 5
Flowchart Elements

Each branch of a decision can contain tasks and further decisions.

The basic idea is simple enough. Link tasks and input/output boxes in the sequence in which they should be executed. Whenever you need to make a decision, draw a diamond with two outcomes (see Figure 6).

Each branch can contain a sequence of tasks and even additional decisions. If there are multiple choices for a value, lay them out as in Figure 7.

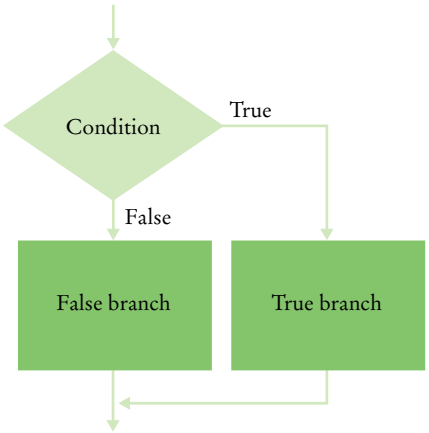


Figure 6 Flowchart with Two Outcomes

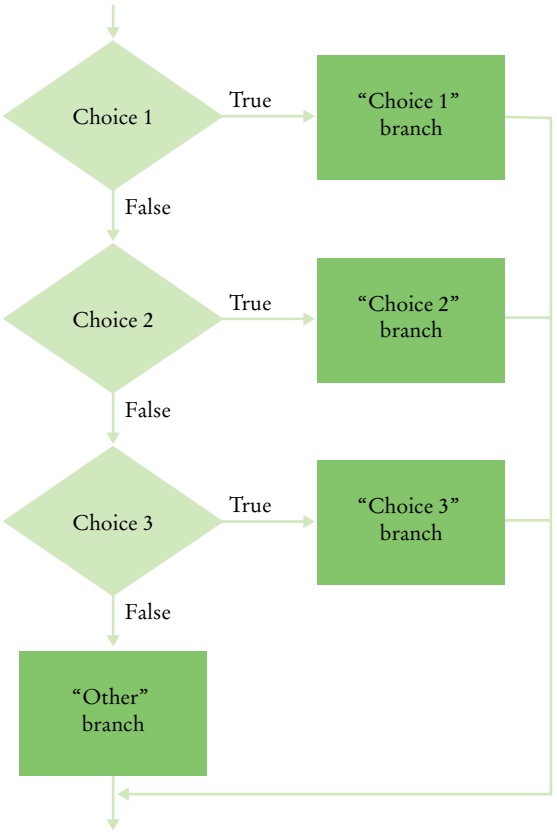


Figure 7 Flowchart with Multiple Choices

There is one issue that you need to be aware of when drawing flowcharts. Unconstrained branching and merging can lead to "spaghetti code", a messy network of possible pathways through a program.

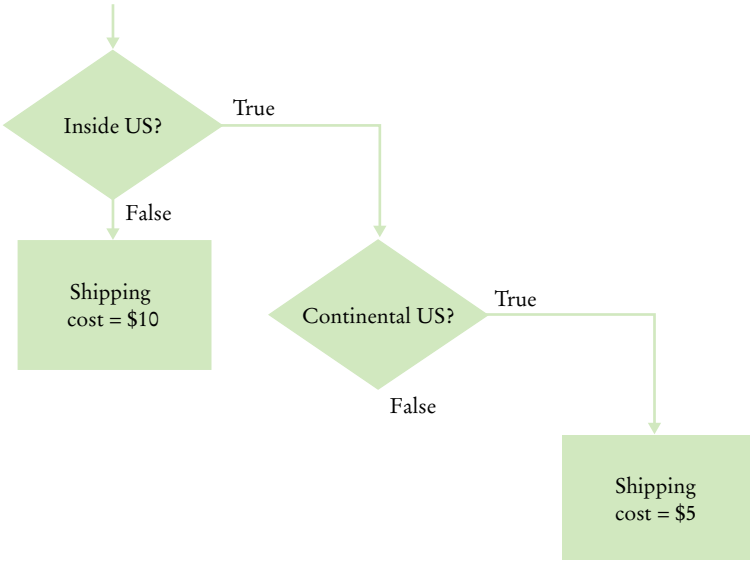
Never point an arrow inside another branch.

There is a simple rule for avoiding spaghetti code: Never point an arrow *inside another branch*.

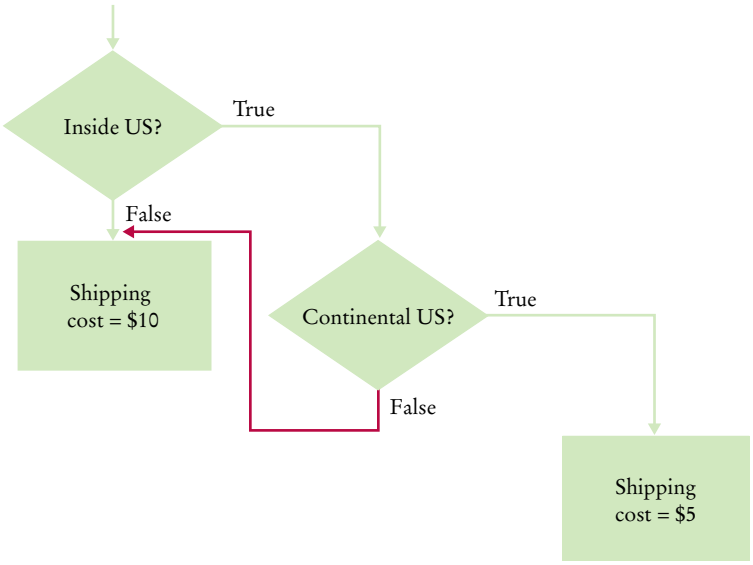
To understand the rule, consider this example: Shipping costs are $5 inside the United States, except that to Hawaii and Alaska they are $10. International shipping costs are also $10. You might start out with a flowchart like the following:

© Ekspansio/iStockphoto.

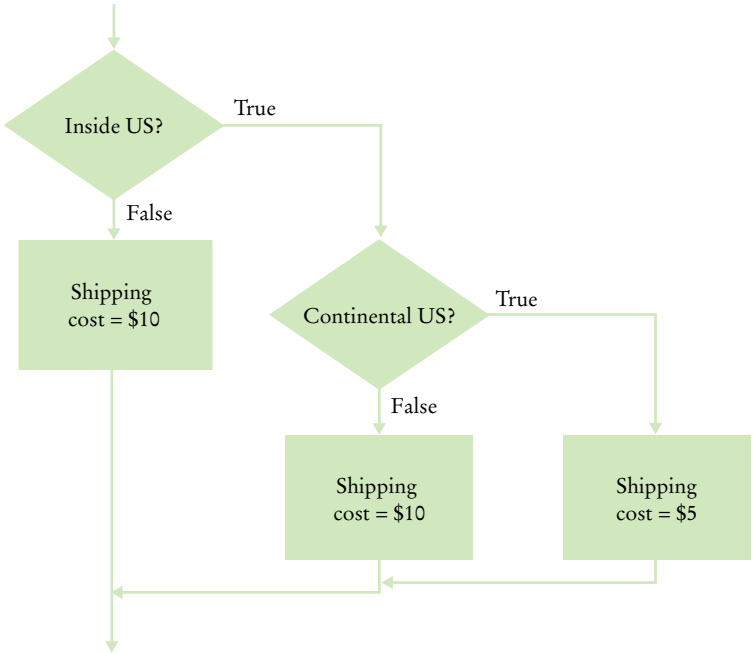
Spaghetti code has so many pathways that it becomes impossible to understand.



Now you may be tempted to reuse the "shipping cost = $10" task:



Don't do that! The red arrow points inside a different branch. Instead, add another task that sets the shipping cost to $10, like this:



Not only do you avoid spaghetti code, but it is also a better design. In the future it may well happen that the cost for international shipments is different from that to Alaska and Hawaii.

Flowcharts can be very useful for getting an intuitive understanding of the flow of an algorithm. However, they get large rather quickly when you add more details. At that point, it makes sense to switch from flowcharts to pseudocode.

EXAMPLE CODE See sec05 of your companion code for a program that computes shipping costs.

3.6 Problem Solving: Test Cases

Each branch of your program should be covered by a test case.

Consider how to test the tax computation program from Section 3.4. Of course, you cannot try out all possible inputs of filing status and income level. Even if you could, there would be no point in trying them all. If the program correctly computes one or two tax amounts in a given bracket, then we have a good reason to believe that all amounts will be correct.

You want to aim for complete *coverage* of all decision points. Here is a plan for obtaining a comprehensive set of test cases:

- There are two possibilities for the filing status and two tax brackets for each status, yielding four test cases.
- Test a handful of *boundary* conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking (which is discussed in Section 3.8), also test an invalid input, such as a negative income.

Make a list of the test cases and the expected outputs:

Test Case		Expected Output	Comment
30,000	s	3,000	10% bracket
72,000	s	13,200	3,200 + 25% of 40,000
50,000	m	5,000	10% bracket
104,000	m	16,400	6,400 + 25% of 40,000
32,000	s	3,200	boundary case
0		0	boundary case

When you develop a set of test cases, it is helpful to have a flowchart of your program (see Section 3.5). Check off each branch that has a test case. Include **boundary test cases** for each decision. For example, if a decision checks whether an input is less than 100, test with an input of 100.

It is a good idea to design test cases before implementing a program.

It is always a good idea to design test cases *before* starting to code. Working through the test cases gives you a better understanding of the algorithm that you are about to implement.

Programming Tip 3.7

Make a Schedule and Make Time for Unexpected Problems

Commercial software is notorious for being delivered later than promised. For example, Microsoft originally promised that its Windows Vista operating system would be available late in 2003, then in 2005, then in March 2006; it finally was released in January 2007. Some of the early promises might not have been realistic. It was in Microsoft's interest to let prospective customers expect the imminent availability of the product. Had customers known the actual delivery date, they might have switched to a different product in the meantime. Undeniably, though, Microsoft had not anticipated the full complexity of the tasks it had set itself to solve.

Microsoft can delay the delivery of its product, but it is likely that you cannot. As a student or a programmer, you are expected to manage your time wisely and to finish your assignments on time. You can probably do simple programming exercises the night before the due date, but an assignment that looks twice as hard may well take four times as long, because more things can go wrong. You should therefore make a schedule whenever you start a programming project.

First, estimate realistically how much time it will take you to:

- Design the program logic.
- Develop test cases.
- Type in the program and fix syntax errors.
- Test and debug the program.

For example, for the income tax program I might estimate an hour for the design; 30 minutes for developing test cases; an hour for data entry and fixing syntax errors; and an hour for testing and debugging. That is a total of 3.5 hours. If I work two hours a day on this project, it will take me almost two days.

Then think of things that can go wrong. Your computer might break down. You might be stumped by a problem with the computer system.

Bananastock/Media Bakery.

Make a schedule for your programming work and build in time for problems.

(That is a particularly important concern for beginners. It is *very* common to lose a day over a trivial problem just because it takes time to track down a person who knows the magic command to overcome it.) As a rule of thumb, *double* the time of your estimate. That is, you should start four days, not two days, before the due date. If nothing went wrong, great; you have the program done two days early. When the inevitable problem occurs, you have a cushion of time that protects you from embarrassment and failure.

3.7 Boolean Variables and Operators

Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a *Boolean variable*. Boolean variables are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

The Boolean type `bool` has two values, `false` and `true`.

In C++, the `bool` data type represents the Boolean type. Variables of type `bool` can hold exactly two values, denoted `false` and `true`. These values are not strings or integers; they are special values, just for Boolean variables.

Here is a definition of a Boolean variable:

```
bool failed = true;
```

Jon Patton/E+/iStockphoto.

A Boolean variable is also called a flag because it can be either up (true) or down (false).

You can use the value later in your program to make a decision:

```
if (failed) // Only executed if failed has been set to true
{
   . . .
}
```

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**. In C++, the `&&` operator (called *and*) yields `true` only when *both* conditions are true. The `||` operator (called *or*) yields the result `true` if *at least one* of the conditions is true.

Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water. (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.) Water is liquid if the temperature is greater than zero *and* less than 100:

```
if (temp > 0 && temp < 100) { cout << "Liquid"; }
```

© toos/iStockphoto.

At this geyser in Iceland, you can see ice, liquid water, and steam.

C++ has two Boolean operators that combine conditions: && (*and*) and || (*or*).

The condition of the test has two parts, joined by the && operator. (As shown in Table 5 and Appendix B, the > and < operators have higher precedence than the && operator.)

Table 5 Selected Operators and Their Precedence

Operator	Description
++ -- + (unary) – (unary) !	Increment, decrement, positive, negative, Boolean *not*
* / %	Multiplication, division, remainder
+ -	Addition, subtraction
< <= > >=	Comparisons
== !=	Equal, not equal
&&	Boolean *and*
||	Boolean *or*

Each part is a Boolean value that can be true or false. The combined expression is true if both individual expressions are true. If either one of the expressions is false, then the result is also false (see Figure 8).

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

A	!A
true	false
false	true

Figure 8 Boolean Truth Tables

Conversely, let's test whether water is *not* liquid at a given temperature. That is the case when the temperature is at most 0 *or* at least 100. Use the || (*or*) operator to combine the expressions:

```
if (temp <= 0 || temp >= 100) { cout << "Not liquid"; }
```

Figure 9 shows flowcharts for these examples.

EXAMPLE CODE See sec07 of your companion code for a program that compares numbers using Boolean expressions.

To invert a condition, use the ! (*not*) operator.

Sometimes you need to *invert* a condition with the *not* logical operator. The ! operator takes a single condition and evaluates to true if that condition is false and to false if the condition is true. In this example, output occurs if the value of the Boolean variable frozen is false:

```
if (!frozen)
{
   cout << "Not frozen";
}
```

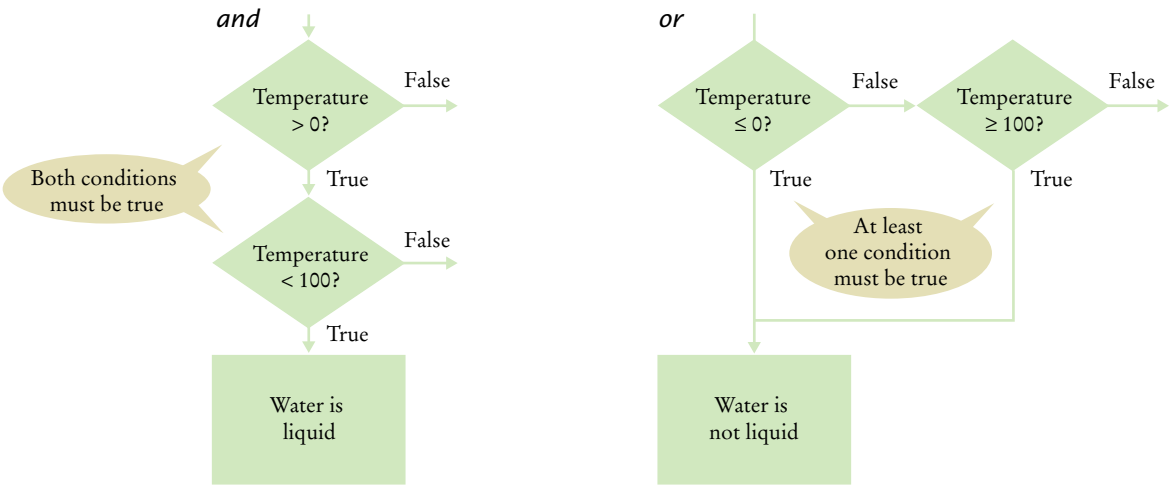


Figure 9 Flowcharts for *and* and *or* Combinations

Table 6 illustrates additional examples of evaluating Boolean operators.

Table 6 Boolean Operators

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true. Note that the < operator has a higher precedence than the && operator.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < 200 < 100`	`true`	**Error:** The expression `0 < 200` is `true`, which is converted to `1`. The expression `1 < 100` is `true`. You never want to write such an expression; see Common Error 3.5.
`-10 && 10 > 0`	`true`	**Error:** –10 is not zero. It is converted to `true`. You never want to write such an expression; see Common Error 3.5.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The && operator has a higher precedence than the `||` operator.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Common Error 3.5

Combining Multiple Relational Operators

Consider the expression

```
if (0 <= temp <= 100) // Error
```

This looks just like the mathematical test $0 \leq \texttt{temp} \leq 100$. Unfortunately, it is not.

Let us dissect the expression `0 <= temp <= 100`. The first half, `0 <= temp`, is a test with outcome `true` or `false`, depending on the value of `temp`. The outcome of that test (`true` or `false`) is then compared against `100`. Can one compare truth values and floating-point numbers? Is `true` larger than `100` or not? Unfortunately, to stay compatible with the C language, C++ converts `false` to `0` and `true` to 1. Therefore, the expression will always evaluate to true.

You must be careful not to mix logical and arithmetic expressions in your programs. Instead, use *and* to combine two separate tests:

```
if (0 <= temp && temp <= 100) . . .
```

Another common error, along the same lines, is to write

```
if (x && y > 0) . . . // Error
```

instead of

```
if (x > 0 && y > 0) . . .
```

Unfortunately, the compiler will not issue an error message. Instead, it converts `x` to `true` or `false`. Zero is converted to `false`, and any nonzero value is converted to `true`. If `x` is not zero, then it tests whether `y` is greater than `0`, and finally it computes the *and* of these two truth values. Naturally, that computation makes no sense.

Common Error 3.6

Confusing && and || Conditions

It is a surprisingly common error to confuse *and* and *or* conditions. A value lies between `0` and `100` if it is at least `0` *and* at most `100`. It lies outside that range if it is less than `0` *or* greater than `100`. There is no golden rule; you just have to think carefully.

Often the *and* or *or* is clearly stated, and then it isn't too hard to implement it. But sometimes the wording isn't as explicit. It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined. Consider these instructions for filing a tax return. You can claim single filing status if any one of the following is true:

- You were never married.
- You were legally separated or divorced on the last day of the tax year.
- You were widowed, and did not remarry.

Because the test passes if *any one* of the conditions is true, you must combine the conditions with *or*. Elsewhere, the same instructions state that you may use the more advantageous status of married filing jointly if all five of the following conditions are true:

- Your spouse died less than two years ago and you did not remarry.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of the tax year.
- You paid over half the cost of keeping up your home for this child.
- You filed a joint return with your spouse the year he or she died.

Because *all* of the conditions must be true for the test to pass, you must combine them with an *and*.

Special Topic 3.4

Short-Circuit Evaluation of Boolean Operators

When the && and || operators are computed, evaluation stops as soon as the truth value is determined. When an && is evaluated and the first condition is false, the second condition is not evaluated, because it does not matter what the outcome of the second test is.

For example, consider the expression

```
quantity > 0 && price / quantity < 10
```

The && and || operators are computed using *short-circuit evaluation:* As soon as the truth value is determined, no further conditions are evaluated.

Suppose the value of `quantity` is zero. Then the test `quantity > 0` fails, and the second test is not attempted. That is just as well, because it is illegal to divide by zero.

Similarly, when the first condition of an || expression is true, then the remainder is not evaluated since the result must be true.

This process is called **short-circuit evaluation**.

In a short circuit, electricity travels along the path of least resistance. Similarly, short-circuit evaluation takes the fastest path for computing the result of a Boolean expression.

Special Topic 3.5

De Morgan's Law

Humans generally have a hard time comprehending logical conditions with *not* operators applied to *and*/*or* expressions. **De Morgan's Law**, named after the logician Augustus De Morgan (1806–1871), can be used to simplify these Boolean expressions.

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States.

```
if (!(country == "USA"
      && state != "AK"
      && state != "HI"))
   shipping_charge = 20.00;
```

This test is a little bit complicated, and you have to think carefully through the logic. When it is *not* true that the country is USA *and* the state is not Alaska *and* the state is not Hawaii, then charge $20.00. Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but it takes human programmers to write and maintain the code. Therefore, it is useful to know how to simplify such a condition.

De Morgan's Law has two forms: one for the negation of an *and* expression and one for the negation of an *or* expression:

```
!(A && B)    is the same as    !A || !B
!(A || B)    is the same as    !A && !B
```

De Morgan's Law tells you how to negate && and || conditions.

Pay particular attention to the fact that the *and* and *or* operators are *reversed* by moving the *not* inward. For example, the negation of "the state is Alaska *or* it is Hawaii",

```
!(state == "AK" || state == "HI")
```

is "the state is not Alaska *and* it is not Hawaii":

```
!(state == "AK") && !(state == "HI")
```

That is, of course, the same as

```
state != "AK" && state != "HI"
```

Now apply the law to our shipping charge computation:

```
!(country == "USA"
   && state != "AK"
   && state != "HI")
```

is equivalent to

```
!(country == "USA")
   || !(state != "AK")
   || !(state != "HI")
```

That yields the simpler test

```
country != "USA"
   || state == "AK"
   || state == "HI"
```

To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.

3.8 Application: Input Validation

An important application for the `if` statement is *input validation*. Whenever your program accepts user input, you need to make sure that the user-supplied values are valid before you use them in your computations.

Consider our elevator program. Assume that the elevator panel has buttons labeled 1 through 20 (but not 13). The following are illegal inputs:

- The number 13
- Zero or a negative number
- A number larger than 20
- An input that is not a sequence of digits, such as `five`

In each of these cases, we will want to give an error message and exit the program.

Like a quality control worker, you want to make sure that user input is correct before processing it.

Tetra Images/Media Bakery.

It is simple to guard against an input of 13:

```
if (floor == 13)
{
   cout << "Error: There is no thirteenth floor." << endl;
   return 1;
}
```

The statement

```
return 1;
```

immediately exits the `main` function and therefore terminates the program. It is a convention to return with the value `0` if the program completed normally, and with a non-zero value when an error was encountered.

When reading a value, check that it is within the required range.

Here is how you ensure that the user doesn't enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
   cout << "Error: The floor must be between 1 and 20." << endl;
   return 1;
}
```

However, dealing with an input that is not a valid integer is a more serious problem. When the statement

```
cin >> floor;
```

is executed, and the user types in an input that is not an integer (such as `five`), then the integer variable `floor` is not set. Instead, the input stream `cin` is set to a failed state. You call the `fail` member function to test for that failed state.

```
if (cin.fail())
{
   cout << "Error: Not an integer." << endl;
   return 1;
}
```

Use the `fail` function to test whether the input stream has failed.

The order of the `if` statements is important. You must *first* test for `cin.fail()`. After all, if the input failed, no value has been assigned to `floor`, and it makes no sense to compare it against other values.

Input failure is quite serious in C++. Once input has failed, all subsequent attempts at input will fail as well. Special Topic 4.1 shows how to write programs that are more tolerant of bad input. For now, our goal is simply to detect bad input and to exit the program when it occurs.

Here is the complete elevator program with input validation.

sec08/elevator2.cpp

```
#include <iostream>

using namespace std;

int main()
{
   int floor;
   cout << "Floor: ";
   cin >> floor;

   // The following statements check various input errors
```

```
12    if (cin.fail())
13    {
14       cout << "Error: Not an integer." << endl;
15       return 1;
16    }
17    if (floor == 13)
18    {
19       cout << "Error: There is no thirteenth floor." << endl;
20       return 1;
21    }
22    if (floor <= 0 || floor > 20)
23    {
24       cout << "Error: The floor must be between 1 and 20." << endl;
25       return 1;
26    }
27
28    // Now we know that the input is valid
29    int actual_floor;
30    if (floor > 13)
31    {
32       actual_floor = floor - 1;
33    }
34    else
35    {
36       actual_floor = floor;
37    }
38
39    cout << "The elevator will travel to the actual floor "
40       << actual_floor << endl;
41
42    return 0;
43 }
```

Program Run

```
Floor: 13
Error: There is no thirteenth floor.
```

Computing & Society 3.2 Artificial Intelligence

When one uses a sophisticated computer program such as a tax preparation package, one is bound to attribute some intelligence to the computer. The computer asks sensible questions and makes computations that we find a mental challenge. After all, if doing one's taxes were easy, we wouldn't need a computer to do it for us.

As programmers, however, we know that all this apparent intelligence is an illusion. Human programmers have carefully "coached" the software in all possible scenarios, and it simply replays the actions and decisions that were programmed into it.

Would it be possible to write computer programs that are genuinely intelligent in some sense? From the earliest days of computing, there was a sense that the human brain might be nothing but an immense computer, and that it might well be feasible to program computers to imitate some processes of human thought. Serious research into *artificial intelligence* began in the mid-1950s, and the first twenty years brought some impressive successes. Programs that play chess—surely an activity that appears to require remarkable intellectual powers—have become so good that they now routinely beat all but the best human players. As far back as 1975, an *expert-system* program called Mycin gained fame for being better in diagnosing meningitis in patients than the average physician.

From the very outset, one of the stated goals of the AI community was to produce software that could translate text from one language to another, for example from English to Russian. That undertaking proved to be enormously complicated. Human language appears to be much more subtle and interwoven with the human experience than had originally been thought. Systems such as Apple's Siri can answer common questions about the

weather, appointments, and traffic. However, beyond a narrow range, they are more entertaining than useful.

In some areas, artificial intelligence technology has seen substantial advances. One of the most astounding examples is the rapid development of self-driving car technology. Starting in 2004, the Defense Advanced Research Projects Agency (DARPA) organized a series of competitions in which computer-controlled vehicles had to complete an obstacle course without a human driver or remote control. The first event was a disappointment, with none of the entrants finishing the route. In 2005, five vehicles completed a grueling 212 km course in the Mojave desert. Stanford's Stanley came in first, with an average speed of 30 km/h. In 2007, DARPA moved the competition to an "urban" environment, an abandoned air force base. Vehicles had to be able to interact with each other, following California traffic laws. Self-driving cars are now tested on public roads in several states, and it is expected that they will become commercially available within a decade.

When a system with artificial intelligence replaces a human in an activity such as giving medical advice or driving a vehicle, an important question arises. Who is responsible for mistakes? We accept that human doctors and drivers occasionally make mistakes with lethal consequences. Will we do the same for medical expert systems and self-driving cars?

Vaughn Youtz/Zuma Press.

Winner of the 2007 DARPA Urban Challenge

CHAPTER SUMMARY

Use the `if` statement to implement a decision.

- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

Implement comparisons of numbers and objects.

- Relational operators (`<` `<=` `>` `>=` `==` `!=`) are used to compare numbers and strings.
- Lexicographic order is used to compare strings.

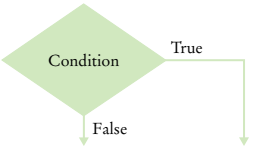
Implement complex decisions that require multiple `if` statements.

- Multiple alternatives are required for decisions that have more than two cases.
- When using multiple `if` statements, pay attention to the order of the conditions.

Implement decisions whose branches require further decisions.

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

Draw flowcharts for visualizing the control flow of a program.



- Flowcharts are made up of elements for tasks, input/outputs, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.

Design test cases for your programs.

- Each branch of your program should be covered by a test case.
- It is a good idea to design test cases before implementing a program.

Use the Boolean data type to store and combine conditions that can be true or false.

- The Boolean type `bool` has two values, `false` and `true`.
- C++ has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
- To invert a condition, use the `!` (*not*) operator.
- The `&&` and `||` operators are computed using *short-circuit evaluation*: As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's Law tells you how to negate `&&` and `||` conditions.

Apply `if` statements to detect whether user input is valid.

- When reading a value, check that it is within the required range.
- Use the `fail` function to test whether the input stream has failed.

REVIEW EXERCISES

■ **R3.1** What is the value of each variable after the `if` statement?

a.
```
int n = 1; int k = 2; int r = n;
if (k < n) { r = k; }
```

b.
```
int n = 1; int k = 2; int r;
if (n < k) { r = k; }
else { r = k + n; }
```

c.
```
int n = 1; int k = 2; int r = k;
if (r < k) { n = r; }
else { k = n; }
```

d.
```
int n = 1; int k = 2; int r = 3;
if (r < n + k) { r = 2 * n; }
else { k = 2 * r; }
```

■■ **R3.2** Explain the difference between

```
s = 0;
if (x > 0) { s++; }
if (y > 0) { s++; }
```

and

```
s = 0;
if (x > 0) { s++; }
else if (y > 0) { s++; }
```

■■ **R3.3** Find the errors in the following `if` statements.

a. `if x > 0 then cout << x;`

b. `if (x > 0) ; { y = 1; } else ; { y = -1; }`

c. `if (1 + x > pow(x, sqrt(2)) { y = y + x; }`

d. `if (x = 1) { y++; }`

e. `cin >> x; if (cin.fail()) { y = y + x; }`

■ **R3.4** What do these code fragments print?

a.
```
int n = 1; int m = -1;
if (n < -m) { cout << n; } else { cout << m; }
```

b.
```
int n = 1; int m = -1;
if (-n >= m) { cout << n; } else { cout << m; }
```

c.
```
double x = 0; double y = 1;
if (abs(x - y) < 1) { cout << x; } else { cout << y; }
```

d.
```
double x = sqrt(2); double y = 2;
if (x * x == y) { cout << x; } else { cout << y; }
```

■■ **R3.5** Suppose `x` and `y` are variables of type `double`. Write a code fragment that sets `y` to `x` if `x` is positive and to `0` otherwise.

■■ **R3.6** Suppose `x` and `y` are variables of type `double`. Write a code fragment that sets `y` to the absolute value of `x` without calling the `abs` function. Use an `if` statement.

■■ **R3.7** Explain why it is more difficult to compare floating-point numbers than integers. Write C++ code to test whether an integer `n` equals 10 and whether a floating-point number `x` equals 10.

■ **R3.8** Common Error 3.2 explains that a C++ compiler will not report an error when you use an assignment operator instead of a test for equality, but it may issue a warning. Write a test program containing a statement

```
if (floor = 13)
```

What does your compiler do when you compile the program?

■■ **R3.9** Given two pixels on a computer screen with integer coordinates (x_1, y_1) and (x_2, y_2), write conditions to test whether they are

a. The same pixel.

b. Very close together (with distance < 5).

■ **R3.10** It is easy to confuse the = and == operators. Write a test program containing the statement

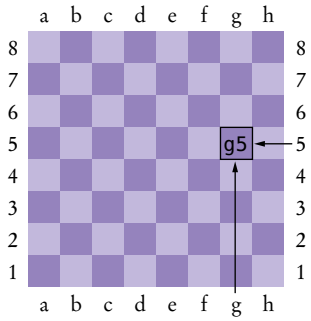
```
if (floor = 13)
```

What error message do you get? Write another test program containing the statement

```
count == 0;
```

What does your compiler do when you compile the program?

■■ **R3.11** Each square on a chess board can be described by a letter and number, such as `g5` in this example:



The following pseudocode describes an algorithm that determines whether a square with a given letter and number is dark (black) or light (white).

```
If the letter is an a, c, e, or g
   If the number is odd
      color = "black"
   Else
      color = "white"
Else
   If the number is even
      color = "black"
   Else
      color = "white"
```

Using the procedure in Programming Tip 3.6, trace this pseudocode with input `g5`.

■■ **R3.12** Give a set of four test cases for the algorithm of Exercise R3.11 that covers all branches.

■■ **R3.13** In a scheduling program, we want to check whether two appointments overlap. For simplicity, appointments start at a full hour, and we use military time (with hours 0–23). The following pseudocode describes an algorithm that determines whether the appointment with start time *start1* and end time *end1* overlaps with the appointment with start time *start2* and end time *end2*.

```
If start1 > start2
   s = start1
Else
   s = start2
If end1 < end2
   e = end1
Else
   e = end2
If s < e
   The appointments overlap.
Else
   The appointments don't overlap.
```

Trace this algorithm with an appointment from 10–12 and one from 11–13, then with an appointment from 10–11 and one from 12–13.

■ **R3.14** Draw a flowchart for the algorithm in Exercise R3.13.

■ **R3.15** Draw a flowchart for the algorithm in Exercise E3.13.

■ **R3.16** Draw a flowchart for the algorithm in Exercise E3.14.

■■ **R3.17** Develop a set of test cases for the algorithm in Exercise R3.13.

■■ **R3.18** Develop a set of test cases for the algorithm in Exercise E3.14.

■■ **R3.19** Write pseudocode for a program that prompts the user for a month and day and prints out whether it is one of the following four holidays:

- New Year's Day (January 1)
- Independence Day (July 4)
- Veterans Day (November 11)
- Christmas Day (December 25)

■■ **R3.20** Write pseudocode for a program that assigns letter grades for a quiz, according to the following table:

Score	Grade
90-100	A
80-89	B
70-79	C
60-69	D
< 60	F

■■ **R3.21** Explain how the lexicographic ordering of strings in C++ differs from the ordering of words in a dictionary or telephone book. *Hint:* Consider strings such as `IBM`, `wiley.com`, `Century 21`, and `While-U-Wait`.

■■ **R3.22** Of the following pairs of strings, which comes first in lexicographic order?

a. `"Tom", "Dick"`

b. `"Tom", "Tomato"`

c. `"church", "Churchill"`
d. `"car manufacturer", "carburetor"`
e. `"Harry", "hairy"`
f. `"C++", " Car"`
g. `"Tom", "Tom"`
h. `"Car", "Carl"`
i. `"car", "bar"`

■ **R3.23** Explain the difference between a sequence of `else if` clauses and nested `if` statements. Give an example for each.

■■ **R3.24** Give an example of a sequence of `else if` clauses where the order of the tests does not matter. Give an example where the order of the tests matters.

■ **R3.25** Rewrite the condition in Section 3.3 to use `<` operators instead of `>=` operators. What is the impact on the order of the comparisons?

■■ **R3.26** Give a set of test cases for the tax program in Exercise P3.8. Manually compute the expected results.

■ **R3.27** Make up another C++ code example that shows the dangling `else` problem, using the following statement. A student with a GPA of at least 1.5, but less than 2, is on probation. With less than 1.5, the student is failing.

■■■ **R3.28** Complete the following truth table by finding the truth values of the Boolean expressions for all combinations of the Boolean inputs `p`, `q`, and `r`.

p	q	r	(p && q) || !r	!(p && (q || !r))
false	false	false		
false	false	true		
false	true	false		
...				
5 more combinations				
...				

■■■ **R3.29** True or false? *A* `&&` *B* is the same as *B* `&&` *A* for any Boolean conditions *A* and *B*.

■ **R3.30** The "advanced search" feature of many search engines allows you to use Boolean operators for complex queries, such as "(cats OR dogs) AND NOT pets". Contrast these search operators with the Boolean operators in C++.

■■ **R3.31** Suppose the value of `b` is `false` and the value of `x` is `0`. What is the value of each of the following expressions?

a. `b && x == 0`
b. `b || x == 0`
c. `!b && x == 0`

d. `!b || x == 0`
e. `b && x != 0`
f. `b || x != 0`
g. `!b && x != 0`
h. `!b || x != 0`

■■ **R3.32** Simplify the following expressions. Here, `b` is a variable of type `bool`.

a. `b == true`
b. `b == false`
c. `b != true`
d. `b != false`

■■■ **R3.33** Simplify the following statements. Here, `b` is a variable of type `bool` and `n` is a variable of type `int`.

a. `if (n == 0) { b = true; } else { b = false; }`
(*Hint:* What is the value of `n == 0`?)
b. `if (n == 0) { b = false; } else { b = true; }`
c. `b = false; if (n > 1) { if (n < 2) { b = true; } }`
d. `if (n < 1) { b = true; } else { b = n > 2; }`

■ **R3.34** What is wrong with the following program?

```
cout << "Enter the number of quarters: ";
cin >> quarters;
total = total + quarters * 0.25;
cout << "Total: " << total << endl;
if (cin.fail()) { cout << "Input error."; }
```

■■■ **R3.35** Reading numbers is surprisingly difficult because a C++ input stream looks at the input one character at a time. First, white space is skipped. Then the stream consumes those input characters that can be a part of a number. Once the stream has recognized a number, it stops reading if it finds a character that cannot be a part of a number. However, if the first non-white space character is not a digit or a sign, or if the first character is a sign and the second one is not a digit, then the stream fails.

Consider a program reading an integer:

```
cout << "Enter the number of quarters: ";
int quarters;
cin >> quarters;
```

For each of the following user inputs, circle how many characters have been read and whether the stream is in the failed state or not.

a. `15.9`
b. `15 9`
c. `+159`
d. `-15A9`
e. `Fifteen`
f. `-Fifteen`
g. `+ 15`
h. `1.5E3`
i. `+1+5`

PRACTICE EXERCISES

- **E3.1** Write a program that reads an integer and prints whether it is negative, zero, or positive.
- **E3.2** Write a program that reads a floating-point number and prints "zero" if the number is zero. Otherwise, print "positive" or "negative". Add "small" if the absolute value of the number is less than 1, or "large" if it exceeds 1,000,000.
- **E3.3** Write a program that reads an integer and prints how many digits the number has, by checking whether the number is ≥ 10, ≥ 100, and so on. (Assume that all integers are less than ten billion.) If the number is negative, first multiply it with –1.
- **E3.4** Write a program that reads three numbers and prints "all the same" if they are all the same, "all different" if they are all different, and "neither" otherwise.
- **E3.5** Write a program that reads three numbers and prints "increasing" if they are in increasing order, "decreasing" if they are in decreasing order, and "neither" otherwise. Here, "increasing" means "strictly increasing", with each value larger than its predecessor. The sequence 3 4 4 would not be considered increasing.
- **E3.6** Repeat Exercise E3.5, but before reading the numbers, ask the user whether increasing/decreasing should be "strict" or "lenient". In lenient mode, the sequence 3 4 4 is increasing and the sequence 4 4 4 is both increasing and decreasing.
- **E3.7** Write a program that reads in three integers and prints "in order" if they are sorted in ascending *or* descending order, or "not in order" otherwise. For example,

```
1 2 5   in order
1 5 2   not in order
5 2 1   in order
1 2 2   in order
```

- **E3.8** Write a program that reads four integers and prints "two pairs" if the input consists of two matching pairs (in some order) and "not two pairs" otherwise. For example,

```
1 2 2 1   two pairs
1 2 2 3   not two pairs
2 2 2 2   two pairs
```

- **E3.9** A compass needle points a given number of degrees away from North, measured clockwise. Write a program that reads the angle and prints out the nearest compass direction; one of N, NE, E, SE, S, SW, W, NW. In the case of a tie, prefer the nearest principal direction (N, E, S, or W).
- **E3.10** Write a program that reads a temperature value and the letter C for Celsius or F for Fahrenheit. Print whether water is liquid, solid, or gaseous at the given temperature at sea level.
- **E3.11** The boiling point of water drops by about one degree centigrade for every 300 meters (or 1,000 feet) of altitude. Improve the program of Exercise E3.10 to allow the user to supply the altitude in meters or feet.
- **E3.12** Add error handling to Exercise E3.11. If the user does not enter a number when expected, or provides an invalid unit for the altitude, print an error message and end the program.

■■ **E3.13** When two points in time are compared, each given as hours (in military time, ranging from 0 and 23) and minutes, the following pseudocode determines which comes first.

```
If hour1 < hour2
   time1 comes first.
Else if hour1 and hour2 are the same
   If minute1 < minute2
      time1 comes first.
   Else if minute1 and minute2 are the same
      time1 and time2 are the same.
   Else
      time2 comes first.
Else
   time2 comes first.
```

Write a program that prompts the user for two points in time and prints the time that comes first, then the other time.

■■ **E3.14** The following algorithm yields the season (Spring, Summer, Fall, or Winter) for a given month and day.

```
If month is 1, 2, or 3, season = "Winter"
Else if month is 4, 5, or 6, season = "Spring"
Else if month is 7, 8, or 9, season = "Summer"
Else if month is 10, 11, or 12, season = "Fall"
If month is divisible by 3 and day >= 21
   If season is "Winter", season = "Spring"
   Else if season is "Spring", season = "Summer"
   Else if season is "Summer", season = "Fall"
   Else season = "Winter"
```

© rotofrank/iStockphoto.

Write a program that prompts the user for a month and day and then prints the season, as determined by this algorithm.

■■ **E3.15** Write a program that reads in two floating-point numbers and tests whether they are the same up to two decimal places. Here are two sample runs.

```
Enter two floating-point numbers: 2.0 1.99998
They are the same up to two decimal places.
Enter two floating-point numbers: 2.0 1.98999
They are different.
```

■■ **E3.16** *Unit conversion.* Write a unit conversion program that asks the users from which unit they want to convert (fl. oz, gal, oz, lb, in, ft, mi) and to which unit they want to convert (ml, l, g, kg, mm, cm, m, km). Reject incompatible conversions (such as gal → km). Ask for the value to be converted, then display the result:

```
Convert from? gal
Convert to? ml
Value? 2.5
2.5 gal = 9462.5 ml
```

■ **E3.17** Write a program that prompts the user to provide a single character from the alphabet. Print `Vowel` or `Consonant`, depending on the user input. If the user input is not a letter (between `a` and `z` or `A` and `Z`), or is a string of length `> 1`, print an error message.

■■ **E3.18** Write a program that asks the user to enter a month (1 for January, 2 for February, and so on) and then prints the number of days in the month. For February, print "28 or 29 days".

```
Enter a month: 5
30 days
```

Do not use a separate `if/else` branch for each month. Use Boolean operators.

PROGRAMMING PROJECTS

■■ **P3.1** Write a program that translates a letter grade into a number grade. Letter grades are A, B, C, D, and F, possibly followed by + or –. Their numeric values are 4, 3, 2, 1, and 0. There is no F+ or F–. A + increases the numeric value by 0.3, a – decreases it by 0.3. However, an A+ has value 4.0.

```
Enter a letter grade: B-
The numeric value is 2.7.
```

■■ **P3.2** Write a program that translates a number between 0 and 4 into the closest letter grade. For example, the number 2.8 (which might have been the average of several grades) would be converted to B–. Break ties in favor of the better grade; for example 2.85 should be a B.

■■ **P3.3** Write a program that takes user input describing a playing card in the following shorthand notation:

`A`	Ace
`2 ... 10`	Card values
`J`	Jack
`Q`	Queen
`K`	King
`D`	Diamonds
`H`	Hearts
`S`	Spades
`C`	Clubs

Your program should print the full description of the card. For example,

```
Enter the card notation: QS
Queen of Spades
```

■■ **P3.4** Write a program that reads in three floating-point numbers and prints the largest of the three inputs. For example:

```
Please enter three numbers: 4 9 2.5
The largest number is 9.
```

■■ **P3.5** Write a program that reads in three strings and sorts them lexicographically.

```
Enter three strings: Charlie Able Baker
Able
Baker
Charlie
```

■■ **P3.6** Write a program that prompts for the day and month of the user's birthday and then prints a horoscope. Make up fortunes for programmers, like this:

```
Please enter your birthday (month and day): 6 16
Gemini are experts at figuring out the behavior of complicated programs.
You feel where bugs are coming from and then stay one step ahead.
Tonight, your style wins approval from a tough critic.
```

Each fortune should contain the name of the astrological sign. (You will find the names and date ranges of the signs at a distressingly large number of sites on the Internet.)

© lillisphotography/iStockphoto.

■■ **P3.7** The original U.S. income tax of 1913 was quite simple. The tax was

- 1 percent on the first $50,000.
- 2 percent on the amount over $50,000 up to $75,000.
- 3 percent on the amount over $75,000 up to $100,000.
- 4 percent on the amount over $100,000 up to $250,000.
- 5 percent on the amount over $250,000 up to $500,000.
- 6 percent on the amount over $500,000.

There was no separate schedule for single or married taxpayers. Write a program that computes the income tax according to this schedule.

■■■ **P3.8** Write a program that computes taxes for the following schedule:

If your status is Single and if the taxable income is over	but not over	the tax is	of the amount over
$0	$8,000	10%	$0
$8,000	$32,000	$800 + 15%	$8,000
$32,000		$4,400 + 25%	$32,000
If your status is Married and if the taxable income is over	**but not over**	**the tax is**	**of the amount over**
$0	$16,000	10%	$0
$16,000	$64,000	$1,600 + 15%	$16,000
$64,000		$8,800 + 25%	$64,000

■■■ **P3.9** The `tax.cpp` program uses a simplified version of the 2008 U.S. income tax schedule. Look up the tax brackets and rates for the current year, for both single and married filers, and implement a program that computes the actual income tax.

■ **P3.10** Write a program that reads in the *x*- and *y*-coordinates of two corner points of a rectangle and then prints out whether the rectangle is a square, or is in "portrait" or "landscape" orientation.

■■ **P3.11** Write a program that reads in the *x*- and *y*-coordinates of three corner points of a triangle and prints out whether it has an obtuse angle, a right angle, or only acute angles.

■■■ **P3.12** Write a program that reads in the *x*- and *y*-coordinates of four corner points of a quadrilateral and prints out whether it is a square, a rectangle, a trapezoid, a rhombus, or none of those shapes.

■■■ **P3.13** *Roman numbers*. Write a program that converts a positive integer into the Roman number system. The Roman number system has digits

I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

© Straitshooter/iStockphoto.

Numbers are formed according to the following rules.

a. Only numbers up to 3,999 are represented.

b. As in the decimal system, the thousands, hundreds, tens, and ones are expressed separately.

c. The numbers 1 to 9 are expressed as

I	1
II	2
III	3
IV	4
V	5
VI	6
VII	7
VIII	8
IX	9

As you can see, an I preceding a V or X is subtracted from the value, and you can never have more than three I's in a row.

d. Tens and hundreds are done the same way, except that the letters X, L, C and C, D, M are used instead of I, V, X, respectively.

Your program should take an input, such as 1978, and convert it to Roman numerals, MCMLXXVIII.

■■■ **P3.14** A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000). Write a program that asks the user for a year and computes whether that year is a leap year. Use a single `if` statement and Boolean operators.

■■■ **P3.15** French country names are feminine when they end with the letter e, masculine otherwise, except for the following which are masculine even though they end with e:

- le Belize
- le Cambodge
- le Mexique
- le Mozambique
- le Zaïre
- le Zimbabwe

Write a program that reads the French name of a country and adds the article: le for masculine or la for feminine, such as le Canada or la Belgique.

However, if the country name starts with a vowel, use l'; for example, l'Afghanistan.

For the following plural country names, use les:

- les Etats-Unis
- les Pays-Bas

■■■ **Business P3.16** Write a program to simulate a bank transaction. There are two bank accounts: checking and savings. First, ask for the initial balances of the bank accounts; reject negative balances. Then ask for the transactions; options are deposit, withdrawal, and transfer. Then ask for the account; options are checking and savings. Then ask for the amount; reject transactions that overdraw an account. At the end, print the balances of both accounts.

■■ **Business P3.17** Write a program that reads in the name and salary of an employee. Here the salary will denote an *hourly* wage, such as $9.25. Then ask how many hours the employee worked in the past week. Be sure to accept fractional hours. Compute the pay. Any overtime work (over 40 hours per week) is paid at 150 percent of the regular wage. Print a paycheck for the employee.

■■ **Business P3.18** When you use an automated teller machine (ATM) with your bank card, you need to use a personal identification number (PIN) to access your account. If a user fails more than three times when entering the PIN, the machine will block the card. Assume that the user's PIN is "1234" and write a program that asks the user for the PIN no more than three times, and does the following:

- If the user enters the right number, print a message saying, "Your PIN is correct", and end the program.
- If the user enters a wrong number, print a message saying, "Your PIN is incorrect" and, if you have asked for the PIN less than three times, ask for it again.
- If the user enters a wrong number three times, print a message saying "Your bank card is blocked" and end the program.

■ **Business P3.19** Calculating the tip when you go to a restaurant is not difficult, but your restaurant wants to suggest a tip according to the service diners receive. Write a program that calculates a tip according to the diner's satisfaction as follows:

- Ask for the diners' satisfaction level using these ratings: 1 = Totally satisfied, 2 = Satisfied, 3 = Dissatisfied.
- If the diner is totally satisfied, calculate a 20 percent tip.
- If the diner is satisfied, calculate a 15 percent tip.
- If the diner is dissatisfied, calculate a 10 percent tip.
- Report the satisfaction level and tip in dollars and cents.

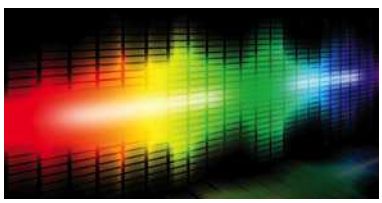
■ **Business P3.20** A supermarket awards coupons depending on how much a customer spends on groceries. For example, if you spend $50, you will get a coupon worth eight percent of that amount. The following table shows the percent used to calculate the coupon awarded for different amounts spent. Write a program that calculates and prints the value of the coupon a person can receive based on groceries purchased.

Here is a sample run:

```
Please enter the cost of your groceries: 14
You win a discount coupon of $ 1.12. (8% of your purchase)
```

Money Spent	Coupon Percentage
Less than $10	No coupon
From $10 to $60	8%
More than $60 to $150	10%
More than $150 to $210	12%
More than $210	14%

■ **Engineering P3.21** Write a program that prompts the user for a wavelength value and prints a description of the corresponding part of the electromagnetic spectrum, as given in Table 7.



© drxy/iStockphoto.

Table 7 Electromagnetic Spectrum

Type	Wavelength (m)	Frequency (Hz)
Radio Waves	$> 10^{-1}$	$< 3 \times 10^{9}$
Microwaves	10^{-3} to 10^{-1}	3×10^{9} to 3×10^{11}
Infrared	7×10^{-7} to 10^{-3}	3×10^{11} to 4×10^{14}
Visible light	4×10^{-7} to 7×10^{-7}	4×10^{14} to 7.5×10^{14}
Ultraviolet	10^{-8} to 4×10^{-7}	7.5×10^{14} to 3×10^{16}
X-rays	10^{-11} to 10^{-8}	3×10^{16} to 3×10^{19}
Gamma rays	$< 10^{-11}$	$> 3 \times 10^{19}$

▪ Engineering P3.22 Repeat Exercise P3.21, modifying the program so that it prompts for the frequency instead.

▪▪ Engineering P3.23 Repeat Exercise P3.21, modifying the program so that it first asks the user whether the input will be a wavelength or a frequency.

▪▪▪ Engineering P3.24 A minivan has two sliding doors. Each door can be opened by either a dashboard switch, its inside handle, or its outside handle. However, the inside handles do not work if a child lock switch is activated. In order for the sliding doors to open, the gear shift must be in park, *and* the master unlock switch must be activated. (This book's author is the long-suffering owner of just such a vehicle.)


© nano/iStockphoto.

Your task is to simulate a portion of the control software for the vehicle. The input is a sequence of values for the switches and the gear shift, in the following order:

- Dashboard switches for left and right sliding door, child lock, and master unlock (`0` for off or `1` for activated)
- Inside and outside handles on the left and right sliding doors (`0` or `1`)
- The gear shift setting (one of `P N D 1 2 3 R`).

A typical input would be `0 0 0 1 0 1 0 0 P`.

Print "left door opens" and/or "right door opens" as appropriate. If neither door opens, print "both doors stay closed".

▪ Engineering P3.25 Sound level L in units of decibel (dB) is determined by

$$L = 20 \log_{10}(p/p_0)$$

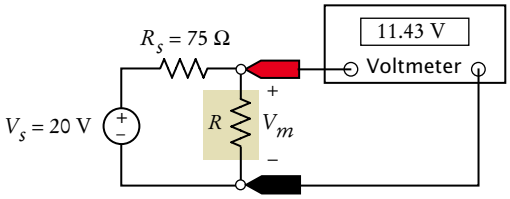

© Photobuff/iStockphoto.

where p is the sound pressure of the sound (in Pascals, abbreviated Pa), and p_0 is a reference sound pressure equal to 20×10^{-6} Pa (where L is 0 dB). The following table gives descriptions for certain sound levels.

Threshold of pain	130 dB
Possible hearing damage	120 dB
Jack hammer at 1 m	100 dB
Traffic on a busy roadway at 10 m	90 dB
Normal conversation	60 dB
Calm library	30 dB
Light leaf rustling	0 dB

Write a program that reads a value and a unit, either dB or Pa, and then prints the closest description from the list above.

▪▪ Engineering P3.26 The electric circuit shown below is designed to measure the temperature of the gas in a chamber.



The resistor R represents a temperature sensor enclosed in the chamber. The resistance R, in Ω, is related to the temperature T, in °C, by the equation

$$R = R_0 + kT$$

In this device, assume $R_0 = 100\ \Omega$ and $k = 0.5$. The voltmeter displays the value of the voltage, V_m, across the sensor. This voltage V_m indicates the temperature, T, of the gas according to the equation

$$T = \frac{R}{k} - \frac{R_0}{k} = \frac{R_s}{k}\frac{V_m}{V_s - V_m} - \frac{R_0}{k}$$

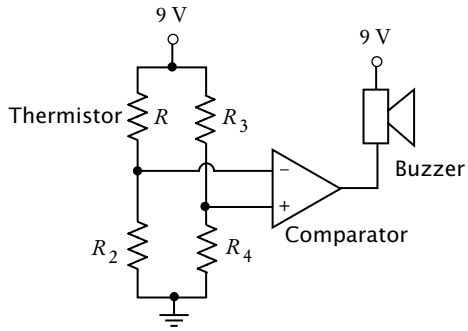
Suppose the voltmeter voltage is constrained to the range $V_{\min} = 12$ volts $\le V_m \le V_{\max} = 18$ volts. Write a program that accepts a value of V_m and checks that it's between 12 and 18. The program should return the gas temperature in degrees Celsius when V_m is between 12 and 18 and an error message when it isn't.

▪▪▪ Engineering P3.27 Crop damage due to frost is one of the many risks confronting farmers. The figure below shows a simple alarm circuit designed to warn of frost. The alarm circuit uses a device called a thermistor to sound a buzzer when the temperature drops below freezing. Thermistors are semiconductor devices that exhibit a temperature dependent resistance described by the equation


© rotofrank/iStockphoto.

$$R = R_0 e^{\beta\left(\frac{1}{T} - \frac{1}{T_0}\right)}$$

where R is the resistance, in Ω, at the temperature T, in °K, and R_0 is the resistance, in Ω, at the temperature T_0, in°K. β is a constant that depends on the material used to make the thermistor.



The circuit is designed so that the alarm will sound when

$$\frac{R_2}{R + R_2} < \frac{R_4}{R_3 + R_4}$$

The thermistor used in the alarm circuit has $R_0 = 33{,}192\ \Omega$ at $T_0 = 40\ ^\circ\text{C}$, and $\beta = 3{,}310\ ^\circ\text{K}$. (Notice that β has units of °K. Recall that the temperature in °K is obtained by adding 273° to the temperature in °C.) The resistors R_2, R_3, and R_4 have a resistance of 156.3 kΩ = 156,300 Ω.

Write a C++ program that prompts the user for a temperature in °F and prints a message indicating whether or not the alarm will sound at that temperature.

■ Engineering P3.28 A mass $m = 2$ kilograms is attached to the end of a rope of length $r = 3$ meters. The mass is whirled around at high speed. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program that accepts a rotation speed v and determines if such a speed will cause the rope to break. *Hint:* $T = mv^2/r$.

■■ Engineering P3.29 A mass m is attached to the end of a rope of length $r = 3$ meters. The rope can only be whirled around at speeds of 1, 10, 20, or 40 meters per second. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program where the user enters the value of the mass m, and the program determines the greatest speed at which it can be whirled without breaking the rope. *Hint:* $T = mv^2/r$.

■■ Engineering P3.30 The average person can jump off the ground with a velocity of 7 mph without fear of leaving the planet. However, if an astronaut jumps with this velocity while standing on Halley's Comet, will the astronaut ever come back down? Create a program that allows the user to input a launch velocity (in mph) from the surface of Halley's Comet and determine whether a jumper will return to the surface. If not, the program should calculate how much more massive the comet must be in order to return the jumper to the surface.

Courtesy NASA/JPL-Caltech.

Hint: Escape velocity is $v_{\text{escape}} = \sqrt{2\frac{GM}{R}}$, where $G = 6.67 \times 10^{-11} N\ \text{m}^2/\text{kg}^2$ is the gravitational constant, $M = 1.3 \times 10^{22}$ kg is the mass of Halley's comet, and $R = 1.153 \times 10^6$ m is its radius.

WORKED EXAMPLE 3.1

Extracting the Middle

Problem Statement Your task is to extract a string containing the middle character from a given string `str`. For example, if the string is "crate", the result is the string "a". However, if the string has an even number of letters, extract the middle two characters. If the string is "crates", the result is "at".

Step 1 Decide on the branching condition.

We need to take different actions for strings of odd and even length. Therefore, the condition is

Is the length of the string odd?

In C++, you use the remainder of division by 2 to find out whether a value is even or odd. Then the test becomes

```
if (str.length() % 2 == 1)
```

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

We need to find the position of the middle character. If the length is 5, the position is 2.

c	r	a	t	e
0	1	2	3	4

In general,

position = str.length() / 2 (with the remainder discarded)
result = str.substr(position, 1)

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

Again, we need to find the position of the middle characters. If the length is 6, the starting position is 2, and the ending position is 3, for a substring length of 2.

c	r	a	t	e	s
0	1	2	3	4	5

In general,

position = str.length() / 2 - 1
result = str.substr(position, 2)

Step 4 Double-check relational operators.

Do we really want `str.length() % 2 == 1`? For example, when the length is 5, `5 % 2` is the remainder of the division 5 / 2, which is 1. In general, dividing an odd number by 2 leaves a remainder of 1. (Actually, dividing a negative odd number by 2 leaves a remainder of –1, but the string length is never negative.) Therefore, our condition is correct.

Step 5 Remove duplication.

Here is the statement that we have developed:

```
If the length of str is odd
   position = str.length() / 2 (with remainder discarded)
   result = str.substr(position, 1)
Else
   position = str.length() / 2 - 1
   result = str.substr(position, 2)
```

The second statement in each branch is almost identical, but the length of the substring differs. Let's set the length in each branch:

```
If the length of str is odd
   position = str.length() / 2 (with remainder discarded)
   length = 1
Else
   position = str.length() / 2 - 1
   length = 2
result = str.substr(position, length)
```

Step 6 Test both branches.

We will use a different set of strings for testing. For an odd-length string, consider `"monitor"`. We get

```
position = str.length() / 2 = 7 / 2 = 3 (with remainder discarded)
length = 1
result = str.substr(3, 1) = "i"
```

For the even-length string `"monitors"`, we get

```
position = str.length() / 2 - 1 = 8 / 2 - 1 = 3 (with remainder discarded)
length = 2
result = str.substr(3, 2) = "it"
```

Step 7 Assemble the `if` statement in C++.

Here's the completed code segment (see `worked_example_1/middle.cpp` for the complete program):

```
if (str.length() % 2 == 1)
{
   position = str.length() / 2;
   length = 1;
}
else
{
   position = str.length() / 2 - 1;
   length = 2;
}
result = str.substr(position, length);
```

CHAPTER 4

LOOPS

© photo75/iStockphoto.

CHAPTER GOALS

- To implement `while`, `for`, and `do` loops
- To avoid infinite loops and off-by-one errors
- To understand nested loops
- To implement programs that read and process data sets
- To use a computer for simulations

CHAPTER CONTENTS

In a loop, a part of a program is repeated over and over, until a specific goal is reached. Loops are important for calculations that require repeated steps and for processing input consisting of many data items. In this chapter you will learn about loop statements in C++, as well as techniques for writing programs that process input and simulate activities in the real world.

4.1 The `while` Loop

In this section, you will learn how to repeatedly execute statements until a goal has been reached.

Recall the investment problem from Chapter 1. You put $10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original investment?

In Chapter 1 we developed the following algorithm for this problem:

Start with a year value of 0, a column for the interest, and a balance of $10,000.

year	interest	balance
0		$10,000

Repeat the following steps while the balance is less than $20,000
  Add 1 to the year value.
  Compute the interest as balance x 0.05 (i.e., 5 percent interest).
  Add the interest to the balance.
Report the final year value as the answer.

You now know how to define and update the variables in C++. What you don't yet know is how to carry out "Repeat steps while the balance is less than $20,000".

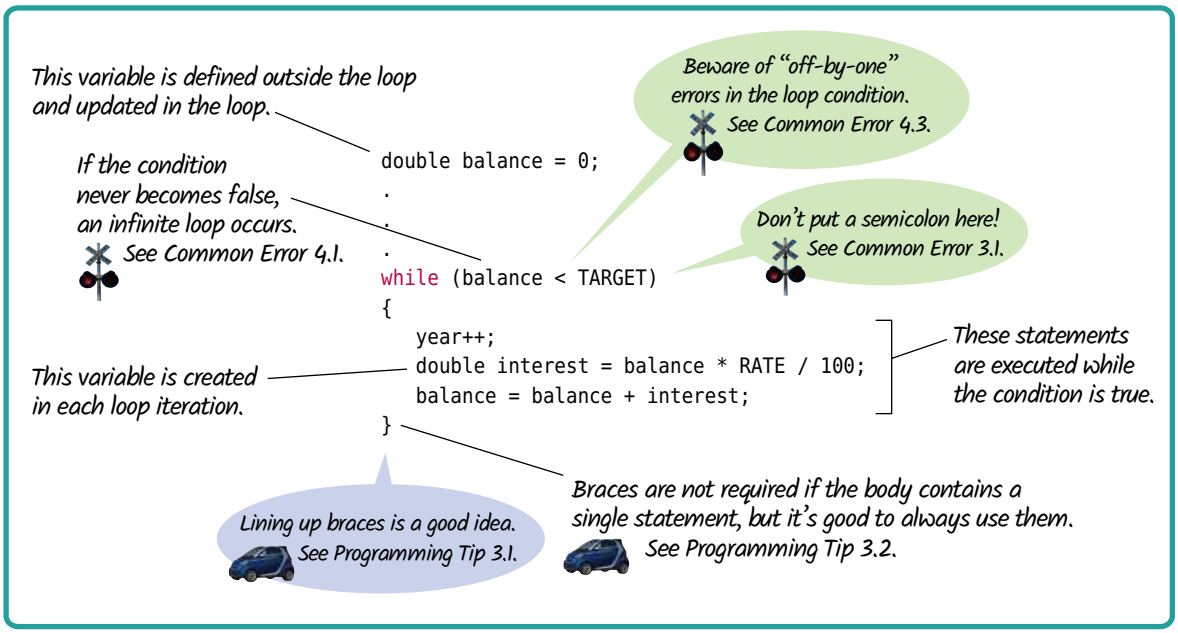
© AlterYourReality/iStockphoto.

Because the interest earned also earns interest, a bank balance grows exponentially.

© mmac72/iStockphoto.

In a particle accelerator, subatomic particles traverse a loop-shaped tunnel multiple times, gaining the speed required for physical experiments. Similarly, in computer science, statements in a loop are executed while a condition is true.

Syntax 4.1 while Statement



Loops execute a block of code repeatedly while a condition remains true.

In C++, the `while` statement implements such a repetition (see Syntax 4.1). The code

```
while (condition)
{
   statements
}
```

keeps executing the statements while the condition is true. In our case, we want to increment the year counter and add interest while the balance is less than the target balance of $20,000:

```
while (balance < TARGET)
{
   year++;
   double interest = balance * RATE / 100;
   balance = balance + interest;
}
```

A `while` statement is an example of a **loop**. If you draw a flowchart, the flow of execution loops again to the point where the condition is tested (see Figure 1).

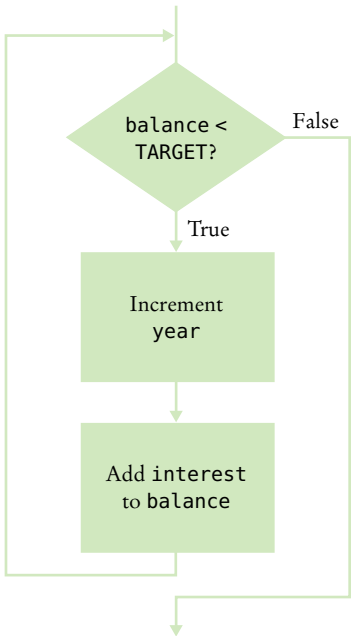


Figure 1 Flowchart of a `while` Loop

When you define a variable *inside* the loop body, the variable is created for each iteration of the loop and removed after the end of each iteration. For example, consider the `interest` variable in this loop:

```
while (balance < TARGET)
{
   year++;
   double interest = balance * RATE / 100;
   balance = balance + interest;
} // interest no longer defined here
```

A new `interest` *variable is created in each iteration.*

In contrast, the `balance` and `years` variables were defined *outside* the loop body. That way, the same variable is used for all iterations of the loop.

Here is the program that solves the investment problem. Figure 2 illustrates the program's execution.

sec01/doublinv.cpp

```
#include <iostream>

using namespace std;

int main()
{
   const double RATE = 5;
   const double INITIAL_BALANCE = 10000;
   const double TARGET = 2 * INITIAL_BALANCE;

   double balance = INITIAL_BALANCE;
   int year = 0;

   while (balance < TARGET)
   {
      year++;
      double interest = balance * RATE / 100;
      balance = balance + interest;
   }

   cout << "The investment doubled after "
      << year << " years." << endl;

   return 0;
}
```

Program Run

```
The investment doubled after 15 years.
```

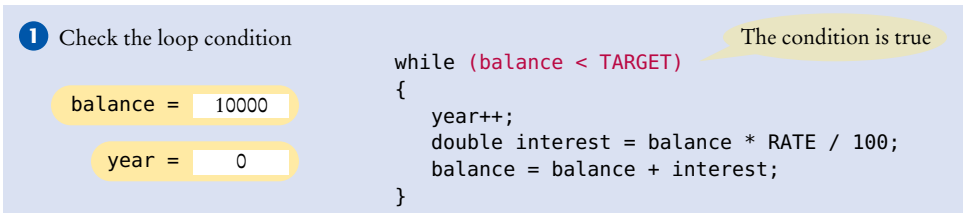


Figure 2 Execution of the `doublinv` Loop

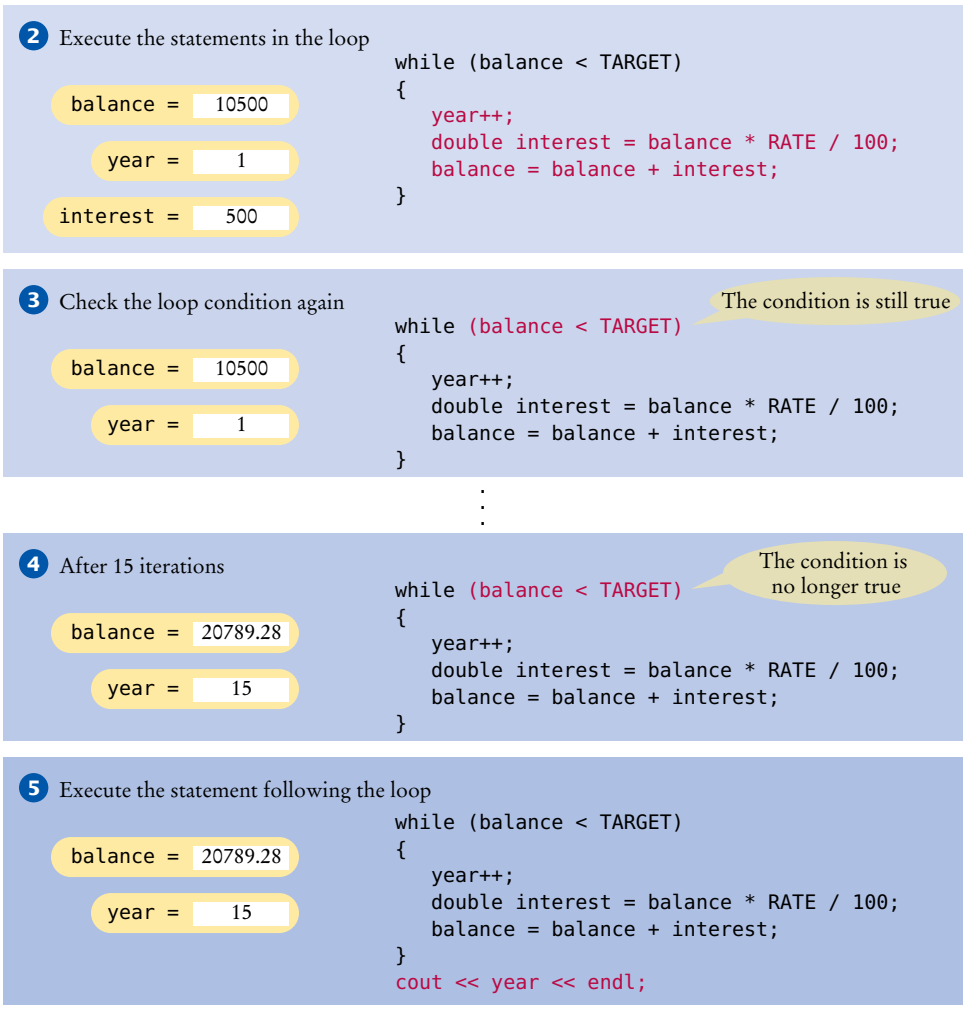


Figure 2 (continued) Execution of the `doublinv` Loop

Table 1 `while` Loop Examples

Loop	Output	Explanation

`i = 5;`
`while (i > 0)`
`{`
`  cout << i << " ";`
`  i--;`
`}`
	`5 4 3 2 1`	When `i` is 0, the loop condition is false, and the loop ends.

`i = 5;`
`while (i > 0)`
`{`
`  cout << i << " ";`
`  i++;`
`}`
	`5 6 7 8 9 10 11 ...`	The `i++` statement is an error causing an "infinite loop" (see Common Error 4.1).

Table 1 `while` Loop Examples

Loop	Output	Explanation

`i = 5;`
`while (i > 5)`
`{`
`  cout << i << " ";`
`  i--;`
`}`
	(No output)	The statement `i > 5` is false, and the loop is never executed.

`i = 5;`
`while (i < 0)`
`{`
`  cout << i << " ";`
`  i--;`
`}`
	(No output)	The programmer probably thought, "Stop when `i` is less than `0`". However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.2).

`i = 5;`
`while (i > 0) ;`
`{`
`  cout << i << " ";`
`  i--;`
`}`
	(No output, program does not terminate)	Note the semicolon before the `{`. This loop has an empty body. It runs forever, checking whether `i > 0` and doing nothing in the body.

Common Error 4.1

Infinite Loops

A very annoying loop error is an *infinite loop:* a loop that runs forever and can be stopped only by killing the program or restarting the computer. If there are output statements in the program, then line after line of output flashes by on the screen. Otherwise, the program just sits there and *hangs*, seeming to do nothing. On some systems, you can terminate a hanging program by hitting Ctrl + C. On others, you can close the window in which the program runs.

A common reason for infinite loops is forgetting to update the variable that controls the loop:

```
year = 1;
while (year <= 20)
{
   balance = balance * (1 + RATE / 100);
}
```

© ohiophoto/iStockphoto.

Like this hamster who can't stop running in the treadmill, an infinite loop never ends.

Here the programmer forgot to add a `year++` command in the loop. As a result, the year always stays at 1, and the loop never comes to an end.

Another common reason for an infinite loop is accidentally incrementing a counter that should be decremented (or vice versa). Consider this example:

```
year = 20;
while (year > 0)
{
```

```
   balance = balance * (1 + RATE / 100);
   year++;
}
```

The year variable really should have been decremented, not incremented. This is a common error because incrementing counters is so much more common than decrementing that your fingers may type the ++ on autopilot. As a consequence, year is always larger than 0, and the loop never ends. (Actually, year may eventually exceed the largest representable positive integer and *wrap around* to a negative number. Then the loop ends—of course, with a completely wrong result.)

Common Error 4.2

Don't Think "Are We There Yet?"

When doing something repetitive, most of us want to know when we are done. For example, you may think, "I want to get at least $20,000," and set the loop condition to

```
balance >= TARGET
```

But the while loop thinks the opposite: How long am I allowed to keep going? The correct loop condition is

```
while (balance < TARGET)
```

In other words: "Keep at it while the balance is less than the target."

When writing a loop condition, don't ask, "Are we there yet?" The condition determines how long the loop will keep going.

© MsSponge/iStockphoto.

Common Error 4.3

Off-by-One Errors

Consider our computation of the number of years that are required to double an investment:

```
int year = 0;
while (balance < TARGET)
{
   year++;
   double interest = balance * RATE / 100;
   balance = balance + interest;
}
cout << "The investment doubled after " << year << " years." << endl;
```

Should year start at 0 or at 1? Should you test for balance < TARGET or for balance <= TARGET? It is easy to be *off by one* in these expressions.

An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Some people try to solve **off-by-one errors** by randomly inserting +1 or -1 until the program seems to work—a terrible strategy. It can take a long time to compile and test all the various possibilities. Expending a small amount of mental effort is a real time saver.

Fortunately, off-by-one errors are easy to avoid, simply by working through a couple of test cases and using the information from the test cases to come up with a rationale for your decisions.

Should year start at 0 or at 1? Look at a scenario with simple values: an initial balance of $100 and an interest rate of 50 percent. After year 1, the balance is $150, and after year 2 it is

$225, or over $200. So the investment doubled after 2 years. The loop executed two times, incrementing year each time. Hence year must start at 0, not at 1.

year	balance
0	$100
1	$150
2	$225

In other words, the balance variable denotes the balance after the end of the year. At the outset, the balance variable contains the balance after year 0 and not after year 1.

Next, should you use a < or <= comparison in the test? If you want to settle this question with an example, you need to find a scenario in which the final balance is exactly twice the initial balance. This happens when the interest is 100 percent. The loop executes once. Now year is 1, and balance is exactly equal to `2 * INITIAL_BALANCE`. Has the investment doubled after one year? It has. Therefore, the loop should not execute again. If the test condition is `balance < TARGET`, the loop stops, as it should. If the test condition had been `balance <= TARGET`, the loop would have executed once more.

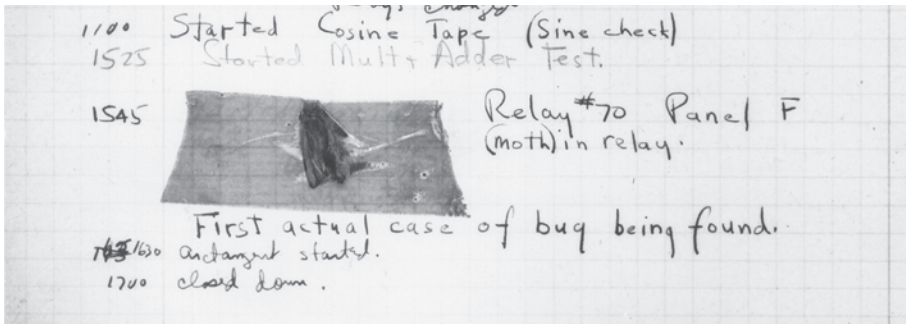
In other words, you keep adding interest while the balance *has not yet doubled*.

Computing & Society 4.1 The First Bug

According to legend, the first bug was found in the Mark II, a huge electromechanical computer at Harvard University. It really was caused by a bug—a moth was trapped in a relay switch.

Actually, from the note that the operator left in the log book next to the moth (see the photo), it appears as if the term "bug" had already been in active use at the time.

The pioneering computer scientist Maurice Wilkes wrote, "Somehow, at the Moore School and afterwards, one had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs."



Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Collection.

The First Bug

4.2 Problem Solving: Hand-Tracing

Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.

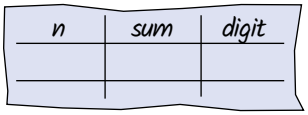
In Programming Tip 3.6, you learned about the method of hand-tracing. When you hand-trace code or pseudocode, you write the names of the variables on a sheet of paper, mentally execute each step of the code and update the variables.

It is best to have the code written or printed on a sheet of paper. Use a marker, such as a paper clip, to mark the current line. Whenever a variable changes, cross out the old value and write the new value below. When a program produces output, also write down the output in another column.

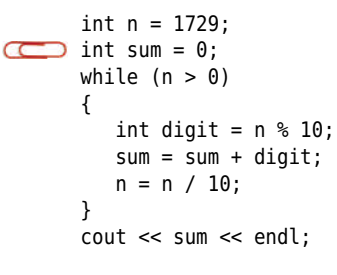
Consider this example. What value is displayed?

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```

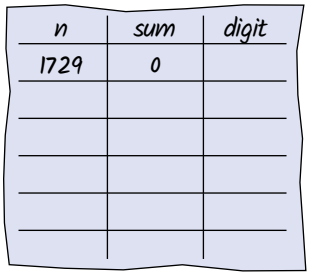
There are three variables: `n`, `sum`, and `digit`.



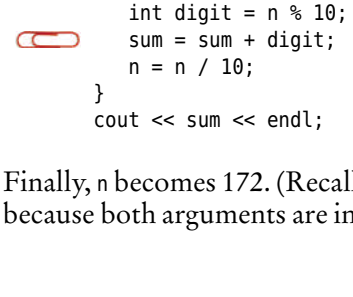
The first two variables are initialized with 1729 and 0 before the loop is entered.



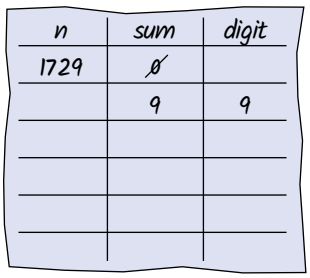
```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```



Because `n` is greater than zero, enter the loop. The variable `digit` is set to 9 (the remainder of dividing 1729 by 10). The variable `sum` is set to 0 + 9 = 9.



```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```



Finally, `n` becomes 172. (Recall that the remainder in the division 1729/10 is discarded because both arguments are integers.)

Cross out the old values and write the new ones under the old ones.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```

n	sum	digit
~~1729~~	~~0~~	
172	9	9

Now check the loop condition again.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```

Because `n` is still greater than zero, repeat the loop. Now `digit` becomes 2, `sum` is set to 9 + 2 = 11, and `n` is set to 17.

n	sum	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
17	11	2

Repeat the loop once again, setting `digit` to 7, `sum` to 11 + 7 = 18, and `n` to 1.

n	sum	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
~~17~~	~~11~~	~~2~~
1	18	7

Enter the loop for one last time. Now `digit` is set to 1, `sum` to 19, and `n` becomes zero.

n	sum	digit
~~1729~~	~~0~~	
~~172~~	~~9~~	~~9~~
~~17~~	~~11~~	~~2~~
~~1~~	~~18~~	~~7~~
0	19	1

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```

Because n *equals zero, this condition is not true.*

The condition `n > 0` is now false. Continue with the statement after the loop.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
cout << sum << endl;
```

n	sum	digit	output
~~1729~~	~~0~~		
~~172~~	~~9~~	~~9~~	
~~17~~	~~11~~	~~2~~	
~~1~~	~~18~~	~~7~~	
0	19	1	19

This statement is an output statement. The value that is output is the value of `sum`, which is 19.

Of course, you can get the same answer by just running the code. However, hand-tracing can give you an *insight* that you would not get if you simply ran the code. Consider again what happens in each iteration:

- We extract the last digit of `n`.
- We add that digit to `sum`.
- We strip the digit off `n`.

Hand-tracing can help you understand how an unfamiliar algorithm works.

In other words, the loop forms the sum of the digits in `n`. You now know what the loop does for any value of `n`, not just the one in the example. (Why would anyone want to form the sum of the digits? Operations of this kind are useful for checking the validity of credit card numbers and other forms of ID numbers—see Exercise P4.21.)

Hand-tracing can show errors in code or pseudocode.

Hand-tracing does not just help you understand code that works correctly. It is a powerful technique for finding errors in your code. When a program behaves in a way that you don't expect, get out a sheet of paper and track the values of the variables as you mentally step through the code.

You don't need a working program to do hand-tracing. You can hand-trace pseudocode. In fact, it is an excellent idea to hand-trace your pseudocode before you go to the trouble of translating it into actual code, to confirm that it works correctly.

4.3 The for Loop

The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
   cout << counter << endl;
   counter++; // Update the counter
}
```

Because this loop type is so common, there is a special form for it, called the `for` loop (see Syntax 4.2).

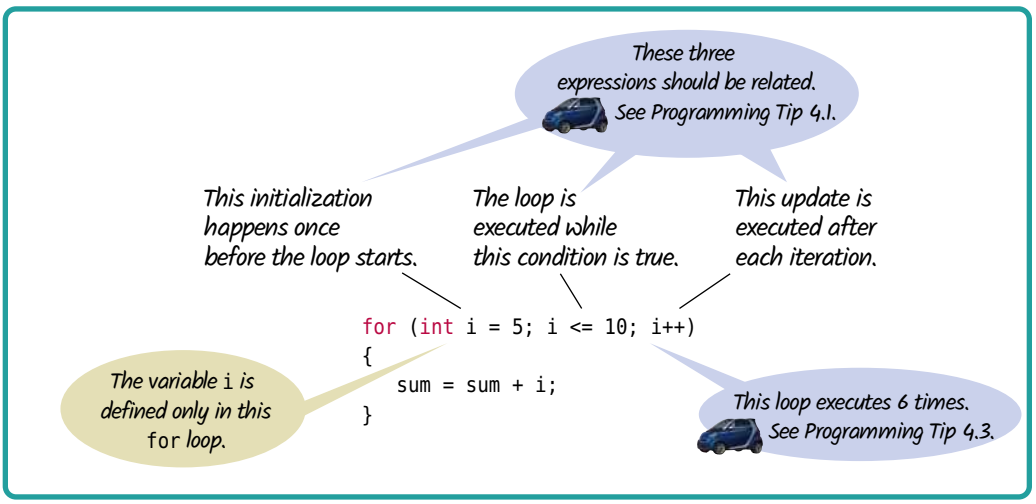
```
for (counter = 1; counter <= 10; counter++)
{
   cout << counter << endl;
}
```

© Enrico Fianchini/iStockphoto.

You can visualize the `for` *loop as an orderly sequence of steps.*

Some people call this loop *count-controlled*. In contrast, the `while` loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs (for example, when the balance reaches the target). Another commonly-used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times—ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.

Syntax 4.2 for Statement



The `for` loop neatly groups the initialization, condition, and update expressions together. However, it is important to realize that these expressions are *not* executed together (see Figure 3).

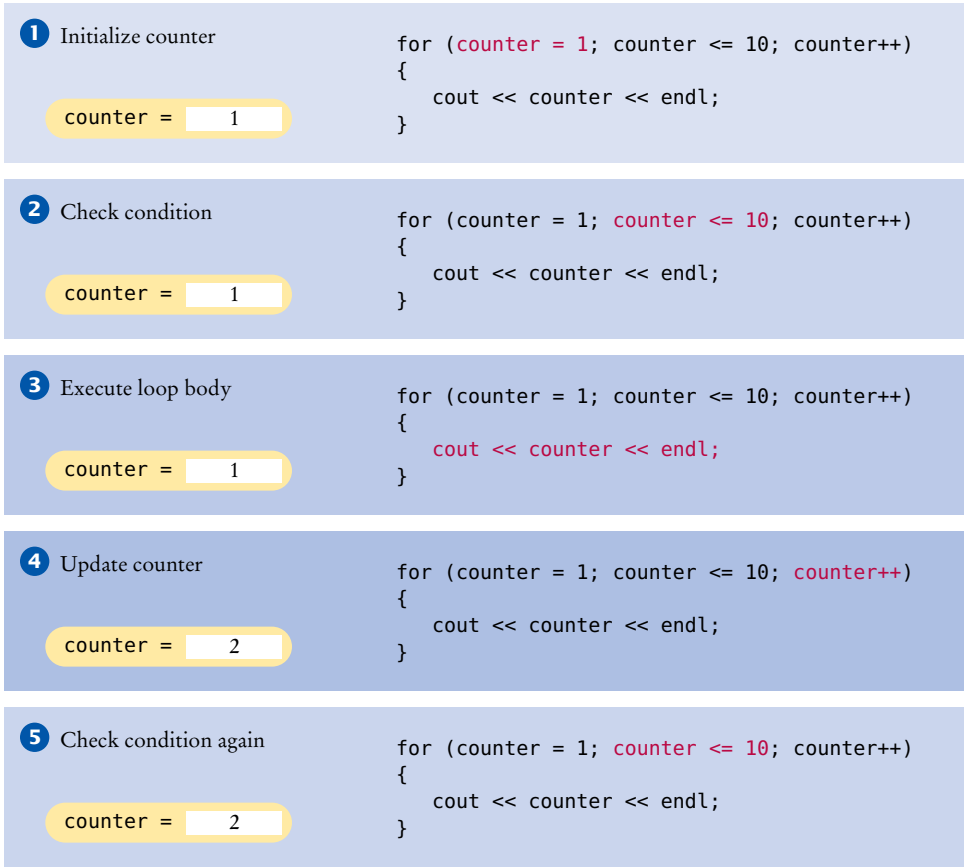


Figure 3 Execution of a `for` Loop

- The initialization is executed once, before the loop is entered. ❶
- The condition is checked before each iteration. ❷ ❺
- The update is executed after each iteration. ❹

A for loop can count down instead of up:

```
for (counter = 10; counter >= 0; counter--) . . .
```

The increment or decrement need not be in steps of 1:

```
for (counter = 0; counter <= 10; counter = counter + 2) . . .
```

So far, we assumed that the `counter` variable had already been defined before the `for` loop. Alternatively, you can define a variable in the loop initialization. Such a variable is defined *only* in the loop:

```
for (int counter = 1; counter <= 10; counter++)
{
   . . .
} // counter no longer defined here
```

See Table 2 for additional variations.

Table 2 `for` Loop Examples

Loop	Values of `i`	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 4.3.)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i = i + 2)`	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use `<` or `<=` instead of `!=` to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying `i`, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 ... until the last valid index of the string `str`	In the loop body, use the expression `str.substr(i, 1)` to get a string containing the `i`th character.

Here is a typical use of the `for` loop. We want to print the balance of our savings account over a period of years, as shown in this table:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The `for` loop pattern applies because the variable `year` starts at 1 and then moves in constant increments until it reaches the target:

```
for (int year = 1; year <= nyears; year++)
{
   Update balance.
   Print year and balance.
}
```

Here is the complete program. Figure 4 shows the corresponding flowchart.

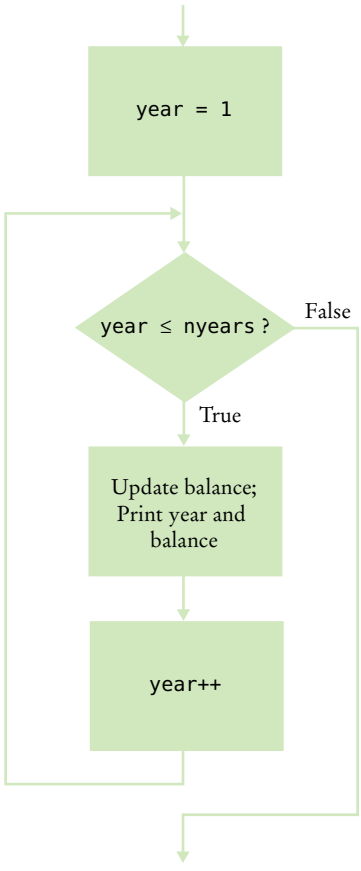


Figure 4 Flowchart of a `for` Loop

sec03/invtable.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   const double RATE = 5;
   const double INITIAL_BALANCE = 10000;
   double balance = INITIAL_BALANCE;
   int nyears;
   cout << "Enter number of years: ";
   cin >> nyears;

   cout << fixed << setprecision(2);
   for (int year = 1; year <= nyears; year++)
   {
      balance = balance * (1 + RATE / 100);
      cout << setw(4) << year << setw(10) << balance << endl;
   }

   return 0;
}
```

Program Run

```
Enter number of years: 10
   1  10500.00
   2  11025.00
   3  11576.25
   4  12155.06
   5  12762.82
   6  13400.96
   7  14071.00
   8  14774.55
   9  15513.28
  10  16288.95
```

Programming Tip 4.1

Use `for` Loops for Their Intended Purpose Only

A `for` loop is an *idiom* for a loop of a particular form. A value runs from the start to the end, with a constant increment or decrement.

The compiler won't check whether the initialization, condition, and update expressions are related. For example, the following loop is legal:

```
// Confusing—unrelated expressions
for (cout << "Inputs: "; cin >> x; sum = sum + x)
{
   count++;
}
```

However, programmers reading such a `for` loop will be confused because it does not match their expectations. Use a `while` loop for iterations that do not follow the `for` idiom.

Programming Tip 4.2

Choose Loop Bounds That Match Your Task

Suppose you want to print line numbers that go from 1 to 10. Of course, you will want to use a loop

```
for (int i = 1; i <= 10; i++)
```

The values for `i` are bounded by the relation $1 \leq i \leq 10$. Because there are $\leq$ on both bounds, the bounds are called **symmetric**.

When traversing the characters in a string, it is more natural to use the bounds

```
for (int i = 0; i < str.length(); i++)
```

In this loop, `i` traverses all valid positions in the string. You can access the `i`th character as `str.substr(i, 1)`. The values for `i` are bounded by $0 \leq i <$ `str.length()`, with a $\leq$ to the left and a $<$ to the right. That is appropriate, because `str.length()` is not a valid position. Such bounds are called **asymmetric**.

In this case, it is not a good idea to use symmetric bounds:

```
for (int i = 0; i <= str.length() - 1; i++) // Use < instead
```

The asymmetric form is easier to understand.

Programming Tip 4.3

Count Iterations

Finding the correct lower and upper bounds for an iteration can be confusing. Should you start at 0 or at 1? Should you use <= or < in the termination condition?

Counting the number of iterations is a very useful device for better understanding a loop. Counting is easier for loops with asymmetric bounds. The loop

```
for (int i = a; i < b; i++)
```

is executed `b - a` times. For example, the loop

```
for (int i = 0; i < 10; i++)
```

runs ten times, with values 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

The loop with symmetric bounds,

```
for (int i = a; i <= b; i++)
```

is executed `b - a + 1` times. That "+1" is the source of many programming errors.

For example,

```
for (int i = 0; i <= 10; i++)
```

runs 11 times. Maybe that is what you want; if not, start at 1 or use `< 10`.

One way to visualize this "+1" error is by looking at a fence. Each section has one fence post to the left, and there is a final post on the right of the last section. Forgetting to count the last value is often called a "fence post error".

How many posts do you need for a fence with four sections? It is easy to be "off by one" with problems such as this one.

4.4 The do Loop

Sometimes you want to execute the body of a loop at least once and perform the loop test after the body is executed. The `do` loop serves that purpose:

```
do
{
   statements
}
while (condition);
```

The `do` loop is appropriate when the loop body must be executed at least once.

The body of the `do` loop is executed first, then the condition is tested.

Some people call such a loop a *post-test* loop because the condition is tested after completing the loop body. In contrast, `while` and `for` loops are *pre-test* loops. In those loop types, the condition is tested before entering the loop body.

A typical example for such a loop is input validation. Suppose you ask a user to enter a value < 100. If the user didn't pay attention and entered a larger value, you ask again, until the value is correct. Of course, you cannot test the value until the user has entered it. This is a perfect fit for the `do` loop (see Figure 5):

```
int value;
do
{
   cout << "Enter a value < 100: ";
   cin >> value;
}
while (value >= 100);
```

Here, we assume that the user will enter an integer. You saw in Section 3.8 how to exit the program if a user fails to enter an integer. Special Topic 4.1 shows how to recover from such an error.

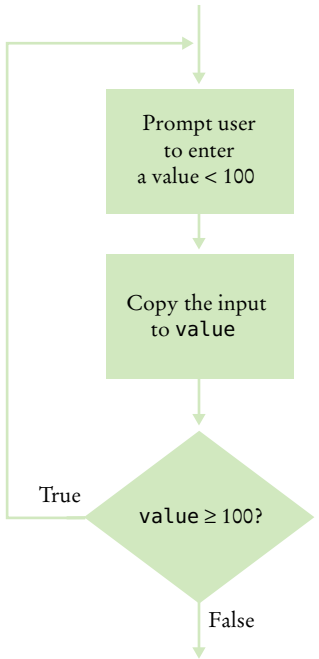


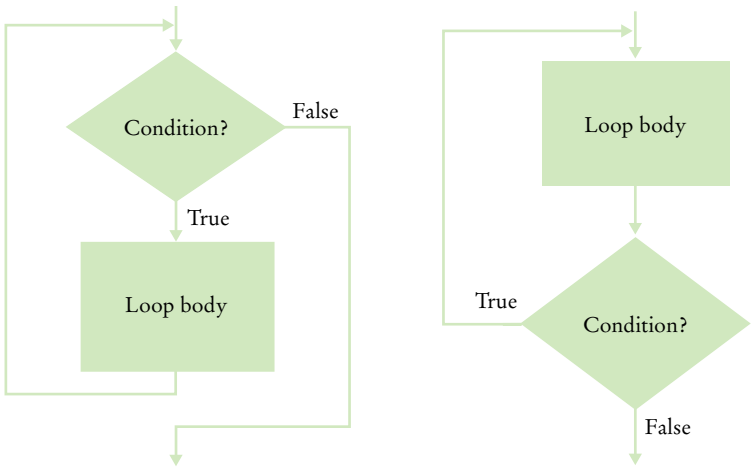
Figure 5 Flowchart of a `do` Loop

EXAMPLE CODE See sec04 of your companion code for a program that illustrates the use of the `do` loop for input validation.

Programming Tip 4.4

Flowcharts for Loops

In Section 3.5, you learned how to use flowcharts to visualize the flow of control in a program. There are two types of loops that you can include in a flowchart; they correspond to a `while` loop and a `do` loop in C++. They differ in the placement of the condition—either before or after the loop body.



As described in Section 3.5, you want to avoid "spaghetti code" in your flowcharts. For loops, that means that you never want to have an arrow that points inside a loop body.

4.5 Processing Input

In the following sections, you will learn how to read and process a sequence of input values.

4.5.1 Sentinel Values

A sentinel value denotes the end of a data set, but it is not part of the data.

Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence. Sometimes you are lucky and no input value can be zero. Then you can prompt the user to keep entering numbers, or `0` to finish the sequence. If zero is allowed but negative numbers are not, you can use –1 to indicate termination. A value that serves as a signal for termination is called a **sentinel**.

Let's put this technique to work in a program that computes the average of a set of salary values. In our sample program, we will use –1 as a sentinel. An employee would surely not work for a negative salary, but there may be volunteers who work for free.

Inside the loop, we read an input. If the input is not –1, we process it. In order to compute the average, we need the total sum of all salaries, and the number of inputs.

```
while (. . .)
{
   cin >> salary;
```

© Rhoberazzi/iStockphoto.

In the military, a sentinel guards a border or passage. In computer science, a sentinel value denotes the end of an input sequence or the border between input sequences.

```
   if (salary != -1)
   {
      sum = sum + salary;
      count++;
   }
}
```

We stay in the loop while the sentinel value is not detected.

```
while (salary != -1)
{
   . . .
}
```

There is just one problem: When the loop is entered for the first time, no data value has been read. Be sure to initialize `salary` with some value other than the sentinel:

```
double salary = 0; // Any value other than –1 will do
```

Alternatively, use a `do` loop

```
do
{
   . . .
}
while (salary != -1)
```

The following program reads inputs until the user enters the sentinel, and then computes and prints the average.

sec05/sentinel.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     double sum = 0;
8     int count = 0;
9     double salary = 0;
10    cout << "Enter salaries, -1 to finish: ";
11    while (salary != -1)
12    {
13       cin >> salary;
14       if (salary != -1)
15       {
16          sum = sum + salary;
17          count++;
18       }
19    }
20    if (count > 0)
21    {
22       double average = sum / count;
23       cout << "Average salary: " << average << endl;
24    }
25    else
26    {
27       cout << "No data" << endl;
28    }
29    return 0;
30 }
```

Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

4.5.2 Reading Until Input Fails

Numeric sentinels only work if there is some restriction on the input. In many cases, though, there isn't. Suppose you want to compute the average of a data set that may contain 0 or negative values. Then you cannot use 0 or –1 to indicate the end of the input.

In such a situation, you can read input data until input fails. As you have seen in Section 3.8, the condition

```
cin.fail()
```

is `true` if the preceding input has failed. For example, suppose that the input was read with these statements:

```
double value;
cin >> value;
```

If the user enters a value that is not a number (such as `Q`), then the input fails.

You can use a Boolean variable to control a loop. Set the variable before entering the loop, then set it to the opposite to leave the loop.

We now encounter an additional complexity. You only know that input failed after you have entered the loop and attempted to read it. To remember the failure, use a Boolean variable:

```
cout << "Enter values, Q to quit: ";
bool done = false;
while (!done)
{
   cin >> value;
   if (cin.fail())
   {
      done = true;
   }
   else
   {
      Process value.
   }
}
```

Some programmers dislike the introduction of a Boolean variable to control a loop. Special Topic 4.2 shows an alternative mechanism for leaving a loop. However, when reading input, there is an easier way. The expression

```
cin >> value
```

can be used in a condition. It evaluates to `true` if `cin` has *not* failed after reading `value`. Therefore, you can read and process a set of inputs with the following loop:

```
cout << "Enter values, Q to quit: ";
while (cin >> value)
{
   Process value.
}
```

This loop is suitable for processing a single sequence of inputs. You will learn more about reading inputs in Chapter 8.

EXAMPLE CODE See sec05 of your companion code for a program that uses a Boolean variable to control a loop.

Special Topic 4.1

Clearing the Failure State

When an input operation has failed, all further input operations also fail. Consider the `doloop.cpp` program in Section 4.4 in which a user is prompted to enter a value that is not negative. Suppose the user enters a value that is not an integer, such as the string `zero`.

Then the operation

```
cin >> value;
```

sets `cin` to the failed state. If you want to give the user another chance to enter a value, you need to *clear* the failed state, by calling the `clear` member function. You also need to read and discard the offending item:

```
cin.clear();
string item;
cin >> item;
```

Now the user can try again. Here is an improved version of the `do` loop:

```
do
{
   cout << "Enter a number >= 0: ";
   cin >> value;

   if (cin.fail())
   {
      // Clear the failed state
      cin.clear();
      // Read and discard the item
      string item;
      cin >> item;
      // Set to an invalid input
      value = -1;
   }
}
while (value < 0);
```

Note that we set `value` to an invalid input if the input was not a number, in order to enter the loop once more.

Here is another situation in which you need to clear the failed state. Suppose you read two number sequences, each of which has a letter as a sentinel. You read the first sequence:

```
cout << "Enter values, Q to quit.\n";
while (cin >> values)
{
   Process input.
}
```

Suppose the user has entered `30 10 5 Q`. The input of `Q` has caused the failure. Because only successfully processed characters are removed from the input, the `Q` character is still present. Clear the stream and read the sentinel into a string variable:

```
cin.clear();
string sentinel;
cin >> sentinel;
```

Now you can go on and read more inputs.

Special Topic 4.2

The Loop-and-a-Half Problem and the `break` Statement

Some programmers dislike loops that are controlled by a Boolean variable, such as:

```
bool done = false;
while (!done)
{
   cin >> value;
   if (cin.fail())
   {
      done = true;
   }
   else
   {
      Process value.
   }
}
```

The actual test for loop termination is in the middle of the loop, not at the top. This is called a **loop and a half** because one must go halfway into the loop before knowing whether one needs to terminate.

As an alternative, you can use the `break` reserved word:

```
while (true)
{
   cin >> value;
   if (cin.fail()) { break; }
   Process value.
}
```

The `break` statement breaks out of the enclosing loop, independent of the loop condition.

In the loop-and-a-half case, `break` statements can be beneficial. But it is difficult to lay down clear rules as to when they are safe and when they should be avoided. We do not use the `break` statement in this book.

Special Topic 4.3

Redirection of Input and Output

Consider the `sentinel.cpp` program that computes the average value of an input sequence. If you use such a program, then it is quite likely that you already have the values in a file, and it seems a shame that you have to type them all in again. The command line interface of your operating system provides a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. If you type

```
sentinel < numbers.txt
```

the program is executed. Its input instructions no longer expect input from the keyboard. All input commands get their input from the file `numbers.txt`. This process is called *input redirection*.

Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Input redirection is an excellent tool for testing programs. When you develop a program and fix its bugs, it is boring to keep entering the same input every time you run the program. Spend a few minutes putting the inputs into a file, and use redirection.

You can also redirect output. In this program, that is not terribly useful. If you run

```
sentinel < numbers.txt > output.txt
```

the file `output.txt` contains the input prompts and the output, such as

```
Enter salaries, -1 to finish:
Average salary: 15
```

However, redirecting output is obviously useful for programs that produce lots of output. You can print the file containing the output or edit it before you turn it in for grading.

4.6 Problem Solving: Storyboards

When you design a program that interacts with a user, you need to make a plan for that interaction. What information does the user provide, and in which order? What information will your program display, and in which format? What should happen when there is an error? When does the program quit?

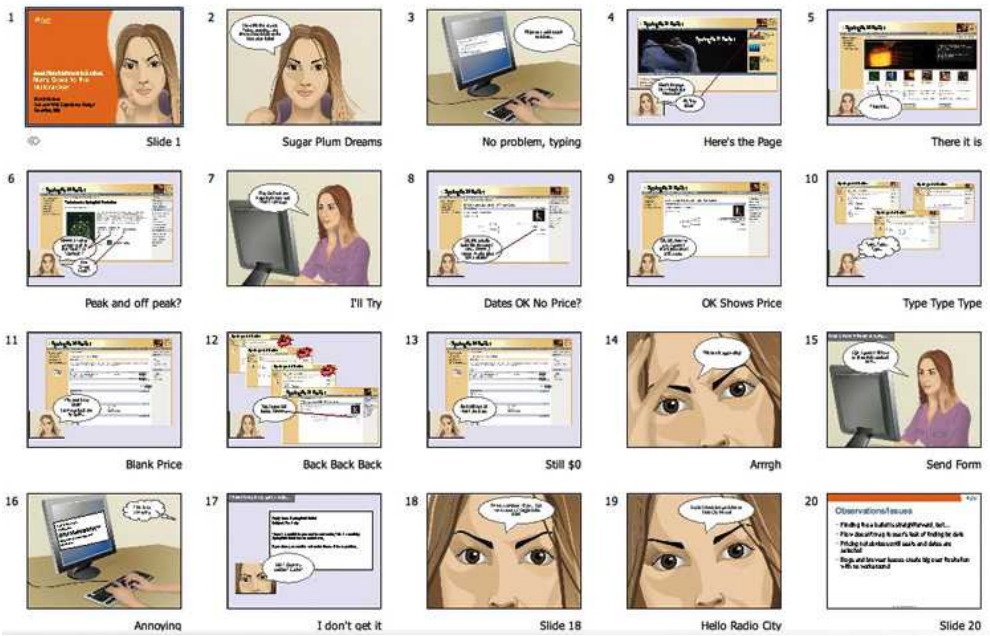
A storyboard consists of annotated sketches for each step in an action sequence.

This planning is similar to the development of a movie or a computer game, where *storyboards* are used to plan action sequences. A storyboard is made up of panels that show a sketch of each step. Annotations explain what is happening and note any special situations. Storyboards are also used to develop software—see Figure 6.

Developing a storyboard helps you understand the inputs and outputs that are required for a program.

Making a storyboard is very helpful when you begin designing a program. You need to ask yourself which information you need in order to compute the answers that the program user wants. You need to decide how to present those answers. These are important considerations that you want to settle before you design an algorithm for computing the answers.

Let's look at a simple example. We want to write a program that helps users with questions such as "How many tablespoons are in a pint?" or "How many inches are 30 centimeters?"



Courtesy of Martin Hardee.

Figure 6 Storyboard for the Design of a Web Application

What information does the user provide?

- The quantity and unit to convert from
- The unit to convert to

What if there is more than one quantity? A user may have a whole table of centimeter values that should be converted into inches.

What if the user enters units that our program doesn't know how to handle, such as ångström?

What if the user asks for impossible conversions, such as inches to gallons?

Let's get started with a storyboard panel. It is a good idea to write the user inputs in a different color. (Underline them if you don't have a color pen handy.)

Converting a Sequence of Values

```
What unit do you want to convert from? cm
What unit do you want to convert to? in
Enter values, terminated by zero
30
30 cm = 11.81 in
100
100 cm = 39.37 in
0
What unit do you want to convert from?
```

Allows conversion of multiple values

Format makes clear what got converted

The storyboard shows how we deal with a potential confusion. A user who wants to know how many inches are 30 centimeters may not read the first prompt carefully and specify inches. But then the output is "30 in = 76.2 cm", alerting the user to the problem.

The storyboard also raises an issue. How is the user supposed to know that "cm" and "in" are valid units? Would "centimeter" and "inches" also work? What happens when the user enters a wrong unit? Let's make another storyboard to demonstrate error handling.

Handling Unknown Units (needs improvement)

```
What unit do you want to convert from? cm
What unit do you want to convert to? inches
Sorry, unknown unit.
What unit do you want to convert to? inch
Sorry, unknown unit.
What unit do you want to convert to? grr
```

To eliminate frustration, it is better to list the units that the user can supply.

```
From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
```

No need to list the units again

We switched to a shorter prompt to make room for all the unit names. Exercise R4.24 explores a different alternative.

There is another issue that we haven't addressed yet. How does the user quit the program? The first storyboard gives the impression that the program will go on forever.

We can ask the user after seeing the sentinel that terminates an input sequence.

Exiting the Program

```
From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
Enter values, terminated by zero
30
30 cm = 11.81 in
0                                   Sentinel triggers the prompt to exit
More conversions (y, n)? n
(Program exits)
```

As you can see from this case study, a storyboard is essential for developing a working program. You need to know the flow of the user interaction in order to structure your program.

4.7 Common Loop Algorithms

In the following sections, we discuss some of the most common algorithms that are implemented as loops. You can use them as starting points for your loop designs.

4.7.1 Sum and Average Value

Computing the sum of a number of inputs is a very common task. Keep a *running total:* a variable to which you add each input value. Of course, the total should be initialized with 0.

```
double total = 0;
double input;
while (cin >> input)
{
   total = total + input;
}
```

To compute an average, count how many values you have, and divide by the count. Be sure to check that the count is not zero.

To compute an average, keep a total and a count of all values.

```
double total = 0;
int count = 0;
double input;
while (cin >> input)
{
   total = total + input;
   count++;
}
double average = 0;
if (count > 0) { average = total / count; }
```

4.7.2 Counting Matches

To count values that fulfill a condition, check all values and increment a counter for each match.

You often want to know how many values fulfill a particular condition. For example, you may want to count how many spaces are in a string. Keep a counter, a variable that is initialized with 0 and incremented whenever there is a match.

```
int spaces = 0;
for (int i = 0; i < str.length(); i++)
{
   string ch = str.substr(i, 1);
   if (ch == " ")
   {
      spaces++;
   }
}
```

For example, if `str` is the string `"My Fair Lady"`, `spaces` is incremented twice (when `i` is 2 and 7).

Note that the `spaces` variable is declared outside the loop. We want the loop to update a single variable. The `ch` variable is declared inside the loop. A separate variable is created for each iteration and removed at the end of each loop iteration.

This loop can also be used for scanning inputs. The following loop reads text, a word at a time, and counts the number of words with at most three letters:

```
int short_words = 0;
string input;
while (cin >> input)
{
   if (input.length() <= 3)
   {
      short_words++;
   }
}
```

© Hiob/iStockphoto.

In a loop that counts matches, a counter is incremented whenever a match is found.

4.7.3 Finding the First Match

If your goal is to find a match, exit the loop when the match is found

When you count the values that fulfill a condition, you need to look at all values. However, if your task is to find a match, then you can stop as soon as the condition is fulfilled.

Here is a loop that finds the first space in a string. Because we do not visit all elements in the string, a `while` loop is a better choice than a `for` loop:

```
bool found = false;
int position = 0;
while (!found && position < str.length())
{
   string ch = str.substr(position, 1);
   if (ch == " ") { found = true; }
```

© drflet/iStockphoto.

When searching, you look at items until a match is found.

```
   else { position++; }
}
```

If a match was found, then `found` is `true` and `position` is the index of the first match. If the loop did not find a match, then `found` remains `false` after the end of the loop.

Note that the variable `position` is declared *outside* the `while` loop because you may want to use it after the loop has finished.

4.7.4 Prompting Until a Match is Found

In the preceding example, we searched a string for a character that matches a condition. You can apply the same process for user input. Suppose you are asking a user to enter a positive value < 100. Keep asking until the user provides a correct input:

```
bool valid = false;
double input;
while (!valid)
{
   cout << "Please enter a positive value < 100: ";
   cin >> input;
   if (0 < input && input < 100) { valid = true; }
   else { cout << "Invalid input." << endl; }
}
```

Note that the variable `input` is declared *outside* the `while` loop because you will want to use the input after the loop has finished. If it had been declared inside the loop body, you would not be able to use it outside the loop.

4.7.5 Maximum and Minimum

To find the largest value, update the largest value seen so far whenever you see a larger one.

To compute the largest value in a sequence, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one:

```
double largest;
cin >> largest;
double input;
while (cin >> input)
{
   if (input > largest)
   {
      largest = input;
   }
}
```

This algorithm requires that there is at least one input.

To compute the smallest value, simply reverse the comparison:

```
double smallest;
cin >> smallest;
double input;
while (cin >> input)
{
```

© CEFutcher/iStockphoto.

To find the height of the tallest bus rider, remember the largest value so far, and update it whenever you see a taller one.

```
   if (input < smallest)
   {
      smallest = input;
   }
}
```

4.7.6 Comparing Adjacent Values

When processing a sequence of values in a loop, you sometimes need to compare a value with the value that just preceded it. For example, suppose you want to check whether a sequence of inputs contains adjacent duplicates such as `1 7 2 9 9 4 9`.

Now you face a challenge. Consider the typical loop for reading a value:

```
double input;
while (cin >> input)
{
   // Now input contains the current input
   . . .
}
```

To compare adjacent inputs, store the preceding input in a variable.

How can you compare the current input with the preceding one? At any time, input contains the current input, overwriting the previous one.

The answer is to store the previous input, like this:

```
double input;
double previous;
while (cin >> input)
{
   if (input == previous) { cout << "Duplicate input" << endl; }
   previous = input;
}
```

One problem remains. When the loop is entered for the first time, `previous` has not yet been set. You can solve this problem with an initial input operation outside the loop:

```
double input;
double previous;
cin >> previous;
while (cin >> input)
{
   if (input == previous) { cout << "Duplicate input" << endl; }
   previous = input;
}
```

When comparing adjacent values, store the previous value in a variable.

EXAMPLE CODE See sec07 of your companion code for a program that demonstrates common loop algorithms.

HOW TO 4.1

Writing a Loop

This How To walks you through the process of implementing a loop statement. We will illustrate the steps with the following example problem:

Problem Statement Read twelve temperature values (one for each month), and display the number of the month with the highest temperature. For example, according to `http://worldclimate.com`, the average maximum temperatures for Death Valley are (in order by month):

18.2 22.6 26.4 31.1 36.6 42.2
45.7 44.5 40.2 33.1 24.2 17.6

In this case, the month with the highest temperature (45.7 degrees Celsius) is July, and the program should display 7.

© Stevegeer/iStockphoto.

Step 1 Decide what work must be done *inside* the loop.

Every loop needs to do some kind of repetitive work, such as

- Reading another item.
- Updating a value (such as a bank balance or total).
- Incrementing a counter.

If you can't figure out what needs to go inside the loop, start by writing down the steps that you would take if you solved the problem by hand. For example, with the temperature reading problem, you might write

```
Read the first value.
Read the second value.
If the second value is higher than the first value
   Set highest temperature to the second value.
   Set highest month to 2.
Read the next value.
If the value is higher than the first and second values
   Set highest temperature to the value.
   Set highest month to 3.
Read the next value.
If the value is higher than the highest temperature seen so far
   Set highest temperature to the value.
   Set highest month to 4.
...
```

Now look at these steps and reduce them to a set of *uniform* actions that can be placed into the loop body. The first action is easy:

```
Read the next value.
```

The next action is trickier. In our description, we used tests "higher than the first", "higher than the first and second", "higher than the highest temperature seen so far". We need to settle on one test that works for all iterations. The last formulation is the most general.

Similarly, we must find a general way of setting the highest month. We need a variable that stores the current month, running from 1 to 12. Then we can formulate the second loop action:

If the value is higher than the highest temperature
  Set highest temperature to the value.
  Set highest month to current month.

Altogether our loop is

While . . .
  Read the next value.
  If the value is higher than the highest temperature
    Set the highest temperature to the value.
    Set highest month to current month.
  Increment current month.

Step 2 Specify the loop condition.

What goal do you want to reach in your loop? Typical examples are:

- Has the counter reached the final value?
- Have you read the last input value?
- Has a value reached a given threshold?

In our example, we simply want the current month to reach 12.

Step 3 Determine the loop type.

We distinguish between two major loop types. A *count-controlled* loop is executed a definite number of times. In an *event-controlled* loop, the number of iterations is not known in advance—the loop is executed until some event happens.

Count-controlled loops can be implemented as `for` statements. For other loops, consider the loop condition. Do you need to complete one iteration of the loop body before you can tell when to terminate the loop? In that case, choose a `do` loop. Otherwise, use a `while` loop.

Sometimes, the condition for terminating a loop changes in the middle of the loop body. In that case, you can use a Boolean variable that specifies when you are ready to leave the loop. Such a variable is called a **flag**. Follow this pattern:

```
bool done = false;
while (!done)
{
   Do some work.
   If all work has been completed
   {
      done = true;
   }
   else
   {
      Do more work.
   }
}
```

In summary,

- If you know in advance how many times a loop is repeated, use a `for` loop.
- If the loop body must be executed at least once, use a `do` loop.
- Otherwise, use a `while` loop.

In our example, we read 12 temperature values. Therefore, we choose a `for` loop.

Step 4 Set up variables for entering the loop for the first time.

List all variables that are used and updated in the loop, and determine how to initialize them. Commonly, counters are initialized with 0 or 1, totals with 0.

In our example, the variables are

current month
highest value
highest month

We need to be careful how we set up the highest temperature value. We can't simply set it to 0. After all, our program needs to work with temperature values from Antarctica, all of which may be negative.

A good option is to set the highest temperature value to the first input value. Of course, then we need to remember to only read in another 11 values, with the current month starting at 2.

We also need to initialize the highest month with 1. After all, in an Australian city, we may never find a month that is warmer than January.

Step 5 Process the result after the loop has finished.

In many cases, the desired result is simply a variable that was updated in the loop body. For example, in our temperature program, the result is the highest month. Sometimes, the loop computes values that contribute to the final result. For example, suppose you are asked to average the temperatures. Then the loop should compute the sum, not the average. After the loop has completed, you are ready to compute the average: divide the sum by the number of inputs.

Here is our complete loop:

```
Read value.
highest temperature = value
highest month = 1
For current month from 2 to 12
   Read next value.
   If the value is higher than the highest temperature
      Set highest temperature to the value.
      Set highest month to current month.
```

Step 6 Trace the loop with typical examples.

Hand-trace your loop code, as described in Section 4.2. Choose example values that are not too complex—executing the loop 3–5 times is enough to check for the most common errors. Pay special attention when entering the loop for the first and last time.

Sometimes, you want to make a slight modification to make tracing feasible. For example, when hand-tracing the investment doubling problem, use an interest rate of 20 percent rather than 5 percent. When hand-tracing the temperature loop, use 4 data values, not 12.

Let's say the data are 22.6 36.6 44.5 24.2. Here is the walkthrough:

current month	current value	highest month	highest value
		~~1~~	~~22.6~~
~~2~~	~~36.6~~	~~2~~	~~36.6~~
~~3~~	~~44.5~~	3	44.5
4	24.2		

The trace demonstrates that *highest month* and *highest value* are properly set.

Step 7 Implement the loop in C++.

Here's the loop for our example. Exercise E4.4 asks you to complete the program.

```
double highest_value;
cin >> highest_value;
int highest_month = 1;
for (int current_month = 2; current_month <= 12; current_month++)
{
   double next_value;
   cin >> next_value;
   if (next_value > highest_value)
   {
      highest_value = next_value;
      highest_month = current_month;
   }
}
cout << highest_month << endl;
```

WORKED EXAMPLE 4.1

Credit Card Processing

Learn how to use a loop to remove spaces from a credit card number. See your E-Text or visit `wiley.com/go/bclo3`.

© MorePixels/iStockphoto.

4.8 Nested Loops

When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

In Section 3.4, you saw how to nest two `if` statements. Similarly, complex iterations sometimes require a **nested loop**: a loop inside another loop statement. When processing tables, nested loops occur naturally. An outer loop iterates over all rows of the table. An inner loop deals with the columns in the current row.

In this section you will see how to print a table. For simplicity, we will simply print powers x^n, as in the table below.

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
...	...	...	...
10	100	1000	10000

Here is the pseudocode for printing the table:

```
Print table header.
For x from 1 to 10
   Print table row.
   Print endl.
```

How do you print a table row? You need to print a value for each exponent. This requires a second loop:

```
For n from 1 to 4
   Print x^n.
```

This loop must be placed inside the preceding loop.

The hour and minute displays in a digital clock are an example of nested loops. The hours loop 12 times, and for each hour, the minutes loop 60 times.

© davejkahn/iStockphoto.

We say that the inner loop is *nested* inside the outer loop (see Figure 7).

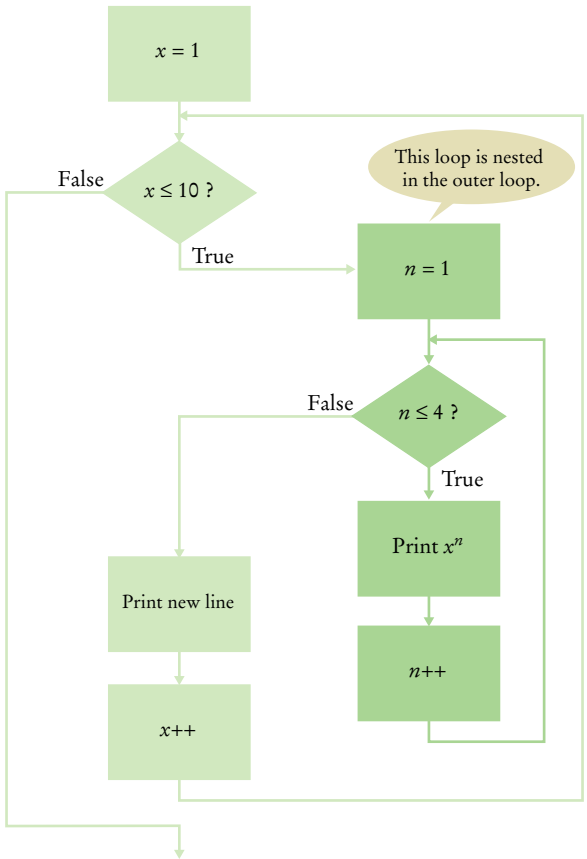


Figure 7 Flowchart of a Nested Loop

There are 10 rows in the outer loop. For each x, the program prints four columns in the inner loop. Thus, a total of $10 \times 4 = 40$ values are printed.

Following is the complete program. Note that we also use loops to print the table header. However, those loops are not nested.

sec08/powtable.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main()
{
   const int NMAX = 4;
   const double XMAX = 10;

   // Print table header

   for (int n = 1; n <= NMAX; n++)
   {
      cout << setw(10) << n;
   }
   cout << endl;
   for (int n = 1; n <= NMAX; n++)
   {
      cout << setw(10) << "x ";
   }
   cout << endl << endl;

   // Print table body

   for (double x = 1; x <= XMAX; x++)
   {
      // Print table row

      for (int n = 1; n <= NMAX; n++)
      {
         cout << setw(10) << pow(x, n);
      }
      cout << endl;
   }

   return 0;
}
```

Program Run

```
         1         2         3         4
         x         x         x         x

         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

Table 3 Nested Loop Examples

Nested Loops	Output	Explanation

`for (i = 1; i <= 3; i++)`
`{`
`  for (j = 1; j <= 4; j++)  { cout << "*"; }`
`  cout << endl;`
`}`
	`****`
	`****`
	`****`
		Prints 3 rows of 4 asterisks each.

`for (i = 1; i <= 4; i++)`
`{`
`  for (j = 1; j <= 3; j++) { cout << "*"; }`
`  cout << endl;`
`}`
	`***`
	`***`
	`***`
	`***`
		Prints 4 rows of 3 asterisks each.

`for (i = 1; i <= 4; i++)`
`{`
`  for (j = 1; j <= i; j++) { cout << "*"; }`
`  cout << endl;`
`}`
	`*`
	`**`
	`***`
	`****`
		Prints 4 rows of lengths 1, 2, 3, and 4.

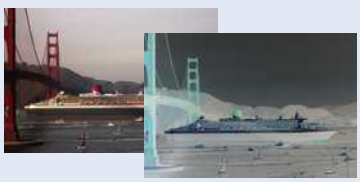
`for (i = 1; i <= 3; i++)`
`{`
`  for (j = 1; j <= 5; j++)`
`  {`
`  if (j % 2 == 0) { cout << "*"; }`
`  else { cout << "-"; }`
`  }`
`  cout << endl;`
`}`
	`-*-*-`
	`-*-*-`
	`-*-*-`
		Prints asterisks in even columns, dashes in odd columns.

`for (i = 1; i <= 3; i++)`
`{`
`  for (j = 1; j <= 5; j++)`
`  {`
`  if ((i + j) % 2 == 0) { cout << "*"; }`
`  else { cout << " "; }`
`  }`
`  cout << endl;`
`}`
	`* * *`
	` * *`
	`* * *`
		Prints a checkerboard pattern.

WORKED EXAMPLE 4.2

Manipulating the Pixels in an Image

Learn how to use nested loops for manipulating the pixels in an image. The outer loop traverses the rows of the image, and the inner loop accesses each pixel of a row. See your E-Text or visit `wiley.com/go/bclo3`.



Cay Horstmann.

4.9 Problem Solving: Solve a Simpler Problem First

When developing a solution to a complex problem, first solve a simpler task.

As you learn more about programming, the complexity of the tasks that you are asked to solve will increase. When you face a complex task, you should apply an important skill: simplifying the problem, and solving the simpler problem first.

This is a good strategy for several reasons. Usually, you learn something useful from solving the simpler task. Moreover, the complex problem can seem unsurmountable, and you may find it difficult to know where to get started. When you are successful with a simpler problem first, you will be much more motivated to try the harder one.

It takes practice and a certain amount of courage to break down a problem into a sequence of simpler ones. The best way to learn this strategy is to practice it. When you work on your next assignment, ask yourself what is the absolutely simplest part of the task that is helpful for the end result, and start from there. With some experience, you will be able to design a plan that builds up a complete solution as a manageable sequence of intermediate steps.

Let us look at an example. You are asked to arrange pictures, lining them up along the top edges, separating them with small gaps, and starting a new row whenever you run out of room in the current row.

National Gallery of Art (see Credits page for details.)

We use the `Picture` type of Worked Example 4.2. The `add` member function can be used to add one picture to another:

```
pic.add(pic2, x, y);
```

The second picture is added so that its top-left corner is at the given `x` and `y` position. The original picture grows to hold all pixels of the added picture. We will use this member function repeatedly in order to build up the result.

Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

Instead of tackling the entire assignment at once, here is a plan that solves a series of simpler problems.

1. Draw one picture.

2. Draw two pictures next to each other.

3. Draw two pictures with a gap between them.

4. Draw all pictures in a long row.

5. Draw a row of pictures until you run out of room, then put one more picture in the next row.

Let's get started with this plan.

1. The purpose of the first step is to become familiar with the `Picture` type. As it turns out, the pictures are in files `a.png` ... `t.png`. Let's load the first one:

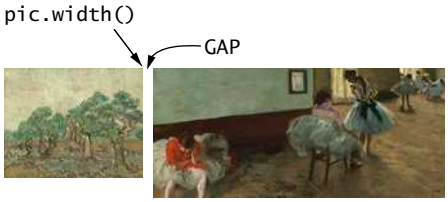
```
Picture pic("a.png");
pic.save("gallery.png");
```

That's enough to produce the picture.

2. Now let's put the next picture after the first. We need to position it at the rightmost *x*-coordinate of the preceding picture.

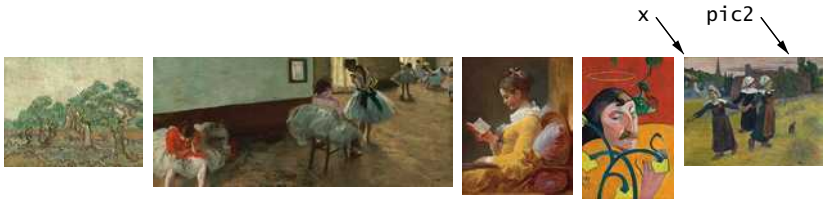

```
Picture pic("a.png");
Picture pic2("b.png");
pic.add(pic2, pic.width(), 0);
```

3. The next step is to separate the two by a small gap when the second is added:




```
const int GAP = 10;

Picture pic("a.png");
Picture pic2("b.png");
pic.add(pic2, pic.width() + GAP, 0);
```

4. Now let's put all pictures in a row. Read the pictures in a loop, and then put each picture to the right of the one that preceded it. In the loop, you need to track the *x*-coordinate at which the next image should be inserted.




```
const int GAP = 10;
const string names = "abcdefghijklmnopqrst";
const int PICTURES = names.length();

Picture pic("a.png");
int x = pic.width() + GAP;
for (int i = 1; i < PICTURES; i++)
{
   Picture pic2(names.substr(i, 1) + ".png");
   pic.add(pic2, x, 0);
   x = x + pic2.width() + GAP;
}
```

5. Of course, we don't want to have all pictures in a row. The right margin of a picture should not extend past `MAX_WIDTH`.

```
if (x + pic.width() < MAX_WIDTH)
{
   Place pic on current row.
}
else
{
   Place pic on next row.
}
```

If the image doesn't fit any more, then we need to put it on the next row, below all the pictures in the current row. We'll set a variable `max_y` to the maximum *y*-coordinate of all placed pictures, updating it whenever a new picture is placed:

```
if (pic2.height() > max_y)
{
   max_y = pic2.height();
}
```

The following statement places a picture on the next row:

```
pic.add(pic2, 0, max_y + GAP);
```

Now we have written complete programs for all preliminary stages. We know how to line up the pictures, how to separate them with gaps, how to find out when to start a new row, and where to start it.

EXAMPLE CODE See sec09 of your companion code for the preliminary stages of the gallery program.

With this knowledge, producing the final version is straightforward. Here is the program listing.

sec09/gallery6.cpp

```
#include "picture.h"
#include <string>

using namespace std;

int main()
{
   const int MAX_WIDTH = 720;
   const int GAP = 10;
   const string names = "abcdefghijklmnopqrst";
   const int PICTURES = names.length();

   // Read the first picture
   Picture pic("a.png");

   // x and y are the top-left corner of the next picture
   int x = pic.width() + GAP;
   int y = 0;

   // max_y is the largest y encountered so far
   int max_y = pic.height();
   for (int i = 1; i < PICTURES; i++)
   {
      // Read the next picture
      Picture pic2(names.substr(i, 1) + ".png");
      if (x + pic2.width() >= MAX_WIDTH) // The picture doesn't fit on the row
      {
         // Place the picture on the next row
         x = 0;
         y = max_y + GAP;
      }
```

```
32          pic.add(pic2, x, y);
33          // Update x and max_y
34          x = x + pic2.width() + GAP;
35          if (y + pic2.height() > max_y)
36          {
37             max_y = y + pic2.height();
38          }
39       }
40       pic.save("gallery.png");
41       return 0;
42    }
```

4.10 Random Numbers and Simulations

In a simulation, you use the computer to simulate an activity. You can introduce randomness by calling the random number generator.

A *simulation program* uses the computer to simulate an activity in the real world (or an imaginary one). Simulations are commonly used for predicting climate change, analyzing traffic, picking stocks, and many other applications in science and business. In the following sections, you will learn how to implement simulations that model phenomena with a degree of randomness.

4.10.1 Generating Random Numbers

Many events in the real world are difficult to predict with absolute precision, yet we can sometimes know the average behavior quite well. For example, a store may know from experience that a customer arrives every five minutes. Of course, that is an average—customers don't arrive in five minute intervals. To accurately model customer traffic, you want to take that random fluctuation into account. Now, how can you run such a simulation in the computer?

The C++ library has a *random number generator*, which produces numbers that appear to be completely random. Calling `rand()` yields a random integer between 0 and `RAND_MAX` (which is an implementation-dependent constant, typically, but not always, the largest valid `int` value). Call `rand()` again, and you get a different number. The `rand` function is declared in the `<cstdlib>` header.

The following program calls the `rand` function ten times.

sec10_01/random.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  int main()
7  {
8     for (int i = 1; i <= 10; i++)
9     {
10       int r = rand();
11       cout << r << endl;
12    }
13    return 0;
14 }
```

Program Run

```
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
118964142
```

Actually, the numbers are not completely random. They are drawn from sequences of numbers that don't repeat for a long time. These sequences are actually computed from fairly simple formulas; they just behave like random numbers. For that reason, they are often called **pseudorandom numbers**.

Try running the program again. You will get the *exact same output*! This confirms that the random numbers are generated by formulas. However, when running simulations, you don't always want to get the same results. To overcome this problem, specify a *seed* for the random number sequence. Every time you use a new seed, the random number generator starts generating a new sequence. The seed is set with the `srand` function. A simple value to use as a seed is the current time:

```
srand(time(0));
```

Simply make this call once in your program, before generating any random numbers. Then the random numbers will be different in every program run. Also include the `<ctime>` header that declares the `time` function.

If you get a compiler warning for the call to `srand`, use

```
srand(static_cast<int>(time(nullptr)));
```

instead, and make sure your compiler uses C++ 11.

4.10.2 Simulating Die Tosses

In actual applications, you need to transform the output from the random number generator into different ranges. For example, to simulate the throw of a die, you need random numbers between 1 and 6.

© ktsimage/iStockphoto.

Here is the general recipe for computing random integers between two bounds `a` and `b`. As you know from Programming Tip 4.3, there are `b - a + 1` values between `a` and `b`, including the bounds themselves. First compute `rand() % (b - a + 1)` to obtain a random value between `0` and `b - a`, then add `a`, yielding a random value between `a` and `b`:

```
int r = rand() % (b - a + 1) + a;
```

Here is a program that simulates the throw of a pair of dice.

sec10_02/dice.cpp

```
1  #include <iostream>
2  #include <string>
```

```
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
   srand(time(0));

   for (int i = 1; i <= 10; i++)
   {
      int d1 = rand() % 6 + 1;
      int d2 = rand() % 6 + 1;
      cout << d1 << " " << d2 << endl;
   }
   cout << endl;
   return 0;
}
```

Program Run

```
5 1
2 1
1 2
5 1
1 2
6 4
4 4
6 1
6 3
5 2
```

4.10.3 The Monte Carlo Method

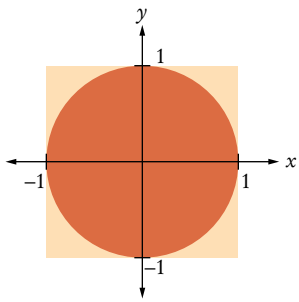
The Monte Carlo method is an ingenious method for finding approximate solutions to problems that cannot be precisely solved. (The method is named after the famous casino in Monte Carlo.) Here is a typical example: It is difficult to compute the number π, but you can approximate it quite well with the following simulation.

© timstarkey/iStockphoto.

Simulate shooting a dart into a square surrounding a circle of radius 1. That is easy: generate random x and y coordinates between –1 and 1.

If the generated point lies inside the circle, we count it as a *hit*. That is the case when $x^2 + y^2 \leq 1$. Because our shots are entirely random, we expect that the ratio of *hits/tries* is approximately equal to the ratio of the areas of the circle and the square,

that is, $\pi/4$. Therefore, our estimate for π is $4 \times hits/tries$. This method yields an estimate for π, using nothing but simple arithmetic.



To run the Monte Carlo simulation, you have to work a little harder with random number generation. When you throw a die, it has to come up with one of six faces. When throwing a dart, however, there are many possible outcomes. You must generate a *random floating-point number*.

First, generate the following value:

```
double r = rand() * 1.0 / RAND_MAX; // Between 0 and 1
```

The value `r` is a random floating-point value between 0 and 1. (You have to multiply by `1.0` to ensure that one of the operands of the `/` operator is a floating-point number. The division `rand() / RAND_MAX` would be an integer division—see Common Error 2.3.)

To generate a random value between –1 and 1, you compute:

```
double x = -1 + 2 * r; // Between –1 and 1
```

As `r` ranges from 0 to 1, `x` ranges from $-1 + 2 \times 0 = -1$ to $-1 + 2 \times 1 = 1$.

Here is the program that carries out the simulation.

sec10_03/montecarlo.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>

using namespace std;

int main()
{
   const int TRIES = 10000;

   srand(time(0));

   int hits = 0;
   for (int i = 1; i <= TRIES; i++)
   {
      double r = rand() * 1.0 / RAND_MAX; // Between 0 and 1
      double x = -1 + 2 * r; // Between –1 and 1
      r = rand() * 1.0 / RAND_MAX;
      double y = -1 + 2 * r;
      if (x * x + y * y <= 1) { hits++; }
   }
   double pi_estimate = 4.0 * hits / TRIES;
   cout << "Estimate for pi: " << pi_estimate << endl;
   return 0;
}
```

Program Run

```
Estimate for pi: 3.1504
```

Computing & Society 4.2 Digital Piracy

As you read this, you will have written a few computer programs and experienced firsthand how much effort it takes to write even the humblest of programs. Writing a real software product, such as a financial application or a computer game, takes a lot of time and money. Few people, and fewer companies, are going to spend that kind of time and money if they don't have a reasonable chance to make more money from their effort. (Actually, some companies give away their software in the hope that users will click on advertisements or upgrade to more elaborate paid versions. Other companies give away the software that enables users to read and use files but sell the software needed to create those files. Finally, there are individuals who donate their time, out of enthusiasm, and produce programs that you can copy freely.)

When selling software, a company must rely on the honesty of its customers. It is an easy matter for an unscrupulous person to make copies of computer programs without paying for them. In most countries that is illegal. Most governments provide legal protection, such as copyright laws and patents, to encourage the development of new products. Countries that tolerate widespread piracy have found that they have an ample cheap supply of foreign software, but no local manufacturers willing to design good software for their own citizens, such as word processors in the local script or financial programs adapted to the local tax laws.

When a mass market for software first appeared, vendors were enraged by the money they lost through piracy. They tried to fight back to ensure that only the legitimate owner could use the software by using various schemes, such as *dongles*—devices that must be attached to a printer port before the software will run. Legitimate users hated these measures. They paid for the software, but they had to suffer through inconveniences, such as having multiple dongles sticking out from their computer.

Because it is so easy and inexpensive to pirate software, and the chance of being found out is minimal, you have to make a moral choice for yourself. If a package that you would really like to have is too expensive for your budget, do you steal it, or do you stay honest and get by with a more affordable product?

Of course, piracy is not limited to software. The same issues arise for other digital products as well. You may have had the opportunity to obtain copies of songs or movies without payment. Or you may have been frustrated by a copy protection device on your music player that made it difficult for you to listen to songs that you paid for. Admittedly, it can be difficult to have a lot of sympathy for a musical ensemble whose publisher charges a lot of money for what seems to have been very little effort on their part, at least when compared to the effort that goes into designing and implementing a software package. Nevertheless, it seems only fair that artists and authors receive some compensation for their efforts.

How to pay artists, authors, and programmers fairly, without burdening honest customers, is an unsolved problem at the time of this writing, and many computer scientists are engaged in research in this area.

© RapidEye/iStockphoto.

CHAPTER SUMMARY

Explain the flow of execution in a loop.

- Loops execute a block of code repeatedly while a condition remains true.
- An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Use the technique of hand-tracing to analyze the behavior of a program.

- Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.
- Hand-tracing can help you understand how an unfamiliar algorithm works.
- Hand-tracing can show errors in code or pseudocode.

Use `for` loops for implementing counting loops.

- The `for` loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

Choose between the `while` loop and the `do` loop.

- The `do` loop is appropriate when the loop body must be executed at least once.

Implement loops that read sequences of input data.

- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a Boolean variable to control a loop. Set the variable before entering the loop, then set it to the opposite to leave the loop.
- Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Use the technique of storyboarding for planning user interactions.

- A storyboard consists of annotated sketches for each step in an action sequence.
- Developing a storyboard helps you understand the inputs and outputs that are required for a program.

Know the most common loop algorithms.

- To compute an average, keep a total and a count of all values.
- To count values that fulfill a condition, check all values and increment a counter for each match.
- If your goal is to find a match, exit the loop when the match is found.
- To find the largest value, update the largest value seen so far whenever you see a larger one.
- To compare adjacent inputs, store the preceding input in a variable.

Use nested loops to implement multiple levels of iteration.

- When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

Design programs that carry out complex tasks.

- When developing a solution to a complex problem, first solve a simpler task.
- Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

Apply loops to the implementation of simulations.

- In a simulation, you use the computer to simulate an activity. You can introduce randomness by calling the random number generator.

REVIEW EXERCISES

■ **R4.1** Given the variables

```
string stars = "*****";
string stripes = "=====";
```

what do these loops print?

a.
```
int i = 0;
while (i < 5)
{
   cout << stars.substr(0, i) << endl;
   i++;
}
```

b.
```
int i = 0;
while (i < 5)
{
   cout << stars.substr(0, i);
   cout << stripes.substr(i, 5 - i) << endl;
   i++;
}
```

c.
```
int i = 0;
while (i < 10)
{
   if (i % 2 == 0) { cout << stars << endl; }
   else { cout << stripes << endl; }
}
```

■ **R4.2** What do these loops print?

a.
```
int i = 0; int j = 10;
while (i < j) { cout << i + " " + j << endl; i++; j--; }
```

b.
```
int i = 0; int j = 10;
while (i < j) { cout << i + j << endl; i++; j++; }
```

■ **R4.3** What do these code snippets print?

a.
```
int result = 0;
for (int i = 1; i <= 10; i++) { result = result + i; }
cout << result << endl;
```

b.
```
int result = 1;
for (int i = 1; i <= 10; i++) { result = i - result; }
cout << result << endl;
```

c.
```
int result = 1;
for (int i = 5; i > 0; i--) { result = result * i; }
cout << result << endl;
```

d.
```
int result = 1;
for (int i = 1; i <= 10; i = i * 2) { result = result * i; }
cout << result << endl;
```

■ **R4.4** Write a `while` loop that prints

a. All squares less than `n`. For example, if `n` is 100, print 0 1 4 9 16 25 36 49 64 81.

b. All positive numbers that are divisible by 10 and less than `n`. For example, if `n` is 100, print 10 20 30 40 50 60 70 80 90

c. All powers of two less than `n`. For example, if `n` is 100, print 1 2 4 8 16 32 64.

■■ **R4.5** Write a loop that computes

a. The sum of all even numbers between 2 and 100 (inclusive).

b. The sum of all squares between 1 and 100 (inclusive).

c. The sum of all odd numbers between `a` and `b` (inclusive).

d. The sum of all odd digits of `n`. (For example, if `n` is 32677, the sum would be 3 + 7 + 7 = 17.)

■ **R4.6** Provide trace tables for these loops.

a.
```
int i = 0; int j = 10; int n = 0;
while (i < j) { i++; j--; n++; }
```

b.
```
int i = 0; int j = 0; int n = 0;
while (i < 10) { i++; n = n + i + j; j++; }
```

c.
```
int i = 10; int j = 0; int n = 0;
while (i > 0) { i--; j++; n = n + i - j; }
```

d.
```
int i = 0; int j = 10; int n = 0;
while (i != j) { i = i + 2; j = j - 2; n++; }
```

■ **R4.7** What do these loops print?

a. `for (int i = 1; i < 10; i++) { cout << i << " "; }`

b. `for (int i = 1; i < 10; i += 2) { cout << i << " "; }`

c. `for (int i = 10; i > 1; i--) { cout << i << " "; }`

d. `for (int i = 0; i < 10; i++) { cout << i << " "; }`

e. `for (int i = 1; i < 10; i = i * 2) { cout << i << " "; }`

f. `for (int i = 1; i < 10; i++) { if (i % 2 == 0) { cout << i << " "; } }`

■ **R4.8** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?

■■ **R4.9** What is an "off-by-one" error? Give an example from your own programming experience.

■ **R4.10** Write a program trace for the pseudocode in Exercise E4.6, assuming the input values are 4 7 –2 –5 0.

■■ **R4.11** Is the following code legal?

```
for (int i = 0; i < 10; i++)
{
   for (int i = 0; i < 10; i++)
   {
      cout << i << " ";
   }
   cout << endl;
}
```

What does it print? Is it good coding style? If not, how would you improve it?

■ **R4.12** How many iterations do the following loops carry out? Assume that `i` is not changed in the loop body.

a. `for (int i = 1; i <= 10; i++) . . .`

b. `for (int i = 0; i < 10; i++) . . .`

c. `for (int i = 10; i > 0; i--) . . .`

d. `for (int i = -10; i <= 10; i++) . . .`

e. `for (int i = 10; i >= 0; i++) . . .`

f. `for (int i = -10; i <= 10; i = i + 2) . . .`

g. `for (int i = -10; i <= 10; i = i + 3) . . .`

■■ **R4.13** Write pseudocode for a program that prints a calendar such as the following:

```
Su  M  T  W Th  F Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

■ **R4.14** Write pseudocode for a program that prints a Celsius/Fahrenheit conversion table such as the following:

```
Celsius | Fahrenheit
--------+-----------
      0 |         32
     10 |         50
     20 |         68
    ...         ...
    100 |        212
```

■ **R4.15** Write pseudocode for a program that reads a student record, consisting of the student's first and last name, followed by a sequence of test scores and a sentinel of –1. The program should print the student's average score. Then provide a trace table for this sample input:

```
Harry Morgan 94 71 86 95 -1
```

■■ **R4.16** Write pseudocode for a program that reads a sequence of student records and prints the total score for each student. Each record has the student's first and last name, followed by a sequence of test scores and a sentinel of –1. The sequence is terminated by the word `END`. Here is a sample sequence:

```
Harry Morgan 94 71 86 95 -1
Sally Lin 99 98 100 95 90 -1
END
```

Provide a trace table for this sample input.

■ **R4.17** Rewrite the following `for` loop into a `while` loop.

```
int s = 0;
for (int i = 1; i <= 10; i++)
{
   s = s + i;
}
```

■ **R4.18** Rewrite the following `do/while` loop into a `while` loop.

```
int n;
cin >> n;
double x = 0;
double s;
do
{
   s = 1.0 / (1 + n * n);
   n++;
   x = x + s;
}
while (s > 0.01);
```

■ **R4.19** Provide trace tables of the following loops.

a.
```
int s = 1;
int n = 1;
while (s < 10) { s = s + n; }
n++;
```

b.
```
int s = 1;
for (int n = 1; n < 5; n++) { s = s + n; }
```

c.
```
int s = 1;
int n = 1;
do
{
   s = s + n;
   n++;
}
while (s < 10 * n);
```

■ **R4.20** What do the following loops print? Work out the answer by tracing the code, not by using the computer.

a.
```
int s = 1;
for (int n = 1; n <= 5; n++)
{
   s = s + n;
   cout << s << " ";
}
```

b.
```
int s = 1;
for (int n = 1; s <= 10; cout << s << " ")
{
   n = n + 2;
   s = s + n;
}
```

c.
```
int s = 1;
int n;
for (n = 1; n <= 5; n++)
{
   s = s + n;
   n++;
}
cout << s << " " << n;
```

■ **R4.21** What do the following program segments print? Find the answers by tracing the code, not by using the computer.

a.
```
int n = 1;
for (int i = 2; i < 5; i++) { n = n + i; }
cout << n;
```

b.
```
int i;
double n = 1 / 2;
for (i = 2; i <= 5; i++) { n = n + 1.0 / i; }
cout << i;
```

c.
```
double x = 1;
double y = 1;
int i = 0;
do
{
   y = y / 2;
   x = x + y;
   i++;
```

```
   }
   while (x < 1.8);
   cout << i;
```

d.
```
double x = 1;
double y = 1;
int i = 0;
while (y >= 1.5)
{
   x = x / 2;
   y = x + y;
   i++;
}
cout << i;
```

■■ **R4.22** Give an example of a `for` loop where symmetric bounds are more natural. Give an example of a `for` loop where asymmetric bounds are more natural.

■ **R4.23** Add a storyboard panel for the conversion program in Section 4.6 that shows a scenario where a user enters incompatible units.

■ **R4.24** In Section 4.6, we decided to show users a list of all valid units in the prompt. If the program supports many more units, this approach is unworkable. Give a storyboard panel that illustrates an alternate approach: If the user enters an unknown unit, a list of all known units is shown.

■ **R4.25** Change the storyboards in Section 4.6 to support a menu that asks users whether they want to convert units, see program help, or quit the program. The menu should be displayed at the beginning of the program, when a sequence of values has been converted, and when an error is displayed.

■ **R4.26** Draw a flowchart for a program that carries out unit conversions as described in Section 4.6.

■■ **R4.27** In Section 4.7.5, the code for finding the largest and smallest input initializes the `largest` and `smallest` variables with an input value. Why can't you initialize them with zero?

■ **R4.28** What are nested loops? Give an example where a nested loop is typically used.

■■ **R4.29** The nested loops

```
for (int i = 1; i <= height; i++)
{
   for (int j = 1; j <= width; j++) { cout << "*"; }
   cout << endl;
}
```

display a rectangle of a given width and height, such as

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

Write a *single* `for` loop that displays the same rectangle.

■■ **R4.30** Suppose you design an educational game to teach children how to read a clock. How do you generate random values for the hours and minutes?

■■■ **R4.31** In a travel simulation, Harry will visit one of his friends that are located in three states. He has ten friends in California, three in Nevada, and two in Utah. How do

you produce a random number between 1 and 3, denoting the destination state, with a probability that is proportional to the number of friends in each state?

PRACTICE EXERCISES

■ **E4.1** Write programs with loops that compute

a. The sum of all even numbers between 2 and 100 (inclusive).

b. The sum of all squares between 1 and 100 (inclusive).

c. All powers of 2 from 2^0 up to 2^{20}.

d. The sum of all odd numbers between `a` and `b` (inclusive), where `a` and `b` are inputs.

e. The sum of all odd digits of an input. (For example, if the input is 32677, the sum would be 3 + 7 + 7 = 17.)

■■ **E4.2** Write programs that read a sequence of integer inputs and print

a. The smallest and largest of the inputs.

b. The number of even and odd inputs.

c. Cumulative totals. For example, if the input is 1 7 2 9, the program should print 1 8 10 19.

d. All adjacent duplicates. For example, if the input is 1 3 3 4 5 5 6 6 2, the program should print 3 5 6.

■■ **E4.3** Write programs that read a line of input as a string and print

a. Only the uppercase letters in the string.

b. Every second letter of the string.

c. The string, with all vowels replaced by an underscore.

d. The number of vowels in the string.

e. The positions of all vowels in the string.

■■ **E4.4** Complete the program in How To 4.1. Your program should read twelve temperature values and print the month with the highest temperature.

■■ **E4.5** Write a program that reads a set of floating-point values. Ask the user to enter the values, then print

- the average of the values.
- the smallest of the values.
- the largest of the values.
- the range, that is the difference between the smallest and largest.

Of course, you may only prompt for the values once.

■ **E4.6** Translate the following pseudocode for finding the minimum value from a set of inputs into a C++ program.

```
Set a Boolean variable "first" to true.
While another value has been read successfully
   If first is true
      Set the minimum to the value.
      Set first to false.
```

Else if the value is less than the minimum
Set the minimum to the value.
Print the minimum.

■■■ **E4.7** Translate the following pseudocode for randomly permuting the characters in a string into a C++ program.

Read a word.
Repeat word.length() times
Pick a random position i in the word, but not the last position.
Pick a random position j > i in the word.
Swap the letters at positions j and i.
Print the word.


© Anthony Rosenberg/iStockphoto.

To swap the letters, construct substrings as follows:

Then replace the string with

```
first + word.substr(j, 1) + middle + word.substr(i, 1) + last
```

■ **E4.8** Write a program that reads a word and prints each character of the word on a separate line. For example, if the user provides the input `"Harry"`, the program prints

```
H
a
r
r
y
```

■■ **E4.9** Write a program that reads a word and prints the word in reverse. For example, if the user provides the input `"Harry"`, the program prints

```
yrraH
```

■ **E4.10** Write a program that reads a word and prints the number of vowels in the word. For this exercise, assume that `a e i o u y` are vowels. For example, if the user provides the input `"Harry"`, the program prints `2 vowels`.

■■■ **E4.11** Write a program that reads a word and prints the number of syllables in the word. For this exercise, assume that syllables are determined as follows: Each sequence of vowels `a e i o u y`, except for the last `e` in a word, is a vowel. However, if that algorithm yields a count of 0, change it to 1. For example,

Word	Syllables
Harry	2
hairy	2
hare	1
the	1

■■■ **E4.12** Write a program that reads a word and prints all substrings, sorted by length. For example, if the user provides the input `"rum"`, the program prints

```
r
u
m
ru
um
rum
```

■■■ **E4.13** Write a program that reads a string and prints the most frequently occurring letter. If there are multiple letters that occur with the same maximum frequency, print them all. For example, if the word is `mississippi`, print `is` because `i` and `s` occur four times each, and no letter occurs more frequently.

■■■ **E4.14** Write a program that reads a string and prints the longest sequence of vowels. If there are multiple sequences of the same length, print them all. For example, if the word is `oiseau`, print `eau`, and if the word is `teakwood`, print `ea` and `oo`.

■■ **E4.15** Write a program that reads a sequence of words and then prints them in a box, with each word centered, like this:

```
+----------+
|  Hello   |
|   C++    |
|programmer|
+----------+
```

■ **E4.16** Write a program that prints all powers of 2 from 2^0 up to 2^{20}.

■■ **E4.17** Write a program that reads a number and prints all of its *binary digits:* Print the remainder `number % 2`, then replace the number with `number / 2`. Keep going until the number is `0`. For example, if the user provides the input 13, the output should be

```
1
0
1
1
```

■ **E4.18** Write a program that prints a multiplication table, like this:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 ...
10  20  30  40  50  60  70  80  90 100
```

■■ **E4.19** Write a program that reads an integer and displays, using asterisks, a filled and hollow square, placed next to each other. For example if the side length is 5, the program should display

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

■■ **E4.20** Write a program that reads an integer and displays, using asterisks, a filled diamond of the given side length. For example, if the side length is 4, the program should display

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

■■ **Media E4.21** Using the `Picture` type from Worked Example 4.2, write a program that reads in the file names of two pictures and superimposes them. The result is a picture whose width and height are the larger of the widths and heights of both pictures. In the area where both pictures have pixels, average the colors.

■■ **Media E4.22** Using the `Picture` type from Worked Example 4.2, write a program that reads in the file names of two pictures and superimposes them. The result is a picture whose width and height are the larger of the widths and heights of both pictures. In the area where both pictures have pixels, use the pixels from the first picture. However, when those pixels are green, use the pixels from the second picture.

■ **Media E4.23** Using the `Picture` type from Worked Example 4.2, apply a sunset effect to a picture, increasing the red value of each pixel by 30 percent (up to a maximum of 255).

■■ **Media E4.24** Using the `Picture` type from Worked Example 4.2, apply a "telescope" effect, turning all pixels black that are outside a circle. The center of the circle should be the image center, and the radius should be 40 percent of the width or height, whichever is smaller.

© Cay Horstmann

■ **Media E4.25** Using the `Picture` type from Worked Example 4.2, write a program that displays a checkerboard with 64 squares, alternating white and black.

■■ **Media E4.26** Reorganize the gallery program in Section 4.9 so that all images, starting with `a.png`, are added to an empty picture.

PROGRAMMING PROJECTS

■■ **P4.1** Enhance Worked Example 4.1 to check that the credit card number is valid. A valid credit card number will yield a result divisible by 10 when you:

Form the sum of all digits. Add to that sum every second digit, starting with the second digit from the right. Then add the number of digits in the second step that are greater than four. The result should be divisible by 10.

For example, consider the number 4012 8888 8888 1881. The sum of all digits is 89. The sum of the colored digits is 46. There are five colored digits larger than four, so the result is 140. 140 is divisible by 10 so the card number is valid.

■■ **P4.2** *Mean and standard deviation*. Write a program that reads a set of floating-point data values. Choose an appropriate mechanism for prompting for the end of the data set. When all values have been read, print out the count of the values, the average, and the standard deviation. The average of a data set $\{x_1, \ldots, x_n\}$ is $\bar{x} = \sum x_i / n$, where $\sum x_i = x_1 + \cdots + x_n$ is the sum of the input values. The standard deviation is

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

However, this formula is not suitable for the task. By the time the program has computed $\bar{x}$, the individual x_i are long gone. Until you know how to save these values, use the numerically less stable formula

$$s = \sqrt{\frac{\sum x_i^2 - \frac{1}{n}\left(\sum x_i\right)^2}{n - 1}}$$

You can compute this quantity by keeping track of the count, the sum, and the sum of squares as you process the input values.

■■ **P4.3** The *Fibonacci numbers* are defined by the sequence

$f_1 = 1$

$f_2 = 1$

$f_n = f_{n-1} + f_{n-2}$

Reformulate that as

```
fold1 = 1;
fold2 = 1;
fnew = fold1 + fold2;
```

© GlobalP/iStockphoto.

Fibonacci numbers describe the growth of a rabbit population.

After that, discard `fold2`, which is no longer needed, and set `fold2` to `fold1` and `fold1` to `fnew`. Repeat `fnew` an appropriate number of times.

Implement a program that computes the Fibonacci numbers in that way.

■■■ **P4.4** *Factoring of integers*. Write a program that asks the user for an integer and then prints out all its factors. For example, when the user enters 150, the program should print

```
2
3
5
5
```

■■■ **P4.5** *Prime numbers*. Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print

```
2
3
5
7
11
13
17
19
```

Recall that a number is a prime number if it is not divisible by any number except 1 and itself.

■ **P4.6** Following Section 4.9, develop a program that reads text and displays the average number of words in each sentence. Assume words are separated by spaces, and a

sentence ends when a word ends in a period. Start small and just print the first word. Then print the first two words. Then print all words in the first sentence. Then print the number of words in the first sentence. Then print the number of words in the first two sentences. Then print the average number of words in the first two sentences. At this time, you should have gathered enough experience that you can complete the program.

■ **P4.7** Following Section 4.9, develop a program that reads a string and removes all duplicates. For example, if the input is `Mississippi`, print `Misp`. Start small and just print the first letter. Then print the first letter and `true` if the letter is not duplicated elsewhere, `false` otherwise. (Look for it in the remaining string.) Next, do the same for the first two letters, and print out for each letter whether or not they occur in the substring before and after the letter. Try with a string like `oops`. Extend to all characters in the string. Have a look at the output when the input is `Mississippi`. Which characters should you not report? At this time, you should have gathered enough experience that you can complete the program.

■■■ **P4.8** *The game of Nim*. This is a well-known game with a number of variants. The following variant has an interesting winning strategy. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble loses.

You will write a program in which the computer plays against a human opponent. Generate a random integer between 10 and 100 to denote the initial size of the pile. Generate a random integer between 0 and 1 to decide whether the computer or the human takes the first turn. Generate a random integer between 0 and 1 to decide whether the computer plays *smart* or *stupid*. In stupid mode the computer simply takes a random legal value (between 1 and $n/2$) from the pile whenever it has a turn. In smart mode the computer takes off enough marbles to make the size of the pile a power of two minus 1—that is, 3, 7, 15, 31, or 63. That is always a legal move, except when the size of the pile is currently one less than a power of two. In that case, the computer makes a random legal move.

You will note that the computer cannot be beaten in smart mode when it has the first move, unless the pile size happens to be 15, 31, or 63. Of course, a human player who has the first turn and knows the winning strategy can win against the computer.

■■ **P4.9** *The Drunkard's Walk.* A drunkard in a grid of streets randomly picks one of four directions and stumbles to the next intersection, then again randomly picks one of four directions, and so on. You might think that on average the drunkard doesn't move very far because the choices cancel each other out, but that is actually not the case.

Represent locations as integer pairs (x, y). Implement the drunkard's walk over 100 intersections and print the beginning and ending location.

■■ **P4.10** *The Monty Hall Paradox*. Marilyn vos Savant described the following problem (loosely based on a game show hosted by Monty Hall) in a popular magazine: "Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?"

Ms. vos Savant proved that it is to your advantage, but many of her readers, including some mathematics professors, disagreed, arguing that the probability would not change because another door was opened.

Your task is to simulate this game show. In each iteration, randomly pick a door number between 1 and 3 for placing the car. Randomly have the player pick a door. Randomly have the game show host pick a door having a goat (but not the door that the player picked). Increment a counter for strategy 1 if the player wins by switching to the host's choice, and increment a counter for strategy 2 if the player wins by sticking with the original choice. Run 1,000 iterations and print both counters.

■■ **P4.11** *The Buffon Needle Experiment*. The following experiment was devised by Comte Georges-Louis Leclerc de Buffon (1707–1788), a French naturalist. A needle of length 1 inch is dropped onto paper that is ruled with lines 2 inches apart. If the needle drops onto a line, we count it as a *hit*. (See Figure 8.) Buffon conjectured that the quotient *tries*/*hits* approximates π.

For the Buffon needle experiment, you must generate two random numbers: one to describe the starting position and one to describe the angle of the needle with the x-axis. Then you need to test whether the needle touches a grid line.

Generate the *lower* point of the needle. Its x-coordinate is irrelevant, and you may assume its y-coordinate y_{low} to be any random number between 0 and 2. The angle α between the needle and the x-axis can be any value between 0 degrees and 180 degrees (π radians). The upper end of the needle has y-coordinate

$$y_{\text{high}} = y_{\text{low}} + \sin \alpha$$

The needle is a hit if y_{high} is at least 2, as shown in Figure 9.

Stop after 10,000 tries and print the quotient tries/hits. (This program is not suitable for computing the value of π. You need π in the computation of the angle.)

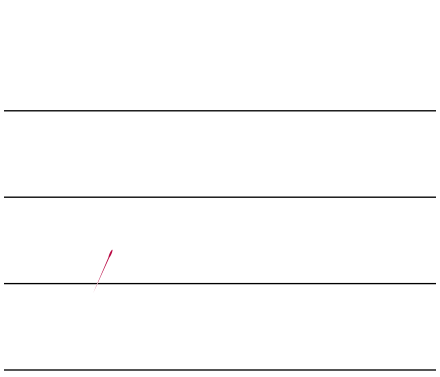


Figure 8 The Buffon Needle Experiment

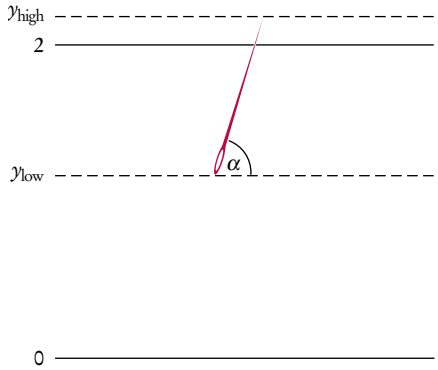


Figure 9 A Hit in the Buffon Needle Experiment

■ **P4.12** A simple random generator is obtained by the formula

$$r_{\text{new}} = (a \cdot r_{\text{old}} + b) \ \% \ m$$

and then setting r_{old} to r_{new}. If m is chosen as 2^{32}, then you can compute

$$r_{\text{new}} = a \cdot r_{\text{old}} + b$$

because the truncation of an overflowing result to the `int` type is equivalent to computing the remainder.

Write a program that asks the user to enter a value for r_{old}. (Such a value is often called a *seed*). Then print the first 100 random integers generated by this formula, using $a = 32310901$ and $b = 1729$.

■■ P4.13 In the 17th century, the discipline of probability theory got its start when a gambler asked a mathematician friend to explain some observations about dice games. Why did he, on average, win a bet that at least one six would appear when rolling a die four times? And why did he seem to lose a similar bet, getting at least one double-six when rolling a pair of dice 24 times?

Nowadays, it seems astounding that any person would roll a pair of dice 24 times in a row, and then repeat that many times over. Let's do that experiment on a computer instead. Simulate each game a million times and print out the wins and losses, assuming each bet was for $1.

■■ P4.14 Write a program that reads an integer n and a digit d between 0 and 9. Use one or more loops to count how many of the integers between 1 and n

- start with the digit d.
- end with the digit d.
- contain the digit d.

■ Business P4.15 Write a program that reads an initial investment balance and an interest rate, then prints the number of years it takes for the investment to reach one million dollars.

■■ Business P4.16 *Currency conversion*. Write a program that first asks the user to type today's exchange rate between U.S. dollars and Japanese yen, then reads U.S. dollar values and converts each to yen. Use 0 as a sentinel.

■■ Business P4.17 Write a program that first asks the user to type in today's exchange rate between U.S. dollars and Japanese yen, then reads U.S. dollar values and converts each to Japanese yen. Use 0 as the sentinel value to denote the end of dollar inputs. Then the program reads a sequence of yen amounts and converts them to dollars. The second sequence is terminated by another zero value.

© hatman12/iStockphoto.

■■ Business P4.18 Your company has shares of stock it would like to sell when their value exceeds a certain target price. Write a program that reads the target price and then reads the current stock price until it is at least the target price. Your program should use a `Scanner` to read a sequence of `double` values from standard input. Once the minimum is reached, the program should report that the stock price exceeds the target price.

■■ Business P4.19 Write an application to pre-sell a limited number of cinema tickets. Each buyer can buy as many as 4 tickets. No more than 100 tickets can be sold. Implement a program called `ticket_seller` that prompts the user for the desired number of tickets and then displays the number of remaining tickets. Repeat until all tickets have been sold, and then display the total number of buyers.

■■ Business P4.20 You need to control the number of people who can be in an oyster bar at the same time. Groups of people can always leave the bar, but a group cannot enter the bar if they would make the number of people in the bar exceed the maximum of 100 occupants. Write a program that reads the sizes of the groups that arrive or depart. Use negative numbers for departures. After each input, display the current number of occupants. As soon as the bar holds the maximum number of people, report that the bar is full and exit the program.

■■■ Business P4.21 *Credit Card Number Check.* The last digit of a credit card number is the *check digit*, which protects against transcription errors such as an error in a single digit or switching two digits. The following method is used to verify actual credit card numbers but, for simplicity, we will describe it for numbers with 8 digits instead of 16:

- Starting from the rightmost digit, form the sum of every other digit. For example, if the credit card number is 43589795, then you form the sum $5 + 7 + 8 + 3 = 23$.
- Double each of the digits that were not included in the preceding step. Add all digits of the resulting numbers. For example, with the number given above, doubling the digits, starting with the next-to-last one, yields 18 18 10 8. Adding all digits in these values yields $1 + 8 + 1 + 8 + 1 + 0 + 8 = 27$.
- Add the sums of the two preceding steps. If the last digit of the result is 0, the number is valid. In our case, $23 + 27 = 50$, so the number is valid.

Write a program that implements this algorithm. The user should supply an 8-digit number, and you should print out whether the number is valid or not. If it is not valid, you should print out the value of the check digit that would make the number valid.

■■ Engineering P4.22 In a predator-prey simulation, you compute the populations of predators and prey, using the following equations:

$$prey_{n+1} = prey_n \times (1 + A - B \times pred_n)$$

$$pred_{n+1} = pred_n \times (1 - C + D \times prey_n)$$

Here, A is the rate at which prey birth exceeds natural death, B is the rate of predation, C is the rate at which predator deaths exceed births without food, and D represents predator increase in the presence of food.

© Charles Gibson/iStockphoto.

Write a program that prompts users for these rates, the initial population sizes, and the number of periods. Then print the populations for the given number of periods. As inputs, try $A = 0.1$, $B = C = 0.01$, and $D = 0.00002$ with initial prey and predator populations of 1,000 and 20.

■■ Engineering P4.23 *Projectile flight*. Suppose a cannonball is propelled straight into the air with a starting velocity v_0. Any calculus book will state that the position of the ball after t seconds is $s(t) = -1/2gt^2 + v_0t$, where $g = 9.81$ m/s^2 is the gravitational force of the earth. No calculus book ever mentions why someone would want to carry out such an obviously dangerous experiment, so we will do it in the safety of the computer.

In fact, we will confirm the theorem from calculus by a simulation. In our simulation, we will consider how the ball moves in very short time intervals Δt. In a short time interval the velocity v is nearly constant, and we can compute the distance the ball moves as $\Delta s = v\Delta t$. In our program, we will simply set

```
const double DELTA_T = 0.01;
```

and update the position by

```
s = s + v * DELTA_T;
```

The velocity changes constantly—in fact, it is reduced by the gravitational force of the earth. In a short time interval, $\Delta v = -g\Delta t$, we must keep the velocity updated as

```
v = v - g * DELTA_T;
```

In the next iteration the new velocity is used to update the distance.

Now run the simulation until the cannonball falls back to the earth. Get the initial velocity as an input (100 m/sec is a good value). Update the position and velocity 100 times per second, but print out the position only every full second. Also printout the values from the exact formula $s(t) = -1/2gt^2 + v_0t$ for comparison.

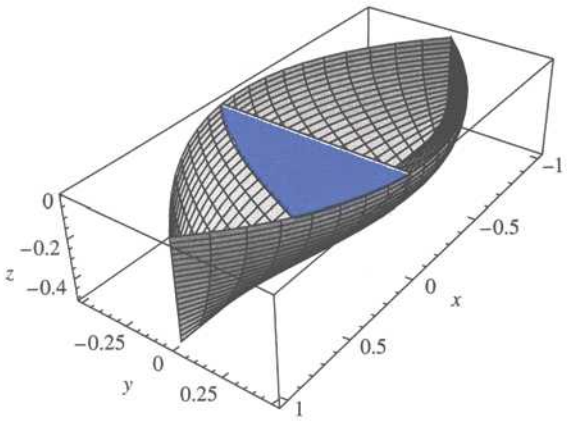
© MOF/iStockphoto.

Note: You may wonder whether there is a benefit to this simulation when an exact formula is available. Well, the formula from the calculus book is *not* exact. Actually, the gravitational force diminishes the farther the cannonball is away from the surface of the earth. This complicates the algebra sufficiently that it is not possible to give an exact formula for the actual motion, but the computer simulation can simply be extended to apply a variable gravitational force. For cannonballs, the calculus-book formula is actually good enough, but computers are necessary to compute accurate trajectories for higher-flying objects such as ballistic missiles.

▪▪▪ Engineering P4.24 A simple model for the hull of a ship is given by

$$|y| = \frac{B}{2}\left[1 - \left(\frac{2x}{L}\right)^2\right]\left[1 - \left(\frac{z}{T}\right)^2\right]$$

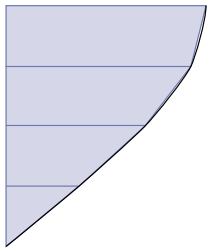
where B is the beam, L is the length, and T is the draft.



Courtesy of James P. Holloway, John Wiley & Sons, Inc.

(*Note:* There are two values of y for each x and z because the hull is symmetric from starboard to port.)

The cross-sectional area at a point x is called the "section" in nautical parlance. To compute it, let z go from 0 to $-T$ in n increments, each of size T/n. For each value of z, compute the value for y. Then sum the areas of trapezoidal strips. At right are the strips where $n = 4$.



Courtesy of James P. Holloway, John Wiley & Sons, Inc.

Write a program that reads in values for B, L, T, x, and n and then prints out the cross-sectional area at x.

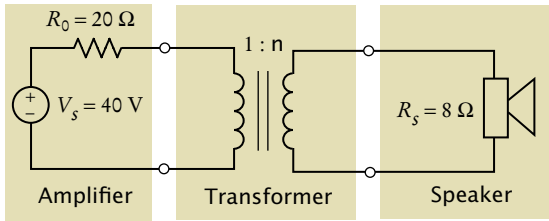
▪ Engineering P4.25 Radioactive decay of radioactive materials can be modeled by the equation $A = A_0 e^{-t(\log 2/h)}$, where A is the amount of the material at time t, A_0 is the amount at time 0, and h is the half-life.

© Snowleopard1/iStockphoto.

Technetium-99 is a radioisotope that is used in imaging of the brain. It has a half-life of 6 hours. Your program should display the relative amount A/A_0 in a patient body every hour for 24 hours after receiving a dose.

▪▪▪ Engineering P4.26 The photo at left shows an electric device called a "transformer". Transformers are often constructed by wrapping coils of wire around a ferrite core. The figure below illustrates a situation that occurs in various audio devices such as cell phones and music players. In this circuit, a transformer is used to connect a speaker to the output of an audio amplifier.

© zig4photo/iStockphoto.



The symbol used to represent the transformer is intended to suggest two coils of wire. The parameter n of the transformer is called the "turns ratio" of the transformer. (The number of times that a wire is wrapped around the core to form a coil is called the number of turns in the coil. The turns ratio is literally the ratio of the number of turns in the two coils of wire.)

When designing the circuit, we are concerned primarily with the value of the power delivered to the speakers—that power causes the speakers to produce the sounds we want to hear. Suppose we were to connect the speakers directly to the amplifier without using the transformer. Some fraction of the power available from the amplifier would get to the speakers. The rest of the available power would be lost in the amplifier itself. The transformer is added to the circuit to increase the fraction of the amplifier power that is delivered to the speakers.

The power, P_s, delivered to the speakers is calculated using the formula

$$P_s = R_s \left(\frac{nV_s}{n^2 R_0 + R_s} \right)^2$$

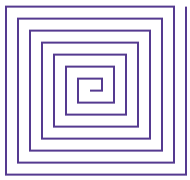
Write a C++ program that models the circuit shown and varies the turns ratio from 0.01 to 2 in 0.01 increments, then determines the value of the turns ratio that maximizes the power delivered to the speakers.

■■ **Media P4.27** Using the `Picture` type of Worked Example 4.2, it is easy to draw mathematical curves. Simply make a white picture and color pixels $(x, f(x))$ black. Draw the curve $f(x) = 0.00005x^3 - 0.03x^2 + 4x + 200$, where x ranges from 0 to 400.

■■■ **Media P4.28** Draw a picture of the "four-leaved rose" whose equation in polar coordinates is $r = \cos(2\theta)$. Let θ go from 0 to 2π in 1,000 steps. Each time, compute r and then compute the (x, y) coordinates from the polar coordinates by using the formula

$$x = 200r \cdot \cos(\theta) + 200, y = 200r \cdot \sin(\theta) + 200$$

■■■ **Media P4.29** Write a program that makes a picture of a spiral, such as the following:



■■■ **Media P4.30** Write a program that adds a frame to a given `.png` image, like this:

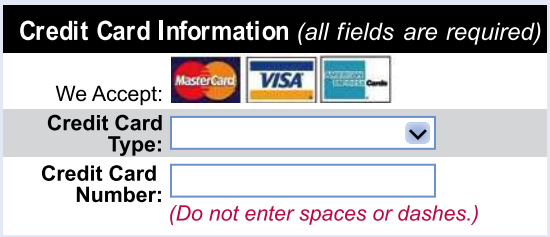
Cay Horstmann.

Add the image and a slightly larger white picture to a black picture.

WORKED EXAMPLE 4.1

Credit Card Processing

One of the minor annoyances of online shopping is that many web sites require you to enter a credit card without spaces or dashes, which makes double-checking the number rather tedious. How hard can it be to remove dashes or spaces from a string? Not hard at all, as this worked example shows.



Problem Statement Your task is to remove all spaces or dashes from a string `credit_card_number`. For example, if `credit_card_number` is `"4123-5678-9012-3450"`, then you should set it to `"4123567890123450"`.

Step 1 Decide what work must be done *inside* the loop.

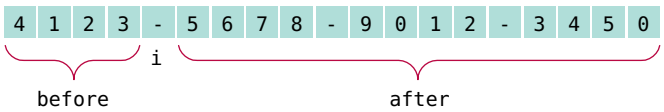
In the loop, we visit each character in turn. You can get the `i`th character as

```
string ch = credit_card_number.substr(i, 1);
```

If it is not a dash or space, we move on to the next character. If it is a dash or space, we remove the offending character.

```
i = 0
While ...
   Set ch to the ith character of credit_card_number.
   If ch is a space or dash
      Remove the character from credit_card_number.
   Else
      Increment i.
```

You may wonder how to remove a character from a string in C++. Here is the procedure for removing the character at position `i`: Take the substrings that end before `i` and start after `i`, and concatenate them.




```
string before = credit_card_number.substr(0, i);
string after = credit_card_number.substr(i + 1);
credit_card_number = before + after;
```

Note that we do *not* increment `i` after removing a character. For example, in the figure above, `i` was 4, and we removed the dash at position 4. The next time we enter the loop, we want to reexamine position 4 which now contains the character 5.

Step 2 Specify the loop condition.

We stay in the loop while the index `i` is a valid position. That is,

```
i < credit_card_number.length()
```

Step 3 Determine the loop type.

We don't know at the outset how often the loop is repeated. It depends on the number of dashes and spaces that we find. Therefore, we will choose a `while` loop. Why not a `do` loop? If we are given an empty string (because the user has not provided any credit card number at all), we do not want to enter the loop at all.

Step 4 Process the result after the loop has finished.

In this case, the result is simply the string.

Step 5 Trace the loop with typical examples.

The complete loop is

```
i = 0
While i < credit_card_number.length()
   ch = the ith character of credit_card_number.
   If ch is a space or dash
      Remove the character from credit_card_number.
   Else
      Increment i.
```

It is a bit tedious to trace a string with 20 characters, so we will use a shorter example:

credit_card_number	i	ch
4-56-7	0	4
4-56-7	1	-
456-7	1	5
456-7	2	6
456-7	3	-
4567	3	7

Step 6 Implement the loop in C++.

Here's the complete program, `worked_example_1/ccnumber.cpp`.

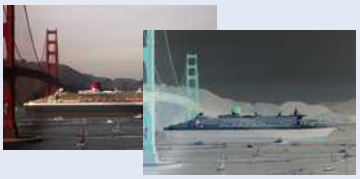
```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   string credit_card_number = "4123-5678-9012-3450";

   int i = 0;
   while (i < credit_card_number.length())
   {
      string ch = credit_card_number.substr(i, 1);
      if (ch == " " || ch == "-")
      {
         string before = credit_card_number.substr(0, i);
         string after = credit_card_number.substr(i + 1);
         credit_card_number = before + after;
      }
      else
      {
```

```
            i++;
        }
    }

    cout << credit_card_number << endl;

    return 0;
}
```

WORKED EXAMPLE 4.2

Manipulating the Pixels in an Image

A digital image is made up of *pixels*. Each pixel is a tiny square of a given color. In this Worked Example, we will use a `Picture` type that has member functions for loading an image and accessing its pixels.



Cay Horstmann.

Problem Statement Your task is to convert an image into its negative, turning white to black, cyan to red, and so on. The result is a negative image of the kind that old-fashioned film cameras used to produce.

Step 1 We provide you with a `Picture` type for manipulating images in the `.png` format. In order to use this type, you need to copy the files `picture.cpp`, `picture.h`, `lodepng.cpp`, and `lodepng.h` into the same folder as your source file, and you need to include them in the compilation. You can find these files in the `ch04/worked_example_2` folder of the companion code.

You also need to include the `picture.h` file into your source file, with the statement

```
#include "picture.h"
```

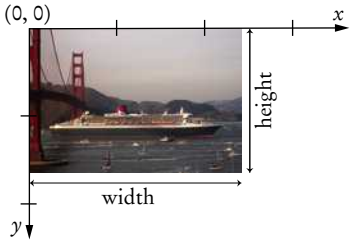
Note that the header file name, `picture.h`, is surrounded by quotation marks, not angle brackets, to indicate that you include a local file and not a system file.

To obtain a picture from a `.png` file, supply the file name like this:

```
Picture pic("queen-mary.png");
```

This statement defines a variable `pic` of type `Picture` that holds the pixels from the provided file.

Each pixel in the picture has an x- and y-coordinate, with $0 \le x <$ width and $0 \le y <$ height. The pixel with coordinates (0, 0) is in the top-left corner, and the y-axis points downward.



Each pixel has an RGB color value that is composed of the three primary colors: red, green, and blue. Each primary color amount is given as an integer between 0 (primary color not present) and 255 (maximum amount present). For example, a color with red value 255, green value 0, and blue value 255 is a bright purple color called magenta.

Step 2 Once you have a picture, you can call member functions to find out about the picture, or to modify it.

Here is what you can do:

- You can obtain the width and height of the picture by calling `pic.width()` and `pic.height()`.
- You can get the red, green, and blue values of a pixel at a particular location by calling `pic.red(x, y)`, `pic.green(x, y)`, and `pic.blue(x, y)`.
- You can set a pixel to any RGB color value by calling `pic.set(x, y, red, green, blue)`.
- You can save the changes to a file: `pic.save("result.png")`.
- You can add another picture to this picture—see Section 4.9.

Instead of loading a picture from an image file, you can also start with a monochromatic picture:

```
Picture all_magenta(300, 200, 255, 0, 255); // Specify width, height, and RGB color
```

Step 3 Now consider the task of converting an image into its negative.

Cay Horstmann.

A pixel is turned into its negative like this:

```
int red = pic.red(x, y);
int green = pic.green(x, y);
int blue = pic.blue(x, y);
pic.set(x, y, 255 - red, 255 - green, 255 - blue);
```

We want to apply this operation to each pixel in the image. To process all pixels, we can use one of the following two strategies:

For each row
  For each pixel in the row
    Process the pixel.

or

For each column
  For each pixel in the column
    Process the pixel.

Because our pixel class uses x/y coordinates to access a pixel, it turns out to be more natural to use the second strategy. (In Chapter 6, you will encounter two-dimensional arrays that are accessed with row/column coordinates. In that situation, use the first form.)

To traverse each column, the x-coordinate starts at `0`. Because there are `pic.width()` columns, we use the loop

```
for (int x = 0; x < pic.width(); x++)
```

Once a column has been fixed, we need to traverse all y-coordinates in that column, starting from `0`. There are `pic.height()` rows, so our nested loops are

```
for (int x = 0; x < pic.width(); x++)
{
   for (int y = 0; y < pic.height(); y++)
   {
      . . .
   }
}
```

The following program solves our image manipulation problem:

worked_example_2/negative.cpp

```cpp
#include "picture.h"

int main()
{
   Picture pic("queen-mary.png");

   for (int x = 0; x < pic.width(); x++)
   {
      for (int y = 0; y < pic.height(); y++)
      {
         int red = pic.red(x, y);
         int green = pic.green(x, y);
         int blue = pic.blue(x, y);
         pic.set(x, y, 255 - red, 255 - green, 255 - blue);
      }
   }
   pic.save("out.png");
   return 0;
}
```

CHAPTER 5

FUNCTIONS

© attator/iStockphoto.

CHAPTER GOALS

To be able to implement functions

To become familiar with the concept of parameter passing

To appreciate the importance of function comments

To develop strategies for decomposing complex tasks into simpler ones

To be able to determine the scope of a variable

To recognize when to use value and reference parameters

CHAPTER CONTENTS

Functions are a fundamental building block of C++ programs. A function packages a computation into a form that can be easily understood and reused. (The person in the photo is executing the function "make two cups of espresso".) In this chapter, you will learn how to design and implement your own functions. Using the process of stepwise refinement, you will be able to break up complex tasks into sets of cooperating functions.

5.1 Functions as Black Boxes

A function is a named sequence of instructions.

A **function** is a sequence of instructions with a name. You have already encountered several functions. For example, the function named `pow`, which was introduced in Chapter 2, contains instructions to compute a power x^y. Moreover, every C++ program has a function called `main`.

You *call* a function in order to execute its instructions. For example, consider the following program:

```
int main()
{
   double z = pow(2, 3);
   . . .
}
```

Arguments are supplied when a function is called. The return value is the result that the function computes.

By using the expression `pow(2, 3)`, `main` *calls* the `pow` function, asking it to compute the power 2^3. The `main` function is temporarily suspended. The instructions of the `pow` function execute and compute the result. The `pow` function *returns* its result (that is, the value 8) back to `main`, and the `main` function resumes execution (see Figure 1).

When another function calls the `pow` function, it provides "inputs", such as the expressions `2` and `3` in the call `pow(2, 3)`. These expressions are called **arguments**. This

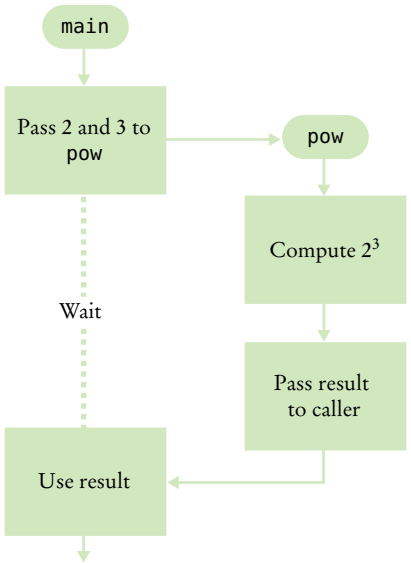


Figure 1 Execution Flow During a Function Call

terminology avoids confusion with other inputs, such as those provided by a human user. Similarly, the "output" that the `pow` function computes is called the **return value**.

Functions can have multiple arguments, but they have only one return value.

Note that the return value of a function is returned to the calling function, not displayed on the screen. For example, suppose your program contains a statement

```
double z = pow(2, 3);
```

When the `pow` function returns its result, the return value is stored in the variable `z`. If you want the value to be displayed, you need to add a statement such as `cout << z`.

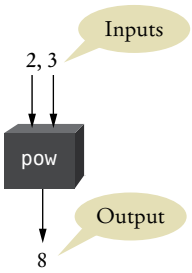


Figure 2
The `pow` Function as a Black Box

At this point, you may wonder how the `pow` function performs its job. For example, how does `pow(2, 3)` compute that 2^3 is 8? By multiplying $2 \times 2 \times 2$? With logarithms? Fortunately, as a user of the function, you *don't need to know* how the function is implemented. You just need to know the *specification* of the function: If you provide arguments x and y, the function returns x^y. Engineers use the term **black box** for a device with a given specification but unknown implementation. You can think of `pow` as a black box, as shown in Figure 2.

When you design your own functions, you will want to make them appear as black boxes to other programmers. Those programmers want to use your functions without knowing what goes on inside. Even if you are the only person working on a program, making each function into a black box pays off: there are fewer details that you need to keep in mind.

Although a thermostat is usually white, you can think of it as a black box. The input is the desired temperature, and the output is a signal to the heater or air conditioner.

5.2 Implementing Functions

In this section, you will learn how to implement a function from a given specification. We will use a very simple example: a function to compute the volume of a cube with a given side length.

When writing this function, you need to

- Pick a name for the function (`cube_volume`).
- Declare a variable for each argument (`double side_length`). These variables are called **parameter variables**.
- Specify the type of the return value (`double`).

The `cube_volume` function uses a given side length to compute the volume of a cube.

When defining a function, you provide a name for the function, a variable for each argument, and a type for the result.

Put all this information together to form the first line of the function's definition:

```
double cube_volume(double side_length)
```

Next, specify the **body** of the function: the statements that are executed when the function is called.

The volume of a cube of side length s is $s \times s \times s$. However, for greater clarity, our parameter variable has been called `side_length`, not s, so we need to compute `side_length * side_length * side_length`.

We will store this value in a variable called `volume`:

```
double volume = side_length * side_length * side_length;
```

In order to return the result of the function, use the `return` statement:

```
return volume;
```

The body of a function is enclosed in braces. Here is the complete function:

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Let's put this function to use. We'll supply a `main` function that calls the `cube_volume` function twice.

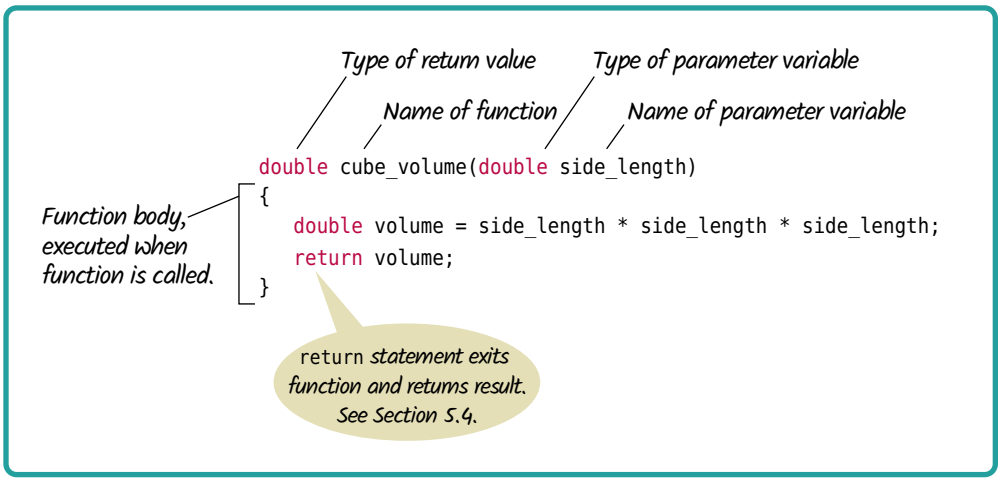
```
int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume " << result1 << endl;
   cout << "A cube with side length 10 has volume " << result2 << endl;

   return 0;
}
```

When the function is called with different arguments, the function returns different results. Consider the call `cube_volume(2)`. The argument `2` corresponds to the `side_length` parameter variable. Therefore, in this call, `side_length` is 2. The function computes `side_length * side_length * side_length`, or `2 * 2 * 2`. When the function is called with a different argument, say 10, then the function computes `10 * 10 * 10`.

Now we combine both functions into a test program. Because `main` calls `cube_volume`, the `cube_volume` function must be known before the `main` function is defined. This is easily achieved by placing `cube_volume` first and `main` last in the source file. (See Special Topic 5.1 for an alternative.)

The `return` *statement gives the function's result to the caller.*

Syntax 5.1 Function Definition



Here is the complete program. Note the comment that describes the behavior of the function. (Programming Tip 5.1 describes the format of the comment.)

sec02/cube.cpp

```
#include <iostream>

using namespace std;

/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}

int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume " << result1 << endl;
   cout << "A cube with side length 10 has volume " << result2 << endl;

   return 0;
}
```

Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

Programming Tip 5.1

Function Comments

Whenever you write a function, you should *comment* its behavior. Comments are for human readers, not compilers, and there is no universal standard for the layout of a function comment. In this book, we will use the following layout:

```
/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

Function comments explain the purpose of the function, the meaning of the parameter variables and return value, as well as any special requirements.

This particular documentation style is borrowed from the Java programming language. It is widely supported by C++ tools as well, for example by the Doxygen tool (`www.doxygen.org`).

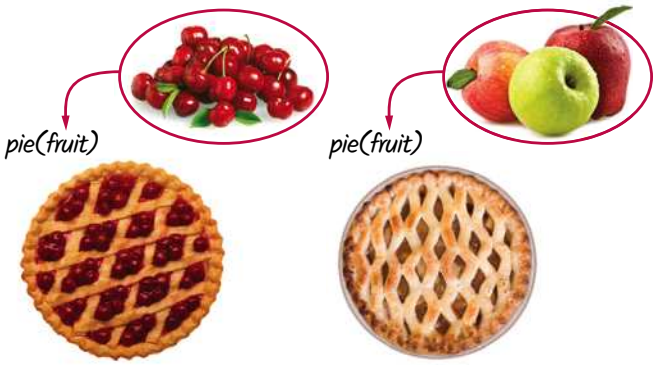
The comment starts with a `/**` delimiter. The first line of the comment describes the purpose of the function. Each `@param` clause describes a parameter variable and the `@return` clause describes the return value.

Note that the function comment does not document the implementation (*how* the function does what it does) but rather the design (*what* the function does, its inputs, and its results). The comment allows other programmers to use the function as a "black box".

5.3 Parameter Passing

Parameter variables hold the argument values supplied in the function call.

In this section, we examine the mechanism of passing arguments into functions. When a function is called, its **parameter variables** are created (Another commonly used term for a parameter variable is *formal parameter*.) In the function call, an expression is supplied for each parameter variable, called the **argument**. (Another commonly used term for this expression is *actual parameter*.) Each parameter variable is initialized with the value of the corresponding argument.



A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.

Consider the function call illustrated in Figure 3:

```
double result1 = cube_volume(2);
```

- The parameter variable `side_length` of the `cube_volume` function is created. ❶
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, `side_length` is set to 2. ❷
- The function computes the expression `side_length * side_length * side_length`, which has the value 8. That value is stored in the variable `volume`. ❸
- The function returns. All of its variables are removed. The return value is transferred to the *caller*, that is, the function calling the `cube_volume` function. ❹

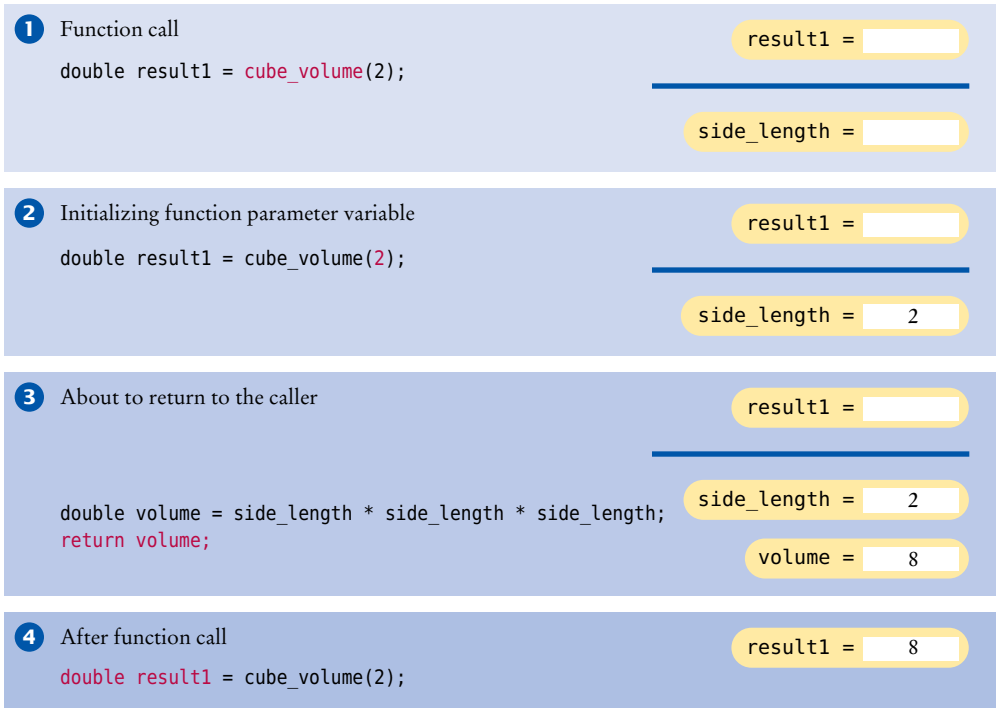


Figure 3 Parameter Passing

Now consider what happens in a subsequent call, `cube_volume(10)`. A new parameter variable is created. (Recall that the previous parameter variable was removed when the first call to `cube_volume` returned.) It is initialized with the argument 10, and the process repeats. After the second function call is complete, its variables are again removed.

Like any other variables, parameter variables can only be set to values of compatible types. For example, the `side_length` parameter variable of the `cube_volume` function has type `double`. It is valid to call `cube_volume(2.0)` or `cube_volume(2)`. In the latter call, the integer `2` is automatically converted to the `double` value `2.0`. However, a call `cube_volume("two")` is not legal.

Programming Tip 5.2

Do Not Modify Parameter Variables

In C++, a parameter variable is just like any other variable. You *can* modify the values of the parameter variables in the body of a function. For example,

```
int total_cents(int dollars, int cents)
{
   cents = dollars * 100 + cents; // Modifies parameter variable
   return cents;
}
```

However, many programmers find this practice confusing. It mixes the concept of a parameter (input to the function) with that of a variable (storage for a value). To avoid the confusion, simply introduce a separate variable:

```
int total_cents(int dollars, int cents)
{
   int result = dollars * 100 + cents;
   return result;
}
```

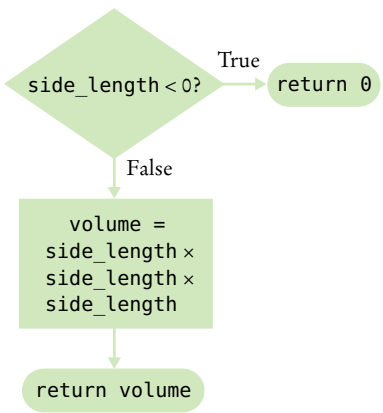
5.4 Return Values

The return statement terminates a function call and yields the function result.

You use the `return` statement to specify the result of a function. When the `return` statement is processed, the function exits *immediately*. This behavior is convenient for handling exceptional cases at the beginning:

```
double cube_volume(double side_length)
{
   if (side_length < 0) { return 0; }
   double volume = side_length * side_length * side_length;
   return volume;
}
```

If the function is called with a negative value for `side_length`, then the function returns `0` and the remainder of the function is not executed. (See Figure 4.)



© Tashka/iStockphoto.

Figure 4 A return Statement Exits a Function Immediately

In the preceding example, each `return` statement returned a constant or a variable. Actually, the `return` statement can return the value of any expression. Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

It is important that every branch of a function return a value. Consider the following incorrect function:

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length * side_length;
   } // Error
}
```

Suppose you call `cube_volume` with a negative value for the side length. Of course, you aren't supposed to call that, but it might happen as the result of a coding error. Because the `if` condition is not true, the `return` statement is not executed. However, the function must return *something*. Depending on the circumstances, the compiler might flag this as an error, or the function might return a random value. Protect against this problem by returning some safe value:

```
double cube_volume(double side_length)
{
   if (side_length >= 0)
   {
      return side_length * side_length * side_length;
   }
   return 0;
}
```

The last statement of every function ought to be a `return` statement. This ensures that *some* value gets returned when the function reaches the end.

EXAMPLE CODE See sec04 of your companion code for another implementation of the earthquake program that you saw in Section 3.3. Note that the `get_description` function has multiple `return` statements.

Common Error 5.1

Missing Return Value

A function always needs to return something. If the code of the function contains several branches, make sure that each one of them returns a value:

```
int sign(double x)
{
   if (x < 0) { return -1; }
   if (x > 0) { return 1; }
   // Error: missing return value if x equals 0
}
```

This function computes the sign of a number: –1 for negative numbers and +1 for positive numbers. If the argument is zero, however, no value is returned. Most compilers will issue a

warning in this situation, but if you ignore the warning and the function is ever called with an argument of 0, a random quantity will be returned.

Special Topic 5.1

Function Declarations

It is a compile-time error to call a function that the compiler does not know, just as it is an error to use an undefined variable. You can avoid this error if you define all functions before they are first used. First define lower-level helper functions, then the mid-level workhorse functions, and finally `main` in your program.

Some programmers prefer to list the `main` function first in their programs. If you share that preference, you need to learn how to declare the other functions at the top of the program. A **declaration** of a function lists the return type, function name, and parameter variables, but it contains no body:

```
double cube_volume(double side_length);
```

This is an advertisement that promises that the function is implemented elsewhere. It is easy to distinguish declarations from definitions: Declarations end in a semicolon, whereas definitions are followed by a `{...}` block. Declarations are also called **prototypes**.

In a function prototype, the names of the parameters are optional. You could also write

```
double cube_volume(double);
```

However, it is a good idea to include parameter names in order to document the purpose of each parameter.

The declarations of common functions such as `pow` are contained in header files. If you have a look inside `<cmath>`, you will find the declaration of `pow` and the other math functions.

Here is an alternate organization of the `cube.cpp` file:

```
#include <iostream>

using namespace std;

// Declaration of cube_volume
double cube_volume(double side_length);

int main()
{
   double result1 = cube_volume(2); // Use of cube_volume
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume " << result1 << endl;
   cout << "A cube with side length 10 has volume " << result2 << endl;
   return 0;
}

// Definition of cube_volume
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

If you prefer this approach, go ahead and use it in your programs. You just need to be aware of one drawback. Whenever you change the name of a function or one of the parameter types, you need to fix it in both places: in the declaration and in the definition.

HOW TO 5.1

Implementing a Function

A function is a computation that can be used multiple times with different parameters, either in the same program or in different programs. Whenever a computation is needed more than once, turn it into a function.

Problem Statement To illustrate this process, suppose that you are helping archaeologists who research Egyptian pyramids. You have taken on the task of writing a function that determines the volume of a pyramid, given its height and base length.

Step 1 Describe what the function should do.

Provide a simple English description, such as "Compute the volume of a pyramid whose base is a square."

Step 2 Determine the function's "inputs".

Make a list of *all* the parameters that can vary. It is common for beginners to implement functions that are overly specific. For example, you may know that the great pyramid of Giza, the largest of the Egyptian pyramids, has a height of 146 meters and a base length of 230 meters. You should *not* use these numbers in your calculation, even if the original problem only asked about the great pyramid. It is just as easy—and far more useful—to write a function that computes the volume of *any* pyramid.

© Holger Mette/iStockphoto.

In our case, the parameters are the pyramid's height and base length. At this point, we have enough information to document the function:

```
/**
   Computes the volume of a pyramid whose base is a square.
   @param height the height of the pyramid
   @param base_length the length of one side of the pyramid's base
   @return the volume of the pyramid
*/
```

Step 3 Determine the types of the parameter variables and the return value.

The height and base length can both be floating-point numbers. Therefore, we will choose the type `double` for both parameter variables. The computed volume is also a floating-point number, yielding a return type of `double`. Therefore, the function will be defined as

```
double pyramid_volume(double height, double base_length)
```

Step 4 Write pseudocode for obtaining the desired result.

In most cases, a function needs to carry out several steps to find the desired answer. You may need to use mathematical formulas, branches, or loops. Express your function in pseudocode.

An Internet search yields the fact that the volume of a pyramid is computed as

volume = 1/3 x height x base area

Since the base is a square, we have

base area = base length x base length

Using these two equations, we can compute the volume from the parameter variables.

Step 5 Implement the function body.

In our example, the function body is quite simple. Note the use of the `return` statement to return the result.

```
{
   double base_area = base_length * base_length;
   return height * base_area / 3;
}
```

Step 6 Test your function.

After implementing a function, you should test it in isolation. Such a test is called a **unit test**. Work out test cases by hand, and make sure that the function produces the correct results. For example, for a pyramid with height 9 and base length 10, we expect the area to be $1/3 \times 9 \times 100 = 300$. If the height is 0, we expect an area of 0.

```
int main()
{
   cout << "Volume: " << pyramid_volume(9, 10) << endl;
   cout << "Expected: 300";
   cout << "Volume: " << pyramid_volume(0, 10) << endl;
   cout << "Expected: 0";
   return 0;
}
```

The output confirms that the function worked as expected:

```
Volume: 300
Expected: 300
Volume: 0
Expected: 0
```

EXAMPLE CODE See how_to_1 of your companion code for a program that calculates a pyramid's volume.

WORKED EXAMPLE 5.1

Generating Random Passwords

This Worked Example creates a function that generates passwords of a given length with at least one digit and one special character. See your E-Text or visit `wiley.com/go/bclo3`.

WORKED EXAMPLE 5.2

Using a Debugger

Learn how to use a debugger to find errors in a program. See your E-Text or visit `wiley.com/go/bclo3`.

5.5 Functions Without Return Values

Use a return type of `void` to indicate that a function does not return a value.

Sometimes, you need to carry out a sequence of instructions that does not yield a value. If that instruction sequence occurs multiple times, you will want to package it into a function. In C++, you use the return type `void` to indicate the absence of a return value.

Here is a typical example. Your task is to print a string in a box, like this:

```
-------
!Hello!
-------
```

© jgroup/iStockphoto.

A `void` function returns no value, but it can produce output.

However, different strings can be substituted for `Hello`. A function for this task can be declared as follows:

```
void box_string(string str)
```

Now you develop the body of the function in the usual way, by formulating a general method for solving the task.

n = the length of the string
Print a line that contains the - character n + 2 times.
Print a line containing the string, surrounded with a ! to the left and right.
Print another line containing the - character n + 2 times.

Here is the function implementation:

```
/**
   Prints a string in a box.
   @param str the string to print
*/
void box_string(string str)
{
   int n = str.length();
   for (int i = 0; i < n + 2; i++) { cout << "-"; }
   cout << endl;
   cout << "!" << str << "!" << endl;
   for (int i = 0; i < n + 2; i++) { cout << "-"; }
   cout << endl;
}
```

This function doesn't compute any value. It performs some actions and then returns to the caller. Note that there is no need for a `return` statement.

Because there is no return value, you cannot use `box_string` in an expression. You can call

```
box_string("Hello");
```

but not

```
result = box_string("Hello"); // Error: box_string doesn't return a result.
```

If you want to return from a `void` function before reaching the end, you use a `return` statement without a value. For example,

```
void box_string(string str)
{
   int n = str.length();
   if (n == 0)
   {
      return; // Return immediately
   }
   . . .
}
```

EXAMPLE CODE See sec05 of your companion code for a program that demonstrates the `box_string` function.

5.6 Problem Solving: Reusable Functions

Eliminate replicated code or pseudocode by defining a function.

You have used many functions from the C++ standard library. These functions have been provided as a part of standard C++ so that programmers need not recreate them. Of course, the C++ library doesn't cover every conceivable need. You will often be able to save yourself time by designing your own functions that can be used for multiple problems.

When you write nearly identical code or pseudocode multiple times, either in the same program or in separate programs, consider introducing a function. Here is a typical example of code replication:

```
int hours;
do
{
   cout << "Enter a value between 0 and 23: ";
   cin >> hours;
}
while (hours < 0 || hours > 23);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
}
while (minutes < 0 || minutes > 59);
```

This program segment reads two variables, making sure that each of them is within a certain range. It is easy to extract the common behavior into a function:

```
/**
   Prompts a user to enter a value up to a given maximum until the user
   provides a valid input.
   @param high the largest allowable input
   @return the value provided by the user (between 0 and high, inclusive)
*/
int read_int_up_to(int high)
{
   int input;
   do
   {
      cout << "Enter a value between 0 and " << high << ": ";
      cin >> input;
   }
   while (input < 0 || input > high);
```

```
   return input;
}
```

Then use this function twice:

```
int hours = read_int_up_to(23);
int minutes = read_int_up_to(59);
```

We have now removed the replication of the loop—it only occurs once, inside the function.

Design your functions to be reusable. Supply parameter variables for the values that can vary when the function is reused.

Note that the function can be reused in other programs that need to read integer values. However, we should consider the possibility that the smallest value need not always be zero.

Here is a better alternative:

```
/**
   Prompts a user to enter a value within a given range until the user
   provides a valid input.
   @param low the smallest allowable input
   @param high the largest allowable input
   @return the value provided by the user (between low and high, inclusive)
*/
int read_int_between(int low, int high)
{
   int input;
   do
   {
      cout << "Enter a value between " << low << " and " << high << ": ";
      cin >> input;
   }
   while (input < low || input > high);
   return input;
}
```

In our program, we call

```
int hours = read_int_between(0, 23);
```

Another program can call

```
int month = read_int_between(1, 12);
```

In general, you will want to provide parameter variables for the values that vary when a function is reused.

EXAMPLE CODE See sec06 of your companion code for a program that demonstrates the `read_int_between` function.

When carrying out the same task multiple times, use a function.

5.7 Problem Solving: Stepwise Refinement

Use the process of stepwise refinement to decompose complex tasks into simpler ones.

One of the most powerful strategies for problem solving is the process of **stepwise refinement**. To solve a difficult task, break it down into simpler tasks. Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve.

© AdShooter/iStockphoto.

A production process is broken down into sequences of assembly steps.

Here is an application of this process to a problem of everyday life. You get up in the morning and simply must *get coffee*. How do you get coffee? You see whether you can get someone else, such as your mother or mate, to bring you some. If that fails, you must *make coffee*. How do you make coffee? If there is instant coffee available, you can *make instant coffee*. How do you make instant coffee? Simply *boil water* and mix the boiling water with the instant coffee. How do you boil water? If there is a microwave, then you fill a cup with water, place it in the microwave and heat it for three minutes. Otherwise, you fill a kettle with water and heat it on the stove until the water comes to a boil. On the other hand, if you don't have instant coffee, you must *brew coffee*. How do you brew coffee? You add water to the coffee maker, put in a filter, *grind coffee*, put the coffee in the filter, and turn the coffee maker on. How do you grind coffee? You add coffee beans to the coffee grinder and push the button for 60 seconds.

Figure 5 shows a flowchart view of the coffee-making solution. Refinements are shown as expanding boxes. In C++, you implement a refinement as a function. For example, a function `brew_coffee` would call `grind_coffee`, and it would be called from a function `make_coffee`.

Let us apply the process of stepwise refinement to a programming problem.

When printing a check, it is customary to write the check amount both as a number ("$274.15") and as a text string ("two hundred seventy four dollars and 15 cents"). Doing so reduces the recipient's temptation to add a few digits in front of the amount.

© YinYang/iStockphoto.

For a human, this isn't particularly difficult, but how can a computer do this? There is no built-in function that turns 274 into `"two hundred seventy four"`. We need to program this function. Here is the description of the function we want to write:

```
/**
   Turns a number into its English name.
   @param number a positive integer < 1,000
   @return the name of number (e.g., "two hundred seventy four")
*/
string int_name(int number)
```

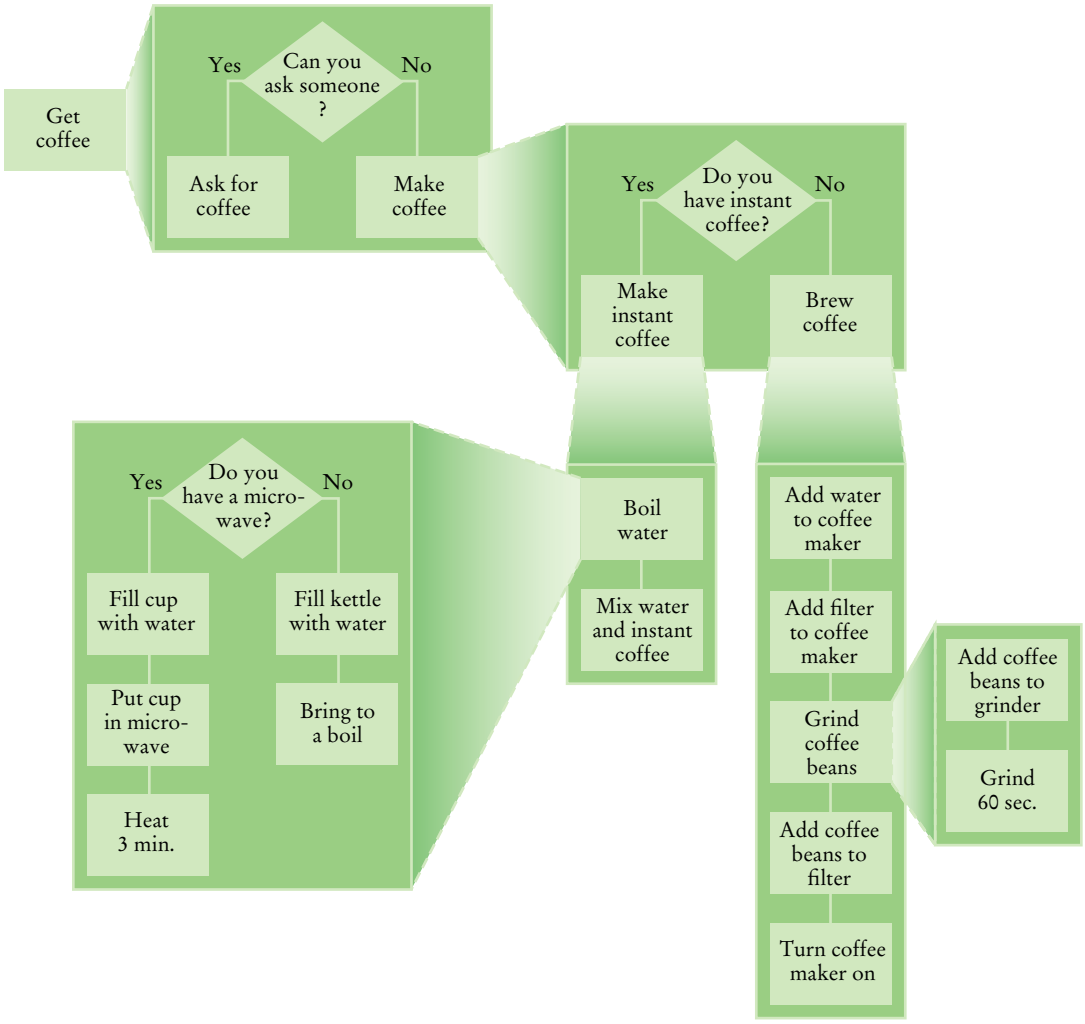


Figure 5 Flowchart of Coffee-Making Solution

When you discover that you need a function, write a description of the parameter variables and return values.

How can this function do its job? Let's look at a simple case first. If the number is between 1 and 9, we need to compute `"one"` … `"nine"`. In fact, we need the same computation *again* for the hundreds (`two hundred`). Using the stepwise decomposition process, we design another function for this simpler task. Again, rather than implementing the function, we first write the comment:

```
/**
   Turns a digit into its English name.
   @param digit an integer between 1 and 9
   @return the name of digit ("one" ... "nine")
*/
string digit_name(int digit)
```

A function may require simpler functions to carry out its work.

This sounds simple enough to implement, using an `if` statement with nine branches. No further functions should be required for completing the `digit_name` function, so we will worry about the implementation later.

Numbers between 10 and 19 are special cases. Let's have a separate function `teen_name` that converts them into strings `"eleven"`, `"twelve"`, `"thirteen"`, and so on:

```
/**
   Turns a number between 10 and 19 into its English name.
   @param number an integer between 10 and 19
   @return the name of the number ("ten" ... "nineteen")
*/
string teen_name(int number)
```

Next, suppose that the number is between 20 and 99. Then we show the tens as `"twenty"`, `"thirty"`, ..., `"ninety"`. For simplicity and consistency, put that computation into a separate function:

```
/**
   Gives the name of the tens part of a number between 20 and 99.
   @param number an integer between 20 and 99
   @return the name of the tens part of the number ("twenty" ... "ninety")
*/
string tens_name(int number)
```

Now suppose the number is at least 20 and at most 99. If the number is evenly divisible by 10, we use `tens_name`, and we are done. Otherwise, we print the tens with `tens_name` and the ones with `digit_name`. If the number is between 100 and 999, then we show a digit, the word `"hundred"`, and the remainder as described previously.

Here is the pseudocode of the algorithm:

```
int_name(number)
part = number // The part that still needs to be converted
name = ""

If part >= 100
   name = name of hundreds in part + " hundred"
   Remove hundreds from part.

If part >= 20
   Append tens_name(part) to name.
   Remove tens from part.
Else if part >= 10
   Append teen_name(part) to name.
   part = 0

If (part > 0)
   Append digit_name(part) to name.
```

This pseudocode has a number of important improvements over the verbal description. It shows how to arrange the tests, starting with the comparisons against the larger numbers, and it shows how the smaller number is subsequently processed in further `if` statements.

On the other hand, this pseudocode is vague about the actual conversion of the pieces, just referring to "name of hundreds" and the like. Furthermore, we were vague about spaces. As it stands, the code would produce strings with no spaces, `twohundredseventyfour`, for example. Compared to the complexity of the main problem, one would hope that spaces are a minor issue. It is best not to muddy the pseudocode with minor details.

Now turn the pseudocode into real code. The last three cases are easy, because helper functions are already developed for them:

```
if (part >= 20)
{
   name = name + " " + tens_name(part);
   part = part % 10;
}
else if (part >= 10)
{
   name = name + " " + teen_name(part);
   part = 0;
}

if (part > 0)
{
   name = name + " " + digit_name(part);
}
```

Finally, let us tackle the case of numbers between 100 and 999. Because `part < 1000`, `part / 100` is a single digit, and we obtain its name by calling `digit_name`. Then we add the "hundred" suffix:

```
if (part >= 100)
{
   name = digit_name(part / 100) + " hundred";
   part = part % 100;
}
```

Now you have seen all the important building blocks for the `int_name` function. Here is the complete program:

sec07/intname.cpp

```
#include <iostream>
#include <string>

using namespace std;

/**
   Turns a digit into its English name.
   @param digit an integer between 1 and 9
   @return the name of digit ("one" ... "nine")
*/
string digit_name(int digit)
{
   if (digit == 1) return "one";
   if (digit == 2) return "two";
   if (digit == 3) return "three";
   if (digit == 4) return "four";
   if (digit == 5) return "five";
   if (digit == 6) return "six";
   if (digit == 7) return "seven";
   if (digit == 8) return "eight";
   if (digit == 9) return "nine";
   return "";
}

/**
   Turns a number between 10 and 19 into its English name.
```

```
27    @param number an integer between 10 and 19
28    @return the name of the given number ("ten" ... "nineteen")
29 */
30 string teen_name(int number)
31 {
32    if (number == 10) return "ten";
33    if (number == 11) return "eleven";
34    if (number == 12) return "twelve";
35    if (number == 13) return "thirteen";
36    if (number == 14) return "fourteen";
37    if (number == 15) return "fifteen";
38    if (number == 16) return "sixteen";
39    if (number == 17) return "seventeen";
40    if (number == 18) return "eighteen";
41    if (number == 19) return "nineteen";
42    return "";
43 }
44
45 /**
46    Gives the name of the tens part of a number between 20 and 99.
47    @param number an integer between 20 and 99
48    @return the name of the tens part of the number ("twenty" ... "ninety")
49 */
50 string tens_name(int number)
51 {
52    if (number >= 90) return "ninety";
53    if (number >= 80) return "eighty";
54    if (number >= 70) return "seventy";
55    if (number >= 60) return "sixty";
56    if (number >= 50) return "fifty";
57    if (number >= 40) return "forty";
58    if (number >= 30) return "thirty";
59    if (number >= 20) return "twenty";
60    return "";
61 }
62
63 /**
64    Turns a number into its English name.
65    @param number a positive integer < 1,000
66    @return the name of the number (e.g. "two hundred seventy four")
67 */
68 string int_name(int number)
69 {
70    int part = number; // The part that still needs to be converted
71    string name; // The return value
72
73    if (part >= 100)
74    {
75       name = digit_name(part / 100) + " hundred";
76       part = part % 100;
77    }
78
79    if (part >= 20)
80    {
81       name = name + " " + tens_name(part);
82       part = part % 10;
83    }
84    else if (part >= 10)
85    {
```

```
86          name = name + " " + teen_name(part);
87          part = 0;
88       }
89
90       if (part > 0)
91       {
92          name = name + " " + digit_name(part);
93       }
94
95       return name;
96    }
97
98    int main()
99    {
100      cout << "Please enter a positive integer: ";
101      int input;
102      cin >> input;
103      cout << int_name(input) << endl;
104      return 0;
105   }
```

Program Run

```
Please enter a positive integer: 729
seven hundred twenty nine
```

Programming Tip 5.3

Keep Functions Short

There is a certain cost for writing a function. You need to design, code, and test the function. The function needs to be documented. You need to spend some effort to make the function reusable rather than tied to a specific context. To avoid this cost, it is always tempting just to stuff more and more code in one place rather than going through the trouble of breaking up the code into separate functions. It is quite common to see inexperienced programmers produce functions that are several hundred lines long.

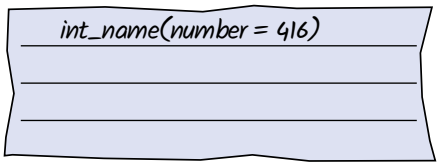
As a rule of thumb, a function that is so long that its code will not fit on a single screen in your development environment should probably be broken up.

Programming Tip 5.4

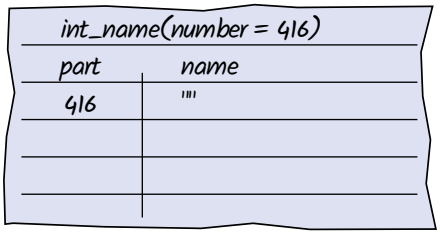
Tracing Functions

When you design a complex set of functions, it is a good idea to carry out a manual **walkthrough** before entrusting your program to the computer.

Take an index card, or some other piece of paper, and write down the function call that you want to study. Write the name of the function and the names and values of the parameter variables, like this:

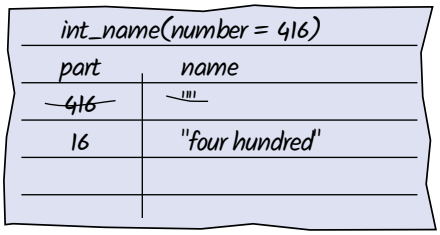


Then write the names and initial values of the function variables. Write them in a table, since you will update them as you walk through the code.

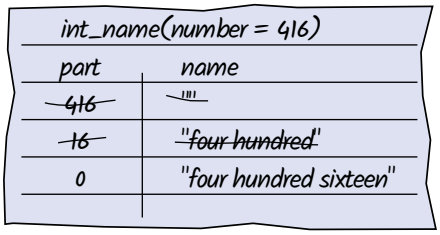


We enter the test `part >= 100`. `part / 100` is 4 and `part % 100` is 16. `digit_name(4)` is easily seen to be `"four"`. (Had `digit_name` been complicated, you would have started another sheet of paper to figure out that function call. It is quite common to accumulate several sheets in this way.)

Now `name` has changed to `name + " " + digit_name(part / 100) + " hundred"`, that is `"four hundred"`, and `part` has changed to `part % 100`, or 16.



Now you enter the branch `part >= 10`. `teen_name(16)` is `sixteen`, so the variables now have the values



Now it becomes clear why you need to set `part` to `0` in line 87. Otherwise, you would enter the next branch and the result would be `"four hundred sixteen six"`. Tracing the code is an effective way to understand the subtle aspects of a function.

Programming Tip 5.5

Stubs

When writing a larger program, it is not always feasible to implement and test all functions at once. You often need to test a function that calls another, but the other function hasn't yet been implemented. Then you can temporarily replace the missing function with a **stub**. A stub is a function that returns a simple value that is sufficient for testing another function. Here are examples of stub functions:

```
/**
   Turns a digit into its English name.
   @param digit an integer between 1 and 9
   @return the name of digit ("one" ... "nine")
```

```
*/
string digit_name(int digit)
{
   return "mumble";
}

/**
   Gives the name of the tens part of a number between 20 and 99.
   @param number an integer between 20 and 99
   @return the tens name of the number ("twenty" ... "ninety")
*/
string tens_name(int number)
{
   return "mumblety";
}
```

If you combine these stubs with the `int_name` function and test it with an argument of 274, you will get a result of `"mumble hundred mumblety mumble"`, which indicates that the basic logic of the `int_name` function is working correctly.

Stubs are incomplete functions that can be used for testing.

WORKED EXAMPLE 5.3

Calculating a Course Grade

Learn how to use stepwise refinement to solve the problem of converting a set of letter grades into an average grade for a course. See your E-Text or visit `wiley.com/go/bclo3`.

5.8 Variable Scope and Global Variables

The scope of a variable is the part of the program in which it is visible.

It is possible to define the same variable name more than once in a program. When the variable name is used, you need to know to which definition it belongs. In this section, we discuss the rules for dealing with multiple definitions of the same name.

A variable that is defined within a function is visible from the point at which it is defined until the end of the block in which it was defined. This area is called the **scope** of the variable.

Consider the `volume` variables in the following example:

```
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}

int main()
{
   double volume = cube_volume(2);
   cout << volume << endl;

   return 0;
}
```

Each `volume` variable is defined in a separate function, and their scopes do not overlap.

It is not legal to define two variables with the same name in the same scope. For example, the following is *not* legal:

```
int main()
{
   double volume = cube_volume(2);
   double volume = cube_volume(10);
      // ERROR: cannot define another volume variable in this scope
   . . .
}
```

However, you can define another variable with the same name in a **nested block**. Here, we define two variables called `amount`.

```
double withdraw(double balance, double amount)
{
   if (. . .)
   {
      double amount = 10; // Another variable named amount
      . . .
   }
   . . .
}
```

In the same way that there can be a street named "Main Street" in different cities, a C++ program can have multiple variables with the same name.

A variable in a nested block shadows a variable with the same name in an outer block.

The scope of the parameter variable `amount` is the entire function, *except* inside the nested block. Inside the nested block, `amount` refers to the variable that was defined in that block. We say that the inner variable *shadows* the variable that is defined in the outer block. You should avoid this potentially confusing situation in the functions that you write, simply by renaming one of the variables.

A local variable is defined inside a function. A global variable is defined outside a function.

Variables that are defined inside functions are called **local variables**. C++ also supports **global variables**: variables that are defined outside functions. A global variable is visible to all functions that are defined after it. For example, the `<iostream>` header defines global variables `cin` and `cout`.

Here is an example of a global variable:

```
int balance = 10000; // A global variable

void withdraw(double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}

int main()
{
   withdraw(1000);
   cout << balance << endl;
   return 0;
}
```

The scope of the variable `balance` extends over both the `withdraw` and the `main` functions.

Avoid global variables in your programs.

Generally, global variables are not a good idea. When multiple functions update global variables, the result can be difficult to predict. Particularly in larger programs that are developed by multiple programmers, it is very important that the effect of each function be clear and easy to understand. You should avoid global variables in your programs.

Programming Tip 5.6

Avoid Global Variables

There are a few cases where global variables are required (such as `cin` and `cout`), but they are quite rare. Programs with global variables are difficult to maintain and extend because you can no longer view each function as a "black box" that simply receives arguments and returns a result. When functions modify global variables, it becomes more difficult to understand the effect of function calls. As programs get larger, this difficulty mounts quickly. Instead of using global variables, use function parameters to transfer information from one part of a program to another.

5.9 Reference Parameters

If you want to write a function that changes the value of an argument, you must use a **reference parameter** in order to allow the change. We first explain why a different parameter type is necessary, then we show you the syntax for reference parameters.

Consider a function that simulates withdrawing a given amount of money from a bank account, provided that sufficient funds are available. If the amount of money is insufficient, a $10 penalty is deducted instead. The function would be used as follows:

```
double harrys_account = 1000;
withdraw(harrys_account, 100); // Now harrys_account is 900
withdraw(harrys_account, 1000); // Insufficient funds. Now harrys_account is 890
```

Here is a first attempt:

```
void withdraw(double balance, double amount) // Does not work
{
   const double PENALTY = 10;
   if (balance >= amount)
   {
      balance = balance - amount;
   }
   else
   {
      balance = balance - PENALTY;
   }
}
```

But this doesn't work.

Let's walk through the function call `withdraw(harrys_account, 100)`—see Figure 6. As the function starts, the parameter variable `balance` is created ❶ and set to the same value as `harrys_account`, and `amount` is set to 100 ❷. Then `balance` is modified ❸. Of course, that modification has no effect on `harrys_account`, because `balance` is a separate variable. When the function returns, `balance` is forgotten, and no money was withdrawn from `harrys_account` ❹.

Modifying a value parameter has no effect on the caller.

The parameter variable `balance` is called a **value parameter**, because it is initialized with the value of the supplied argument. All functions that we have written so far use value parameters. In this situation, though, we don't just want `balance` to have the same value as `harrys_account`. We want `balance` to refer to the actual variable `harrys_account` (or `joes_account` or whatever variable is supplied in the call). The contents of *that* variable should be updated.

A reference parameter refers to a variable that is supplied in a function call.

You use a reference parameter when you want to update a variable that was supplied in the function call. When we make `balance` into a reference parameter, then `balance` is not a new variable but a reference to an existing variable. Any change in `balance` is actually a change in the variable to which `balance` refers in that particular call.

A reference parameter for a bank balance is like an ATM card—it allows you to change the balance. In contrast, a value parameter can only tell you the balance.

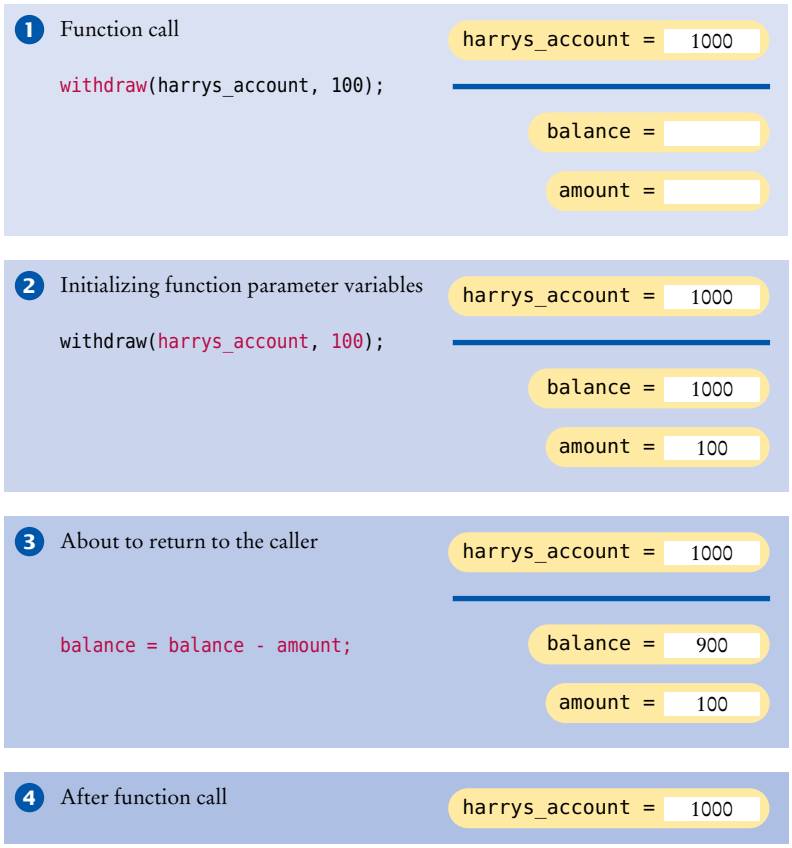


Figure 6 When `balance` and `account` are Value Parameters

Figure 7 shows the difference between value and reference parameters. To indicate a reference parameter, you place an `&` after the type name.

```
void withdraw(double& balance, double amount)
```

The type `double&` is read "a reference to a `double`" or, more briefly, "`double` ref".

The `withdraw` function now has two parameter variables: one of type "`double` ref" and the other a value parameter of type `double`. The body of the function is unchanged. What has changed is the meaning of the assignments to the `balance` variable.

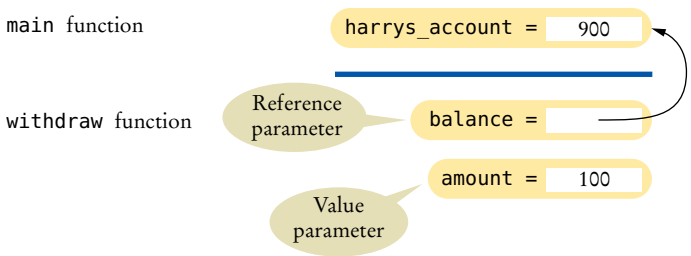


Figure 7 Reference and Value Parameters

The assignment

```
balance = balance - amount;
```

now changes the variable that was passed to the function (see Figure 8). For example, the call

```
withdraw(harrys_account, 100);
```

modifies the variable `harrys_account`, and the call

```
withdraw(sallys_account, 150);
```

modifies the variable `sallys_account`.

Modifying a reference parameter updates the variable that was supplied in the call.

The argument for a reference parameter must always be a *variable*. It would be an error to supply a number:

```
withdraw(1000, 500); // Error: argument for reference parameter must be a variable
```

The reason is clear—the function modifies the reference parameter, but it is impossible to change the value of a number. For the same reason, you cannot supply an expression:

```
withdraw(harrys_account + 150, 500);
   // Error: argument for reference parameter must be a variable
```

A function with a reference parameter can also have a return value. For example, you can modify the `withdraw` function to return a `bool` value indicating whether the withdrawal was successful:

```
bool withdraw(double& balance, double amount)
```

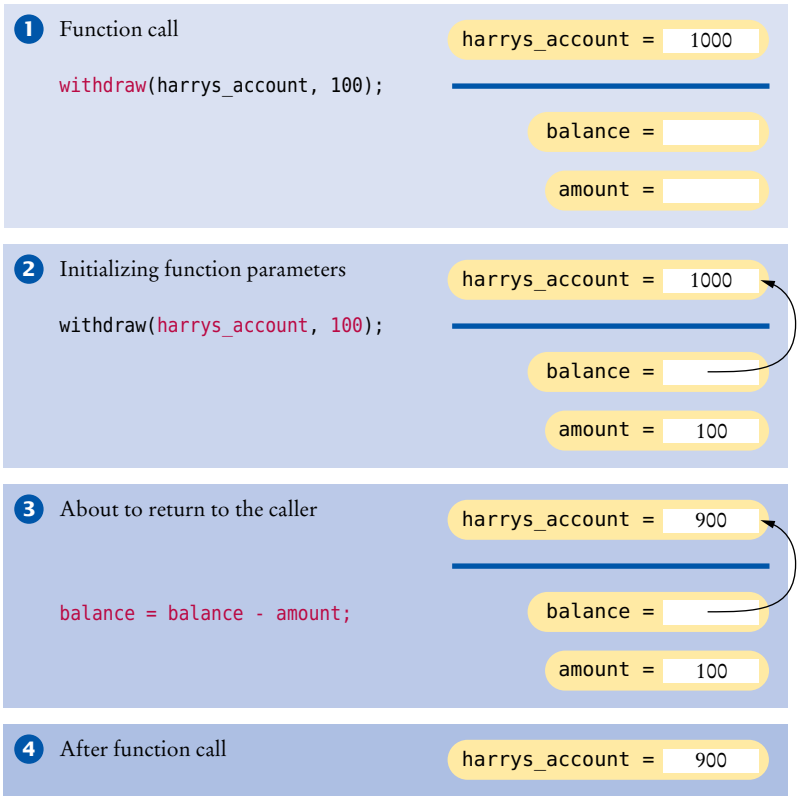


Figure 8
When `balance` is a Reference Parameter

Alternatively, you can supply a second reference parameter for that purpose:

```
void withdraw(double& balance, double amount, double& success)
```

sec09/account.cpp

```
#include <iostream>

using namespace std;

/**
   Withdraws the amount from the given balance, or withdraws
   a penalty if the balance is insufficient.
   @param balance the balance from which to make the withdrawal
   @param amount the amount to withdraw
*/
void withdraw(double& balance, double amount)
{
   const double PENALTY = 10;
   if (balance >= amount)
   {
      balance = balance - amount;
   }
   else
   {
      balance = balance - PENALTY;
   }
}

int main()
{
   double harrys_account = 1000;
   double sallys_account = 500;
   withdraw(harrys_account, 100);
      // Now harrys_account is 900
   withdraw(harrys_account, 1000); // Insufficient funds
      // Now harrys_account is 890
   withdraw(sallys_account, 150);
   cout << "Harry's account: " << harrys_account << endl;
   cout << "Sally's account: " << sallys_account << endl;

   return 0;
}
```

Program Run

```
Harry's account: 890
Sally's account: 350
```

Programming Tip 5.7

Prefer Return Values to Reference Parameters

Some programmers use reference parameters as a mechanism for setting the result of a function. For example,

```
void cube_volume(double side_length, double& volume)
{
   volume = side_length * side_length * side_length;
```

```
}
```

However, this function is less convenient than our previous `cube_volume` function. It cannot be used in expressions such as `cout << cube_volume(2)`.

Use a reference parameter only when a function needs to update a variable.

Special Topic 5.2

Constant References

It is not very efficient to have a value parameter that is a large object (such as a `string` value). Copying the object into a parameter variable is less efficient than using a reference parameter. With a reference parameter, only the location of the variable, not its value, needs to be transmitted to the function.

You can instruct the compiler to give you the efficiency of a reference parameter and the meaning of a value parameter, by using a *constant reference* as shown below. The function

```
void shout(const string& str)
{
   cout << str << "!!!" << endl;
}
```

works exactly the same as the function

```
void shout(string str)
{
   cout << str << "!!!" << endl;
}
```

There is just one difference: Calls to the first function execute a bit faster.

5.10 Recursive Functions (Optional)

A **recursive function** is a function that calls itself. This is not as unusual as it sounds at first. Suppose you face the arduous task of cleaning up an entire house. You may well say to yourself, "I'll pick a room and clean it, and then I'll clean the other rooms." In other words, the cleanup task calls itself, but with a simpler input. Eventually, all the rooms will be cleaned.

In C++, a recursive function uses the same principle. Here is a typical example. We want to print triangle patterns like this:

```
[]
[][]
[][][]
[][][][]
```

Cleaning up a house can be solved recursively: Clean one room, then clean up the rest.

Specifically, our task is to provide a function

```
void print_triangle(int side_length)
```

The triangle given above is printed by calling `print_triangle(4)`.

To see how recursion helps, consider how a triangle with side length 4 can be obtained from a triangle with side length 3.

```
[]
[][]
[][][]
[][][][]
```

Print the triangle with side length 3.
Print a line with four [].

More generally, for an arbitrary side length:

Print the triangle with side length – 1.
Print a line with side length [].

Here is the pseudocode translated to C++:

```
void print_triangle(int side_length)
{
   print_triangle(side_length - 1);
   for (int i = 0; i < side_length; i++)
   {
      cout << "[]";
   }
   cout << endl;
}
```

There is just one problem with this idea. When the side length is 1, we don't want to call `print_triangle(0)`, `print_triangle(-1)`, and so on. The solution is simply to treat this as a special case, and not to print anything when `side_length` is less than 1.

```
void print_triangle(int side_length)
{
   if (side_length < 1) { return; }
   print_triangle(side_length - 1);
   for (int i = 0; i < side_length; i++)
   {
      cout << "[]";
   }
   cout << endl;
}
```

A recursive computation solves a problem by using the solution of the same problem with simpler inputs.

Look at the `print_triangle` function one more time and notice how utterly reasonable it is. If the side length is 0, nothing needs to be printed. The next part is just as reasonable. Print the smaller triangle *and don't think about why that works*. Then print a row of `[]`. Clearly, the result is a triangle of the desired size.

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly.

For a recursion to terminate, there must be special cases for the simplest inputs.

The `print_triangle` function calls itself again with smaller and smaller side lengths. Eventually the side length must reach 0, and the function stops calling itself.

Here is what happens when we print a triangle with side length 4.

- The call `print_triangle(4)` calls `print_triangle(3)`.
  - The call `print_triangle(3)` calls `print_triangle(2)`.
    - The call `print_triangle(2)` calls `print_triangle(1)`.
      - The call `print_triangle(1)` calls `print_triangle(0)`.
        - `The call print_triangle(0) returns, doing nothing.`
      - The call `print_triangle(1)` prints `[]`.
    - The call `print_triangle(2)` prints `[][]`.
  - The call `print_triangle(3)` prints `[][][]`.
- The call `print_triangle(4)` prints `[][][][]`.

The call pattern of a recursive function looks complicated, and the key to the successful design of a recursive function is *not to think about it*.

This set of Russian dolls looks similar to the call pattern of a recursive function.

sec10/triangle.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Prints a triangle with a given side length.
7     @param side_length the side length (number of [] along the base)
8  */
9  void print_triangle(int side_length)
10 {
11    if (side_length < 1) { return; }
12    print_triangle(side_length - 1);
13    for (int i = 0; i < side_length; i++)
14    {
15       cout << "[]";
16    }
17    cout << endl;
18 }
19
20 int main()
21 {
22    cout << "Enter the side length: ";
23    int input;
24    cin >> input;
25    print_triangle(input);
```

```
26    return 0;
27 }
```

Program Run

```
Enter the side length: 10
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
[][][][][][][][]
[][][][][][][][][]
[][][][][][][][][][]
```

Recursion is not really necessary to print triangle shapes. You can use nested loops, like this:

```
for (int i = 0; i < side_length; i++)
{
   for (int j = 0; j < i; j++)
   {
      cout << "[]";
   }
   cout << endl;
}
```

However, this pair of loops is a bit tricky. Many people find the recursive solution simpler to understand.

HOW TO 5.2

Thinking Recursively

To solve a problem recursively requires a different mindset than to solve it by programming loops. In fact, it helps if you are, or pretend to be, a bit lazy and let others do most of the work for you. If you need to solve a complex problem, pretend that "someone else" will do most of the heavy lifting and solve the problem for all simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem.

Problem Statement To illustrate the recursive thinking process, consider the problem of Section 4.2, computing the sum of the digits of a number. We want to design a function `digit_sum` that computes the sum of the digits of an integer `n`. For example, `digit_sum(1729)` = 1 + 7 + 2 + 9 = 19.

Step 1 Break the input into parts that can themselves be inputs to the problem.

The key to finding a recursive solution is reducing the input to a simpler input for the same problem.

In your mind, fix a particular input or set of inputs for the task that you want to solve, and think how you can simplify the inputs. Look for simplifications that can be solved by the same task, and whose solutions are related to the original task.

In the digit sum problem, consider how we can simplify an input such as `n = 1729`. Would it help to subtract 1? After all, `digit_sum(1729) = digit_sum(1728) + 1`. But consider `n = 1000`. There seems to be no obvious relationship between `digit_sum(1000)` and `digit_sum(999)`.

A much more promising idea is to remove the last digit, that is, compute `n / 10 = 172`. The digit sum of 172 is directly related to the digit sum of 1729.

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

In your mind, consider the solutions for the simpler inputs that you have discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

In the case of the digit sum task, ask yourself how you can obtain `digit_sum(1729)` if you know `digit_sum(172)`. You simply add the last digit (9), and you are done. How do you get the last digit? As the remainder `n % 10`. The value `digit_sum(n)` can therefore be obtained as

```
digit_sum(n / 10) + n % 10
```

Don't worry how `digit_sum(n / 10)` is computed. The input is smaller, and therefore it just works.

Step 3 Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them. That is usually very easy.

Look at the simplest inputs for the `digit_sum` test:

- A number with a single digit
- `0`

A number with a single digit is its own digit sum, so you can stop the recursion when `n < 10`, and return `n` in that case. Or, if you prefer, you can be even lazier. If `n` has a single digit, then `digit_sum(n / 10) + n % 10` equals `digit_sum(0) + n`. You can simply terminate the recursion when `n` is zero.

Step 4 Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2.

Here is the complete `digit_sum` function:

```
int digit_sum(int n)
{
   // Special case for terminating the recursion
   if (n == 0) { return 0; }
   // General case
   return digit_sum(n / 10) + n % 10;
}
```

Computing & Society 5.1 The Explosive Growth of Personal Computers

In 1971, Marcian E. "Ted" Hoff, an engineer at Intel Corporation, was working on a chip for a manufacturer of electronic calculators. He realized that it would be a better idea to develop a *general-purpose* chip that could be *programmed* to interface with the keys and display of a calculator, rather than to do yet another custom design. Thus, the *microprocessor* was born. At the time, its primary application was as a controller for calculators, washing machines, and the like. It took years for the computer industry to notice that a genuine central processing unit was now available as a single chip.

Hobbyists were the first to catch on. In 1974 the first computer *kit*, the Altair 8800, was available from MITS Electronics for about $350. The kit consisted of the microprocessor, a circuit board, a very small amount of memory, toggle switches, and a row of display lights. Purchasers had to solder and assemble it, then program it in machine language through the toggle switches. It was not a big hit.

The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a $3,000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not

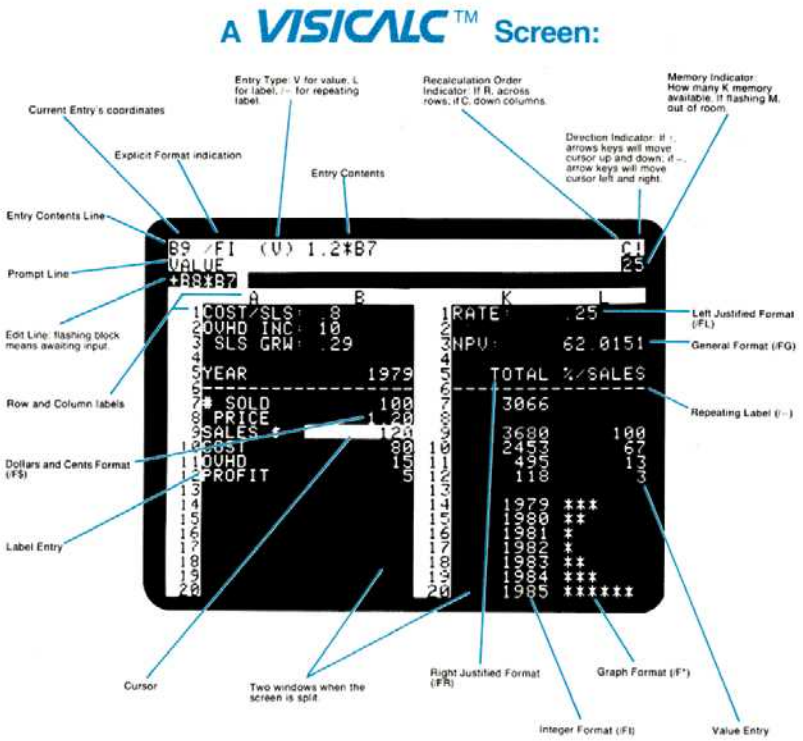
even support lowercase letters, making it worthless for word processing. The breakthrough came in 1979 with a new spreadsheet program, VisiCalc. In a spreadsheet, you enter financial data and their relationships into a grid of rows and columns (see the figure). Then you modify some of the data and watch in real time how the others change. For example, you can see how changing the mix of widgets in a manufacturing plant might affect estimated costs and profits. Corporate managers snapped up VisiCalc and the computer that was needed to run it. For them, the computer was a spreadsheet machine.

More importantly, it was a personal device. The managers were free to do the calculations that they wanted to do, not just the ones that the "high priests" in the data center provided.

Personal computers have been with us ever since, and countless users have tinkered with their hardware and software, sometimes establishing highly successful companies or creating free software for millions of users. This "freedom to tinker" is an important part of personal computing. On a personal device, you should be able to install the software that you want to install to make you more productive or creative, even if that's not the same software that most people use. You should be able to add peripheral equipment of your choice. For the first thirty years of personal computing, this freedom was largely taken for granted.

We are now entering an era where smart phones, tablets, and smart TV sets are replacing functions that were traditionally fulfilled by personal computers. While it is amazing to carry more computing power in your cell phone than in the best personal computers of the 1990s, it is disturbing that we lose a degree of personal control. With some phone or tablet brands, you can install only those applications that the manufacturer publishes on the "app store". For example, Apple rejected MIT's iPad app for the educational language Scratch because it contained a virtual machine. You'd think it would be in Apple's interest to encourage the next generation to be enthusiastic about programming, but they have a general policy of denying programmability on "their" devices, in order to thwart competitive environments such as Flash or Java.

When you select a device for making phone calls or watching movies, it is worth asking who is in control. Are you purchasing a personal device that you can use in any way you choose, or are you being tethered to a flow of data that is controlled by somebody else?



Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

The Visicalc Spreadsheet Running on an Apple II

CHAPTER SUMMARY

Understand the concepts of functions, arguments, and return values.

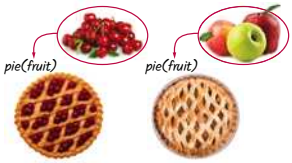
- A function is a named sequence of instructions.
- Arguments are supplied when a function is called. The return value is the result that the function computes.

Be able to implement functions.

- When defining a function, you provide a name for the function, a variable for each argument, and a type for the result.
- Function comments explain the purpose of the function, the meaning of the parameter variables and return value, as well as any special requirements.

Describe the process of parameter passing.

- Parameter variables hold the argument values supplied in the function call.



Describe the process of returning a value from a function.

- The return statement terminates a function call and yields the function result.

Design and implement functions without return values.

- Use a return type of `void` to indicate that a function does not return a value.

Develop functions that can be reused for multiple problems.

- Eliminate replicated code or pseudocode by defining a function.
- Design your functions to be reusable. Supply parameter variables for the values that can vary when the function is reused.

Apply the design principle of stepwise refinement.

- Use the process of stepwise refinement to decompose complex tasks into simpler ones.
- When you discover that you need a function, write a description of the parameter variables and return values.
- A function may require simpler functions to carry out its work.

Determine the scope of variables in a program.

- The scope of a variable is the part of the program in which it is visible.
- A variable in a nested block shadows a variable with the same name in an outer block.
- A local variable is defined inside a function. A global variable is defined outside a function.
- Avoid global variables in your programs.

Describe how reference parameters work.

- Modifying a value parameter has no effect on the caller.
- A reference parameter refers to a variable that is supplied in a function call.
- Modifying a reference parameter updates the variable that was supplied in the call.

Understand recursive function calls and implement simple recursive functions.

- A recursive computation solves a problem by using the solution of the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.
- The key to finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

REVIEW EXERCISES

■ **R5.1** What is the difference between an argument and a return value? How many arguments can a function have? How many return values?

■ **R5.2** In which sequence are the lines of the program `cube.cpp` in Section 5.2 executed, starting with the first line of `main`?

■ **R5.3** Give examples of the following, either from the C++ library or from the functions discussed in this chapter:

a. A function with two `double` arguments and a `double` return value

b. A function with a `double` argument and a `double` return value

c. A function with two `int` arguments and an `int` return value

d. A function with an `int` argument and a `string` return value

e. A function with a `string` argument and no return value

f. A function with a reference parameter and no return value

g. A function with no arguments and an `int` return value

■ **R5.4** True or false?

a. A function has exactly one `return` statement.

b. A function has at least one `return` statement.

c. A function has at most one return value.

d. A function with return value `void` never has a `return` statement.

e. When executing a `return` statement, the function exits immediately.

f. A function with return value `void` must print a result.

g. A function without arguments always returns the same value.

■■ **R5.5** Consider these functions:

```
double f(double x) { return g(x) + sqrt(h(x)); }
double g(double x) { return 4 * h(x); }
double h(double x) { return x * x + k(x) - 1; }
double k(double x) { return 2 * (x + 1); }
```

Without actually compiling and running a program, determine the results of the following function calls:

a. `double x1 = f(2);`

b. `double x2 = g(h(2));`

c. `double x3 = k(g(2) + h(2));`

d. `double x4 = f(0) + f(1) + f(2);`

e. `double x5 = f(-1) + g(-1) + h(-1) + k(-1);`

■■ **Business R5.6** Write pseudocode for a function that translates a telephone number with letters in it (such as 1-800-FLOWERS) into the actual phone number. Use the standard letters on a phone pad.

© stacey_newman/iStockphoto.

■■ **R5.7** Design a function that prints a floating-point number as a currency value (with a $ sign and two decimal digits).

a. Indicate how the programs `ch02/sec03/volume2.cpp` and `ch04/sec03/invtable.cpp` should change to use your function.

b. What change is required if the programs should show a different currency, such as euro?

■■ **R5.8** For each of the variables in the following program, indicate the scope. Then determine what the program prints, without actually running the program.

```
 1  int a = 0;
 2  int b = 0;
 3  int f(int c)
 4  {
 5     int n = 0;
 6     a = c;
 7     if (n < c)
 8     {
 9        n = a + b;
10     }
11     return n;
12  }
13
14  int g(int c)
15  {
16     int n = 0;
17     int a = c;
18     if (n < f(c))
19     {
20        n = a + b;
21     }
22     return n;
23  }
24
25  int main()
26  {
27     int i = 1;
28     int b = g(i);
29     cout << a + b + i << endl;
30     return 0;
31  }
```

■■ **R5.9** We have seen three kinds of variables in C++: global variables, parameter variables, and local variables. Classify the variables of Exercise R5.8 according to these categories.

■■ **R5.10** Use the process of stepwise refinement to describe the process of making scrambled eggs. Discuss what you do if you do not find eggs in the refrigerator.

■ **R5.11** How many parameters does the following function have? How many return values does it have? *Hint:* The C++ notions of "parameter " and "return value" are not the same as the intuitive notions of "input" and "output".

```
void average(double& avg)
{
   cout << "Please enter two numbers: ";
   double x;
   double y;
```

```
   cin >> x >> y;
   avg = (x + y) / 2;
}
```

■ **R5.12** Perform a walkthrough of the `int_name` function with the following arguments:

a. `5`
b. `12`
c. `21`
d. `301`
e. `324`
f. `0`
g. `-2`

■■ **R5.13** Consider the following function:

```
int f(int n)
{
   if (n <= 1) { return 1; }
   if (n % 2 == 0) // n is even
   {
      return f(n / 2);
   }
   else { return f(3 * n + 1); }
}
```

Perform traces of the computations `f(1)`, `f(2)`, `f(3)`, `f(4)`, `f(5)`, `f(6)`, `f(7)`, `f(8)`, `f(9)`, and `f(10)`.

■■ **R5.14** Eliminate the global variable in the code at the end of Section 5.8 by

a. passing the balance to the `withdraw` function and returning the updated balance.
b. passing the balance as a reference parameter to the `withdraw` function.

■■ **R5.15** Given the following functions, trace the function call `print_roots(4)`.

```
int i;

int isqrt(int n)
{
   i = 1;
   while (i * i <= n) { i++; }
   return i - 1;
}

void print_roots(int n)
{
   for (i = 0; i <= n; i++) { cout << isqrt(i) << " "; }
}
```

How can you fix the code so that the output is as expected (that is, `0 1 1 1 2`)?

■■ **R5.16** Consider the following function that is intended to swap the values of two integers:

```
void false_swap1(int& a, int& b)
{
   a = b;
   b = a;
}

int main()
{
```

```
   int x = 3;
   int y = 4;
   false_swap1(x, y);
   cout << x << " " << y << endl;
   return 0;
}
```

Why doesn't the function swap the contents of `x` and `y`? How can you rewrite the function to work correctly?

■■■ **R5.17** Consider the following function that is intended to swap the values of two integers:

```
void false_swap2(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

int main()
{
   int x = 3;
   int y = 4;
   false_swap2(x, y);
   cout << x << " " << y << endl;
   return 0;
}
```

Why doesn't the function swap the contents of `x` and `y`? How can you rewrite the function to work correctly?

■■■ **R5.18** The following function swaps two integers, without requiring a temporary variable:

```
void tricky_swap(int& a, int& b)
{
   a = a - b;
   b = a + b;
   a = b - a;
}
```

However, it fails in one important case, namely when calling `tricky_swap(x, x)`. Explain what should happen and what actually happens.

■■■ **R5.19** In How To 5.1, it is easy enough to measure the width of a real pyramid. But how can you measure the height without climbing to the top? You need to measure the angle between the ground and the top from a point, and the distance from that point to the base. Develop a concrete plan and write pseudocode for a program that determines the volume of a pyramid from these inputs.

■■■ **R5.20** Suppose an ancient civilization had constructed circular pyramids. Write pseudocode for a program that determines the surface area from measurements that you can determine from the ground.

■■■ **R5.21** Give pseudocode for a recursive function for printing all substrings of a given string. For example, the substrings of the string `"rum"` are `"rum"` itself, `"ru"`, `"um"`, `"r"`, `"u"`, `"m"`, and the empty string. You may assume that all letters of the string are different.

■■■ **R5.22** Give pseudocode for a recursive function that sorts all letters in a string. For example, the string `"goodbye"` would be sorted into `"bdegooy"`.

PRACTICE EXERCISES

■ **E5.1** The `max` function that is declared in the `<algorithm>` header returns the larger of its two arguments. Write a program that reads three floating-point numbers, uses the `max` function, and displays

- the larger of the first two inputs.
- the larger of the last two inputs.
- the largest of all three inputs.

■ **E5.2** Write a function that computes the balance of a bank account with a given initial balance and interest rate, after a given number of years. Assume interest is compounded yearly.

■ **E5.3** Write the following functions and provide a program to test them.

a. `double smallest(double x, double y, double z)`, returning the smallest of the arguments

b. `double average(double x, double y, double z)`, returning the average of the arguments

■■ **E5.4** Write the following functions and provide a program to test them:

a. `bool all_the_same(double x, double y, double z)`, returning true if the arguments are all the same

b. `bool all_different(double x, double y, double z)`, returning true if the arguments are all different

c. `bool sorted(double x, double y, double z)`, returning true if the arguments are sorted, with the smallest one coming first

■■ **E5.5** Write the following functions:

a. `int first_digit(int n)`, returning the first digit of the argument

b. `int last_digit(int n)`, returning the last digit of the argument

c. `int digits(int n)`, returning the number of digits of the argument

For example, `first_digit(1729)` is 1, `last_digit(1729)` is 9, and `digits(1729)` is 4. Provide a program that tests your functions.

■ **E5.6** Write a function

```
string middle(string str)
```

that returns a string containing the middle character in `str` if the length of `str` is odd, or the two middle characters if the length is even. For example, `middle("middle")` returns `"dd"`.

■ **E5.7** Write a function

```
string repeat(string str, int n)
```

that returns the string `str` repeated `n` times. For example, `repeat("ho", 3)` returns `"hohoho"`.

■■ **E5.8** Write a function

```
int count_vowels(string str)
```

that returns a count of all vowels in the string `str`. Vowels are the letters a, e, i, o, and u, and their uppercase variants. Use a helper function `is_vowel` that checks whether a character is a vowel.

■■ **E5.9** Write a function

```
int count_words(string str)
```

that returns a count of all words in the string `str`. Words are separated by spaces. For example, `count_words("Mary had a little lamb")` should return 5. Your function should work correctly if there are multiple spaces between words. Use helper functions to find the position of the next space following a given position, and to find the position of the next non-space character following a given position.

■■ **E5.10** Write a function that returns the average length of all words in the string `str`. Use the same helper functions as in Exercise E5.9.

■ **E5.11** Write functions

```
double sphere_volume(double r)
double sphere_surface(double r)
double cylinder_volume(double r, double h)
double cylinder_surface(double r, double h)
double cone_volume(double r, double h)
double cone_surface(double r, double h)
```

that compute the volume and surface area of a sphere with radius `r`, a cylinder with a circular base with radius `r` and height `h`, and a cone with a circular base with radius `r` and height `h`. Then write a program that prompts the user for the values of `r` and `h`, calls the six functions, and prints the results.

■■ **E5.12** Write functions

```
double distance(double x1, double x2, double y1, double y2)
void midpoint(double x1, double x2, double y1, double y2, double& xmid, double& ymid)
void slope(double x1, double x2, double y1, double y2, bool& vertical, double& s)
```

that compute the distance, midpoint, and slope of the line segment joining the points (x_1, y_1) and (x_2, y_2). The `slope` function should either set `vertical` to `true` and not set `s`, or set `vertical` to `false` and set `s` to the slope.

■■ **E5.13** Write a function

```
double read_double(string prompt)
```

that displays the prompt string, followed by a space, reads a floating-point number in, and returns it. Here is a typical usage:

```
salary = read_double("Please enter your salary:");
perc_raise = read_double("What percentage raise would you like?");
```

■■ **E5.14** Write a function `void sort2(int& a, int& b)` that swaps the values of `a` and `b` if `a` is greater than `b` and otherwise leaves `a` and `b` unchanged. For example,

```
int u = 2;
int v = 3;
int w = 4;
int x = 1;
sort2(u, v); // u is still 2, v is still 3
sort2(w, x); // w is now 1, x is now 4
```

■■ **E5.15** Write a function `sort3(int& a, int& b, int& c)` that swaps its three arguments to arrange them in sorted order. For example,

```
int v = 3;
int w = 4;
int x = 1;
sort3(v, w, x); // v is now 1, w is now 3, x is now 4
```

Hint: Use multiple calls to the `sort2` function of Exercise E5.14.

■■ **E5.16** Write a function that computes the weekday of a given date, using a formula known as *Zeller's congruence*. Let

d = the day of the month

mm = the modified month (3 = March, ..., 12 = December, 13 = January, 14 = February)

w = the weekday (0 = Monday, 1 = Tuesday, ..., 6 = Sunday)

Then

$$w = \left(d + 5 + \frac{\left(26 \times \left(mm + 1\right)\right)}{10} + \frac{5 \times \left(year \ \% \ 100\right)}{4} + \frac{21 \times \left(year / 100\right)}{4} \right) \% \ 7$$

Here, all / denote integer division and % denotes the remainder operation.

■■ **E5.17** Write a program that prints instructions to get coffee, asking the user for input whenever a decision needs to be made. Decompose each task into a function, for example:

```
void brew_coffee()
{
   cout << "Add water to the coffee maker." << endl;
   cout << "Put a filter in the coffee maker." << endl;
   grind_coffee();
   cout << "Put the coffee in the filter." << endl;
   . . .
}
```

■■ **E5.18** Write a function that computes the Scrabble score of a word. Look up the letter values on the Internet. For example, with English letter values, the word `CODE` is worth 3 + 0 + 2 + 1 = 6 points.

■■ **E5.19** Write a function that tests whether a file name should come before or after another. File names are first sorted by their extension (the string after the last period) and then by the name part (the string that remains after removing the extension). For example, `before("report.doc", "notes.txt")` should return `true` and `before("report.txt", "notes.txt")` should return `false`. Provide helper functions that return the extension and name part of a file name.

■■■ **E5.20** When comparing strings, the comparison is not always satisfactory. For example, `"file10" < "file2"`, even though we would prefer it to come afterwards. Produce a `num_less` function that, when comparing two strings that are identical except for a positive integer at the end, compares the integers. For example, `num_less("file12", "file2")` should return `true`, but `num_less("file12", "file11")` and `num_less("file2", "doc12")` should return `false`. Use a helper function that returns the starting position of the number, or –1 if there is none.

■■ **E5.21** Write a recursive function

```
string reverse(string str)
```

that computes the reverse of a string. For example, `reverse("flow")` should return `"wolf"`. *Hint:* Reverse the substring starting at the second character, then add the first character at the end. For example, to reverse `"flow"`, first reverse `"low"` to `"wol"`, then add the `"f"` at the end.

■■ **E5.22** Write a recursive function

```
bool is_palindrome(string str)
```

that returns `true` if `str` is a palindrome, that is, a word that is the same when reversed. Examples of palindrome are "deed", "rotor", or "aibohphobia". *Hint:* A word is a palindrome if the first and last letters match and the remainder is also a palindrome.

■■ **E5.23** Use recursion to implement a function `bool find(string str, string match)` that tests whether `match` is contained in `str`:

```
bool b = find("Mississippi", "sip"); // Sets b to true
```

Hint: If `str` starts with `match`, then you are done. If not, consider the string that you obtain by removing the first character.

■ **E5.24** Use recursion to determine the number of digits in a number `n`. *Hint:* If `n` is < 10, it has one digit. Otherwise, it has one more digit than `n / 10`.

PROGRAMMING PROJECTS

■■ **P5.1** It is a well-known phenomenon that most people are easily able to read a text whose words have two characters flipped, provided the first and last letter of each word are not changed. For example:

> I dn'ot gvie a dman for a man taht can olny sepll a wrod one way. (Mrak Taiwn)

Write a function `string scramble(string word)` that constructs a scrambled version of a given word, randomly flipping two characters other than the first and last one. Then write a program that reads words from `cin` and prints the scrambled words.

■■ **P5.2** Enhance the `int_name` function so that it works correctly for values < 1,000,000,000.

■■ **P5.3** Enhance the `int_name` function so that it works correctly for negative values and zero. *Caution:* Make sure the improved function doesn't print 20 as `"twenty zero"`.

■■■ **P5.4** For some values (for example, 20), the `int_name` function returns a string with a leading space (`" twenty"`). Repair that blemish and ensure that spaces are inserted only when necessary. *Hint:* There are two ways of accomplishing this. Either ensure that leading spaces are never inserted, or remove leading spaces from the result before returning it.

■■ **P5.5** Implement the `number_to_grade` function of Worked Example 5.3 so that you use two sets of branches: one to determine whether the grade should be `A`, `B`, `C`, `D`, or `F`, and another to determine whether a `+` or `-` should be appended.

■■ **P5.6** Write a program that reads two strings containing section numbers such as `"1.13.2"` and `"1.2.4.1"` and determines which one comes first. Provide appropriate helper function.

■■ **P5.7** Write a program that prompts the user for a regular English verb (such as play), the first, second, or third person, singular or plural, and present, past, or future tense. Provide these values to a function that yields the conjugated verb and prints it. For example, the input `play 3 singular present` should yield `"he/she plays"`.

■■■ **P5.8** Write a program that reads words and arranges them in a paragraph so that all lines other than the last one are exactly forty characters long. Add spaces between words to make the last word extend to the margin. Distribute the spaces evenly. Use a helper function for that purpose. A typical output would be:

```
Four  score  and  seven  years  ago our
fathers brought forth on this continent
a new nation, conceived in liberty, and
dedicated  to  the proposition that all
men are created equal.
```

■■■ **P5.9** Write a program that, given a month and year, prints a calendar, such as

```
     June 2016
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

To find out the weekday of the first day of the month, call this function:

```
/**
   Computes the weekday of a given date.
   @param year the year
   @param month the month (1 = January ... 12 = December)
   @param day the day of the month
   @return the weekday (0 = Sunday ... 6 = Saturday)
*/
int day_of_week(int year, int month, int day)
{
   int y = year;
   int m = month - 1;
   if (month < 3) { y--; m = m + 4; }
   return (y + y / 4 - y / 100 + y / 400
      + 3 * m + 4 - (m - m / 8) / 2 + day) % 7;
}
```

Make a helper function to print the header and a helper function to print each row.

■■ **P5.10** Write a program that reads two fractions, adds them, and prints the result so that the numerator and denominator have no common factor. For example, when adding `3/4` and `5/6`, the result is `19/12`. Use helper functions for obtaining the numerator and denominator of strings such as `"3/4"`, and for computing the greatest common divisor of two integers.

■■■ **P5.11** Write a program that reads two positive integers and prints out the "long division". For example,

```
3640 : 15 = 242 Remainder 10
30
--
 64
 60
```

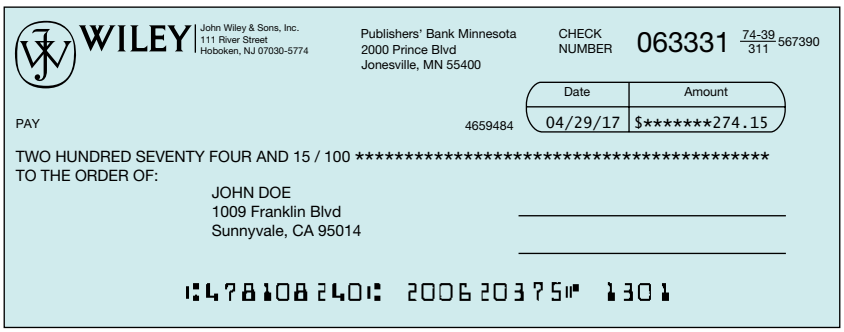
```
 --
  40
  30
 --
  10
```

Format the computation in the way that you learned in elementary school. Use helper functions to structure your program.

■■ **P5.12** Write a program that prints the decimal expansion of a fraction, marking the repeated part with an `R`. For example, `5/2` is `2.5R0`, `1/12` is `0.08R3`, and `1/7` is `0.R142857`. Use helper functions to structure your program. *Hint:* Use the long division algorithm.

■■ **P5.13** Write a program that reads a decimal expansion with a repeated part, as in Exercise P5.12, and displays the fraction that it represents. Use helper functions to structure your program.

■■ **Business P5.14** Write a program that prints a paycheck. Ask the program user for the name of the employee, the hourly rate, and the number of hours worked. If the number of hours exceeds 40, the employee is paid "time and a half", that is, 150 percent of the hourly rate on the hours exceeding 40. Your check should look similar to that in the figure below. Use fictitious names for the payer and the bank. Be sure to use stepwise refinement and break your solution into several functions. Use the `int_name` function to print the dollar amount of the check.



■■ **P5.15** *Leap years.* Write a function

```
bool leap_year(int year)
```

© mbbirdy/iStockphoto.

that tests whether a year is a leap year: that is, a year with 366 days. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000).

■■ Business P5.16 Write a program that converts a Roman number such as MCMLXXVIII to its decimal number representation. *Hint:* First write a function that yields the numeric value of each of the letters. Then use the following algorithm:

```
total = 0
str = roman number string
While str is not empty
   If str has length 1, or value(first character of str) is at least value(second character of str)
      Add value(first character of str) to total.
      Remove first character from str.
   Else
      difference = value(second character of str) - value(first character of str)
      Add difference to total.
      Remove first character and second character from str.
```

■■ P5.17 In Exercise P3.13 you were asked to write a program to convert a number to its representation in Roman numerals. At the time, you did not know how to eliminate duplicate code, and as a consequence the resulting program was rather long. Rewrite that program by implementing and using the following function:

© Straitshooter/iStockphoto.

```
string roman_digit(int n, string one, string five, string ten)
```

That function translates one digit, using the strings specified for the one, five, and ten values. You would call the function as follows:

```
roman_ones = roman_digit(n % 10, "I", "V", "X");
n = n / 10;
roman_tens = roman_digit(n % 10, "X", "L", "C");
. . .
```

■■ Business P5.18 *Postal bar codes*. For faster sorting of letters, the United States Postal Service encourages companies that send large volumes of mail to use a bar code denoting the zip code (see Figure 9).

The encoding scheme for a five-digit zip code is shown in Figure 10. There are full-height frame bars on each side. The five encoded digits are followed by a check digit, which is computed as follows: Add up all digits, and choose the check digit to make the sum a multiple of 10. For example, the zip code 95014 has a sum of 19, so the check digit is 1 to make the sum equal to 20.

Each digit of the zip code, and the check digit, is encoded according to the following table where 0 denotes a half bar and 1 a full bar.

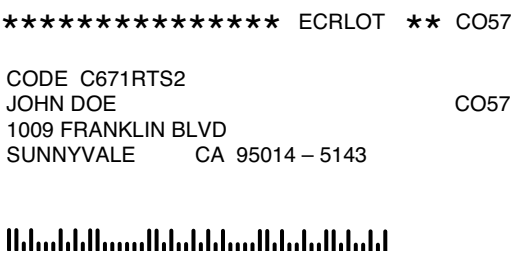


Figure 9 A Postal Bar Code

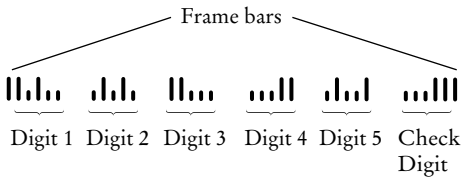


Figure 10 Encoding for Five-Digit Bar Codes

Digit	Bar 1 (weight 7)	Bar 2 (weight 4)	Bar 3 (weight 2)	Bar 4 (weight 1)	Bar 5 (weight 0)
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	1	0	0	0	1
8	1	0	0	1	0
9	1	0	1	0	0
0	1	1	0	0	0

The digit can be easily computed from the bar code using the column weights 7, 4, 2, 1, 0. For example, 01100 is $0 \times 7 + 1 \times 4 + 1 \times 2 + 0 \times 1 + 0 \times 0 = 6$. The only exception is 0, which would yield 11 according to the weight formula.

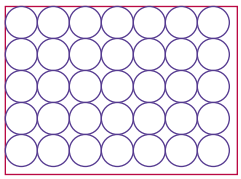
Write a program that asks the user for a zip code and prints the bar code. Use : for half bars, | for full bars. For example, 95014 becomes

```
||:|:::|:|:||:::::::||:|::|:::|||
```

■■■ **Business P5.19** Write a program that reads in a bar code (with : denoting half bars and | denoting full bars) and prints out the zip code it represents. Print an error message if the bar code is not correct.

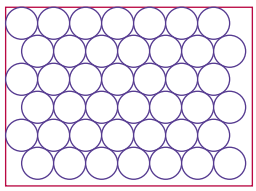
■ **P5.20** Use recursion to compute a^n, where n is a positive integer. *Hint:* If n is 1, then $a^n = a$. Otherwise, $a^n = a \times a^{n-1}$.

■■ **Media P5.21** Write a program that reads in the width and height of a rectangle and the diameter of a circle. Then fill the rectangle with as many circles as possible, using "square circle packing":



Use the `Picture` class from Worked Example 4.2 and a `.png` file with a circle on a transparent background (supplied in the companion code). Provide a helper function to draw each row.

■■■ **Media P5.22** Repeat Exercise P5.21 with "hexagonal circle packing":

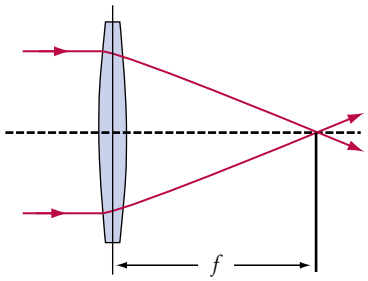


■■ **Engineering P5.23** The effective focal length f of a lens of thickness d that has surfaces with radii of curvature R_1 and R_2 is given by

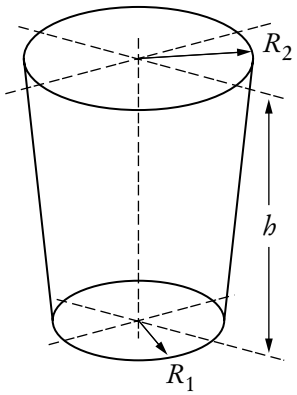
$$\frac{1}{f} = (n-1)\left[\frac{1}{R_1} - \frac{1}{R_2} + \frac{(n-1)d}{nR_1R_2}\right]$$

where n is the refractive index of the lens medium.

Write a function that computes f in terms of the other parameters. Write a C++ program to test this function.



■■ **Engineering P5.24** A laboratory container is shaped like the frustum of a cone:



Write functions to compute the volume and surface area, using these equations:

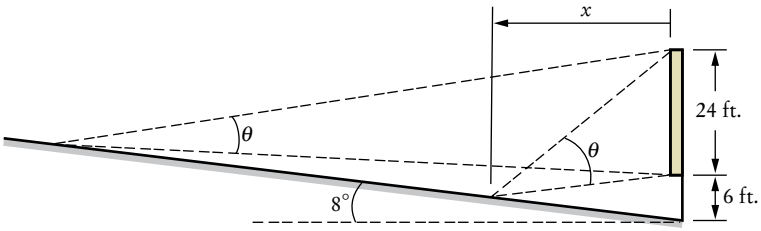
$$V = \tfrac{1}{3}\pi h\left(R_1^2 + R_2^2 + R_1R_2\right)$$

$$S = \pi\left(R_1 + R_2\right)\sqrt{\left(R_2 - R_1\right)^2 + h^2} + \pi R_1^2$$

Write a C++ program to test these functions.

■■ **Engineering P5.25** In a movie theater, the angle θ at which a viewer sees the picture on the screen depends on the distance x of the viewer from the screen. For a movie theater with the

dimensions shown in the picture below, write a function that computes the angle for a given distance.



Next, provide a more general function that works for theaters with arbitrary dimensions. Write a C++ program to test these functions.

▪▪ Engineering P5.26 Electric wire, like that in the photo, is a cylindrical conductor covered by an insulating material. The resistance of a piece of wire is given by the formula


© alacatr/iStockphoto.

$$R = \frac{\rho L}{A} = \frac{4\rho L}{\pi d^2}$$

where ρ is the resistivity of the conductor, and L, A, and d are the length, cross-sectional area, and diameter of the wire. The resistivity of copper is 1.678×10^{-8} Ω m. The wire diameter, d, is commonly specified by the American wire gauge (AWG), which is an integer, n. The diameter of an AWG n wire is given by the formula

$$d = 0.127 \times 92^{\frac{36-n}{39}} \text{ mm}$$

Write a C++ function

```
double diameter(int wire_gauge)
```

that accepts the wire gauge and returns the corresponding wire diameter. Write another C++ function

```
double copper_wire_resistance(double length, int wire_gauge)
```

that accepts the length and gauge of a piece of copper wire and returns the resistance of that wire. The resistivity of aluminum is 2.82×10^{-8} Ω m. Write a third C++ function

```
double aluminum_wire_resistance(double length, int wire_gauge)
```

that accepts the length and gauge of a piece of aluminum wire and returns the resistance of that wire.

Write a C++ program to test these functions.

▪▪ Engineering P5.27 The drag force on a car is given by

$$F_D = \frac{1}{2}\rho v^2 A C_D$$

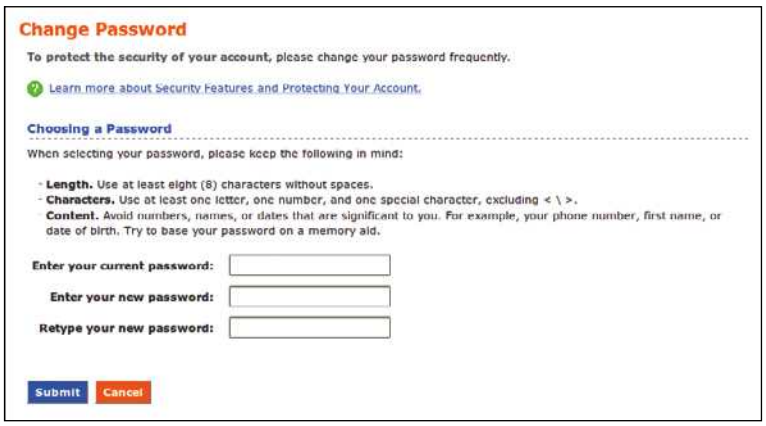
where ρ is the density of air (1.23 kg/m^3), v is the velocity in units of m/s, A is the projected area of the car (2.5 m^2), and C_D is the drag coefficient (0.2).

The amount of power in watts required to overcome such drag force is $P = F_D v$, and the equivalent horsepower required is Hp = P / 746. Write a program that accepts a car's velocity and computes the power in watts and in horsepower needed to overcome the resulting drag force. *Note:* 1 mph = 0.447 m/s.

WORKED EXAMPLE 5.1

Generating Random Passwords

Problem Statement Many web sites and software packages require you to create passwords that contain at least one digit and one special character. Your task is to write a program that generates such a password of a given length. The characters should be chosen randomly.



Step 1 Describe what the function should do.

The problem description asks you to write a program, not a function. We will write a password-generating function and call it from the program's `main` function.

Let us be more precise about the function. It will generate a password with a given number of characters. We could include multiple digits and special characters, but for simplicity, we decide to include just one of each. We need to decide which special characters are valid. For our solution, we will use the following set:

```
+ - * / ? ! @ # $ % &
```

The remaining characters of the password are letters. For simplicity, we will use only lowercase letters in the English alphabet.

Step 2 Determine the function's "inputs".

There is just one parameter: the length of the password.

At this point, we have enough information to document the function:

```
/**
   Generates a random password.
   @param length the length of the password
   @return a password of the given length with one digit and one
   special symbol
*/
```

Step 3 Determine the types of the parameter variables and the return value.

The parameter is an integer. The function returns the password, that is, a string. The function will be declared as

```
string make_password(int length)
```

Step 4 Write pseudocode for obtaining the desired result.

Here is one approach for making a password:

Make an empty string called password.
Randomly generate length - 2 letters and append them to password.
Randomly generate a digit and insert it at a random location in password.
Randomly generate a symbol and insert it at a random location in password.

How do we generate a random letter, digit, or symbol? How do we insert a digit or symbol in a random location? In the spirit of stepwise refinement, we will delegate those tasks to helper functions. Each of those functions starts a new sequence of steps, which, for greater clarity, we will place after the steps for this function.

Step 5 Implement the function body.

We need to know the "black box" descriptions of the two helper functions described in Step 4 (which we will complete after this function). Here they are:

```
/**
   Returns a string containing one character randomly chosen from a given string.
   @param characters the string from which to randomly choose a character
   @return a substring of length 1, taken at a random index
*/
string random_character(string characters)

/**
   Inserts one string into another at a random position.
   @param str the string into which another string is inserted
   @param to_insert the string to be inserted
   @return the result of inserting to_insert into str
*/
string insert_at_random(string str, string to_insert)
```

Now we can translate the pseudocode of Step 4 into C++:

```
string make_password(int length)
{
   string password = "";
   for (int i = 0; i < length - 2; i++)
   {
      password = password + random_character("abcdefghijklmnopqrstuvwxyz");
   }
   string random_digit = random_character("0123456789");
   password = insert_at_random(password, random_digit);
   string random_symbol = random_character("+-*/?!@#$%&");
   password = insert_at_random(password, random_symbol);
   return password;
}
```

Step 6 Test your function.

Because our function depends on several helper functions, we must implement the helper functions first, as described in the following sections. (If you are impatient, you can use the technique of stubs that is described in Programming Tip 5.5.)

Here is a simple program that calls the `make_password` function:

```
int main()
{
   srand(time(0));
   string result = make_password(8);
   cout << result << endl;
   return 0;
```

```
}
```

Run the program a few times. Typical outputs are

```
u@taqr8f
i?fs1dgh
ot$3rvdv
```

Each output has length 8 and contains a digit and special symbol. Note that without seeding the random number generator, the program would always print the same result.

Repeat for the First Helper Function

Now it is time to turn to the helper function for generating a random letter, digit, or special symbol.

Step 1 Describe what the function should do.

How do we deal with the choice between letter, digit, or special symbol? Of course, we could write three separate functions, but it is better if we can solve all three tasks with a single function. We could require a parameter, such as 1 for letter, 2 for digit, and 3 for special symbol. But stepping back a bit, we can supply a more general function that simply selects a random character from *any* set. Passing the string `"abcdefghijklmnopqrstuvwxyz"` generates a random lowercase letter. To get a random digit, pass the string `"0123456789"` instead.

Now we know what our function should do. Given any string, it should return a random character in it.

Step 2 Determine the function's "inputs".

The input is any string.

Step 3 Determine the types of the parameter variables and the return value.

The input type is clearly `string`. We want to make a string from the random characters. It is easy to concatenate strings, so we choose to return strings of length 1. The return type is `string`.

The function will be declared as

```
string random_character(string characters)
```

Step 4 Write pseudocode for obtaining the desired result.

random_character(characters)
n = length of characters
r = a random integer between 0 and n - 1
Return the substring of characters of length 1 that starts at r.

Step 5 Implement the function body.

Simply translate the pseudocode into C++:

```
/**
   Returns a string containing one character randomly chosen from a given string.
   @param characters the string from which to randomly choose a character
   @return a substring of length 1, taken at a random index
*/
string random_character(string characters)
{
   int n = characters.length();
   int r = rand() % n;
   return characters.substr(r, 1);
}
```

Step 6 Test your function.

Supply a program for testing this function only:

```
int main()
{
   srand(time(0));
   for (int i = 1; i <= 10; i++)
   {
      cout << random_character("abcdef");
   }
   cout << endl;
   return 0;
}
```

When you run this program, you might get an output such as

```
afcdfeefac
```

This confirms that the function works correctly.

Repeat for the Second Helper Function

Finally, we implement the second helper function, which inserts a string containing a single character at a random location in a string.

Step 1 Describe what the function should do.

Suppose we have a string `"arxcsw"` and a string `"8"`. Then the second string should be inserted at a random location, returning a string such as `"ar8xcsw"` or `"arxcsw8"`. Actually, it doesn't matter that the second string has length 1, so we will simply specify that our function should insert an arbitrary string into a given string.

Step 2 Determine the function's "inputs".

The first input is the string into which another string should be inserted. The second input is the string to be inserted.

Step 3 Determine the types of the parameter variables and the return value.

The inputs are both of type `string`. The return value is also a `string`. We can now fully describe our function:

```
/**
   Inserts one string into another at a random position.
   @param str the string into which another string is inserted
   @param to_insert the string to be inserted
   @return the result of inserting to_insert into str
*/
string insert_at_random(string str, string to_insert)
```

Step 4 Write pseudocode for obtaining the desired result.

There is no predefined function for inserting a string into another. Instead, we need to find the insertion position and then "break up" the first string by taking two substrings: the characters up to the insertion position, and the characters following it.

How many choices are there for the insertion position? If `str` has length 6, there are seven choices:

```
1. |arxcsw
2. a|rxcsw
3. ar|xcsw
4. arx|csw
5. arxc|sw
```

```
6. arxcs|w
7. arxcsw|
```

In general, if the string has length n, there are $n + 1$ choices, ranging from 0 (before the start of the string) to n (after the end of the string). Here is the pseudocode:

```
insert_at_random(str, to_insert)
n = length of str
r = a random integer between 0 and n
Return the substring of str from 0 to r - 1 + to_insert + the remainder of str.
```

Step 5 Implement the function body.

Translate the pseudocode into C++:

```
/**
   Inserts one string into another at a random position.
   @param str the string into which another string is inserted
   @param to_insert the string to be inserted
   @return the result of inserting to_insert into str
*/
string insert_at_random(string str, string to_insert)
{
   int n = str.length();
   int r = rand() % (n + 1);
   return str.substr(0, r) + to_insert + str.substr(r);
}
```

Step 6 Test your function.

Supply a program for testing this function only:

```
int main()
{
   srand(time(0));
   for (int i = 1; i <= 10; i++)
   {
      cout << insert_at_random("arxcsw", "8");
   }
   cout << endl;
   return 0;
}
```

When you run this program, you might get an output such as

```
arxcsw8
ar8xcsw
arxc8sw
a8rxcsw
arxcsw8
ar8xcsw
arxcsw8
a8rxcsw
8arxcsw
8arxcsw
```

The output shows that the second string is being inserted at an arbitrary position, including the beginning and end of the first string.

Here is the complete program:

worked_example_1/password.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
```

```
using namespace std;

/**
   Returns a string containing one character randomly chosen from a given string.
   @param characters the string from which to randomly choose a character
   @return a substring of length 1, taken at a random index
*/
string random_character(string characters)
{
   int n = characters.length();
   int r = rand() % n;
   return characters.substr(r, 1);
}

/**
   Inserts one string into another at a random position.
   @param str the string into which another string is inserted
   @param to_insert the string to be inserted
   @return the result of inserting to_insert into str
*/
string insert_at_random(string str, string to_insert)
{
   int n = str.length();
   int r = rand() % (n + 1);
   return str.substr(0, r) + to_insert + str.substr(r);
}

/**
   Generates a random password.
   @param length the length of the password
   @return a password of the given length with one digit and one
   special symbol
*/
string make_password(int length)
{
   string password = "";

   // Pick random letters

   for (int i = 0; i < length - 2; i++)
   {
      password = password + random_character("abcdefghijklmnopqrstuvwxyz");
   }

   // Insert two random digits

   string random_digit = random_character("0123456789");
   password = insert_at_random(password, random_digit);
   string random_symbol = random_character("+-*/?!@#$%&");
   password = insert_at_random(password, random_symbol);
   return password;
}

int main()
{
   srand(time(0));
   string result = make_password(8);
   cout << result << endl;
   return 0;
}
```

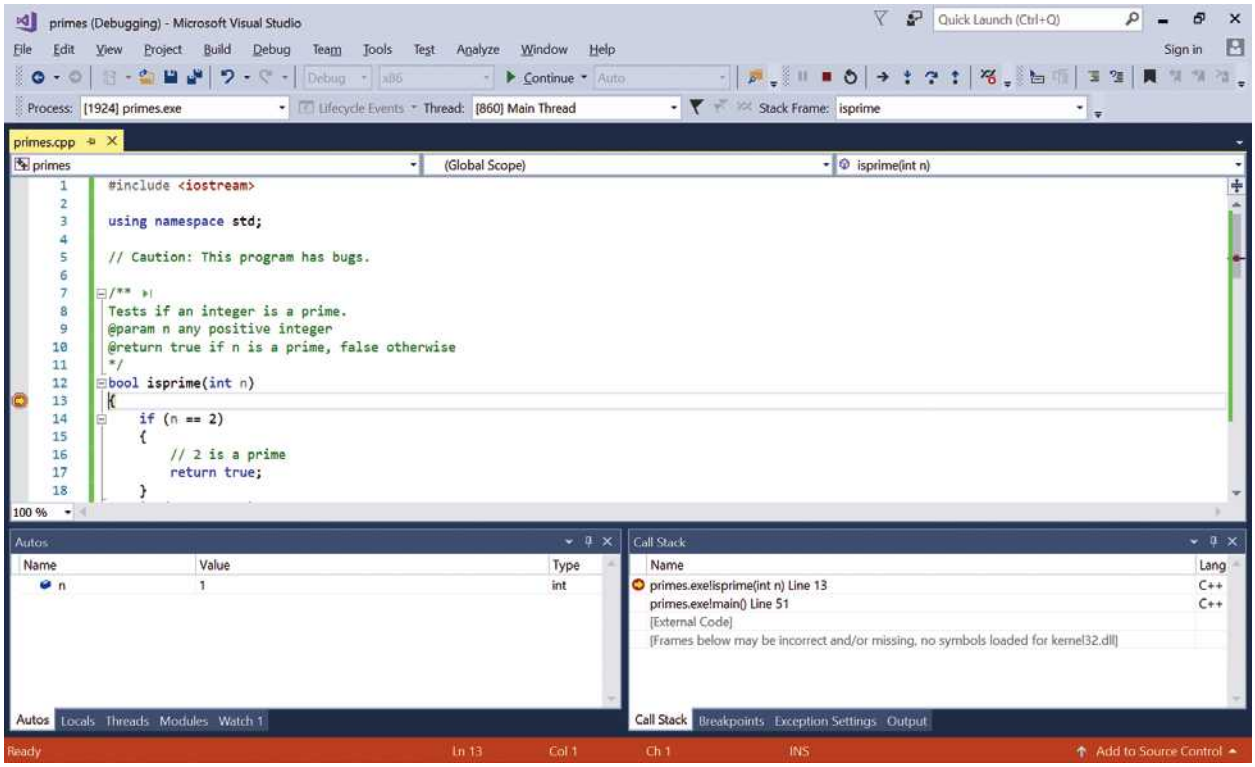
WORKED EXAMPLE 5.2

Using a Debugger

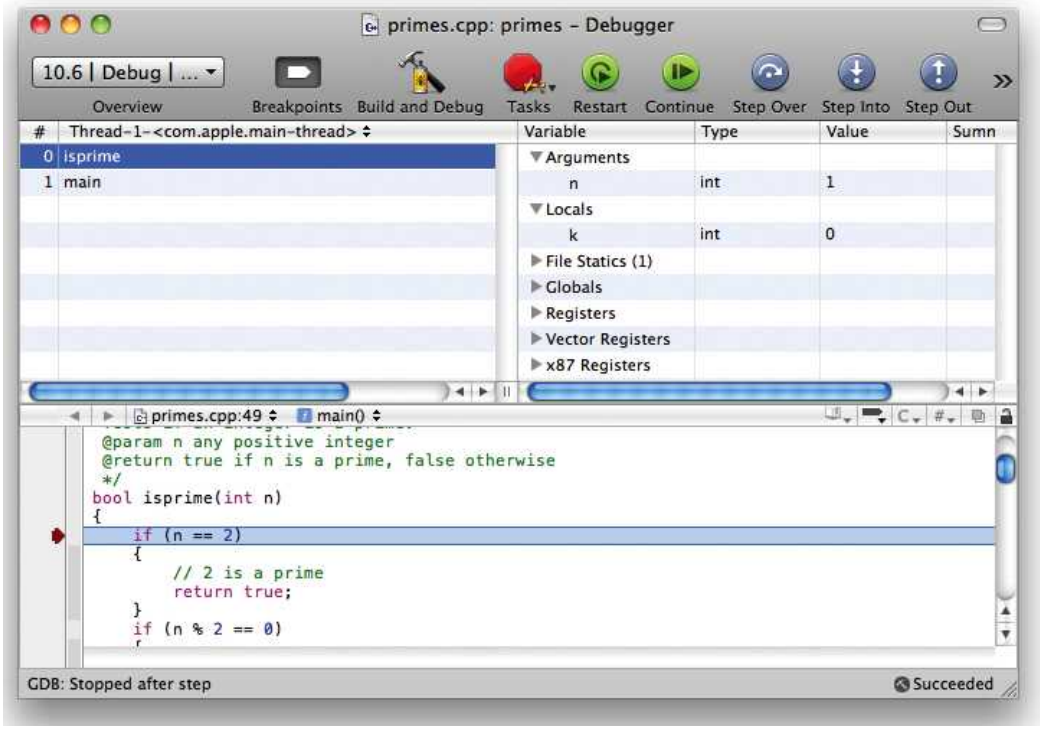
As you have undoubtedly realized by now, computer programs rarely run perfectly the first time. At times, it can be quite frustrating to find the errors, or bugs, as they are called by programmers. Of course, you can insert print statements into your code that show the program flow and values of key variables. You then run the program and try to analyze the printout. But if the printout does not clearly point to the problem, you need to add and remove print statements and run the program again. That can be a time-consuming process.

Modern development environments contain a **debugger**, a program that helps you locate bugs by letting you follow the execution of a program. You can stop and restart the program and see the contents of variables whenever the program is temporarily stopped. At each stop, you can decide how many program steps to run until the next stop.

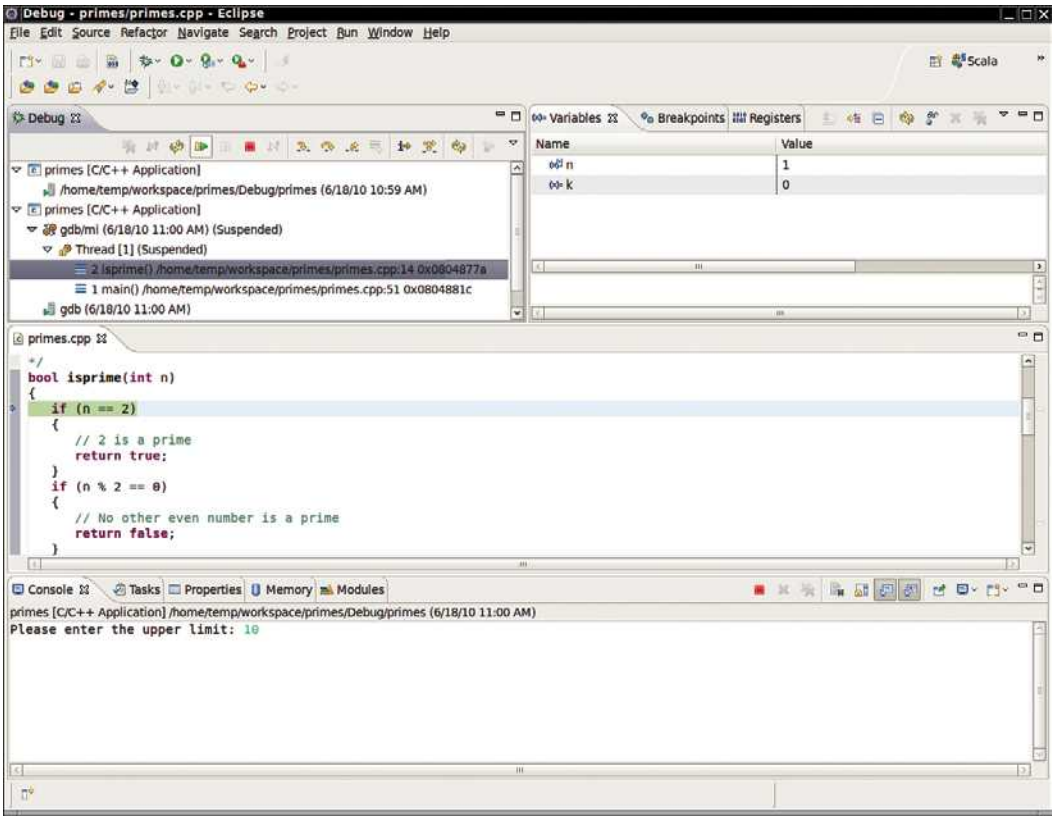
Step 1 Just like compilers, debuggers vary widely from one system to another. The debuggers of most integrated environments have a similar layout—see the examples below. You will have to find out how to prepare a program for debugging, and how to start the debugger on your system. With many development environments, you can simply pick a menu command to build your program for debugging and start the debugger.



The Visual Studio Debugger



The XCode Debugger



The Eclipse Debugger

Step 2 Once you have started the debugger, you can go a long way with just three debugging commands: "set breakpoint", "single step", and "inspect variable". The names and keystrokes or mouse clicks for these commands differ widely between debuggers, but all debuggers support these basic commands. You can find out how, either from the documentation or a lab manual, or by asking someone who has used the debugger before.

Step 3 When you start the debugger, it runs at full speed until it reaches a **breakpoint**. Then execution stops, and the breakpoint that causes the stop is displayed.

You can now inspect variables and step through the program a line at a time, or continue running the program at full speed until it reaches the next breakpoint. When the program terminates, the debugger stops as well.

Breakpoints stay active until you remove them, so you should periodically clear the breakpoints that you no longer need.

Once the program has stopped, you can look at the current values of variables. Some debuggers always show you a window with the current local variables. On other debuggers you issue a command such as "inspect variable" and type the variable name. If all variables contain what you expected, you can run the program until the next point where you want to stop.

Step 4 Running to a breakpoint gets you there speedily, but you don't know what the program did along the way. For a better understanding of the program flow, you can step through the program a line at a time. Most debuggers have two step commands, one usually called "step into", which steps inside function calls, and one called "step over", which skips over function calls. You should step into a function to check whether it carries out its job correctly. Step over a function if you know it works correctly.

Step 5 Finally, when the program has finished running, the debugging session is also finished. To run the program again, you need to start another debugging session.

A debugger can be an effective tool for finding and removing bugs in your program. However, it is no substitute for good design and careful programming. If the debugger does not find any errors, it does not mean that your program is bug-free. Testing and debugging can only show the presence of bugs, not their absence.

Sample Session

Here is a simple program for practicing the use of a debugger. The program is supposed to compute all prime numbers up to a number *n*. (An integer is defined to be prime if it is not evenly divisible by any number except by 1 and itself. Also, mathematicians find it convenient not to call 1 a prime. Thus, the first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19.)

worked_example_2/primes.cpp

```
#include <iostream>

using namespace std;

// Caution: This program has bugs.

/**
   Tests if an integer is a prime.
   @param n any positive integer
   @return true if n is a prime, false otherwise
*/
bool isprime(int n)
{
```

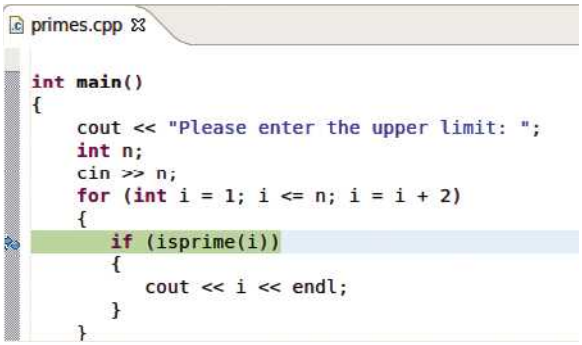
```
14    if (n == 2)
15    {
16       // 2 is a prime
17       return true;
18    }
19    if (n % 2 == 0)
20    {
21       // No other even number is a prime
22       return false;
23    }
24
25    // Try finding a number that divides n
26
27    int k = 3; // No need to divide by 2 since n is odd
28    // Only need to try divisors up to sqrt(n)
29    while (k * k < n)
30    {
31       if (n % k == 0)
32       {
33          // n is not a prime since it is divisible by k
34          return false;
35       }
36       // Try next odd number
37       k = k + 2;
38    }
39
40    // No divisor found. Therefore, n is a prime
41    return true;
42 }
43
44 int main()
45 {
46    cout << "Please enter the upper limit: ";
47    int n;
48    cin >> n;
49    for (int i = 1; i <= n; i = i + 2)
50    {
51       if (isprime(i))
52       {
53          cout << i << endl;
54       }
55     }
56     return 0;
57 }
```

When you run this program with an input of 10, then the output is

```
1
3
5
7
9
```

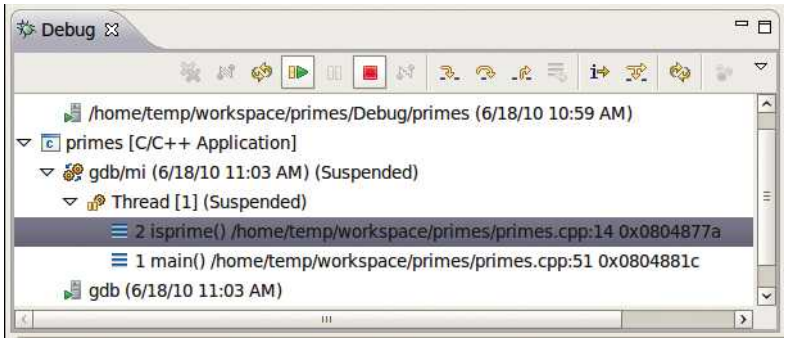
That is not very promising. It looks as if the program just prints all odd numbers. Let us find out what it does wrong by using the debugger.

First, set a breakpoint in line 51 and start debugging the program. On the way, the program will stop to input a value into `n`. Type 10 at the input prompt. The program will then stop at the breakpoint.

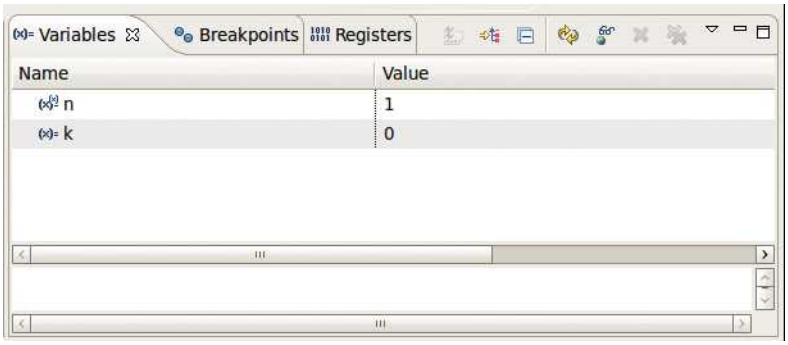


Now we wonder why the program treats 1 as a prime. Step into the `isprime` function.

Note the call stack display. It shows that the `isprime` function is currently active, and it is called by the `main` function.



Inspect the variable `n` to confirm that it is currently 1.



Execute the "step over" command a few times. You will notice that the program skips the two `if` statements. That's not surprising—1 is an odd number. Then the program skips over the `while` statement and is ready to return `true`, indicating that 1 is a prime.

Inspect the value of `k`. It is 3, which explains why the `while` loop was never entered. It looks like the `isprime` function needs to be rewritten to treat 1 as a special case.

Next, we would like to know why the program doesn't print 2 as a prime even though the `isprime` function recognizes that 2 is a prime. Continue the debugger. It will stop at the breakpoint in line 51.

Note that `i` is 3. Now it becomes clear. The `for` loop in the `main` function only tests odd numbers. Either `main` should test both odd and even numbers, or better, it should just handle 2 as a special case.

Finally, we would like to find out why the program believes 9 is a prime. Continue debugging until the breakpoint is hit with `i = 9`. Step into the `isprime` function. Now use "step over" repeatedly. The two `if` statements are skipped, which is correct since 9 is an odd number. The program again skips past the `while` loop. Inspect `k` to find out why. `k` is 3. Look at the condition in the `while` loop. It tests whether `k * k < n`. Now `k * k` is 9 and `n` is also 9, so the test fails.

When checking whether `n` is prime, it makes sense to only test divisors up to $\sqrt{n}$. If n can be factored as $p \times q$, *then the factors can't both be greater than* $\sqrt{n}$. But actually that isn't quite true. If n is a perfect square of a prime, then its sole nontrivial divisor is equal to $\sqrt{n}$. That is exactly the case for $9 = 3 \times 3$. We should have tested for `k * k <= n`.

By running the debugger, we discovered three bugs in the program:

- `isprime` falsely claims 1 to be a prime.
- `main` doesn't test 2.
- There is an off-by-one error in `isprime`. The condition of the `while` statement should be `k * k <= n`.

Here is the improved program:

```
#include <iostream>

using namespace std;

/**
   Tests if an integer is a prime.
   @param n any positive integer
   @return true if n is a prime, false otherwise
*/
bool isprime(int n)
{
   if (n == 1)
   {
      // 1 is not a prime
      return false;
   }
   if (n == 2)
   {
      // 2 is a prime
      return true;
   }
   if (n % 2 == 0)
   {
      // No other even number is a prime
      return false;
   }

   // Try finding a number that divides n

   int k = 3; // No need to divide by 2 since n is odd
   // Only need to try divisors up to sqrt(n)
   while (k * k <= n)
   {
      if (n % k == 0)
      {
         // n is not a prime since it is divisible by k
         return false;
      }
      // Try next odd number
      k = k + 2;
   }
```

```
42
43    // No divisor found. Therefore, n is a prime
44    return true;
45 }
46
47 int main()
48 {
49    cout << "Please enter the upper limit: ";
50    int n;
51    cin >> n;
52    if (n >= 2)
53    {
54       cout << 2 << endl;
55    }
56    for (int i = 3; i <= n; i = i + 2)
57    {
58       if (isprime(i))
59       {
60          cout << i << endl;
61       }
62    }
63    return 0;
64 }
```

Is our program now free from bugs? That is not a question the debugger can answer. Remember, testing can only show the presence of bugs, not their absence.

WORKED EXAMPLE 5.3

Calculating a Course Grade

Students in this course take four exams and earn a letter grade (A+, A, A–, B+, B, B–, C+, C, C–, D+, D, D–, or F) for each of them. The course grade is determined by dropping the lowest grade and averaging the three remaining grades. To average grades, first convert them to number grades, using the usual scheme A+ = 4.3, A = 4.0, A– = 3.7, B+ = 3.3, ..., D– = 0.7, F = 0. Then compute their average and convert it back to the closest letter grade. For example, an average of 3.51 would be an A–.

© paul kline/iStockphoto.

Problem Statement Your task is to read inputs of the form:

letter_grade1 letter_grade2 letter_grade3 letter_grade4

For example,

```
A- B+ C A
```

For each input line, your output should be

letter_grade

where the letter grade is the grade earned in the course, as just described. For example,

```
A-
```

The end of inputs will be indicated by a *letter_grade1* value of `Q`.

Step 1 Carry out stepwise refinement.

We will use the process of stepwise refinement. To process the inputs, we can process each line individually. Therefore, we define a task *process line*.

To process a line, we read the first grade and bail out if it is a `Q`. Otherwise, we read the four grades. Since we need them in their numeric form, we identify a task *convert letter grade to number*.

We then have four numbers and need to find the smallest one. That is another task, *find smallest of four numbers*. To average the remaining ones, we compute the sum of all values, subtract the smallest, and divide by three. Let's say that is not worth making into a subtask.

Next, we need to convert the result back into a letter grade. That is yet another subtask *convert number grade to letter*. Finally, we print the letter grade. That is again so simple that it requires no subtask.

Step 2 Convert letter grade to number.

How do we convert a letter grade to a number? Follow this algorithm:

```
grade_to_number(grade)
ch = first character of grade
If ch is A / B / C / D / F
   Set result to 4 / 3 / 2 / 1 / 0.
If the second character of grade is +
   Add 0.3 to result.
If the second character of grade is -
   Subtract 0.3 from result.
Return result.
```

Here is a function for that task.

```
/**
   Converts a letter grade to a number.
   @param grade a letter grade (A+, A, A–, ..., D–, F)
   @return the equivalent number grade
*/
double grade_to_number(string grade)
{
   double result = 0;
   string first = grade.substr(0, 1);
   if (first == "A") { result = 4; }
   else if (first == "B") { result = 3; }
   else if (first == "C") { result = 2; }
   else if (first == "D") { result = 1; }
   if (grade.length() > 1)
   {
      if (grade.substr(1, 1) == "+")
      {
         result = result + 0.3;
      }
      else
      {
         result = result - 0.3;
      }
   }
   return result;
}
```

Step 3 Convert number grade to letter.

How do we do the opposite conversion? Here, the challenge is that we need to convert to the *nearest* letter grade. For example, suppose x is 2.9. This value is closer to a B (23.0) than to a B– (2.7). When a value is exactly in the middle between two letter grade values, we give the higher grade. For example, 2.85 is rounded up to a B, and 3.15 is rounded up to a B+ (3.3). That is, any grade that is at least 2.85 but less than 3.15 is a B.

We can make a function with 13 branches, one for each valid letter grade.

```
/**
   Converts a number to the nearest letter grade.
   @param x a number between 0 and 4.3
   @return the nearest letter grade
*/
string number_to_grade(double x)
{
   if (x >= 4.15) { return "A+"; }
   if (x >= 3.85) { return "A"; }
   if (x >= 3.5) { return "A-"; }
   if (x >= 3.15) { return "B+"; }
   if (x >= 2.85) { return "B"; }
   if (x >= 2.5) { return "B-"; }
   if (x >= 2.15) { return "C+"; }
   if (x >= 1.85) { return "C"; }
   if (x >= 1.5) { return "C-"; }
   if (x >= 1.15) { return "D+"; }
   if (x >= 0.85) { return "D"; }
   if (x >= 0.5) { return "D-"; }
   return "F";
}
```

Exercise P5.5 suggests an alternate approach.

Step 4 Find the minimum of four numbers.

Finally, how do we find the smallest of four numbers? Let's suppose we can find the smallest of two numbers, with a function `min(x, y)`. Then the smallest of four numbers is `min(min(x1, x2), min(x3, x4))`. Finding the smallest of two numbers is easy:

```
/**
   Returns the smaller of two numbers.
   @param x a number
   @param y a number
   @return the smaller of x and y
*/
double min(double x, double y)
{
   if (x < y)
   {
      return x;
   }
   else
   {
      return y;
   }
}
```

Step 5 Process a line.

As previously described, to process a line, we read in the four input strings, convert grades to numbers, and compute the average after dropping the lowest grade. Then we print the grade corresponding to that average.

However, if we read the first input string and find a `Q`, we need to signal to the caller that we have reached the end of the input set and that no further calls should be made.

Our function will return a `bool` value: `true` if it was successful, `false` if it encountered the sentinel.

```
/**
   Processes one line of input.
   @return true if the sentinel was not encountered
*/
bool process_line()
{
   cout << "Enter four grades or Q to quit: ";
   string g1;
   cin >> g1;
   if (g1 == "Q") { return false; }
   string g2;
   string g3;
   string g4;
   cin >> g2 >> g3 >> g4;
   double x1 = grade_to_number(g1);
   double x2 = grade_to_number(g2);
   double x3 = grade_to_number(g3);
   double x4 = grade_to_number(g4);
   double xlow = min(min(x1, x2), min(x3, x4));
   double avg = (x1 + x2 + x3 + x4 - xlow) / 3;
   cout << number_to_grade(avg) << endl;
   return true;
}
```

Step 6 Write the `main` function.

The `main` function is now utterly trivial. We keep calling `process_line` while it returns `true`.

```
int main()
{
   while (process_line())
   {
   }
   return 0;
}
```

EXAMPLE CODE Here is the complete program.

worked_example_3/grades.cpp

```
#include <iostream>
#include <string>

using namespace std;

/**
   Converts a letter grade to a number.
   @param grade a letter grade (A+, A, A-, ..., D-, F)
   @return the equivalent number grade
*/
double grade_to_number(string grade)
{
   double result = 0;
   string first = grade.substr(0, 1);
   if (first == "A") { result = 4; }
   else if (first == "B") { result = 3; }
   else if (first == "C") { result = 2; }
   else if (first == "D") { result = 1; }
   if (grade.length() > 1)
   {
      if (grade.substr(1, 1) == "+")
      {
         result = result + 0.3;
      }
      else
      {
         result = result - 0.3;
      }
   }
   return result;
}

/**
   Converts a number to the nearest letter grade.
   @param x a number between 0 and 4.3
   @return the nearest letter grade
*/
string number_to_grade(double x)
{
   if (x >= 4.15) { return "A+"; }
   if (x >= 3.85) { return "A"; }
   if (x >= 3.5) { return "A-"; }
   if (x >= 3.15) { return "B+"; }
   if (x >= 2.85) { return "B"; }
   if (x >= 2.5) { return "B-"; }
   if (x >= 2.15) { return "C+"; }
   if (x >= 1.85) { return "C"; }
   if (x >= 1.5) { return "C-"; }
```

```
   if (x >= 1.15) { return "D+"; }
   if (x >= 0.85) { return "D"; }
   if (x >= 0.5) { return "D-"; }
   return "F";
}

/**
   Returns the smaller of two numbers.
   @param x a number
   @param y  a number
   @return the smaller of x and y
*/
double min(double x, double y)
{
   if (x < y)
   {
      return x;
   }
   else
   {
      return y;
   }
}

/**
   Processes one line of input.
   @return true if the sentinel was not encountered
*/
bool process_line()
{
   cout << "Enter four grades or Q to quit: ";
   string g1;
   cin >> g1;
   if (g1 == "Q") { return false; }
   string g2;
   string g3;
   string g4;
   cin >> g2 >> g3 >> g4;
   double x1 = grade_to_number(g1);
   double x2 = grade_to_number(g2);
   double x3 = grade_to_number(g3);
   double x4 = grade_to_number(g4);
   double xlow = min(min(x1, x2), min(x3, x4));
   double avg = (x1 + x2 + x3 + x4 - xlow) / 3;
   cout << number_to_grade(avg) << endl;
   return true;
}

int main()
{
   while (process_line())
   {
   }
   return 0;
}
```

CHAPTER 6

ARRAYS AND VECTORS

© traveler1116/iStockphoto.

CHAPTER GOALS

- To become familiar with using arrays and vectors to collect values
- To learn about common algorithms for processing arrays and vectors
- To write functions that process arrays and vectors
- To be able to use two-dimensional arrays

CHAPTER CONTENTS

In many programs, you need to collect large numbers of values. In standard C++, you use arrays and vectors for this purpose. Arrays are a fundamental structure of the C++ language. The standard C++ library provides the vector construct as a more convenient alternative when working with collections whose size is not fixed. In this chapter, you will learn about arrays, vectors, and common algorithms for processing them.

6.1 Arrays

We start this chapter by introducing the **array** data type. Arrays are the fundamental mechanism in C++ for collecting multiple values. In the following sections, you will learn how to define arrays and how to access array elements.

6.1.1 Defining Arrays

Suppose you write a program that reads a sequence of values and prints out the sequence, marking the largest value, like this:

```
32
54
67.5
29
34.5
80
115 <= largest value
44.5
100
65
```

You do not know which value to mark as the largest one until you have seen them all. After all, the last value might be the largest one. Therefore, the program must first store all values before it can print them.

Could you simply store each value in a separate variable? If you know that there are ten inputs, then you can store the values in ten variables `value1`, `value2`, `value3`, ..., `value10`. However, such a sequence of variables is not very practical to use. You would

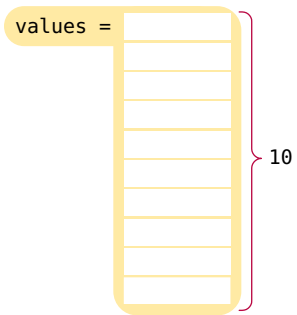


Figure 1 An Array of Size 10

have to write quite a bit of code ten times, once for each of the variables. To solve this problem, use an array: a structure for storing a sequence of values.

Here we define an array that can hold ten values:

```
double values[10];
```

Use an array to collect a sequence of values of the same type.

This is the definition of a variable `values` whose type is "array of `double`". That is, `values` stores a sequence of floating-point numbers. The `[10]` indicates the *size* of the array. (See Figure 1.) The array size must be a constant that is known at compile time.

When you define an array, you can specify the initial values. For example,

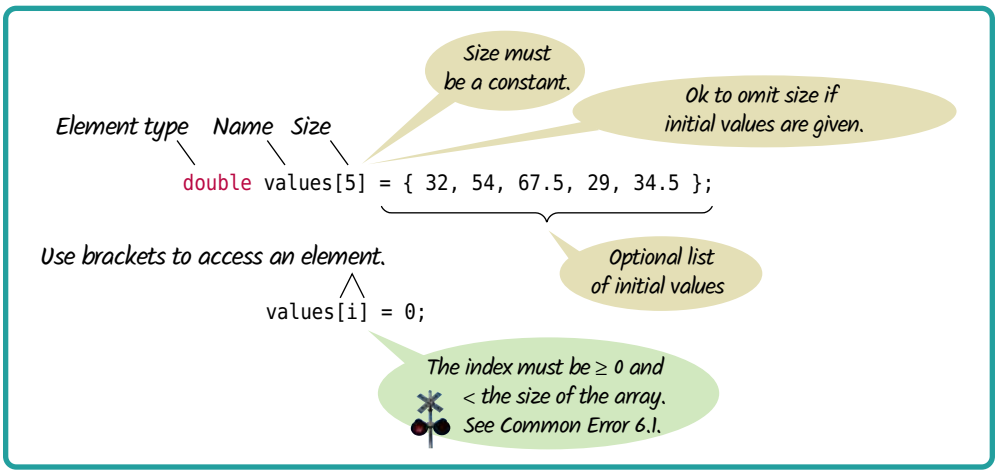
```
double values[] = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
```

When you supply initial values, you don't need to specify the array size. The compiler determines the size by counting the values.

Table 1 Defining Arrays

`int numbers[10];`	An array of ten integers.
`const int SIZE = 10;` `int numbers[SIZE];`	It is a good idea to use a named constant for the size.
`int size = 10;` `int numbers[size];`	**Caution:** In standard C++, the size must be a constant. This array definition will not work with all compilers.
`int squares[5] = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`int squares[] = { 0, 1, 4, 9, 16 };`	You can omit the array size if you supply initial values. The size is set to the number of initial values.
`int squares[5] = { 0, 1, 4 };`	If you supply fewer initial values than the size, the remaining values are set to 0. This array contains 0, 1, 4, 0, 0.
`string names[3];`	An array of three strings.

Syntax 6.1 Defining an Array



6.1.2 Accessing Array Elements

Individual elements in an array *values* are accessed by an integer index `i`, using the notation *values*`[i]`.

The values stored in an array are called its **elements**. Each element has a position number, called an **index**. To access a value in the `values` array, you must specify which index you want to use. That is done with the `[]` operator:

```
values[4] = 34.5;
```

Now the element with index 4 is filled with 34.5. (See Figure 2).

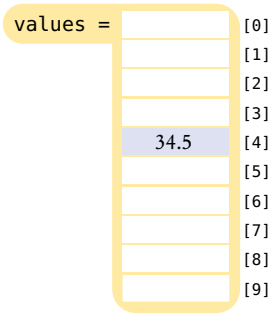


Figure 2 Filling an Array Element

You can display the contents of the element with index 4 with the following command:

```
cout << values[4] << endl;
```

An array element can be used like any variable.

As you can see, the element `values[4]` can be used like any variable of type `double`.

In C++, array positions are counted in a way that you may find surprising. If you look carefully at Figure 2, you will find that the *fifth* element was filled when we changed `values[4]`. In C++, the elements of arrays are numbered *starting at 0*. That is, the legal elements for the `values` array are

`values[0]`, the first element

`values[1]`, the second element

`values[2]`, the third element

`values[3]`, the fourth element

`values[4]`, the fifth element

. . .

`values[9]`, the tenth element

You will see in Chapter 7 why this numbering scheme was chosen in C++.

An array index must be at least zero and less than the size of the array.

You have to be careful about index values. Trying to access a element that does not exist in the array is a serious error. For example, if `values` has twenty elements, you are not allowed to access `values[20]`.

Like a post office box that is identified by a box number, an array element is identified by an index.

A bounds error, which occurs if you supply an invalid array index, can corrupt data or cause your program to terminate.

Attempting to access an element whose index is not within the valid index range is called a **bounds error**. The compiler does not catch this type of error. Even the running program generates *no error message*. If you make a bounds error, you silently read or overwrite another memory location. As a consequence, your program may have random errors, and it can even crash.

The most common bounds error is the following:

```
double values[10];
cout << values[10];
```

There is no `values[10]` in an array with ten elements—the legal index values range from 0 to 9.

To visit all elements of an array, use a variable for the index. Suppose `values` has ten elements and the integer variable `i` takes values 0, 1, 2, and so on, up to 9. Then the expression `values[i]` yields each element in turn. For example, this loop displays all elements:

```
for (int i = 0; i < 10; i++)
{
   cout << values[i] << endl;
}
```

Note that in the loop condition the index is *less than* 10 because there is no element corresponding to `values[10]`.

6.1.3 Partially Filled Arrays

With a partially filled array, you need to remember how many elements are filled.

An array cannot change size at run time. This is a problem when you don't know in advance how many elements you need. In that situation, you must come up with a good guess on the maximum number of elements that you need to store. We call this quantity the *capacity*. For example, we may decide that we sometimes want to store more than ten values, but never more than 100:

```
const int CAPACITY = 100;
double values[CAPACITY];
```

In a typical program run, only part of the array will be occupied by actual elements. We call such an array a *partially filled array*. You must keep a *companion variable* that counts how many elements are actually used. In Figure 3 we call the companion variable `current_size`.

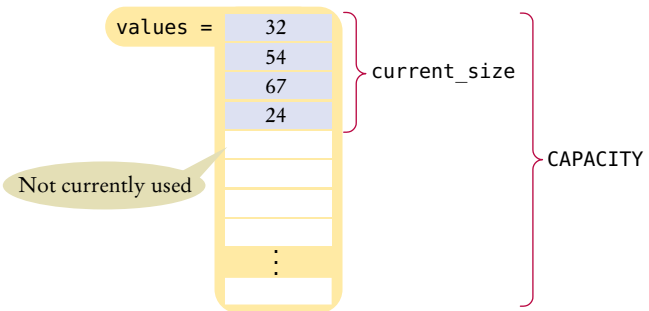


Figure 3 A Partially Filled Array

The following loop collects values and fills up the `values` array.

```
int current_size = 0;
double input;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

With a partially filled array, keep a companion variable for the current size.

At the end of this loop, `current_size` contains the actual number of elements in the array. Note that you have to stop accepting inputs if the size of the array reaches the capacity.

To process the gathered array elements, you again use the companion variable, not the capacity. This loop prints the partially filled array:

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

Common Error 6.1

Bounds Errors

Perhaps the most common error in using arrays is accessing a nonexistent element.

```
double values[10];
values[10] = 5.4;
   // Error—values has 10 elements with index values ranging from 0 to 9
```

If your program accesses an array through an out-of-bounds index, there is often no visible error message. Instead, the program may quietly (or not so quietly) corrupt some memory. Except for very short programs, in which the problem may go unnoticed, that corruption can make the program act unpredictably. Some C++ programming environments make an effort to report certain bounds errors, but they cannot detect them in all cases. Bounds errors are a serious issue (see also Computing & Society 6.1). You should develop the habit of scanning all array element accesses in your programs and asking yourself why they are valid.

Programming Tip 6.1

Use Arrays for Sequences of Related Values

Arrays are intended for storing sequences of values with the same meaning. For example, an array of test scores makes perfect sense:

```
int scores[NUMBER_OF_SCORES];
```

But an array

```
double personal_data[3];
```

that holds a person's age, bank balance, and shoe size in positions 0, 1, and 2 is bad design. It would be tedious for the programmer to remember which of these data values is stored in which array location. In this situation, it is far better to use three separate variables, or, as you will learn in Chapter 9, an object.

Computing & Society 6.1 Computer Viruses

In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected a significant fraction of computers connected to the Internet (which was much smaller then than it is now). Morris was sentenced to three years probation, 400 hours of community service, and a $10,000 fine.

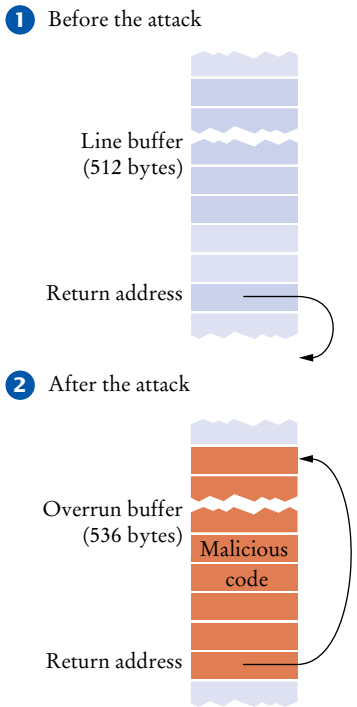
In order to attack a computer, a virus has to find a way to get its instructions executed. This particular program carried out a "buffer overrun" attack, providing an unexpectedly large input to a program on another machine. That program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, that program was written in the C programming language. C, like C++, does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. C programmers are supposed to provide safety checks, but that had not happened in the program under attack. The virus program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes overwrote a return address, which the attacker knew was stored just after the array. When the function that read the input was finished, it didn't return to its caller but to code supplied by the virus (see the figure). The virus was thus able to execute its code on a remote machine and infect it.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often take permanent residence in the attacked computers. Criminal enterprises rent out the processing power of millions of hijacked computers for sending spam e-mail. Other viruses monitor every keystroke and send those that look like credit card numbers or banking passwords to their master, or encrypt all disk files and demand a monetary ransom payment from the unfortunate user to unlock the files.

Typically, a machine gets infected because a user executes code downloaded from the Internet, clicking on an icon or link that purports to lead to something interesting, such as a game or video clip. Antivirus programs check all downloaded programs against an ever-growing list of known viruses.

When you use a computer for managing your finances, you need to be aware of the risk of infection. If a virus reads your banking password and empties your account, you will have a hard time convincing your financial institution that it wasn't your act, and you will most likely lose your money. It is a good idea to use banks that require "two-factor authentication" for major transactions, such as a callback on your cell phone.

Viruses are even used for military purposes. In 2010, a virus dubbed Stuxnet spread through Microsoft Windows and infected USB sticks. The virus looked for Siemens industrial computers and reprogrammed them in subtle ways. It appears that the virus was designed to damage the centrifuges of the Iranian nuclear enrichment operation. The computers controlling the centrifuges were not connected to the Internet, but they were configured with USB sticks, some of which were infected. Security researchers believe that the virus was developed by U.S. and Israeli intelligence agencies, and that it was successful in slowing down the Iranian nuclear program. Neither country has officially acknowledged or denied their role in the attacks.



A "Buffer Overrun" Attack

6.2 Common Array Algorithms

In the following sections, we discuss some of the most common algorithms for processing sequences of values. We present the algorithms so that you can use them with fully and partially filled arrays as well as vectors (which we will introduce in Section 6.7). When we use the expression *size of* values, you should replace it with a constant or variable that yields the number of elements in the array (or the expression `values.size()` if `values` is a vector.)

6.2.1 Filling

This loop fills an array with zeroes:

```
for (int i = 0; i < size of values; i++)
{
   values[i] = 0;
}
```

Next, let us fill an array `squares` with the numbers 0, 1, 4, 9, 16, and so on. Note that the element with index 0 contains 0^2, the element with index 1 contains 1^2, and so on.

```
for (int i = 0; i < size of squares; i++)
{
   squares[i] = i * i;
}
```

6.2.2 Copying

Consider two arrays:

```
int squares[5] = { 0, 1, 4, 9, 16 };
int lucky_numbers[5];
```

To copy an array, use a loop to copy its elements to a new array.

Now suppose you want to copy all values from the first array to the second. The following assignment is an error:

```
lucky_numbers = squares; // Error
```

In C++, you cannot assign one array to another. Instead, you must use a loop to copy all elements:

```
for (int i = 0; i < 5; i++)
{
   lucky_numbers[i] = squares[i];
}
```

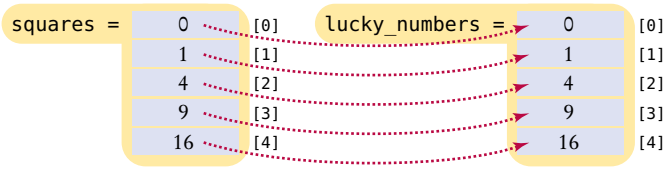


Figure 4 Copying Elements to Copy an Array

6.2.3 Sum and Average Value

You have already encountered this algorithm in Section 4.7.1. Here is the code for computing the sum of all elements in an array:

```
double total = 0;
for (int i = 0; i < size of values; i++)
{
   total = total + values[i];
}
```

To obtain the average, divide by the number of elements:

```
double average = total / size of values;
```

Be sure to check that the size is not zero.

6.2.4 Maximum and Minimum

© CEFutcher/iStockphoto.

Use the algorithm from Section 4.7.5 that keeps a variable for the largest element that you have encountered so far. Here is the implementation for arrays:

```
double largest = values[0];
for (int i = 1; i < size of values; i++)
{
   if (values[i] > largest)
   {
      largest = values[i];
   }
}
```

Note that the loop starts at 1 because we initialize `largest` with `values[0]`.

To compute the smallest value, reverse the comparison.

These algorithms require that the array contain at least one element.

6.2.5 Element Separators

When separating elements, don't place a separator before the first element.

When you display the elements of a collection, you usually want to separate them, often with commas or vertical lines, like this:

```
1 | 4 | 9 | 16 | 25
```

Note that there is one fewer separator than there are numbers. Print the separator before each element *except the initial one* (with index 0):

```
for (int i = 0; i < size of values; i++)
{
   if (i > 0)
   {
      cout << " | ";
   }
   cout << values[i];
}
```


© trutenka/iStockphoto.

To print five elements, you need four separators.

6.2.6 Counting Matches

Suppose you are asked to count how many elements of an array match a certain criterion. Visit each element, test whether it is a match, and if so, increment a counter. For example, the following loop counts the number of elements that are at least 100:

```
int count = 0;
for (int i = 0; i < size of values; i++)
{
   if (values[i] >= 100)
   {
      count++;
   }
}
```

6.2.7 Linear Search

A linear search inspects elements in sequence until a match is found.

© yekorzh/Getty Images.

To search for a specific element, visit the elements and stop when you encounter the match.

You often need to search for the position of an element so that you can replace or remove it. Visit all elements until you have found a match or you have come to the end of the array. Here we search for the position of the first element equal to 100.

```
int pos = 0;
bool found = false;
while (pos < size of values && !found)
{
   if (values[pos] == 100)
   {
      found = true;
   }
   else
   {
      pos++;
   }
}
```

If `found` is `true`, then `pos` is the position of the first match.

6.2.8 Removing an Element

Consider a partially filled array `values` whose current size is stored in the variable `current_size`. Suppose you want to remove the element with index `pos` from `values`. If the elements are not in any particular order, that task is easy to accomplish. Simply overwrite the element to be removed with the *last* element, then decrement the variable tracking the size. (See Figure 5.)

```
values[pos] = values[current_size - 1];
current_size--;
```

The situation is more complex if the order of the elements matters. Then you must move all elements following the element to be removed to a lower index, then decrement the variable holding the size of the array. (See Figure 6.)

```
for (int i = pos + 1; i < current_size; i++)
{
   values[i - 1] = values[i];
}
current_size--;
```

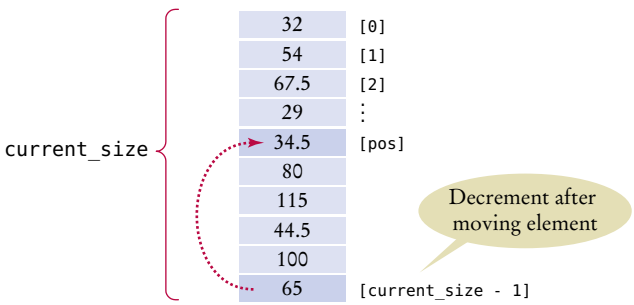


Figure 5
Removing an Element in an Unordered Array

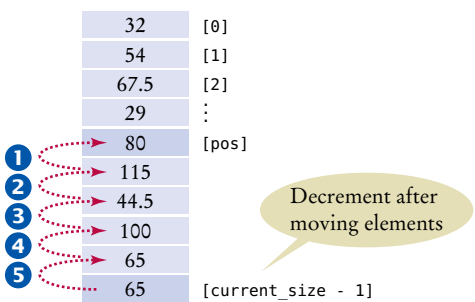


Figure 6
Removing an Element in an Ordered Array

6.2.9 Inserting an Element

If the order of the elements does not matter, you can simply insert new elements at the end, incrementing the variable tracking the size. (See Figure 7.) For a partially filled array:

```
if (current_size < CAPACITY)
{
   current_size++;
   values[current_size - 1] = new_element;
}
```

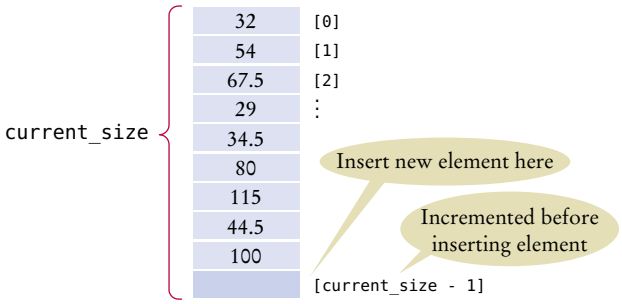


Figure 7
Inserting an Element in an Unordered Array

Before inserting an element, move elements to the end of the array *starting with the last one.*

It is more work to insert an element at a particular position in the middle of a sequence. First, increase the variable holding the current size. Next, move all elements above the insertion location to a higher index. Finally, insert the new element. Here is the code for a partially filled array:

```
if (current_size < CAPACITY)
{
   current_size++;
   for (int i = current_size - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = new_element;
}
```

Note the order of the movement: When you remove an element, you first move the next element down to a lower index, then the one after that, until you finally get to the end of the array. When you insert an element, you start at the end of the array, move that element to a higher index, then move the one before that, and so on until you finally get to the insertion location (see Figure 8).

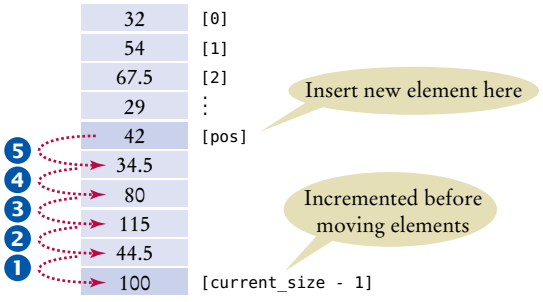
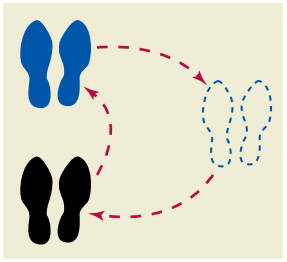


Figure 8
Inserting an Element in an Ordered Array

6.2.10 Swapping Elements



To swap two elements, you need a temporary variable.

You often need to swap elements of an array. For example, the sorting algorithm in Special Topic 6.2 sorts an array by repeatedly swapping elements.

Consider the task of swapping the elements at positions `i` and `j` of an array `values`. We'd like to set `values[i]` to `values[j]`. But that overwrites the value that is currently stored in `values[i]`, so we want to save that first:

```
double temp = values[i];
values[i] = values[j];
```

Use a temporary variable when swapping two elements.

Now we can set `values[j]` to the saved value.

```
values[j] = temp;
```

Figure 9 shows the process.

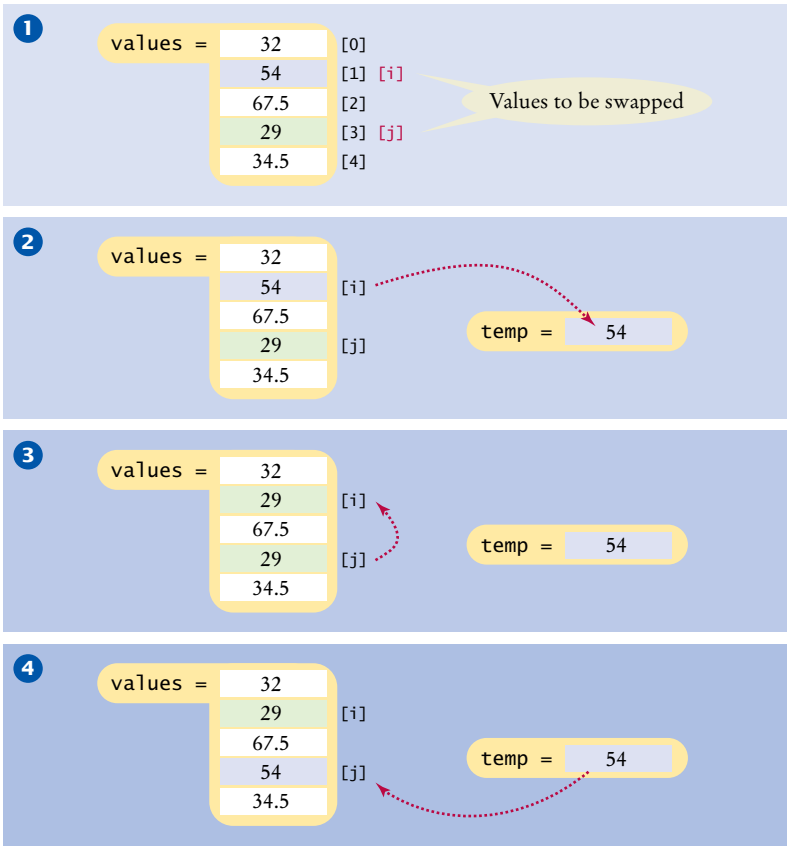


Figure 9 Swapping Array Elements

6.2.11 Reading Input

If you know how many input values the user will supply, it is simple to place them into an array:

```
double values[NUMBER_OF_INPUTS];
for (i = 0; i < NUMBER_OF_INPUTS; i++)
{
   cin >> values[i];
}
```

However, this technique does not work if you need to read an arbitrary number of inputs. In that case, add the values to an array until the end of the input has been reached.

```
double values[CAPACITY];
int current_size = 0;
double input;
while (cin >> input)
{
   if (current_size < CAPACITY)
   {
      values[current_size] = input;
      current_size++;
   }
}
```

Now `values` is a partially filled array, and the companion variable `current_size` is set to the number of input values.

This loop discards any inputs that won't fit in the array. A better approach would be to copy `values` to a new larger array when the capacity is reached (see Section 7.4).

The following program solves the task that we set ourselves at the beginning of this chapter, to mark the largest value in an input sequence:

sec02/largest.cpp

```
#include <iostream>

using namespace std;

int main()
{
   const int CAPACITY = 1000;
   double values[CAPACITY];
   int current_size = 0;

   cout << "Please enter values, Q to quit:" << endl;
   double input;
   while (cin >> input)
   {
      if (current_size < CAPACITY)
      {
         values[current_size] = input;
         current_size++;
      }
   }

   double largest = values[0];
   for (int i = 1; i < current_size; i++)
   {
```

```
25          if (values[i] > largest)
26          {
27             largest = values[i];
28          }
29       }
30
31       for (int i = 0; i < current_size; i++)
32       {
33          cout << values[i];
34          if (values[i] == largest)
35          {
36             cout << " <== largest value";
37          }
38          cout << endl;
39       }
40
41       return 0;
42    }
```

Program Run

```
Please enter values, Q to quit:
34.5 80 115 44.5 Q
34.5
80
115 <== largest value
44.5
```

Special Topic 6.1

Sorting with the C++ Library

You often want to sort the elements of an array or vector. Special Topic 6.2 shows you a sorting algorithm that is relatively simple but not very efficient. Efficient sorting algorithms are significantly more complex. Fortunately, the C++ library provides an efficient `sort` function. To sort an array `a` with `size` elements, call

```
sort(a, a + size);
```

To sort a vector `values`, make this call:

```
sort(values.begin(), values.end());
```

To fully understand why the `sort` function is called in this way, you will need to know advanced C++ that is beyond the scope of this book. But don't hesitate to call the `sort` function whenever you need to sort an array or vector.

To use the `sort` function, include the `<algorithm>` header in your program.

Special Topic 6.2

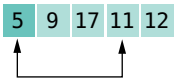
A Sorting Algorithm

A *sorting algorithm* rearranges the elements of a sequence so that they are stored in sorted order. Here is a simple sorting algorithm, called **selection sort**. Consider sorting the following array `values`:

[0]	[1]	[2]	[3]	[4]
11	9	17	5	12

An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in `values[3]`. You should move the 5 to the beginning of the array. Of course, there is already

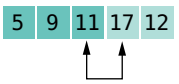
an element stored in `values[0]`, namely 11. Therefore you cannot simply move `values[3]` into `values[0]` without moving the 11 somewhere else. You don't yet know where the 11 should end up, but you know for certain that it should not be in `values[0]`. Simply get it out of the way by *swapping it* with `values[3]`:



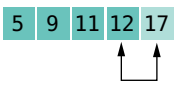
Now the first element is in the correct place. In the foregoing figure, the darker color indicates the portion of the array that is already sorted.

Next take the minimum of the remaining entries `values[1]...values[4]`. That minimum value, 9, is already in the correct place. You don't need to do anything in this case, simply extend the sorted area by one to the right:

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:



Now the unsorted region is only two elements long; keep to the same successful strategy. The minimum element is 12. Swap it with the first value, 17:



That leaves you with an unprocessed region of length 1, but of course a region of length 1 is always sorted. You are done.

Here is the C++ code:

```
for (int unsorted = 0; unsorted < size - 1; unsorted++)
{
   // Find the position of the minimum
   int min_pos = unsorted;
   for (int i = unsorted + 1; i < size; i++)
   {
      if (values[i] < values[min_pos]) { min_pos = i; }
   }
   // Swap the minimum into the sorted area
   if (min_pos != unsorted)
   {
      double temp = values[min_pos];
      values[min_pos] = values[unsorted];
      values[unsorted] = temp;
   }
}
```

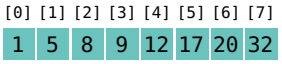
This algorithm is simple to understand, but it is not very efficient. Computer scientists have studied sorting algorithms extensively and discovered significantly better algorithms. The `sort` function of the C++ library provides one such algorithm—see Special Topic 6.1.

Special Topic 6.3

Binary Search

When an array is sorted, there is a much faster search algorithm than the linear search of Section 6.2.7.

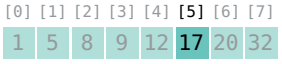
Consider the following sorted array `values`:



We would like to see whether the value 15 is in the array. Let's narrow our search by finding whether the value is in the first or second half of the array. The last point in the first half of the data set, `values[3]`, is 9, which is smaller than the value we are looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:

Now the last value of the first half of this sequence is 17; hence, the value must be located in the sequence:

The last value of the first half of this very short sequence is 12, which is smaller than the value that we are searching, so we must look in the second half:



We still don't have a match because $15 \neq 17$, and we cannot divide the subsequence further. If we wanted to insert 15 into the sequence, we would need to insert it just before `values[5]`.

This search process is called a **binary search**, because we cut the size of the search in half in each step. That cutting in half works only because we know that the sequence of values is sorted. Here is an implementation in C++:

```
bool found = false;
int low = 0;
int high = size - 1;
int pos = 0;
while (low <= high && !found)
{
   pos = (low + high) / 2; // Midpoint of the subsequence
   if (values[pos] == searched_value) { found = true; }
   else if (values[pos] < searched_value) { low = pos + 1; } // Look in second half
   else { high = pos - 1; } // Look in first half
}
if (found) { cout << "Found at position " << pos; }
else { cout << "Not found. Insert before position " << pos; }
```

6.3 Arrays and Functions

In this section, we will explore how to write functions that process arrays.

When passing an array to a function, also pass the size of the array.

A function that processes the values in an array needs to know the number of valid elements in the array. For example, here is a `sum` function that computes the sum of all elements in an array:

```
double sum(double values[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
```

```
      total = total + values[i];
   }
   return total;
}
```

Note the special syntax for array parameter variables. When writing an array parameter variable, you place an empty `[]` behind the parameter name. Do not specify the size of the array inside the brackets.

When you call the function, supply both the name of the array and the size. For example,

```
const int NUMBER_OF_SCORES = 10;
double scores[NUMBER_OF_SCORES]
   = { 32, 54, 67.5, 29, 34.5, 80, 115, 44.5, 100, 65 };
double total_score = sum(scores, NUMBER_OF_SCORES);
```

You can also pass a smaller size to the function:

```
double partial_score = sum(scores, 5);
```

This call computes the sum of the first five elements of the `scores` array. Remember, the function has no way of knowing how many elements the array has. It simply relies on the size that the caller provides.

Array parameters are always reference parameters.

Array parameters are *always reference parameters*. (You will see the reason in Chapter 7.) Functions can modify array arguments, and those modifications affect the array that was passed into the function. For example, the following `multiply` function updates all elements in the array:

```
void multiply(double values[], int size, double factor)
{
   for (int i = 0; i < size; i++)
   {
      values[i] = values[i] * factor;
   }
}
```

You do *not* use an `&` symbol to denote the reference parameter in this case.

A function's return type cannot be an array.

Although arrays can be function arguments, they cannot be function return types. If a function computes multiple values, the caller of the function must provide an array parameter variable to hold the result.

```
void squares(int n, int result[])
{
   for (int i = 0; i < n; i++)
   {
      result[i] = i * i;
   }
}
```

When a function modifies the size of an array, it needs to tell its caller.

When a function changes the size of an array, it should indicate to the caller how many elements the array has after the call. The easiest way to do this is to return the new size. Here is an example—a function that adds input values to an array:

```
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
```

```
            inputs[current_size] = input;
            current_size++;
         }
      }
      return current_size;
   }
```

A function that adds elements to an array needs to know its capacity.

Note that this function also needs to know the capacity of the array. Generally, a function that adds elements to an array needs to know its capacity. You would call this function like this:

```
const int MAXIMUM_NUMBER_OF_VALUES = 1000;
double values[MAXIMUM_NUMBER_OF_VALUES];
int current_size = read_inputs(values, MAXIMUM_NUMBER_OF_VALUES);
   // values is a partially filled array; the current_size variable specifies its size
```

Alternatively, you can pass the size as a reference parameter. This is more appropriate for functions that modify an existing array:

```
void append_inputs(double inputs[], int capacity, int& current_size)
{
   double input;
   while (cin >> input)
   {
      if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
}
```

This function is called as

```
append_inputs(values, MAXIMUM_NUMBER_OF_VALUES, current_size);
```

After the call, the `current_size` variable contains the new size.

The following example program reads values from standard input, doubles them, and prints the result. The program uses three functions:

- The `read_inputs` function fills an array with the input values. It returns the number of elements that were read.
- The `multiply` function modifies the contents of the array that it receives, demonstrating that arrays are passed by reference.
- The `print` function does not modify the contents of the array that it receives.

sec03/functions.cpp

```
 1  #include <iostream>
 2
 3  using namespace std;
 4
 5  /**
 6     Reads a sequence of floating-point numbers.
 7     @param inputs an array containing the numbers
 8     @param capacity the capacity of that array
 9     @return the number of inputs stored in the array
10  */
11  int read_inputs(double inputs[], int capacity)
12  {
```

```
13    int current_size = 0;
14    cout << "Please enter values, Q to quit:" << endl;
15    bool more = true;
16    while (more)
17    {
18       double input;
19       cin >> input;
20       if (cin.fail())
21       {
22          more = false;
23       }
24       else if (current_size < capacity)
25       {
26          inputs[current_size] = input;
27          current_size++;
28       }
29    }
30    return current_size;
31 }
32
33 /**
34    Multiplies all elements of an array by a factor.
35    @param values a partially filled array
36    @param size the number of elements in values
37    @param factor the value with which each element is multiplied
38 */
39 void multiply(double values[], int size, double factor)
40 {
41    for (int i = 0; i < size; i++)
42    {
43       values[i] = values[i] * factor;
44    }
45 }
46
47 /**
48    Prints the elements of a vector, separated by commas.
49    @param values a partially filled array
50    @param size the number of elements in values
51 */
52 void print(double values[], int size)
53 {
54    for (int i = 0; i < size; i++)
55    {
56       if (i > 0) { cout << ", "; }
57       cout << values[i];
58    }
59    cout << endl;
60 }
61
62 int main()
63 {
64    const int CAPACITY = 1000;
65    double values[CAPACITY];
66    int size = read_inputs(values, CAPACITY);
67    multiply(values, size, 2);
68    print(values, size);
69
70    return 0;
71 }
```

Program Run

```
Please enter values, Q to quit:
12 25 20 Q
24, 50, 40
```

Special Topic 6.4

Constant Array Parameters

When a function doesn't modify an array parameter, it is considered good style to add the `const` reserved word, like this:

```
double sum(const double values[], int size)
```

The `const` reserved word helps the reader of the code, making it clear that the function keeps the array elements unchanged. If the implementation of the function tries to modify the array, the compiler issues a warning.

6.4 Problem Solving: Adapting Algorithms

By combining fundamental algorithms, you can solve complex programming tasks.

In Section 6.2, you were introduced to a number of fundamental array algorithms. These algorithms form the building blocks for many programs that process arrays. In general, it is a good problem-solving strategy to have a repertoire of fundamental algorithms that you can combine and adapt.

Consider this example problem: You are given the quiz scores of a student. You are to compute the final quiz score, which is the sum of all scores after dropping the lowest one. For example, if the scores are

```
8  7  8.5  9.5  7  4  10
```

then the final score is 50.

We do not have a ready-made algorithm for this situation. Instead, consider which algorithms may be related. These include:

- Calculating the sum (Section 6.2.3)
- Finding the minimum value (Section 6.2.4)
- Removing an element (Section 6.2.8)

Now we can formulate a plan of attack that combines these algorithms.

Find the minimum.
Remove the minimum from the array.
Calculate the sum.

Let's try it out with our example. The minimum of

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

is 4. How do we remove it?

Now we have a problem. The removal algorithm in Section 6.2.8 locates the element to be removed by using the *position* of the element, not the value.

But we have another algorithm for that:

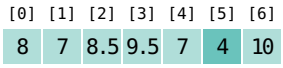
- Linear search (Section 6.2.7)

We need to fix our plan of attack:

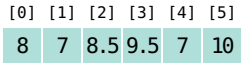
Find the minimum value.
Find the position of the minimum.
Remove the element at the position from the array.
Calculate the sum.

Will it work? Let's continue with our example.

We found a minimum value of 4. Linear search tells us that the value 4 occurs at position 5.



We remove it:



Finally, we compute the sum: 8 + 7 + 8.5 + 9.5 + 7 + 10 = 50.

This walkthrough demonstrates that our strategy works.

Can we do better? It seems a bit inefficient to find the minimum and then make another pass through the array to obtain its position.

You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

We can adapt the algorithm for finding the minimum to yield the position of the minimum. Here is the original algorithm:

```
double smallest = values[0];
for (int i = 1; i < size of values; i++)
{
   if (values[i] < smallest)
   {
      smallest = values[i];
   }
}
```

When we find the smallest value, we also want to update the position:

```
   if (values[i] < smallest)
   {
      smallest = values[i];
      smallest_position = i;
   }
```

In fact, then there is no reason to keep track of the smallest value any longer. It is simply `values[smallest_position]`. With this insight, we can adapt the algorithm as follows:

```
int smallest_position = 0;
for (int i = 1; i < size of values; i++)
{
   if (values[i] < values[smallest_position])
   {
      smallest_position = i;
   }
}
```

With this adaptation, our problem is solved with the following strategy:

Find the position of the minimum.
Remove the element at the position from the array.
Calculate the sum.

EXAMPLE CODE See sec04 of your companion code for a program that implements this strategy. How To 6.1 shows an alternate approach that does not modify the array.

The next section shows you a technique for discovering a new algorithm when none of the fundamental algorithms can be adapted to a task.

HOW TO 6.1

Working with Arrays

In many data processing situations, you need to process a sequence of values. This How To walks you through the steps for storing input values in an array and carrying out computations with the array elements.

Problem Statement Consider again the problem from Section 6.4: A final quiz score is computed by adding all the scores, except for the lowest one. For example, if the scores are

```
8  7  8.5  9.5  7  5  10
```

then the final score is 50.

Thierry Dosogne/The Image Bank/Getty Images, Inc.

Step 1 Decompose your task into steps.

You will usually want to break down your task into multiple steps, such as

- Reading the data into an array.
- Processing the data in one or more steps.
- Displaying the results.

When deciding how to process the data, you should be familiar with the array algorithms in Section 6.2. Most processing tasks can be solved by using one or more of these algorithms.

In our sample problem, we will want to read the data. Then we will remove the minimum and compute the total. For example, if the input is `8 7 8.5 9.5 7 5 10`, we will remove the minimum of `5`, yielding `8 7 8.5 9.5 7 10`. The sum of those values is the final score of `50`.

Thus, we have identified three steps:

Read inputs.
Remove the minimum.
Calculate the sum.

Step 2 Determine which algorithm(s) you need.

Sometimes, a step corresponds to exactly one of the basic array algorithms in Section 6.2. That is the case with calculating the sum (Section 6.2.3) and reading the inputs (Section 6.2.11). At other times, you need to combine several algorithms. To remove the minimum value, you can find the minimum value (Section 6.2.4), find its position (Section 6.2.7), and remove the element at that position (Section 6.2.8).

We have now refined our plan as follows:

Read inputs.
Find the minimum.
Find the position of the minimum.
Remove the element at the position.
Calculate the sum.

This plan will work—see Section 6.4. But here is an alternate approach. It is easy to compute the sum and subtract the minimum. Then we don't have to find its position. The revised plan is

Read inputs.
Find the minimum.
Calculate the sum.
Subtract the minimum from the sum.

Step 3 Use functions to structure the program.

Even though it may be possible to put all steps into the `main` function, this is rarely a good idea. It is better to make each processing step into a separate function. In our example, we will implement three functions:

- `read_inputs`
- `sum`
- `minimum`

The `main` function simply calls these functions:

```
const int CAPACITY = 1000;
double scores[CAPACITY];
int scores_size = read_inputs(scores, CAPACITY);
double total = sum(scores, scores_size) - minimum(scores, scores_size);
```

For each function that processes an array, you will need to pass the array itself and the array size. For example,

```
double sum(double values[], int size)
```

If the function modifies the size, it needs to tell the caller what the new size is. The function reading inputs returns the new size:

```
int read_inputs(double inputs[], int capacity) // Returns the size
```

Step 4 Assemble and test the program.

Combine your functions into a program. Review your code and check that you handle both normal and exceptional situations. What happens with an empty array? One that contains a single element? When no match is found? When there are multiple matches? Consider these boundary conditions and make sure that your program works correctly.

In our example, it is impossible to compute the minimum if the array is empty. In that case, we should terminate the program with an error message *before* attempting to call the `minimum` function.

What if the minimum value occurs more than once? That means that a student had more than one test with the same low score. We subtract only one of the occurrences of that low score, and that is the desired behavior.

The following table shows test cases and their expected output:

Test Case	Expected Output	Comment
`8 7 8.5 9.5 7 5 10`	50	See Step 1.
`8 7 7 9`	24	Only one instance of the low score should be removed.
`8`	0	After removing the low score, no score remains.
(no inputs)	**Error**	That is not a legal input.

This `main` function completes the solution. Here is the complete program:

how_to_1/scores.cpp

```
#include <iostream>

using namespace std;

/**
   Reads a sequence of floating-point numbers.
   @param inputs an array containing the numbers
   @param capacity the capacity of that array
   @return the number of inputs stored in the array
*/
int read_inputs(double inputs[], int capacity)
{
   int current_size = 0;
   cout << "Please enter values, Q to quit:" << endl;
   bool done = false;
   while (!done)
   {
      double input;
      cin >> input;
      if (cin.fail())
      {
         done = true;
      }
      else if (current_size < capacity)
      {
         inputs[current_size] = input;
         current_size++;
      }
   }
   return current_size;
}

/**
   Gets the minimum value from an array.
   @param values a partially filled array of size >= 1
   @param size the number of elements in values
   @return the smallest element in values
*/
double minimum(double values[], int size)
{
   double smallest = values[0];
   for (int i = 1; i < size; i++)
   {
      if (values[i] < smallest)
      {
         smallest = values;
      }
   }
   return smallest;
}

/**
   Computes the sum of the elements in an array.
   @param values a partially filled array
   @param size the number of elements in values
   @return the sum of the elements in values
*/
```

```
58 double sum(double values[], int size)
59 {
60    double total = 0;
61    for (int i = 0; i < size; i++)
62    {
63       total = total + values[i];
64    }
65    return total;
66 }
67
68 int main()
69 {
70    const int CAPACITY = 1000;
71    double scores[CAPACITY];
72    int size = read_inputs(scores, CAPACITY);
73    if (size == 0)
74    {
75       cout << "At least one score is required." << endl;
76    }
77    else
78    {
79       double final_score = sum(scores, size) - minimum(scores, size);
80       cout << "Final score: " << final_score << endl;
81    }
82    return 0;
83 }
```

EXAMPLE CODE See `how_to_1/scores_vector` in your companion code for a solution using vectors instead of arrays.

WORKED EXAMPLE 6.1

Rolling the Dice

Learn how to analyze a set of die tosses to see whether the die is "fair". See your E-Text or visit `wiley.com/go/bclo3`.

© ktsimage/iStockphoto.

6.5 Problem Solving: Discovering Algorithms by Manipulating Physical Objects

In Section 6.4, you saw how to solve a problem by combining and adapting known algorithms. But what do you do when none of the standard algorithms is sufficient for your task? In this section, you will learn a technique for discovering algorithms by manipulating physical objects.

Consider the following task. You are given an array whose size is an even number, and you are to switch the first and the second half. For example, if the array contains the eight numbers

9	13	21	4	11	7	1	3

then you should change it to

11	7	1	3	9	13	21	4

Many students find it quite challenging to come up with an algorithm. They may know that a loop is required, and they may realize that elements should be inserted (Section 6.2.9) or swapped (Section 6.2.10), but they do not have sufficient intuition to draw diagrams, describe an algorithm, or write down pseudocode.

© JenCon/iStockphoto.

Manipulating physical objects can give you ideas for discovering algorithms.

One useful technique for discovering an algorithm is to manipulate physical objects. Start by lining up some objects to denote an array. Coins, playing cards, or small toys are good choices.

Use a sequence of coins, playing cards, or toys to visualize an array of values.

Here we arrange eight coins.

coins: © jamesbenet/iStockphoto; dollar coins: © JordiDelgado/iStockphoto.

Now let's step back and see what we can do to change the order of the coins. We can remove a coin (Section 6.2.8):

Visualizing the removal of an array element

We can insert a coin (Section 6.2.9):

Visualizing the insertion of an array element

Or we can swap two coins (Section 6.2.10).

Visualizing the swapping of two coins

Go ahead—line up some coins and try out these three operations right now so that you get a feel for them.

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as C++ programmers, we will say that we swap the coins in positions 0 and 4:

Next, we swap the coins in positions 1 and 5:

Two more swaps, and we are done:

Now an algorithm is becoming apparent:

```
i = 0
j = . . . // We'll think about that in a minute
While // Don't know yet
   Swap elements at positions i and j.
   i++
   j++
```

Where does the variable *j* start? When we have eight coins, the coin at position zero is moved to position 4. In general, it is moved to the middle of the array, or to position *size* / 2.

And how many iterations do we make? We need to swap all coins in the first half. That is, we need to swap *size / 2* coins. The pseudocode is

```
i = 0
j = size / 2
While i < size / 2
   Swap elements at positions i and j.
   i++
   j++
```

You can use paper clips as position markers or counters.

It is a good idea to make a walkthrough of the pseudocode (see Section 4.2). You can use paper clips to denote the positions of the variables *i* and *j*. If the walkthrough is successful, then we know that there was no "off-by-one" error in the pseudocode. Exercise E6.6 asks you to translate the pseudocode to C++. Exercise R6.19 suggests a different algorithm for switching the two halves of an array, by repeatedly removing and inserting coins.

Many people find that the manipulation of physical objects is less intimidating than drawing diagrams or mentally envisioning algorithms. Give it a try when you need to design a new algorithm!

6.6 Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout. Such data sets commonly occur in financial and scientific applications. An arrangement consisting of rows and columns of values is called a **two-dimensional array**, or a *matrix*.

Let's explore how to store the example data shown in Figure 10: the medal counts of the figure skating competitions at the 2014 Winter Olympics.

© Trub/iStockphoto.

© technotr/iStockphoto.

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Figure 10 Figure Skating Medal Counts

6.6.1 Defining Two-Dimensional Arrays

Use a two-dimensional array to store tabular data.

C++ uses an array with row and column index values to store a two-dimensional array. For example, here is the definition of an array with 8 rows and 3 columns, suitable for storing our medal count data:

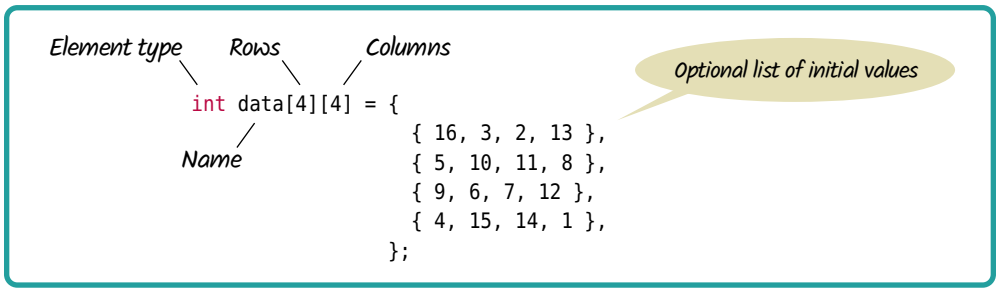
```
const int COUNTRIES = 8;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

You can initialize the array by grouping each row, as follows:

```
int counts[COUNTRIES][MEDALS] =
   {
      { 0, 3, 0 },
      { 0, 0, 1 },
      { 0, 0, 1 },
      { 1, 0, 0 },
      { 0, 0, 1 },
      { 3, 1, 1 },
      { 0, 1, 0 }
      { 1, 0, 1 }
   };
```

Just as with one-dimensional arrays, you cannot change the size of a two-dimensional array once it has been defined.

Syntax 6.2 Two-Dimensional Array Definition



6.6.2 Accessing Elements

Individual elements in a two-dimensional array are accessed by using two index values, *array*`[i][j]`.

To access a particular element in the two-dimensional array, you need to specify two index values in separate brackets to select the row and column, respectively (see Syntax 6.2 and Figure 11):

```
int value = counts[3][1];
```

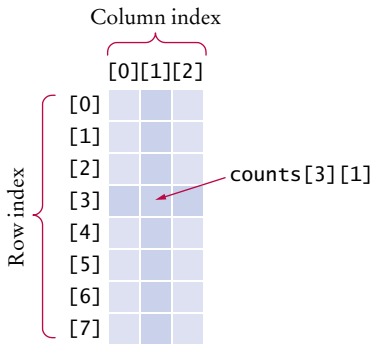


Figure 11
Accessing an Element in a Two-Dimensional Array

To access all values in a two-dimensional array, you use two nested loops. For example, the following loop prints all elements of `counts`.

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      cout << setw(8) << counts[i][j];
   }
   cout << endl; // Start a new line at the end of the row
}
```

6.6.3 Locating Neighboring Elements

Some programs that work with two-dimensional arrays need to locate the elements that are adjacent to an element. This task is particularly common in games. Figure 12 shows how to compute the index values of the neighbors of an element.

For example, the neighbors of `counts[3][1]` to the left and right are `counts[3][0]` and `counts[3][2]`. The neighbors to the top and bottom are `counts[2][1]` and `counts[4][1]`.

You need to be careful about computing neighbors at the boundary of the array. For example, `counts[0][1]` has no neighbor to the top. Consider the task of computing the sum of the neighbors to the top and bottom of the element `count[i][j]`. You need to check whether the element is located at the top or bottom of the array:

```
int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

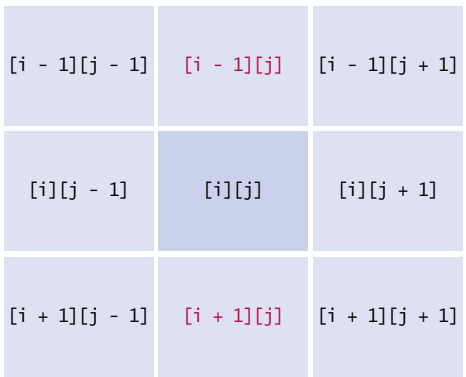
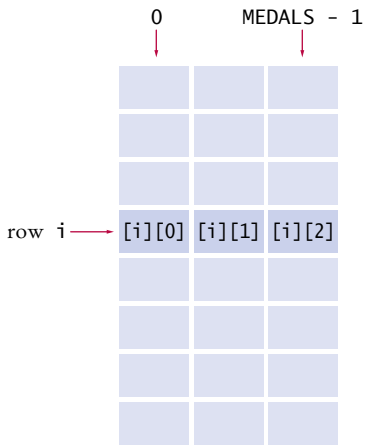


Figure 12 Neighboring Locations in a Two-Dimensional Array

6.6.4 Computing Row and Column Totals

A common task is to compute row or column totals. In our example, the row totals give us the total number of medals won by a particular country.

Finding the right index values is a bit tricky, and it is a good idea to make a quick sketch. To compute the total of row `i`, we need to visit the following elements:

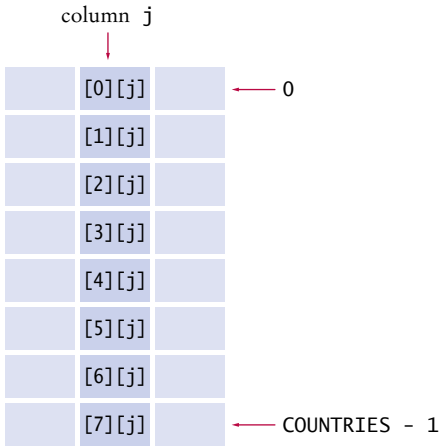


As you can see, we need to compute the sum of `counts[i][j]`, where `j` ranges from `0` to `MEDALS - 1`. The following loop computes the total:

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
   total = total + counts[i][j];
}
```

Computing column totals is similar. Form the sum of `counts[i][j]`, where `i` ranges from `0` to `COUNTRIES - 1`.

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
   total = total + counts[i][j];
}
```



6.6.5 Two-Dimensional Array Parameters

A two-dimensional array parameter must have a fixed number of columns.

When defining a function with a two-dimensional array parameter, you must specify the number of columns *as a constant* with the parameter type. For example, this function computes the total of a given row:

```
const int COLUMNS = 3;

int row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}
```

This function can compute row totals of a two-dimensional array with an arbitrary number of rows, but the array must have 3 columns. You have to write a different function if you want to compute row totals of a two-dimensional array with 4 columns.

To understand this limitation, you need to know how the array elements are stored in memory. Although the array appears to be two-dimensional, the elements are still stored as a linear sequence. Figure 13 shows how the `counts` array is stored, row by row.

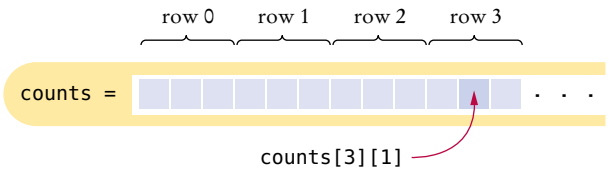


Figure 13 A Two-Dimensional Array is Stored as a Sequence of Rows

For example, to reach

```
counts[3][1]
```

the program must first skip past rows 0, 1, and 2 and then locate offset 1 in row 3. The offset from the start of the array is

$$3 \times \textit{number of columns} + 1$$

Now consider the `row_total` function. The compiler generates code to find the element

```
table[i][j]
```

by computing the offset

```
i * COLUMNS + j
```

The compiler uses the value that you supplied in the second pair of brackets when declaring the parameter:

```
int row_total(int table[][COLUMNS], int row)
```

Note that the first pair of brackets should be empty, just as with one-dimensional arrays.

The row_total function did not need to know the number of rows of the array. If the number of rows is required, pass it as a variable, as in this example:

```
int column_total(int table[][COLUMNS], int rows, int column)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][column];
   }
   return total;
}
```

Working with two-dimensional arrays is illustrated in the following program. The program prints out the medal counts and the row totals.

sec06/medals.cpp

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

const int COLUMNS = 3;

/**
   Computes the total of a row in a table.
   @param table a table with 3 columns
   @param row the row that needs to be totaled
   @return the sum of all elements in the given row
*/
double row_total(int table[][COLUMNS], int row)
{
   int total = 0;
   for (int j = 0; j < COLUMNS; j++)
   {
      total = total + table[row][j];
   }
   return total;
}

int main()
{
   const int COUNTRIES = 8;
   const int MEDALS = 3;

   string countries[] =
      {
         "Canada",
         "Italy",
         "Germany",
         "Japan",
         "Kazakhstan",
         "Russia",
         "South Korea",
         "United States"
      };

```

```
42    int counts[COUNTRIES][MEDALS] =
43    {
44       { 0, 3, 0 },
45       { 0, 0, 1 },
46       { 0, 0, 1 },
47       { 1, 0, 0 },
48       { 0, 0, 1 },
49       { 3, 1, 1 },
50       { 0, 1, 0 },
51       { 1, 0, 1 }
52    };
53
54    cout << "        Country    Gold  Silver  Bronze   Total" << endl;
55
56    // Print countries, counts, and row totals
57    for (int i = 0; i < COUNTRIES; i++)
58    {
59       cout << setw(15) << countries[i];
60       // Process the ith row
61       for (int j = 0; j < MEDALS; j++)
62       {
63          cout << setw(8) << counts[i][j];
64       }
65       int total = row_total(counts, i);
66       cout << setw(8) << total << endl;
67    }
68
69    return 0;
70 }
```

Program Run

```
        Country    Gold  Silver  Bronze   Total
         Canada       0       3       0       3
          Italy       0       0       1       1
        Germany       0       0       1       1
          Japan       1       0       0       1
     Kazakhstan       0       0       1       1
         Russia       3       1       1       5
    South Korea       0       1       0       1
  United States       1       0       1       2
```

Common Error 6.2

Omitting the Column Size of a Two-Dimensional Array Parameter

When passing a one-dimensional array to a function, you specify the size of the array as a separate parameter variable:

```
void print(double values[], int size)
```

This function can print arrays of any size. However, for two-dimensional arrays you cannot simply pass the numbers of rows and columns as parameter variables:

```
void print(double table[][], int rows, int cols) // NO!
```

The function must know *at compile time* how many columns the two-dimensional array has. You must specify the number of columns with the array parameter variable. This number must be a constant:

```
const int COLUMNS = 3;
void print(const double table[][COLUMNS], int rows) // OK
```

This function can print tables with any number of rows, but the column size is fixed.

WORKED EXAMPLE 6.2

A World Population Table

Learn how to print world population data in a table with row and column headers, and with totals for each of the data columns. See your E-Text or visit `wiley.com/go/bclo3`.

6.7 Vectors

A vector stores a sequence of values whose size can change.

When you write a program that collects values from user input, you don't always know how many values you will have. Unfortunately, the size of the array has to be known *when the program is compiled*.

In Section 6.1.3, you saw how you can address this problem with partially filled arrays. The **vector** construct, which we discuss in the following sections, offers a more convenient solution. A vector collects a sequence of values, just like an array does, but its size can change.

© Michael Brake/iStockphoto.

A vector expands to hold as many elements as needed.

Syntax 6.3 Defining a Vector

Element type *Name* *Initial size*

```
vector<double> values(10);
```

If you omit the size and the parentheses, the vector has initial size 0.

```
vector<double> values = { 32, 54, 67.5, 29, 34.5 };
```

If you specify initial values, you don't provide the size.

Use brackets to access an element.

```
values[i] = 0;
```

The index must be ≥ *0 and* < `values.size()`.

6.7.1 Defining Vectors

When you define a vector, you specify the type of the elements in angle brackets, like this:

```
vector<double> values;
```

You can optionally specify the initial size. For example, here is a definition of a vector whose initial size is 10:

```
vector<double> values(10);
```

If you define a vector without an initial size, it has size 0. While there would be no point in defining an array of size zero, it is often useful to have vectors with *initial* size zero, and then grow them as needed.

With C++ 11 and above, you can specify initial values for a vector, using the same notation as for arrays:

```
vector<double> values = { 32, 54, 67.5, 29, 34.5 };
```

In order to use vectors in your program, you need to include the `<vector>` header.

You access the vector elements as `values[i]`, just as you do with arrays.

Use the `size` member function to obtain the current size of a vector.

The `size` member function returns the current size of a vector. In a loop that visits all vector elements, use the `size` member function like this:

```
for (int i = 0; i < values.size(); i++)
{
   cout << values[i] << endl;
}
```

Table 2 Defining Vectors

`vector<int> numbers(10);`	A vector of ten integers.
`vector<string> names(3);`	A vector of three strings.
`vector<double> values;`	A vector of size 0.
`vector<double> values();`	**Error:** Does not define a vector; it declares a function returning a vector.

`vector<int> numbers;`
`for (int i = 1; i <= 10; i++)`
`{`
`  numbers.push_back(i);`
`}`
	A vector of ten integers, filled with 1, 2, 3, ..., 10.

`vector<int> numbers(10);`
`for (int i = 0; i < numbers.size(); i++)`
`{`
`  numbers[i] = i + 1;`
`}`
	Another way of defining a vector of ten integers and filling it with 1, 2, 3, ..., 10.

`vector<int> numbers =` `  { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`	This syntax is supported with C++ 11 and above.

6.7.2 Growing and Shrinking Vectors

Use the push_back member function to add more elements to a vector.
Use pop_back to reduce the size.

If you need additional elements, you use the `push_back` function to add an element to the end of the vector, thereby increasing its size by 1. The `push_back` function is a member function that you must call with the dot notation, like this:

```
values.push_back(37.5);
```

Suppose the vector `values` had size 2 and elements 32 and 54 before the call to `push_back` (see Figure 14). After the call, the vector has size 3, and `values[2]` contains the value 37.5.

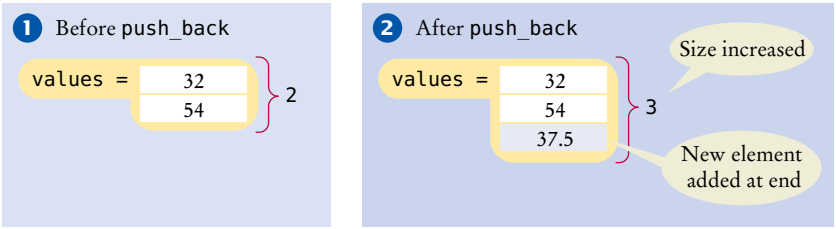


Figure 14 Adding an Element with `push_back`

A common use for the `push_back` member function is to fill a vector with input values.

```
vector<double> values;  // Initially empty

double input;
while (cin >> input)
{
   values.push_back(input);
}
```

Note how this input loop is *much* simpler than the one in Section 6.2.11.

Another member function, `pop_back`, removes the last element of a vector, shrinking its size by one (see Figure 15):

```
values.pop_back();
```

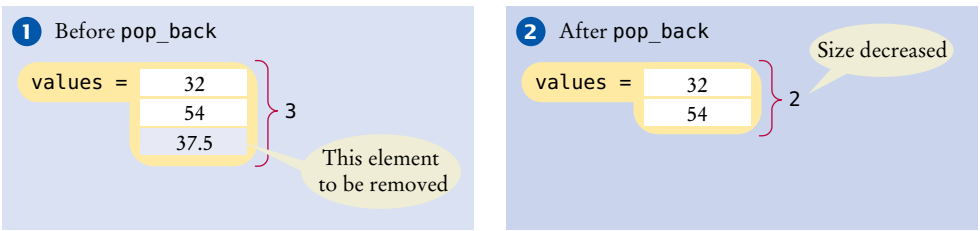


Figure 15 Removing an Element with `pop_back`

6.7.3 Vectors and Functions

Vectors can occur as function arguments and return values.

You can use vectors as function arguments in exactly the same way as any other values. For example, the following function computes the sum of a vector of floating-point numbers:

```
double sum(vector<double> values)
{
   double total = 0;
   for (int i = 0; i < values.size(); i++)
   {
      total = total + values[i];
   }
   return total;
}
```

Use a reference parameter to modify the contents of a vector.

This function visits the vector elements, but it does not modify them. If your function modifies the elements, use a reference parameter. The following function multiplies all values of a vector with a given factor.

```
void multiply(vector<double>& values, double factor) // Note the &
{
   for (int i = 0; i < values.size(); i++)
   {
      values[i] = values[i] * factor;
   }
}
```

It is a good idea to use a constant reference (see Special Topic 5.2) for vector parameters that are not modified, for example:

```
double sum(const vector<double>& values) // const & added for efficiency
```

A function can return a vector.

A function can return a vector. Again, vectors are no different from any other values in this regard. Simply build up the result in the function and return it. In this example, the `squares` function returns a vector of squares from 0^2 up to $(n - 1)^2$:

```
vector<int> squares(int n)
{
   vector<int> result;
   for (int i = 0; i < n; i++)
   {
      result.push_back(i * i);
   }
   return result;
}
```

As you can see, it is easy to use vectors with functions—there are no special rules to keep in mind.

EXAMPLE CODE See sec07_03 of your companion code for a program that demonstrates vectors and functions.

6.7.4 Vector Algorithms

Most of the algorithms in Section 6.2 apply without change to vectors—simply replace *size of* `values` with `values.size()`. In this section, we discuss which of the algorithms are different for vectors.

Copying

As discussed in Section 6.2.2, you need an explicit loop to make a copy of an array. It is much easier to make a copy of a vector. You simply assign it to another vector. Consider this example:

```
vector<int> squares;
for (int i = 0; i < 5; i++) { squares.push_back(i * i); }
vector<int> lucky_numbers; // Initially empty
lucky_numbers = squares; // Now lucky_numbers contains the same elements as squares
```

Finding Matches

Section 6.2.7 shows you how to find the first match, but sometimes you want to have all matches. This is tedious with arrays, but simple using a vector that collects the matches. Here we collect all elements that are greater than 100:

```
vector<double> matches;
for (int i = 0; i < values.size(); i++)
{
   if (values[i] > 100)
   {
      matches.push_back(values[i]);
   }
}
```

Removing an Element

When you remove an element from a vector, you want to adjust the size of the vector by calling the `pop_back` member function. Here is the code for removing an element at `[pos]` when the order doesn't matter.

```
int last_pos = values.size() - 1;
values[pos] = values[last_pos]; // Replace element at pos with last element
values.pop_back(); // Delete last element
```

When removing an element from an ordered vector, first move the elements, then reduce the size:

```
for (int i = pos + 1; i < values.size(); i++)
{
   values[i - 1] = values[i];
}
values.pop_back();
```

Inserting an Element

Inserting an element at the end of a vector requires no special code. Simply use the `push_back` member function.

When you insert an element in the middle, you still want to call `push_back` so that the size of the vector is increased. Use the following code:

```
int last_pos = values.size() - 1;
values.push_back(values[last_pos]);
for (int i = last_pos; i > pos; i--)
{
   values[i] = values[i - 1];
}
values[pos] = new_element;
```

EXAMPLE CODE See sec07_04 of your companion code for a program that uses vectors to solve the problem posed in Section 6.1.

6.7.5 Two-Dimensional Vectors

There are no two-dimensional vectors, but if you want to store rows and columns, you can use a vector of vectors. For example, the medal counts of Section 6.6 can be stored in this variable:

```
vector<vector<int>> counts;
```

Think of `counts` as a vector of rows. Each row is a `vector<int>`.

You need to initialize such a vector of vectors, to make sure there are rows and columns for all the elements.

```
for (int i = 0; i < COUNTRIES; i++)
{
   vector<int> row(MEDALS);
   counts.push_back(row);
}
```

Now you can simply access `counts[i][j]` in the same way as with two-dimensional arrays. The expression `counts[i]` denotes the `i`th row, and `counts[i][j]` is the value in the `j`th column of that row.

Vectors of vectors have an advantage over two-dimensional arrays. The row and column sizes don't have to be fixed at compile time. Suppose you don't know the number of countries and medal types at the outset. You can still construct the table of counts:

```
int countries = . . .;
int medals = . . .;
vector<vector<int>> counts;
for (int i = 0; i < countries; i++)
{
   vector<int> row(medals);
   counts.push_back(row);
}
```

In contrast, you would not be able to declare an array `int[countries][medals]` because the bounds are not constant.

When you process a vector of vectors, you find the number of rows and columns like this:

```
vector<vector<int>> values = . . .;
int rows = values.size();
int columns = values[0].size();
```

Remember that each element of `values` is a row. Therefore, `values.size()` is the number of rows. The size of any one of the rows gives you the number of columns.

It is also possible to declare vectors of vectors in which the row size varies. For example, to achieve a triangular shape, such as

```
t[0][0]
t[1][0] t[1][1]
t[2][0] t[2][1] t[2][2]
t[3][0] t[3][1] t[3][2] t[3][3]
```

you add rows of the appropriate sizes, like this:

```
vector<vector<int>> t;
for (int i = 0; i < 3; i++)
{
   vector<int> row(i + 1);
   t.push_back(row);
```

```
}
```

You still access each array element as `t[i][j]`. The expression `t[i]` selects the `i`th row, and the `[j]` operator selects the `j`th element in that row.

Programming Tip 6.2

Prefer Vectors over Arrays

For most programming tasks, vectors are easier to use than arrays. Vectors can grow and shrink. Even if a vector always stays the same size, it is convenient that a vector remembers its size. For a beginner, the sole advantage of an array is the initialization syntax. Advanced programmers sometimes prefer arrays because they are a bit more efficient. Moreover, you need to know how to use arrays if you work with older programs.

Special Topic 6.5

The Range-Based `for` Loop

C++ 11 introduces a convenient syntax for visiting all elements in a "range" or sequence of elements. This loop displays all elements in a vector:

```
vector<int> values = {1, 4, 9, 16, 25, 36};
for (int v : values)
{
   cout << v << " ";
}
```

In each iteration of the loop, `v` is set to an element of the vector. Note that you do not use an index variable. The value of `v` is the element, not the index of the element.

If you want to modify elements, declare the loop variable as a reference:

```
for (int& v : values)
{
   v++;
}
```

This loop increments all elements of the vector.

You can use the reserved word `auto`, which was introduced in Special Topic 2.3, for the type of the element variable:

```
for (auto v : values) { cout << v << " "; }
```

The range-based `for` loop also works for arrays:

```
int primes[] = { 2, 3, 5, 7, 11, 13 };
for (int p : primes)
{
   cout << p << " ";
}
```

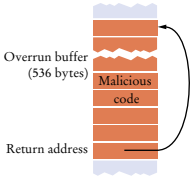
The range-based `for` loop is a convenient shortcut for visiting or updating all elements of a vector or an array. This book doesn't use it because one can achieve the same result by looping over index values. But if you like the more concise form, and use C++ 11 or later, you should certainly consider using it.

EXAMPLE CODE See special_topic_5 of your companion code for a program that demonstrates the range-based `for` loop.

CHAPTER SUMMARY

Use arrays for collecting values.

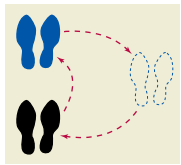
- Use an array to collect a sequence of values of the same type.
- Individual elements in an array *values* are accessed by an integer index `i`, using the notation *values*`[i]`.



- An array element can be used like any variable.
- An array index must be at least zero and less than the size of the array.
- A bounds error, which occurs if you supply an invalid array index, can data or cause your program to terminate.
- With a partially filled array, keep a companion variable for the current size.

Be able to use common array algorithms.

- To copy an array, use a loop to copy its elements to a new array.
- When separating elements, don't place a separator before the first element.
- A linear search inspects elements in sequence until a match is found.
- Before inserting an element, move elements to the end of the array *starting with the last one*.
- Use a temporary variable when swapping two elements.



Implement functions that process arrays.

- When passing an array to a function, also pass the size of the array.
- Array parameters are always reference parameters.
- A function's return type cannot be an array.
- When a function modifies the size of an array, it needs to tell its caller.
- A function that adds elements to an array needs to know its capacity.

Be able to combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

Discover algorithms by manipulating physical objects.

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

Use two-dimensional arrays for data that is arranged in rows and columns.

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two index values, *array*`[i][j]`.
- A two-dimensional array parameter must have a fixed number of columns.

Use vectors for managing collections whose size can change.

- A vector stores a sequence of values whose size can change.
- Use the `size` member function to obtain the current size of a vector.
- Use the `push_back` member function to add more elements to a vector. Use `pop_back` to reduce the size.
- Vectors can occur as function arguments and return values.
- Use a reference parameter to modify the contents of a vector.
- A function can return a vector.

REVIEW EXERCISES

■■ **R6.1** Carry out the following tasks with an array:

a. Allocate an array `a` of ten integers.

b. Put the number 17 as the initial element of the array.

c. Put the number 29 as the last element of the array.

d. Fill the remaining elements with –1.

e. Add 1 to each element of the array.

f. Print all elements of the array, one per line.

g. Print all elements of the array in a single line, separated by commas.

■■ **R6.2** Write code that fills an array `double values[10]` with each set of values below.

a. 1 2 3 4 5 6 7 8 9 10

b. 0 2 4 6 8 10 12 14 16 18 20

c. 1 4 9 16 25 36 49 64 81 100

d. 0 0 0 0 0 0 0 0 0 0

e. 1 4 9 16 9 7 4 9 11

f. 0 1 0 1 0 1 0 1 0 1

g. 0 1 2 3 4 0 1 2 3 4

■■ **R6.3** Consider the following array:

```
int a[] = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What is the value of `total` after the following loops complete?

a.
```
int total = 0;
for (int i = 0; i < 10; i++) { total = total + a[i]; }
```

b.
```
int total = 0;
for (int i = 0; i < 10; i = i + 2) { total = total + a[i]; }
```

c.
```
int total = 0;
for (int i = 1; i < 10; i = i + 2) { total = total + a[i]; }
```

d.
```
int total = 0;
for (int i = 2; i <= 10; i++) { total = total + a[i]; }
```

e.
```
int total = 0;
for (int i = 1; i < 10; i = 2 * i) { total = total + a[i]; }
```

f.
```
int total = 0;
for (int i = 9; i >= 0; i--) { total = total + a[i]; }
```

g.
```
int total = 0;
for (int i = 9; i >= 0; i = i - 2) { total = total + a[i]; }
```

h.
```
int total = 0;
for (int i = 0; i < 10; i++) { total = a[i] - total; }
```

■■ **R6.4** Consider the following array:

```
int a[] = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What are the contents of the array `a` after the following loops complete?

a.
```
for (int i = 1; i < 10; i++) { a[i] = a[i - 1]; }
```

b.
```
for (int i = 9; i > 0; i--) { a[i] = a[i - 1]; }
```

c.
```
for (int i = 0; i < 9; i++) { a[i] = a[i + 1]; }
```

d. `for (int i = 8; i >= 0; i--) { a[i] = a[i + 1]; }`

e. `for (int i = 1; i < 10; i++) { a[i] = a[i] + a[i - 1]; }`

f. `for (int i = 1; i < 10; i = i + 2) { a[i] = 0; }`

g. `for (int i = 0; i < 5; i++) { a[i + 5] = a[i]; }`

h. `for (int i = 1; i < 5; i++) { a[i] = a[9 - i]; }`

■■■ **R6.5** Write a loop that fills an array `int values[10]` with ten random numbers between 1 and 100. Write code for two nested loops that fill `values` with ten *different* random numbers between 1 and 100.

■■ **R6.6** Write C++ code for a loop that simultaneously computes both the maximum and minimum of an array.

■ **R6.7** What is wrong with the following loop?

```
int values[10];
for (int i = 1; i <= 10; i++)
{
   values[i] = i * i;
}
```

Explain two ways of fixing the error.

■ **R6.8** What is an index of an array? What are the legal index values? What is a bounds error?

■ **R6.9** Write a program that contains a bounds error. Run the program. What happens on your computer?

■ **R6.10** Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.

■ **R6.11** Trace the flow of the element separator loop in Section 6.2.5 with the given example. Show two columns, one with the value of `i` and one with the output.

■ **R6.12** Trace the flow of the finding matches loop in Section 6.7.4, where `values` contains the elements 110 90 100 120 80. Show two columns, for `i` and `matches`.

■ **R6.13** Trace the flow of the linear search loop in Section 6.2.7, where `values` contains the elements 80 90 100 120 110. Show two columns, for `pos` and `found`. Repeat the trace when `values` contains 80 90 100 70.

■ **R6.14** Trace both mechanisms for removing an element described in Section 6.2.8. Use an array `values` with elements 110 90 100 120 80, and remove the element at index 2.

■ **R6.15** Consider the following loop for collecting all elements that match a condition; in this case, that the element is larger than 100.

```
vector<double> matches;
for (int i = 0; i < values.size(); i++)
{
   if (values[i] > 100)
   {
      matches.add(values[i]);
   }
}
```

Trace the flow of the loop, where `values` contains the elements 110 90 100 120 80. Show two columns, for `element` and `matches`.

■■ **R6.16** For the operations on partially filled arrays below, provide the header of a function.

a. Sort the elements in decreasing order.

b. Print all elements, separated by a given string.

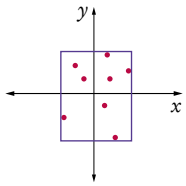
c. Count how many elements are less than a given value.

d. Remove all elements that are less than a given value.

e. Place all elements that are less than a given value in another array.

Do not implement the functions.

■■ **R6.17** You are given two arrays denoting x- and y-coordinates of a set of points in the plane. For plotting the point set, we need to know the x- and y-coordinates of the smallest rectangle containing the points.



How can you obtain these values from the fundamental algorithms in Section 6.2?

■ **R6.18** Solve the problem described in Section 6.4 by sorting the array first. How do you need to modify the algorithm for computing the total?

■■ **R6.19** Solve the task described in Section 6.5 using an algorithm that removes and inserts elements instead of switching them. Write the pseudocode for the algorithm, assuming that functions for removal and insertion exist. Act out the algorithm with a sequence of coins and explain why it is less efficient than the swapping algorithm developed in Section 6.5.

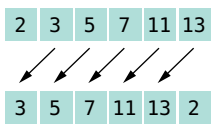
■■ **R6.20** Develop an algorithm for finding the most frequently occurring value in an array of numbers. Use a sequence of coins. Place paper clips below each coin that count how many other coins of the same value are in the sequence. Give the pseudocode for an algorithm that yields the correct answer, and describe how using the coins and paper clips helped you find the algorithm.

■■ **R6.21** Write C++ statements for performing the following tasks with an array declared as

```
int values[ROWS][COLUMNS];
```

- Fill all entries with `0`.
- Fill elements alternately with `0`s and `1`s in a checkerboard pattern.
- Fill only the elements in the top and bottom rows with zeroes.
- Compute the sum of all elements.
- Print the array in tabular form.

■■ **R6.22** Give pseudocode for a function that rotates the elements of an array by one position, moving the initial element to the end of the array, like this:



■■ **R6.23** Give pseudocode for a function that removes all negative values from a partially filled array, preserving the order of the remaining elements.

■■ **R6.24** Suppose `values` is a *sorted* partially filled array of integers. Give pseudocode that describes how a new value can be inserted in its proper position so that the resulting array stays sorted.

■■■ **R6.25** A *run* is a sequence of adjacent repeated values. Give pseudocode for computing the length of the longest run in an array. For example, the longest run in the array with following elements has length 4

```
1 2 5 5 3 1 2 4 3 2 2 2 2 3 6 5 5 6 3 1
```

■■ **R6.26** Write pseudocode for an algorithm that fills the first and last column as well as the first and last row of a two-dimensional array of integers with –1.

■■ **R6.27** True or false?

a. All elements of an array are of the same type.

b. Arrays cannot contain strings as elements.

c. Two-dimensional arrays always have the same number of rows and columns.

d. Elements of different columns in a two-dimensional array can have different types.

e. A function cannot return a two-dimensional array.

f. All array parameters are reference parameters.

g. A function cannot change the dimensions of a two-dimensional array that is passed as a parameter.

■■ **R6.28** How do you perform the following tasks with vectors in C++?

a. Test that two vectors contain the same elements in the same order.

b. Copy one vector to another.

c. Fill a vector with zeroes, overwriting all elements in it.

d. Remove all elements from a vector.

■ **R6.29** True or false?

a. All elements of a vector are of the same type.

b. A vector index can be negative.

c. Vectors cannot contain strings as elements.

d. Vectors cannot use strings as index values.

e. All vector parameters are reference parameters.

f. A function cannot return a vector.

g. A function cannot change the length of a vector that is a reference parameter.

PRACTICE EXERCISES

■■ **E6.1** Write a program that initializes an array with ten random integers and then prints four lines of output, containing

- Every element at an even index.
- Every even element.

- All elements in reverse order.
- Only the first and last element.

■■ **E6.2** Write array functions that carry out the following tasks for an array of integers. For each function, provide a test program.

a. Swap the first and last element in an array.

b. Shift all elements by one to the right and move the last element into the first position. For example, 1 4 9 16 25 would be transformed into 25 1 4 9 16.

c. Replace all even elements with `0`.

d. Replace each element except the first and last by the larger of its two neighbors.

e. Remove the middle element if the array length is odd, or the middle two elements if the length is even.

f. Move all even elements to the front, otherwise preserving the order of the elements.

g. Return true if all elements of the array are identical.

h. Return the second-largest element in the array.

i. Return true if the array is currently sorted in increasing order.

j. Return true if the array contains two adjacent duplicate values.

k. Return true if the array contains duplicate values (which need not be adjacent).

■ **E6.3** Modify the `largest.cpp` program to mark both the smallest and the largest element.

■ **E6.4** Write a function `void remove_min` that removes the minimum value from a partially filled array without calling other functions.

■■ **E6.5** Write a function that computes the *alternating sum* of all elements in an array. For example, if `alternating_sum` is called with an array containing

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

then it computes

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

■ **E6.6** Write a function that implements the algorithm developed in Section 6.5.

■ **E6.7** Write a function `reverse` that reverses the sequence of elements in an array. For example, if `reverse` is called with an array containing

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

then the array is changed to

$$11 \quad 9 \quad 4 \quad 7 \quad 9 \quad 16 \quad 9 \quad 4 \quad 1$$

■■ **E6.8** Write a function

```
bool equals(int a[], int a_size, int b[], int b_size)
```

that checks whether two arrays have the same elements in the same order.

■■ **E6.9** Write a function

```
bool same_set(int a[], int a_size, int b[], int b_size)
```

that checks whether two vectors have the same elements in some order, ignoring duplicates. For example, the two arrays

1 4 9 16 9 7 4 9 11

and

11 11 7 9 16 4 1

would be considered identical. You will probably need one or more helper functions.

■■■ **E6.10** Write a function

```
bool same_elements(int a[], int b[], int size)
```

that checks whether two arrays have the same elements in some order, with the same multiplicities. For example,

1 4 9 16 9 7 4 9 11

and

11 1 4 9 16 9 7 4 9

would be considered identical, but

1 4 9 16 9 7 4 9 11

and

11 11 7 9 16 4 1 4 9

would not. You will probably need one or more helper functions.

■■ **E6.11** Write a function that removes duplicates from an array. For example, if `remove_duplicates` is called with an array containing

1 4 9 16 9 7 4 9 11

then the array is changed to

1 4 9 16 7 11

Your function should have a reference parameter for the array size that is updated when removing the duplicates.

■■ **E6.12** Write a program that generates a sequence of 20 random values between 0 and 99, prints the sequence, sorts it, and prints the sorted sequence. Use the `sort` function from the standard C++ library.

■■ **E6.13** Write a function that computes the average of the neighbors of a two-dimensional array element in the eight directions shown in Figure 12.

```
double neighbor_average(int values[][], int row, int column)
```

However, if the element is located at the boundary of the array, only include the neighbors that are in the array. For example, if `row` and `column` are both 0, there are only three neighbors.

■■ **E6.14** Write a function

```
void bar_chart(double values[], int size)
```

that displays a bar chart of the values in `values`, using asterisks, like this:

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

You may assume that all values in `values` are positive. First figure out the maximum value in `values`. That value's bar should be drawn with 40 asterisks. Shorter bars should use proportionally fewer asterisks.

■■■ **E6.15** Repeat Exercise E6.14, but make the bars vertical, with the tallest bar twenty asterisks high.

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

■■■ **E6.16** Improve the `bar_chart` function of Exercise E6.14 to work correctly when `values` contains negative values.

■■■ **E6.17** Improve the `bar_chart` function of Exercise E6.14 by adding an array of captions for each bar. The output should look like this:

```
      Egypt **********************
     France ****************************************
      Japan ***************************
    Uruguay ************************
Switzerland **************
```

■ **E6.18** Write a function

```
vector<int> append(vector<int> a, vector<int> b)
```

that appends one vector after another. For example, if `a` is

1 4 9 16

and `b` is

9 7 4 9 11

then `append` returns the vector

1 4 9 16 9 7 4 9 11

■■ **E6.19** Write a function

```
vector<int> merge(vector<int> a, vector<int> b)
```

that merges two vectors, alternating elements from both vectors. If one vector is shorter than the other, then alternate as long as you can and then append the remaining elements from the longer vector.

For example, if `a` is

1 4 9 16

and `b` is

9 7 4 9 11

then `merge` returns the vector

1 9 4 7 9 4 16 9 11

■■ **E6.20** Write a function

```
vector<int> merge_sorted(vector<int> a, vector<int> b)
```

that merges two *sorted* vectors, producing a new sorted vector. Keep an index into each vector, indicating how much of it has been processed already. Each time, append the smallest unprocessed element from either vector, then advance the index. For example, if `a` is

1 4 9 16

and `b` is

4 7 9 9 11

then `merge_sorted` returns the vector

1 4 4 7 9 9 9 11 16

■ **E6.21** Write a function that modifies a `vector<string>`, moving all strings starting with an uppercase letter to the front, without otherwise changing the order of the elements.

■■ **E6.22** Write a function that counts the number of distinct elements in a `vector<string>`. Do not modify the vector.

■ **E6.23** Write a function that finds the first occurrence of a value in a two-dimensional array. Return an `int` array of length 2 with the row and column, or `null` if the value was not found.

■ **E6.24** Write a function that checks whether all elements in a two-dimensional array are identical.

■■ **E6.25** Write a function that checks whether all elements in a two-dimensional array are distinct.

■■ **E6.26** Write a function that checks whether two two-dimensional arrays are equal; that is, whether they have the same number of rows and columns, and corresponding elements are equal.

■ **E6.27** Write a function that swaps two rows of a two-dimensional array.

■ **E6.28** Write a function that swaps two columns of a two-dimensional array.

PROGRAMMING PROJECTS

■■ **P6.1** A *run* is a sequence of adjacent repeated values. Write a program that generates a sequence of 20 random die tosses and prints the die values, marking the runs by including them in parentheses, like this:

```
1 2 (5 5) 3 1 2 4 3 (2 2 2 2) 3 6 (5 5) 6 3 1
```

Use the following pseudocode:

```
in_run = false
For each valid index i in the array
   If in_run
      If values[i] is different from the preceding value
         Print ).
         in_run = false
```

```
If not in_run
   If values[i] is the same as the following value
      Print (.
      in_run = true
   Print values[i].
If in_run, print ).
```

■■ **P6.2** Write a program that generates a sequence of 20 random die tosses and that prints the die values, marking only the longest run, like this:

```
1 2 5 5 3 1 2 4 3 (2 2 2 2) 3 6 5 5 6 3 1
```

If there is more than one run of maximum length, mark the first one.

■■■ **P6.3** Write a program that produces ten random permutations of the numbers 1 to 10. To generate a random permutation, you need to fill an array with the numbers 1 to 10 so that no two elements have the same contents. You could do it by brute force, by generating random values until you have a value that is not yet in the array. But that is inefficient. Instead, follow this algorithm:

```
Make a second array and fill it with the numbers 1 to 10.
Repeat 10 times
   Pick a random position in the second array.
   Remove the element at the position from the second array.
   Append the removed element to the permutation array.
```

■■ **P6.4** It is a well-researched fact that men in a restroom generally prefer to maximize their distance from already occupied stalls, by occupying the middle of the longest sequence of unoccupied places.

For example, consider the situation where all ten stalls are empty.

```
_ _ _ _ _ _ _ _ _ _
```

The first visitor will occupy a middle position:

```
_ _ _ _ X _ _ _ _ _
```

The next visitor will be in the middle of the empty area at the right.

```
_ _ _ _ X _ _ X _ _
```

Given an array of `bool` values, where true indicates an occupied stall, find the position for the next visitor. Your computation should be placed in a function

```
next_visitor(bool occupied[], int stalls)
```

■■■ **P6.5** In this assignment, you will model the game of *Bulgarian Solitaire*. The game starts with 45 cards. (They need not be playing cards. Unmarked index cards work just as well.) Randomly divide them into some number of piles of random size. For example, you might start with piles of size 20, 5, 1, 9, and 10. In each round, you take one card from each pile, forming a new pile with these cards. For example, the sample starting configuration would be transformed into piles of size 19, 4, 8, 9, and 5. The solitaire is over when the piles have size 1, 2, 3, 4, 5, 6, 7, 8, and 9, in some order. (It can be shown that you always end up with such a configuration.)

In your program, produce a random starting configuration and print it. Then keep applying the solitaire step and print the result. Stop when the solitaire final configuration is reached.

■■■ **P6.6** *Magic squares.* An $n \times n$ matrix that is filled with the numbers 1, 2, 3, . . ., n^2 is a magic square if the sum of the elements in each row, in each column, and in the two diagonals is the same value.

Write a program that reads in 16 values from the keyboard and tests whether they form a magic square when put into a 4×4 array. You need to test two features:

1. Does each of the numbers 1, 2, ..., 16 occur in the user input?
2. When the numbers are put into a square, are the sums of the rows, columns, and diagonals equal to each other?

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

■■■ **P6.7** Implement the following algorithm to construct magic $n \times n$ squares; it works only if n is odd.

```
Set row = n - 1, column = n / 2.
For k = 1 . . . n * n
   Place k at [row][column].
   Increment row and column.
   If the row or column is n, replace it with 0.
   If the element at [row][column] has already been filled
      Set row and column to their previous value.
      Decrement row.
```

Here is the 5×5 square that you get if you follow this method:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Write a program whose input is the number n and whose output is the magic square of order n if n is odd.

■■ **P6.8** A theater seating chart is implemented as a two-dimensional array of ticket prices, like this:

```
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
20 20 30 30 40 40 30 30 20 20
20 30 30 40 50 50 40 30 30 20
30 40 50 50 50 50 50 50 40 30
```

Write a program that prompts users to pick either a seat or a price. Mark sold seats by changing the price to 0. When a user specifies a seat, make sure it is available. When a user specifies a price, find any seat with that price.

© lepas2004/iStockphoto.

■■■ **P6.9** Write a program that plays tic-tac-toe. The tic-tac-toe game is played on a 3×3 grid as in the photo at right.

The game is played by two players, who take turns. The first player marks moves with a circle, the second with a cross. The player who has formed a horizontal, vertical, or diagonal sequence of three marks wins. Your program should print the game board, ask the user for the coordinates of the next mark, change the players after every successful move, and pronounce the winner.

© KathyMuller/iStockphoto.

■■■ **P6.10** *The Game of Life* is a well-known mathematical game that gives rise to amazingly complex behavior, although it can be specified by a few simple rules. (It is not actually a game in the traditional sense, with players competing for a win.) Here are the rules. The game is played on a rectangular board. Each square can be either empty or occupied. At the beginning, you can specify empty and occupied cells in some way; then the game runs automatically. In each *generation*, the next generation is computed. A new cell is born on an empty square if it is surrounded by exactly three occupied neighbor cells. A cell dies of overcrowding if it is surrounded by four or more neighbors, and it dies of loneliness if it is surrounded by zero or one neighbor. A neighbor is an occupant of an adjacent square to the left, right, top, or bottom or in a diagonal direction. Figure 16 shows a cell and its neighbor cells.

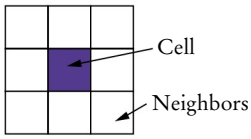


Figure 16
Neighborhood of a Cell

Many configurations show interesting behavior when subjected to these rules. Figure 17 shows a *glider*, observed over five generations. After four generations, it is transformed into the identical shape, but located one square to the right and below.

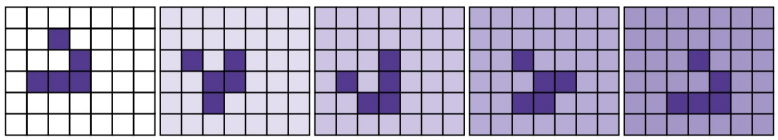


Figure 17 Glider

One of the more amazing configurations is the glider gun: a complex collection of cells that, after 30 moves, turns back into itself and a glider (see Figure 18).

Program the game to eliminate the drudgery of computing successive generations by hand. Use a two-dimensional array to store the rectangular configuration. Write a program that shows successive generations of the game. Ask the user to specify the original configuration, by typing in a configuration of spaces and `o` characters.

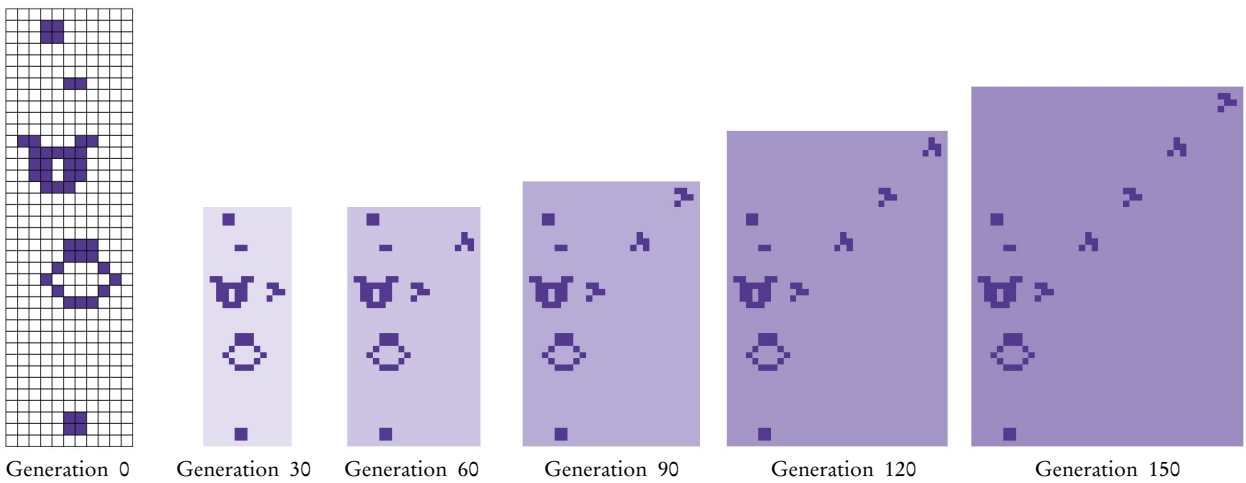


Figure 18 Glider Gun

■■ Business P6.11 A pet shop wants to give a discount to its clients if they buy one or more pets and at least five other items. The discount is equal to 20 percent of the cost of the other items, but not the pets. Implement a function

```
void discount(double prices[], bool is_pet[], int n_items)
```

The function receives information about a particular sale. For the ith item, `prices[i]` is the price before any discount, and `is_pet[i]` is true if the item is a pet.

Write a program that prompts a cashier to enter each price and then a `Y` for a pet or `N` for another item. Use a price of –1 as a sentinel. Save the inputs in an array. Call the function that you implemented, and display the discount.

© joshblake/iStockphoto.

■■ Business P6.12 A supermarket wants to reward its best customer of each day, showing the customer's name on a screen in the supermarket. For that purpose, the customer's purchase amount is stored in a `vector<double>` and the customer's name is stored in a corresponding `vector<string>`.

Implement a function

```
string name_of_best_customer(vector<double> sales,
      vector<string> customers)
```

that returns the name of the customer with the largest sale.

Write a program that prompts the cashier to enter all prices and names, adds them to two vectors, calls the function that you implemented, and displays the result. Use a price of `0` as a sentinel.

■■■ Business P6.13 Improve the program of Exercise P6.12 so that it displays the top customers, that is, the `top_n` customers with the largest sales, where `top_n` is a value that the user of the program supplies.

Implement a function

```
vector<string> name_of_best_customers(vector<double> sales,
      vector<string> customers, int top_n)
```

If there were fewer than `top_n` customers, include all of them.

■■ Engineering P6.14 Sample values from an experiment often need to be smoothed out. One simple approach is to replace each value in an array with the average of the value and its two neighboring values (or one neighboring value if it is at either end of the array). Implement a function

```
void smooth(double values[], int size)
```

that carries out this operation. You should not create another array in your solution.

■■■ Engineering P6.15 You are given a two-dimensional array of values that give the height of a terrain at different points in a square. Write a function

```
void flood_map(double heights[10][10], double water_level)
```

that prints out a flood map, showing which of the points in the terrain would be flooded if the water level was the given value. In the flood map, print a `*` for each flooded point and a space for each point that is not flooded.

Here is a sample map:

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

© nicolamargaret/iStockphoto.

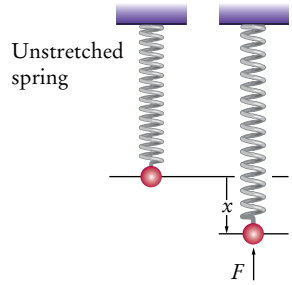
Then write a program that reads one hundred terrain height values and shows how the terrain gets flooded when the water level increases in ten steps from the lowest point in the terrain to the highest.

■■■ Engineering P6.16 Write a program that models the movement of an object with mass m that is attached to an oscillating spring. When a spring is displaced from its equilibrium position by an amount x, Hooke's law states that the restoring force is

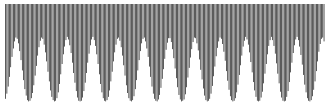
$$F = -kx$$

where k is a constant that depends on the spring. (Use 10 N/m for this simulation.)

Start with a given displacement x (say, 0.5 meter). Set the initial velocity v to 0. Compute the acceleration a from Newton's law ($F = ma$) and Hooke's law, using a mass of 1 kg. Use a small time interval $\Delta t = 0.01$ second. Update the velocity—it changes by $a\Delta t$. Update the displacement—it changes by $v\Delta t$.



Every ten iterations, plot the spring displacement as a bar, where 1 pixel represents 1 cm. Use the `Picture` class from the `media/picture` directory of the companion code for creating an image.



■■ Media P6.17 Sounds can be represented by an array of "sample values" that describe the intensity of the sound at a point in time. The code in the `media/sound` directory of the companion code allows you to modify a sound file. Your task is to introduce an echo. For each sound value, add the value from 0.2 seconds ago. Scale the result so that no value is larger than 32767.

© GordonHeeley/iStockphoto.

■■ Media P6.18 Using the sound processing code from the `media/sound` directory of the companion code, write a program that reverses the sample values in a sound file.

■■ Media P6.19 Using the sound processing code from the `media/sound` directory of the companion code, write a program that computes the average and maximum sample value of a

sound file, then replaces all values that are less than the average with zero, and all values that are greater than the average with the maximum. Try your program with a sound file containing human speech.

■■ **Media P6.20** Using the sound processing code from the `media/sound` directory of the companion code, write a program that increases the volume in the first half, gradually going from the original volume to twice the original, and then gradually decreases the volume in the second half to go back to the original.

■■ **Media P6.21** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, flips it vertically, and writes the result.


© Deejpilot/E+/Getty Images.

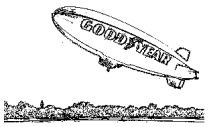
■■ **Media P6.22** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, flips it horizontally, and writes the result.


© tomprout/iStock/Getty Images.

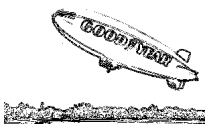
■■ **Media P6.23** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, rotates it clockwise by 90 degrees, and writes the result.


■■ **Media P6.24** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, changes each pixel to a 2×2 array of the same color, and writes the result.


■■ **Media P6.25** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, detects edges, and writes the result. Use the following algorithm. For each pixel, calculate the average gray value of the neighbors to the east, south, and southeast. If the pixel differs by more than 10 from that average, color it black. Otherwise, color it white. Color the last row and column white.

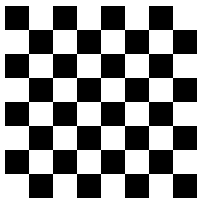
■■ **Media P6.26** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vector of grayscale values, detects edges, and writes the result. Use the following algorithm. For each pixel, calculate the average gray value of the neighbors in all eight compass directions. If the pixel differs by more than 10 from that average, color it black. Otherwise, color it white. You will need a separate two-dimensional vector for the result.

■■ **Media P6.27** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads two image files, obtains the two-dimensional vectors of grayscale values, and superimposes them. Average the values that are present in both images, and use gray (127) where neither image contributes pixels.

■■ **Media P6.28** Using the `Picture` class from the `media/picture` directory of the companion code, write a program that reads an image file, obtains the two-dimensional vectors of grayscale values, and colors every seventh pixel black, starting at the top left corner.

■■ **Media P6.29** Use the `Picture` class from the `media/picture` directory of the companion code to generate the image of a checkerboard.

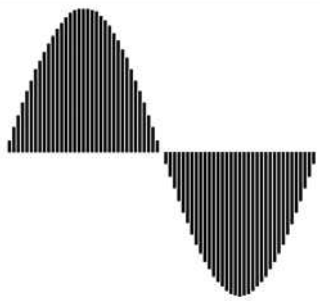


■■ **Media P6.30** Use the code in the `media/animation` directory of the companion code, show a rectangle that travels from the left of the image to the right and then back to the left.

■■■ **Media P6.31** Using the code in the `media/animation` directory of the companion code, show an animation of the oscillating spring described in Exercise P6.16. Just draw a straight line and a square weight, but accurately update the displacement.

■■■ **Media P6.32** Using the code in the `media/animation` directory of the companion code, show an animation of a bouncing ball. The ball is dropped from an initial height, and its velocity is accelerated by the gravitational force, as described in Exercise P4.23. When the ball hits the ground, the velocity is reversed but dampened by a small percentage.

■ **Media P6.33** Use the `Picture` class from the `media/picture` directory of the companion code to generate the image of a sine wave. Draw a line of pixels for every five degrees.



■■ **Media P6.34** Using the code in the `media/animation` directory of the companion code, show an animated sine wave. In the ith frame, shift the sine wave by $5 \times i$ degrees.

WORKED EXAMPLE 6.1

Rolling the Dice

© ktsimage/iStockphoto.

Problem Statement Your task is to analyze whether a die is fair by counting how often the values 1, 2, ..., 6 appear. Your input is a sequence of die toss values, and you should print a table with the frequencies of each die value.

Step 1 Decompose your task into steps.

Our first try at decomposition simply echoes the problem statement:

Read the die values into an array.
Count how often the values 1, 2, ..., 6 appear in the array.
Print the counts.

But let's think about the task a little more. This decomposition suggests that we first read and store all die values. Do we really need to store them? After all, we only want to know how often each face value appears. If we keep an array of counters, we can discard each input after incrementing the counter.

This refinement yields the following outline:

For each input value i
  Increment the counter corresponding to i.
Print the counters.

Step 2 Determine which algorithm(s) you need.

We don't have a ready-made algorithm for reading inputs and incrementing a counter, but it is straightforward to develop one. Suppose we read an input into `value`. This is an integer between 1 and 6. If we have an array `counters` of length 6, then we simply call

```
counters[value - 1]++;
```

Alternatively, we can use an array of seven integers, "wasting" the element `counters[0]`. That trick makes it easier to update the counters. When reading an input value, we simply execute

```
counters[value]++; // value is between 1 and 6
```

That is, we declare the array as

```
const int FACES = 6;
int counters[FACES + 1];
```

Why introduce a `FACES` variable? Suppose you later changed your mind and wanted to investigate 12-sided dice:

© Ryan Ruffatti/iStockphoto.

Then the program can simply be changed by setting `FACES` to 12.

The only remaining task is to print the counts. A typical output might look like this:

```
1: 3
2: 3
3: 2
4: 2
5: 2
6: 0
```

We haven't seen an algorithm for this exact output format. It is similar to the basic loop for printing all elements:

```
for (int i = 0; i <= FACES; i++)
{
   cout << counters[j] << endl;
}
```

However, that loop displays the unused `0` entry, and it does not show the corresponding face values. We adapt the algorithm as follows:

```
for (int i = 1; i <= FACES; i++)
{
   cout << j << ": " << counters[j] << endl;
}
```

Step 3 Use functions to structure your program.

We will provide a function for each step:

- `generate_test_values`
- `count_values`
- `print_counters`

The `generate_test_values` function fills an array of test values.

```
/**
   Generates a sequence of die toss values for testing.
   @param values the array to be filled with die toss values
   @param size the size of the values array
*/
void generate_test_values(int values[], int size)
```

The `count_values` function receives the die toss values, and it must also receive the array of counters so that it can update them:

```
/**
   Counts the number of times each value occurs in a sequence of die tosses.
   @param values an array of die toss values.
   @param size the size of the values array
   @param faces the number of faces on the die
   @param counters an array of counters of length faces + 1. counters[j]
   is filled with the count of elements of values that equal j. counters[0] is not used.
*/
void count_values(int values[], int size, int faces, int counters[])
```

The function modifies the `counters` array. This is not a problem because array parameters are always reference parameters.

Finally, the `print_counters` function prints the value of the counters. Again, we will want to support an arbitrary number of die faces, so we will supply that number as an argument.

```
/**
   Prints a table of die value counters.
   @param faces the number of faces on the die
   @param counters an array of counters of length faces + 1
*/
void print_counters(int faces, int counters[])
```

Step 4 Assemble and test the program.

The listing at the end of this section shows the complete program.

The following table shows test cases and their expected output. To save space, we only show the counters in the output.

Test Case	Expected Output	Comment
1 2 3 4 5 6	1 1 1 1 1 1	Each number occurs once.
1 2 3	1 1 1 0 0 0	Numbers that don't appear should have counts of zero.
1 2 3 1 2 3 4	2 2 2 1 0 0	The counters should reflect how often each input occurs.
(No input)	0 0 0 0 0 0	This is a legal input; all counters are zero.
0 1 2 3 4 5 6 7	**Error**	Each input should be between 1 and 6.

Here's the complete program:

worked_example_1/dice.cpp

```
#include <iostream>

using namespace std;

/**
   Generates a sequence of die toss values for testing.
   @param values the array to be filled with die toss values
   @param size the size of the values array
*/
void generate_test_values(int values[], int size)
{
   int next = 1;
   for (int i = 0; i < size; i++)
   {
      values[i] = next;
      next++;
      if (next == 6) { next = 1; }
   }
}

/**
   Counts the number of times each value occurs in a sequence of die tosses.
   @param values an array of die toss values. Each element is >= 1 and <= faces.
   @param size the size of the values array
   @param faces the number of faces of the die
   @param counters an array of counters of length faces + 1. counters[j]
   is filled with the count of elements of values that equal j. counters[0] is
   not used.
*/
void count_values(int values[], int size, int faces, int counters[])
{
   for (int i = 1; i <= faces; i++) { counters[i] = 0; }
```

```
33    for (int j = 0; j < size; j++)
34    {
35       int value = values[j];
36       counters[value]++;
37    }
38 }
39
40 /**
41    Prints a table of die value counters.
42    @param faces the number of faces of the die
43    @param counters an array of counters of length faces + 1.
44    counters[0] is not printed.
45 */
46 void print_counters(int faces, int counters[])
47 {
48    for (int i = 1; i <= faces; i++)
49    {
50       cout << j << ": " << counters[j] << endl;
51    }
52 }
53
54 int main()
55 {
56    const int FACES = 6;
57    int counters[FACES + 1];
58    const int NUMBER_OF_TOSSES = 12;
59    int tosses[NUMBER_OF_TOSSES];
60
61    generate_test_values(tosses, NUMBER_OF_TOSSES);
62    count_values(tosses, NUMBER_OF_TOSSES, FACES, counters);
63    print_counters(FACES, counters);
64    return 0;
65 }
```

WORKED EXAMPLE 6.2

A World Population Table

Problem Statement You are to print the following population data in tabular format and add column totals that show the total world populations in the given years.

Population Per Continent (in millions)							
Year	1750	1800	1850	1900	1950	2000	2050
Africa	106	107	111	133	221	767	1766
Asia	502	635	809	947	1402	3634	5268
Australia	2	2	2	6	13	30	46
Europe	163	203	276	408	547	729	628
North America	2	7	26	82	172	307	392
South America	16	24	38	74	167	511	809

Step 1 First, we break down the task into steps:

Initialize the table data.
Print the table.
Compute and print the column totals.

Step 2 Initialize the table as a sequence of rows:

```
int data[ROWS][COLUMNS] =
   {
      { 106, 107, 111, 133, 221, 767, 1766 },
      { 502, 635, 809, 947, 1402, 3634, 5268 },
      { 2, 2, 2, 6, 13, 30, 46 },
      { 163, 203, 276, 408, 547, 729, 628 },
      { 2, 7, 26, 82, 172, 307, 392 },
      { 16, 24, 38, 74, 167, 511, 809 }
   };
```

Step 3 To print the row headers, we also need a one-dimensional array of the continent names. Note that it has the same number of rows as our table.

```
string continents[ROWS] =
   {
      "Africa",
      "Asia",
      "Australia",
      "Europe",
      "North America",
      "South America"
   };
```

To print a row, we first print the continent name, then all columns. This is achieved with two nested loops. The outer loop prints each row:

```
// Print data
for (int i = 0; i < ROWS; i++)
{
   // Print the ith row
   . . .
   cout << endl; // Start a new line at the end of the row
}
```

To print a row, we first print the row header, then all columns:

```
cout << setw(20) << continents[i];
for (int j = 0; j < COLUMNS; j++)
{
   cout << setw(5) << data[i][j];
}
```

Step 4 To print the column sums, we use a helper function, as described in Section 6.6.5. We carry out that computation once for each column.

```
for (int j = 0; j < COLUMNS; j++)
{
   cout << setw(5) << column_total(data, ROWS, j);
}
```

Here is the complete program:

worked_example_2/worldpop.cpp

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

const int ROWS = 6;
const int COLUMNS = 7;

/**
   Computes the total of a column in a table.
   @param table a table with 7 columns
   @param rows the number of rows of the table
   @param column the column that needs to be totaled
   @return the sum of all elements in the given column
*/
int column_total(int table[][COLUMNS], int rows, int column)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][column];
   }
   return total;
}

int main()
{
   int data[ROWS][COLUMNS] =
   {
      { 106, 107, 111, 133, 221, 767, 1766 },
      { 502, 635, 809, 947, 1402, 3634, 5268 },
```

```
      { 2, 2, 2, 6, 13, 30, 46 },
      { 163, 203, 276, 408, 547, 729, 628 },
      { 2, 7, 26, 82, 172, 307, 392 },
      { 16, 24, 38, 74, 167, 511, 809 }
   };

   string continents[ROWS] =
      {
         "Africa",
         "Asia",
         "Australia",
         "Europe",
         "North America",
         "South America"
      };

   cout << "                Year 1750 1800 1850 1900 1950 2000 2050"
      << endl;

   // Print data
   for (int i = 0; i < ROWS; i++)
   {
      // Print the ith row
      cout << setw(20) << continents[i];
      for (int j = 0; j < COLUMNS; j++)
      {
         cout << setw(5) << data[i][j];
      }
      cout << endl; // Start a new line at the end of the row
   }

   // Print column totals
   cout << "               World";
   for (int j = 0; j < COLUMNS; j++)
   {
      cout << setw(5) << column_total(data, ROWS, j);
   }
   cout << endl;

   return 0;
}
```

Program Run

```
                Year 1750 1800 1850 1900 1950 2000 2050
              Africa  106  107  111  133  221  767 1766
                Asia  502  635  809  947 1402 3634 5268
           Australia    2    2    2    6   13   30   46
              Europe  163  203  276  408  547  729  628
       North America    2    7   26   82  172  307  392
       South America   16   24   38   74  167  511  809
               World  791  978 1262 1650 2522 5978 8909
```

CHAPTER 7

POINTERS AND STRUCTURES

© Suzanne Tucker/Shutterstock.

CHAPTER GOALS

To be able to declare, initialize, and use pointers

To understand the relationship between arrays and pointers

To be able to convert between string objects and character pointers

To become familiar with dynamic memory allocation and deallocation

To use structures to aggregate data items

CHAPTER CONTENTS

In the game in the photo, the spinner's pointer moves to an item. A player follows the pointer and handles the item to which it points—by taking the ball or following the instructions written in the space. C++ also has pointers that can point to different values throughout a program run. Pointers let you work with data whose locations change or whose size is variable. In the second part of this chapter, you will learn how to use structures. Unlike arrays, structures can be used to group values of different types together. Finally, you will see how to use pointers and structures to model interconnected data.

7.1 Defining and Using Pointers

With a variable, you can access a value at a fixed location. With a pointer, the location can vary. This capability has many useful applications. Pointers can be used to share values among different parts of a program. Pointers allow allocation of values on demand. Furthermore, as you will see in Chapter 10, pointers are necessary for implementing programs that manipulate objects of multiple related types. In the first half of this chapter, you will learn how to define pointers and access the values to which they point.

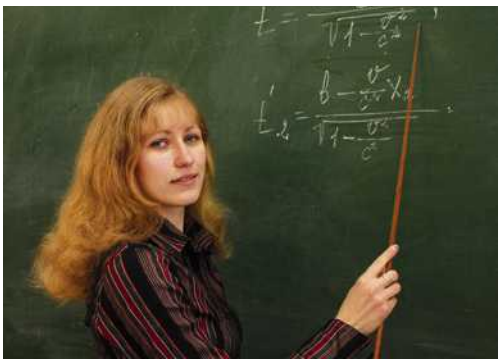
7.1.1 Defining Pointers

Consider a person who wants a program for making bank deposits and withdrawals, but who may not always use the same bank account. By using a pointer, it is possible to switch to a different account without modifying the code for deposits and withdrawals.

Let's start with a variable for storing an account balance:

```
double harrys_account = 0;
```

Now suppose that we want to write an algorithm that manipulates a bank account, but we anticipate that we may sometimes want to use `harrys_account`, sometimes another account. Using a pointer gives us that flexibility. A pointer tells you *where* a value is located, not what the value is.



Like a pointer that points to different locations on a blackboard, a C++ pointer can point to different memory locations.

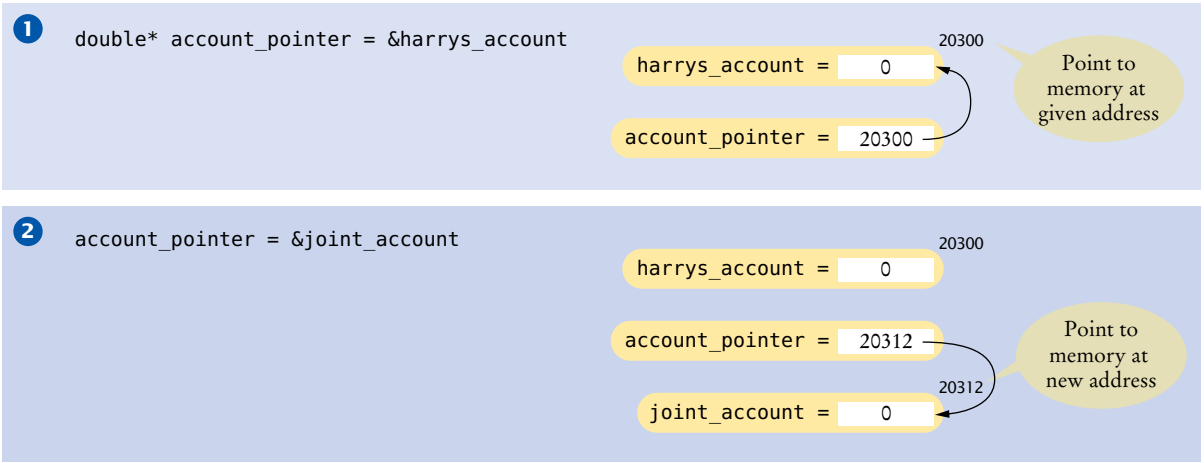


Figure 1 Pointers and Values in Memory

A pointer denotes the location of a variable in memory.

Here is the definition of a pointer variable. The pointer variable is initialized with the location (also called the **address**) of the variable `harrys_account` (see Figure 1):

```
double* account_pointer = &harrys_account;
```
1

The type T* denotes a pointer to a variable of type T.

The type `double*`, or "pointer to `double`", denotes the location of a `double` variable. The `&` operator, also called the *address operator*, yields the location of a variable. Taking the address of a `double` variable yields a value of type `double*`.

The & operator yields the location of a variable.

Thinking about pointers can be rather abstract, but you can use a simple trick to make it more tangible. Every variable in a computer program is located in a specific memory location. You don't know where each variable is stored, but you can pretend you do. Let's pretend that we know that `harrys_account` is stored in location 20300. (That is just a made-up value.) As shown in Figure 1, the value of `harrys_account` is 0, but the value of `&harrys_account` is 20300. The value of `account_pointer` is also 20300. In our diagrams, we will draw an arrow from a pointer to the location, but of course the computer doesn't store arrows, just numbers.

By using a pointer, you can switch to a different account at any time. To access a different account, simply change the pointer value:

```
account_pointer = &joint_account;
```
2

7.1.2 Accessing Variables Through Pointers

The * operator accesses the variable to which a pointer points.

When you have a pointer, you will want to access the variable to which it points. The `*` operator is used to read or update the variable to which a pointer points. When used with pointers, the `*` operator has no relationship with multiplication. In the C++ standard, this operator is called the *indirection operator*, but it is also commonly called the ***dereferencing*** *operator*.

This statement makes an initial deposit into the account to which `account_pointer` points (see Figure 2):

```
*account_pointer = 1000;
```
1

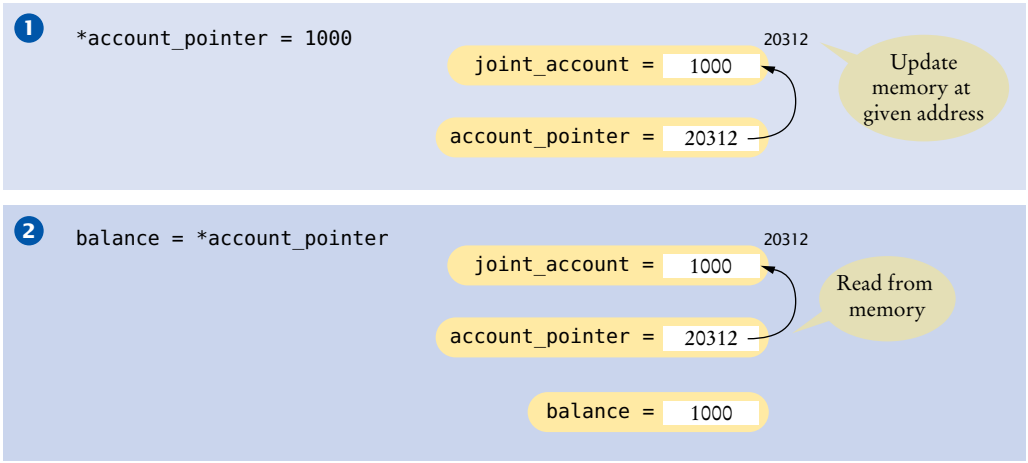


Figure 2 Pointer Variables Can be on Either Side of an Assignment

In other words, you can use `*account_pointer` in exactly the same way as `harrys_account` or `joint_account`. Which account is used depends on the value of the pointer. When the program executes this statement, it fetches the address stored in `account_pointer`. It then uses the variable at that address, as shown in Figure 2.

An expression such as `*account_pointer` can be on the left or the right of an assignment. When it occurs on the left, then the value on the right is stored in the location to which the pointer refers. When it occurs on the right, then the value is fetched from the location and assigned to the variable on the left. For example, the following statement reads the variable to which `account_pointer` currently points, and places its contents into the `balance` variable:

```
balance = *account_pointer; 2
```

You can have `*account_pointer` on both sides of an assignment. The following statement withdraws $100:

```
*account_pointer = *account_pointer - 100;
```

Syntax 7.1 Pointer Syntax

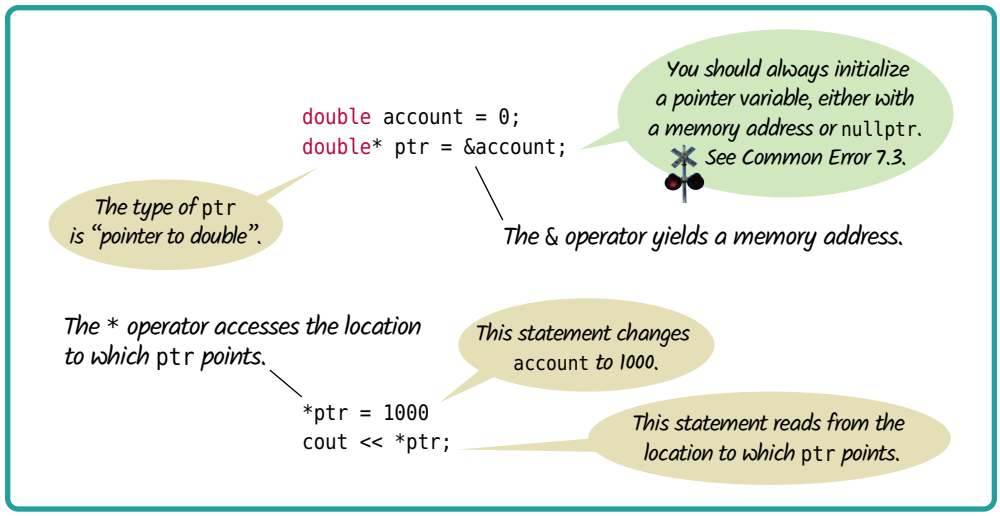


Table 1 contains additional pointer examples.

Table 1 Pointer Syntax Examples

Assume the following declarations:

```
int m = 10; // Assumed to be at address 20300
int n = 20; // Assumed to be at address 20304
int* p = &m;
```

Expression	Value	Comment
`p`	20300	The address of `m`.
`*p`	10	The value stored at that address.
`&n`	20304	The address of `n`.
`p = &n;`		Set `p` to the address of `n`.
`*p`	20	The value stored at the changed address.
`m = *p;`		Stores 20 into `m`.
`m = p;`	**Error**	`m` is an `int` value; `p` is an `int*` pointer. The types are not compatible.
`&10`	**Error**	You can only take the address of a variable.
`&p`	The address of `p`, perhaps 20308	This is the location of a pointer variable, not the location of an integer.
`double x = 0;` `p = &x;`	**Error**	`p` has type `int*`, `&x` has type `double*`. These types are incompatible.

7.1.3 Initializing Pointers

With pointers, it is particularly important that you pay attention to proper initialization.

When you initialize a pointer, be sure that the pointer and the memory address have the same type. For example, the following initialization would be an error:

```
int balance = 1000;
double* account_pointer = &balance; // Error!
```

The address `&balance` is a pointer to an `int` value, that is, an expression of type `int*`. It is never legal to initialize a `double*` pointer with an `int*`.

It is an error to use an uninitialized pointer.

If you define a pointer variable without providing an initial variable, the pointer contains a random address. Using that random address is an error. In practice, your program will likely crash or mysteriously misbehave if you use an uninitialized pointer:

```
double* account_pointer;
   // Forgot to initialize
*account_pointer = 1000;
   // NO! account_pointer contains an unpredictable value
```

The nullptr pointer does not point to any object.

There is a special value, nullptr, that you should use to indicate a pointer that doesn't point anywhere. If you define a pointer variable and are not ready to initialize it quite yet, set it to nullptr.

```
double* account_pointer = nullptr; // Will set later
```

You can later test whether the pointer is still nullptr. If it is, don't use it.

```
if (account_pointer != nullptr) { cout << *account_pointer; } // OK
```

Trying to access data through a nullptr pointer is illegal, and it will cause your program to terminate.

The reserved word nullptr was introduced in C++ 11. If you use an older version of C++, use the NULL constant instead.

The following program demonstrates the behavior of pointers. We execute the same withdrawal statement twice, but with different values for account_pointer. Each time, a different account is modified.

sec01/accounts.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7     double harrys_account = 0;
8     double joint_account = 2000;
9     double* account_pointer = &harrys_account;
10
11    *account_pointer = 1000; // Initial deposit
12
13    *account_pointer = *account_pointer - 100; // Withdraw $100
14    cout << "Balance: " << *account_pointer << endl; // Print balance
15
16    // Change the pointer value
17    account_pointer = &joint_account;
18
19    // The same statements affect a different account
20    *account_pointer = *account_pointer - 100; // Withdraw $100
21    cout << "Balance: " << *account_pointer << endl; // Print balance
22
23    return 0;
24 }
```

Program Run

```
Balance: 900
Balance: 1900
```

Common Error 7.1

Confusing Pointers with the Data to Which They Point

A pointer is a memory address—a number that tells where a value is located in memory. It is a common error to confuse the pointer with the variable to which it points:

```
double* account_pointer = &joint_account;
account_pointer = 1000; // Error
```

The assignment statement does not set the joint account balance to 1000. Instead, it sets the pointer to point to memory address 1000. The pointer `account_pointer` only describes *where* the joint account variable is, and it almost certainly is not located at address 1000. Most compilers will report an error for this assignment.

To actually access the variable, use `*account_pointer`:

```
*account_pointer = 1000; // OK
```

Programming Tip 7.1

Use a Separate Definition for Each Pointer Variable

It is legal in C++ to define multiple variables together, like this:

```
int i = 0, j = 1;
```

This style can lead to problems when used with pointers. The following declaration is not correct:

```
double* p = &i, q = &j;
```

The * associates only with the first variable. That is, `p` is a `double*` pointer, and `q` is a `double` value.

It is legal to define multiple pointers together like this:

```
double *p = &i, *q = &j;
```

That means that both `p` and `q` are variables and that `*p` and `*q` are `double` values. In other words, `p` and `q` are both pointers to `double`. Some programmers find this syntax hard to understand.

To avoid any confusion, it is best to define each pointer variable separately:

```
double* p = &i;
double* q = &j;
```

Special Topic 7.1

Pointers and References

In Section 5.9, you saw how reference parameters enable a function to modify variables that are passed as arguments. Here is an example of a function with a reference parameter:

```
void withdraw(double& balance, double amount)
{
   if (balance >= amount)
   {
      balance = balance - amount;
   }
}
```

If you call

```
withdraw(harrys_checking, 1000);
```

then $1000 is withdrawn from `harrys_checking`, provided that sufficient funds are available.

You can use pointers to achieve the same effect:

```
void withdraw(double* balance, double amount)
{
   if (*balance >= amount)
   {
      *balance = *balance - amount;
   }
}
```

However, then you need to call the function with the *address* of the account:

```
withdraw(&harrys_checking, 1000);
```

These solutions are equivalent. Behind the scenes, the compiler translates reference parameters into pointers.

7.2 Arrays and Pointers

Pointers are particularly useful for understanding the peculiarities of arrays. In the following sections, we describe the relationship between arrays and pointers in C++.

7.2.1 Arrays as Pointers

Consider this declaration of an array:

```
double a[10];
```

As you know, `a[3]` denotes an array element. The array name *without* brackets denotes a pointer to the starting element (see Figure 3).

The name of an array variable is a pointer to the starting element of the array.

You can capture that pointer in a variable:

```
double* p = a; // Now p points to a[0]
```

You can also use the array name as a pointer in expressions. The statement

```
cout << *a;
```

has the same effect as the statement

```
cout << a[0];
```

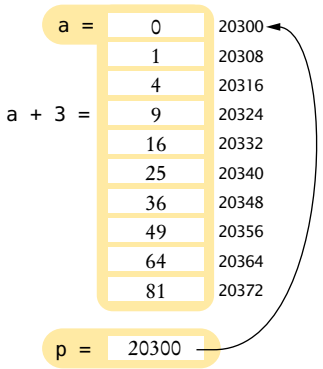


Figure 3 Pointers into an Array

7.2.2 Pointer Arithmetic

Pointers into arrays support *pointer arithmetic*. You can add an integer offset to the pointer to point to another array location. For example, suppose `p` points to the beginning of an array:

```
double a[10];
double* p = a;
```

Pointer arithmetic means adding an integer offset to an array pointer, yielding a pointer that skips past the given number of elements.

Then the expression

```
p + 3
```

is a pointer to the array element with index 3, and

```
*(p + 3)
```

is that array element.

As you saw in the preceding section, we can use the array name as a pointer. That is, `a + 3` is a pointer to the array element with index 3, and `*(a + 3)` has exactly the same meaning as `a[3]`.

In fact, for any integer `n`, it is true that

`a[n]` is the same as `*(a + n)`

This relationship is called the *array/pointer duality law*.

The array/pointer duality law states that `a[n]` is identical to `*(a + n)`, where `a` is a pointer into an array and `n` is an integer offset.

This law explains why all C++ arrays start with an index of zero. The pointer `a` (or `a + 0`) points to the starting element of the array. That element must therefore be `a[0]`.

To better understand pointer arithmetic, let's again pretend that we know actual memory addresses. Suppose the array `a` starts at address 20300. The array contains ten values of type `double`. A `double` value occupies 8 bytes of memory. Therefore, the array occupies 80 bytes, from 20300 to 20379. The starting value is located at address 20300, the next one at address 20308, and so on (see Figure 3). For example, the value of `a + 3` is $20300 + 3 \times 8 = 20324$. (In general, if `p` is a pointer to a type `T`, then the address `p + n` is obtained by adding `n` × the size of a `T` value to the address `p`.)

Table 2 shows pointer arithmetic and the array/pointer duality using this example.

Table 2 Arrays and Pointers

Expression	Value	Comment
`a`	20300	The starting address of the array, here assumed to be 20300.
`*a`	0	The value stored at that address. (The array contains values 0, 1, 4, 9, ...)
`a + 1`	20308	The address of the next `double` value in the array. A `double` occupies 8 bytes.
`a + 3`	20324	The address of the element with index 3, obtained by skipping past 3 × 8 bytes.
`*(a + 3)`	9	The value stored at address 20324.
`a[3]`	9	The same as `*(a + 3)` by array/pointer duality.
`*a + 3`	3	The sum of `*a` and 3. Because there are no parentheses, the `*` refers only to `a`.
`&a[3]`	20324	The address of the element with index 3, the same as `a + 3`.

7.2.3 Array Parameter Variables Are Pointers

Once you understand the connection between arrays and pointers, it becomes clear why array parameter variables are different from other parameter types. As an example, consider this function that computes the sum of all values in an array:

```
double sum(double values[], int size)
{
   double total = 0;
   for (int i = 0; i < size; i++)
   {
      total = total + values[i];
   }
   return total;
}
```

Here is a call to the function (see Figure 4):

```
double a[10];
. . . // Initialize a
double s = sum(a, 10);
```

When passing an array to a function, only the starting address is passed.

The value `a` is passed to the `sum` function. It is actually a pointer of type `double*`, pointing to the starting element of the array. One would therefore expect that the function is declared as

```
double sum(double* values, int size)
```

However, if you look closely at the function definition, you will see that the parameter variable is declared as an array with empty bounds:

```
double sum(double values[], int size)
```

As viewed by the C++ compiler, these parameter declarations are completely equivalent. The `[]` notation is "syntactic sugar" for declaring a pointer. (Computer scientists use the term "syntactic sugar" to describe a notation that is easy to read for humans and that masks a complex implementation detail.) The array notation gives human

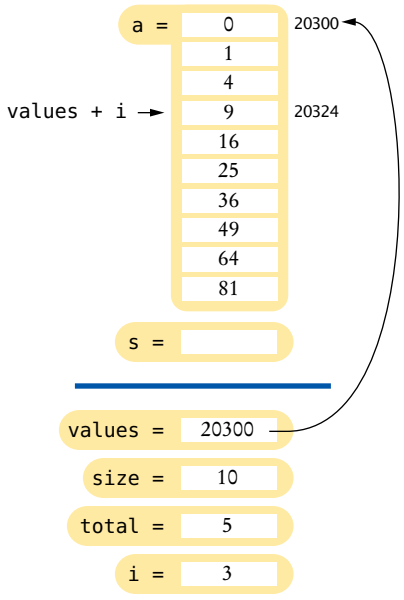


Figure 4
Passing an Array to a Function

readers the illusion that an entire array is passed to the function, but in fact the function receives only the starting address for the array.

Now consider this statement in the body of the function:

```
total = total + values[i];
```

The C++ compiler considers `values` to be a *pointer*, not an array. The expression `values[i]` is syntactic sugar for `*(values + i)`. That expression denotes the storage location that is `i` elements away from the address stored in the variable `values`. Figure 4 shows how `values[i]` is accessed when `i` is 3.

You can now understand why it is always necessary to pass the size of the array. The function receives a single memory address, which tells it where the array starts. That memory address enables the function to locate the values in the array. But the function also needs to know where to stop.

EXAMPLE CODE See sec02 of your companion code for a program that demonstrates pointer values.

Special Topic 7.2

Using a Pointer to Step Through an Array

Consider again the `sum` function of Section 7.2.3. Now that you know that the first parameter variable of the `sum` function is a pointer, you can implement the function in a slightly different way. Rather than incrementing an integer index, you can increment a pointer variable to visit all array elements in turn:

```
double sum(double* values, int size)
{
   double total = 0;
   double* p = values; // p starts at the beginning of the array
   for (int i = 0; i < size; i++)
   {
      total = total + *p; // Add the value to which p points
      p++; // Advance p to the next array element
   }
   return total;
}
```

Initially, the pointer `p` points to the element `values[0]`. The increment

```
p++;
```

moves it to point to the next element (see Figure 5).

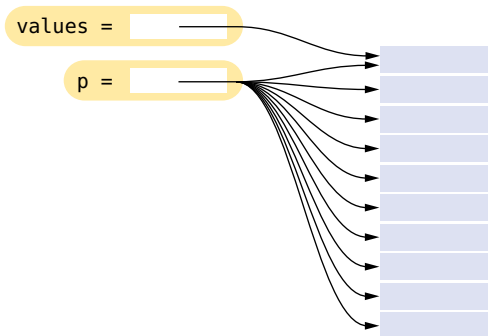


Figure 5 A Pointer Variable Traversing the Elements of an Array

It is a tiny bit more efficient to use and increment a pointer than to access an array element as `values[i]`. For this reason, some programmers routinely use pointers instead of indexes to access array elements.

Common Error 7.2

Returning a Pointer to a Local Variable

Consider this function that tries to return a pointer to an array containing two elements, the first and the last values of an array:

```
double* firstlast(double values[], int size)
{
   double result[2];
   result[0] = values[0];
   result[1] = values[size - 1];
   return result; // Error!
}
```

The function returns a pointer to the starting element of the `result` array. However, that array is a local variable of the `firstlast` function. The local variable no longer exists when the function exits. Its contents will soon be overwritten by other function calls.

You can solve this problem by passing an array to hold the answer:

```
void firstlast(const double values[], int size, double result[])
{
   result[0] = values[0];
   result[1] = values[size - 1];
}
```

Then it is the responsibility of the caller to allocate an array to hold the result.

Programming Tip 7.2

Program Clearly, Not Cleverly

Some programmers take great pride in minimizing the number of instructions, even if the resulting code is hard to understand. For example, here is a legal implementation of the `sum` function:

```
double sum(double* values, int size)
{
   double total = 0;
   while (size-- > 0) // Loop size times
   {
      total = total + *values++; // Add the value to which values points; increment values
   }
   return total;
}
```

This implementation uses two tricks. First, the function parameter variables `values` and `size` are variables, and it is legal to modify them. Moreover, the expressions `size--` and `values++` mean "decrement or increment the variable and return the old value". In other words, the expression

```
size-- > 0
```

combines two tasks: to decrement size, and to test whether `size` was positive before the decrement. Similarly, the expression

```
*values++
```

increments the pointer to the next element, and it returns the element to which it pointed before the increment.

Please do not use this programming style. Your job as a programmer is not to dazzle other programmers with your cleverness, but to write code that is easy to understand and maintain.

Special Topic 7.3

Constant Pointers

The following definition specifies a constant pointer:

```
const double* p = &balance;
```

You cannot modify the value to which p points. That is, the following statement is illegal:

```
*p = 0; // Error
```

Of course, you can read the value:

```
cout << *p; // OK
```

A constant array parameter variable is equivalent to a constant pointer. For example, consider the function

```
double sum(const double values[], int size)
```

Recall from Section 7.2.3 that values is a pointer. The function could have been defined as

```
double sum(const double* values, int size)
```

The function can use the pointer values to read the array elements, but it cannot modify them.

7.3 C and C++ Strings

C++ has two mechanisms for manipulating strings. The string class supports character sequences of arbitrary length and provides convenient operations such as concatenation and string comparison. However, C++ also inherits a more primitive level of string handling from the C language, in which strings are represented as arrays of char values. In this sections, we will discuss the relationships between these types.

7.3.1 The char Type

A value of type char denotes an individual character. Character literals are enclosed in single quotes.

The char type denotes an individual character. Character literals are delimited by single quotes; for example,


```
char input = 'y';
```

Each character is actually encoded as an integer value. (See Appendix C for the encoding using the ASCII code, which is used on the majority of computers today.)

Note that 'y' is a single character, which is quite different from "y", a string containing the 'y' character.

Table 3 shows typical character literals.

Table 3 Character Literals	
`'y'`	The character y
`'0'`	The character for the digit 0. In the ASCII code, `'0'` has the value 48.
`' '`	The space character
`'\n'`	The newline character
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a `char` value

Letters from foreign alphabets and special symbols may require multiple `char` values. See Chapter 8 for more information.

7.3.2 C Strings

A literal string (enclosed in double quotes) is an array of `char` values with a zero terminator.

In the C programming language, strings are always represented as character arrays. C++ programmers often refer to arrays of `char` values as "C strings".

In particular, a literal string, such as `"Harry"`, is *not* an object of type `string`. Instead, it is an array of `char` values. As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
const char* char_pointer = "Harry"; // Points to 'H'
```

The string is declared as `const` because you are not supposed to modify a literal string. (See Special Topic 7.3 for more information on constant pointers.)

A C string is terminated by a special character, called a *null terminator*, denoted `'\0'`. For example, the C string `"Harry"` contains *six* characters, namely `'H'`, `'a'`, `'r'`, `'r'`, `'y'` and `'\0'` (See Figure 6.)

The terminator is a character that is encoded as the number zero—this is different from the character `'0'`, the character denoting the zero digit. (Under the ASCII encoding scheme, the character denoting the zero digit is encoded as the number 48.)

Functions that operate on C strings rely on this terminator. Here is a typical example, the `strlen` function declared in the `<cstring>` header that computes the length of a character array. The function counts the number of characters until it reaches the terminator.

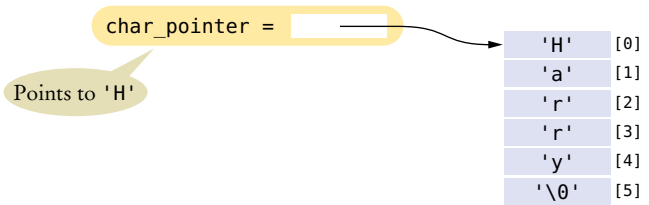


Figure 6 A Character Array

```
int strlen(const char s[])
{
   int i = 0;
   while (s[i] != '\0') { i++; } // Count characters before the null terminator
   return i;
}
```

The call `strlen("Harry")` returns 5.

7.3.3 Character Arrays

A literal string such as `"Harry"` is a constant. You are not allowed to modify its characters. If you want to modify the characters in a string, define a character array instead. For example:

```
char char_array[] = "Harry"; // An array of 6 characters
```

The `char_array` variable is an array of 6 characters, initialized with `'H'`, `'a'`, `'r'`, `'r'`, `'y'`, and a null terminator. The compiler counts the characters in the string that is used for initializing the array, including the null terminator.

You can modify the characters in the array:

```
char_array[0] = 'L';
```

7.3.4 Converting Between C and C++ Strings

Many library functions use pointers of type `char*`.

Before the C++ `string` class became widely available, direct manipulation of character arrays was common, but also quite challenging (see Special Topic 7.4). If you use functions that receive or return C strings, you need to know how to convert between C strings and `string` objects.

For example, the `<cstdlib>` header declares a useful function

```
int atoi(const char s[])
```

The `atoi` function converts a character array containing digits into its integer value:

```
char year[] = "2020";
int y = atoi(year); // Now y is the integer 2020
```

The `c_str` member function yields a `char*` pointer from a `string` object.

In older versions of C++, this functionality was missing from the `string` class. The `c_str` member function of the `string` class offers an "escape hatch". If `s` is a string, then `s.c_str()` yields a `char*` pointer to the characters in the string. Here is how you use that member function to call the `atoi` function:

```
string year = "2020";
int y = atoi(year.c_str());
```

You can initialize C++ `string` variables with C strings.

Conversely, converting from a C string to a C++ string is very easy. Simply initialize a `string` variable with any value of type `char*`, such as a string literal or character array. For example, the definition

```
string name = "Harry";
```

initializes the C++ `string` object `name` with the C string `"Harry"`.

7.3.5 C++ Strings and the [] Operator

Up to this point, we have always used the `substr` member function to access individual characters in a C++ string. For example, if a `string` variable is defined as

```
string name = "Harry";
```

the expression

```
name.substr(3, 1)
```

yields a string of length 1 containing the character at index 3.

You can access characters in a C++ `string` object with the `[]` operator.

You can access individual characters with the `[]` operator:

```
name[0] = 'L';
```

Now the string is `"Larry"`. The `[]` operator is more convenient than the `substr` function if you want to visit a string one character at a time.

Here is a useful example. The following function makes a copy of a string and changes all characters to uppercase:

```
/**
   Makes an uppercase version of a string.
   @param str a string
   @return a string with the characters in str converted to uppercase
*/
string uppercase(string str)
{
   string result = str; // Make a copy of str
   for (int i = 0; i < result.length(); i++)
   {
      result[i] = toupper(result[i]); // Convert each character to uppercase
   }
   return result;
}
```

For example, `uppercase("Harry")` returns a string with the characters `"HARRY"`.

The `toupper` function is defined in the `<cctype>` header. It converts lowercase characters to uppercase. (The `tolower` function does the opposite.)

EXAMPLE CODE See sec03 of your companion code for a program that demonstrates string processing.

Special Topic 7.4

Working with C Strings

Before the `string` class became widely available, it was common to work with character arrays directly.

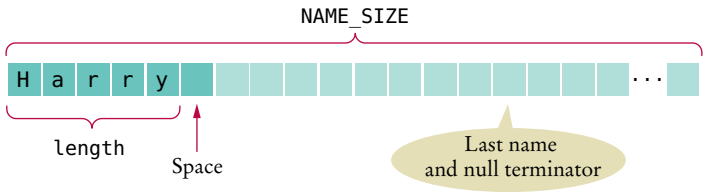
Table 4 shows several commonly used functions for manipulating C strings. All of these functions are declared in the `<cstring>` header.

Consider the task of concatenating a first name and a last name into a string. The `string` class makes this very easy:

```
string first = "Harry";
string last = "Smith";
string name = first + " " + last;
```

Let us implement this task with C strings. Allocate an array of characters for the result:

```
const int NAME_SIZE = 40;
char name[NAME_SIZE];
```



This array can hold strings with a length of at most 39, because one character is required for the null terminator.

Now copy the first name, using `strncpy`:

```
strncpy(name, first, NAME_SIZE - 1);
```

You must be careful not to overrun the target array. It is unlikely that a first name is longer than 39 characters, but a hacker could supply a longer input in order to overwrite memory.

Now, if there is still room, add a space and the last name, again being careful not to overrun the array boundaries.

```
int length = strlen(name);
if (length < NAME_SIZE - 1)
{
   strcat(name, " ");
   int n = NAME_SIZE - 2 - length; // Leave room for space, null terminator
   if (n > 0)
   {
      strncat(name, last, n);
   }
}
```

As you can see, the C string code is over three times as long as the code using C++ strings, and it is not as capable—if the target array is not long enough to hold the result, it is truncated.

Table 4 C String Functions

In this table, `s` and `t` are character arrays; `n` is an integer.

Function	Description
`strlen(s)`	Returns the length of `s`.
`strcpy(t, s)`	Copies the characters from `s` into `t`.
`strncpy(t, s, n)`	Copies at most `n` characters from `s` into `t`.
`strcat(t, s)`	Appends the characters from `s` after the end of the characters in `t`.
`strncat(t, s, n)`	Appends at most `n` characters from `s` after the end of the characters in `t`.
`strcmp(s, t)`	Returns `0` if `s` and `t` have the same contents, a negative integer if `s` comes before `t` in lexicographic order, a positive integer otherwise.

7.4 Dynamic Memory Allocation

Use dynamic memory allocation if you do not know in advance how many values you need.

In many programming situations, you do not know beforehand how many values you need. To solve this problem, you can use **dynamic allocation** and ask the C++ run-time system to create new values whenever you need them. The run-time system keeps a large storage area, called the **free store**, that can allocate values and arrays of any type. When you ask for a

```
new double
```

then a storage location of type `double` is located on the free store, and a pointer to that location is returned.

The `new` operator allocates memory from the free store.

More usefully, the expression

```
new double[n]
```

allocates an array of size `n`, and yields a pointer to the starting element. (Here `n` need *not* be a constant.)

You will want to capture that pointer in a variable:

```
double* account_pointer = new double;
double* account_array = new double[n];
```

You now use the pointer as described previously in this chapter. If you allocated an array, the magic of array/pointer duality lets you use the array notation `account_array[i]` to access the `i`th element.

You must reclaim dynamically allocated objects with the `delete` or `delete[]` operator.

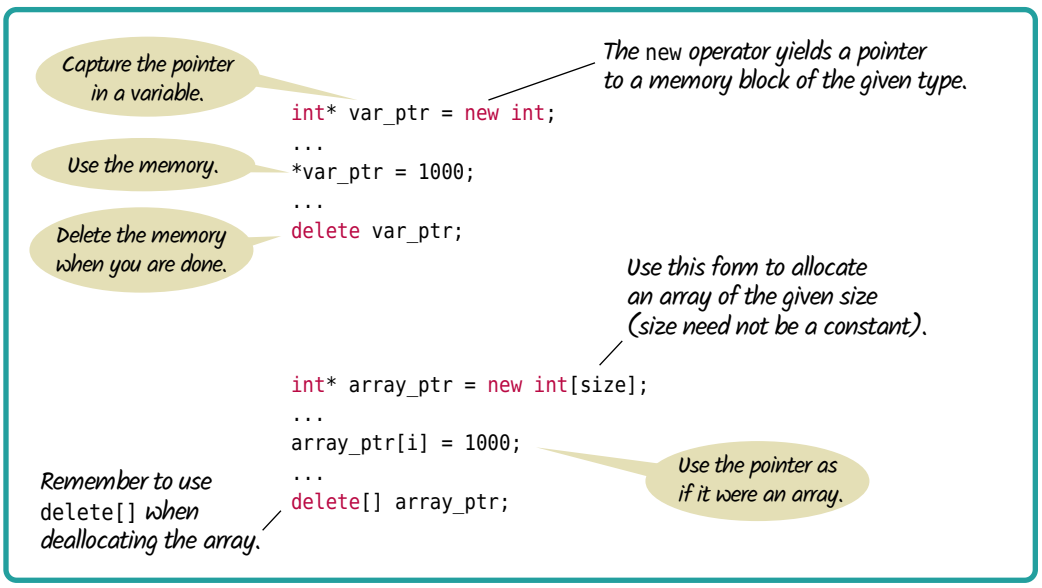
When your program no longer needs memory that you previously allocated with the `new` operator, you must return it to the free store, using the `delete` operator:

```
delete account_pointer;
```

However, if you allocated an array, you must use the `delete[]` operator:

```
delete[] account_array;
```

Syntax 7.2 Dynamic Memory Allocation



This operator reminds the free store that the pointer points to an array, not a single value.

After you delete a memory block, you can no longer use it. The storage space may already be used elsewhere.

```
delete[] account_array;
account_array[0] = 1000; // NO! You no longer own the memory of account_array
```

Arrays that are allocated on the free store have one significant advantage over array variables. If you declare an array variable, you must specify a fixed array size when you compile the program. But when you allocate an array on the free store, you can choose the size at run time.

Moreover, if you later need more elements, you are not stuck. You can allocate a bigger array on the free store, copy the elements from the smaller array into the bigger array, and delete the smaller array (see Figure 7):

```
double* bigger_array = new double[2 * n];
for (int i = 0; i < n; i++)
{
   bigger_array[i] = account_array[i];
}
delete[] account_array;
account_array = bigger_array;
n = 2 * n;
```

This is exactly what a `vector` does behind the scenes.

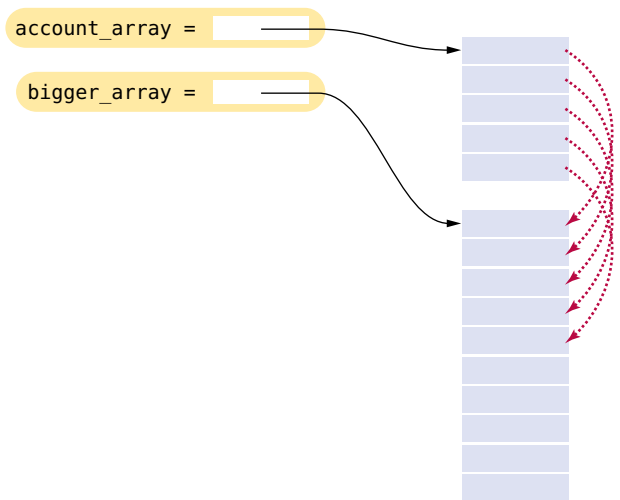


Figure 7 Growing a Dynamic Array

Dynamic allocation is a powerful feature, but you must be very careful to follow all rules precisely:

- Every call to `new` must be matched by exactly one call to `delete`.
- Use `delete[]` to delete arrays.
- Don't access a memory block after it has been deleted.

If you don't follow these rules, your program can crash or run unpredictably. Table 5 shows common errors.

Table 5 Common Memory Allocation Errors

Statements	Error

`int* p;`
`*p = 5;`
`delete p;`
	There is no call to `new int`.

`int* p = new int;`
`*p = 5;`
`p = new int;`
	The first allocated memory block was never deleted.

`int* p = new int[10];`
`*p = 5;`
`delete p;`
	The `delete[]` operator should have been used.

`int* p = new int[10];`
`int* q = p;`
`q[0] = 5;`
`delete p;`
`delete q;`
	The same memory block was deleted twice.

`int n = 4;`
`int* p = &n;`
`*p = 5;`
`delete p;`
	You can only delete memory blocks that you obtained from calling `new`.

EXAMPLE CODE See sec04 of your companion code for a program that demonstrates dynamic memory allocation.

Common Error 7.3

Dangling Pointers

A very common pointer error is to use a pointer that points to memory that has already been deleted. Such a pointer is called a **dangling pointer**. Because the freed memory will be reused for other purposes, you can create real damage by using a dangling pointer. Consider this example:

```
int* values = new int[n];
// Process values
delete[] values;
// Some other work
values[0] = 42;
```

Using a dangling pointer (a pointer that points to memory that has been deleted) is a serious programming error.

This code will compile because the compiler does not track whether a pointer points to a valid memory location. However, the program may run with unpredictable results. If the program calls the `new` operator anywhere after deleting `values`, that call may allocate the same memory again. Now some other part of your program accesses the memory to which `values` points, and that program part will malfunction when you overwrite the memory. This can happen even if you don't see any call to `new`—such calls may occur in library functions.

Never use a pointer that has been deleted. Some programmers take the precaution of setting all deleted pointers to `nullptr`:

```
delete[] values;
values = nullptr;
```

This is not perfect protection—you might have saved `values` into another pointer variable—but it is a reasonable precaution.

Common Error 7.4

Memory Leaks

Another very common pointer error is to allocate memory on the free store and never deallocate it. A memory block that is never deallocated is called a **memory leak**.

If you allocate a few small blocks of memory and forget to deallocate them, this is not a huge problem. When the program exits, all allocated memory is returned to the operating system.

Every call to `new` should have a matching call to `delete`.

But if your program runs for a long time, or if it allocates lots of memory (perhaps in a loop), then it can run out of memory. Memory exhaustion will cause your program to crash. In extreme cases, the computer may freeze up if your program exhausted all available memory. Avoiding memory leaks is particularly important in programs that need to run for months or years, without restarting, and in programs that run on resource-constrained devices such as cell phones.

Be sure to recycle any free store memory that your program no longer needs.

Even if you write short-lived programs, you should make it a habit to avoid memory leaks. Make sure that every call to the `new` operator has a corresponding call to the `delete` operator.

7.5 Arrays and Vectors of Pointers

When you have a sequence of pointers, you can place them into an array or vector. An array and a vector of ten `int*` pointers are defined as

```
int* pointer_array[10];
vector<int*> pointer_vector(10);
```

The expression `pointer_array[i]` or `pointer_vector[i]` denotes the pointer with index `i` in the sequence.

One application of such pointer sequences are two-dimensional arrays in which each row has a different length, such as the triangular array shown in Figure 8.

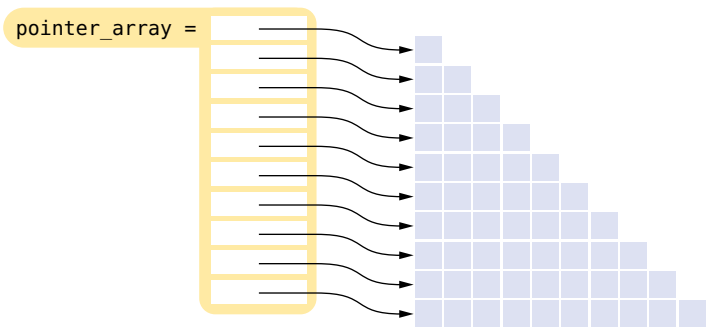


Figure 8 A Triangular Array

In this situation, it would not be very efficient to use a two-dimensional array, because almost half of the elements would be wasted.

We will develop a program that uses such an array for simulating a *Galton board* (Figure 9). A Galton board consists of a pyramidal arrangement of pegs, and a row of bins at the bottom. Balls are dropped onto the top peg and travel toward the bins. At each peg, there is a 50 percent chance of moving left or right. The balls in the bins approximate a bell-curve distribution.

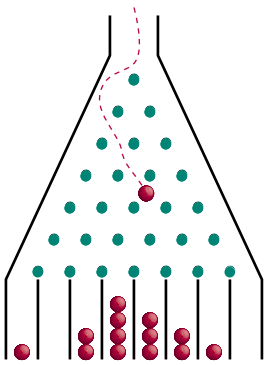


Figure 9 A Galton Board

The Galton board can only show the balls in the bins, but we can do better by keeping a counter for each peg, incrementing it as a ball travels past it.

We will simulate a board with ten rows of pegs. Each row requires an array of counters.

The following statements initialize the triangular array:

```
int* counts[10];
for (int i = 0; i < 10; i++)
{
   counts[i] = new int[i + 1];
}
```

Note that the first element `counts[0]` contains a pointer to an array of length 1, and the last element `counts[9]` contains a pointer to an array of length 10.

Before doing the simulation, let us consider how to print out the values. The element `counts[i]` points to an array. The element at index `j` of that array is

```
counts[i][j]
```

This loop prints all elements in the `i`th row:

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

Now let's simulate a falling ball. The movements to the left and right in Figure 9 correspond to movements to the next row, either straight down or to the right, in Figure 10. More precisely, if the ball is currently at row `i` and column `j`, then it will go to row `i + 1` and, with a 50 percent chance, either stay in column `j` or go to column `j + 1`.

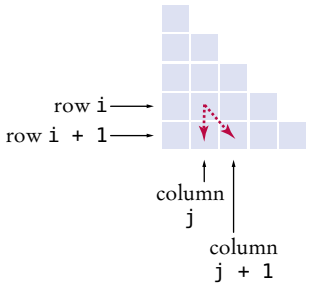


Figure 10 Movement of a Ball in the Galton Board Array

The program below has the details. In the sample program run, notice how 1,000 balls have hit the top peg, and how the bottommost row of pegs approximates a bell-curve distribution.

sec05/galton.cpp

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
   srand(time(0));
   const int SIZE = 10;

   int* counts[SIZE];

   // Allocate the rows
   for (int i = 0; i < SIZE; i++)
   {
      counts[i] = new int[i + 1];
      for (int j = 0; j <= i; j++)
      {
        counts[i][j] = 0;
      }
   }

   const int RUNS = 1000;

   // Simulate 1,000 balls
   for (int run = 0; run < RUNS; run++)
   {
      // Add a ball to the top
      counts[0][0]++;
      // Have the ball run to the bottom
      int j = 0;
      for (int i = 1; i < 10; i++)
      {
         int r = rand() % 2;
         // If r is even, move down, otherwise to the right
         if (r == 1)
         {
```

```
40             j++;
41          }
42          counts[i][j]++;
43       }
44    }
45
46    // Print all counts
47    for (int i = 0; i < SIZE; i++)
48    {
49       for (int j = 0; j <= i; j++)
50       {
51          cout << setw(4) << counts[i][j];
52       }
53       cout << endl;
54    }
55
56    // Deallocate the rows
57    for (int i = 0; i < SIZE; i++)
58    {
59       delete[] counts[i];
60    }
61
62    return 0;
63 }
```

Program Run

```
1000
 480 520
 241 500 259
 124 345 411 120
  68 232 365 271  64
  32 164 283 329 161  31
  16  88 229 303 254  88  22
   9  47 147 277 273 190  44  13
   5  24 103 203 288 228 113  33   3
   1  18  64 149 239 265 186  61  15   2
```

7.6 Problem Solving: Draw a Picture

When designing programs that use pointers, you want to visualize how the pointers connect the data that you are working with. In most situations, it is essential that you draw a diagram that shows the connections.

Start with the data that will be accessed or modified through the pointers. These may be account balances, counters, character strings, or other items. Focus on what is being pointed at.

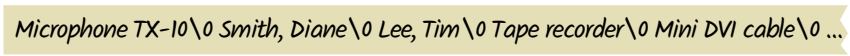
Draw the data that is being processed, then draw the pointer variables. When drawing the pointer arrows, illustrate a typical situation.

Then draw the variable or variables that contain the pointers. These will be the front-end to your system. Processing usually starts with a pointer, then locates the actual data.

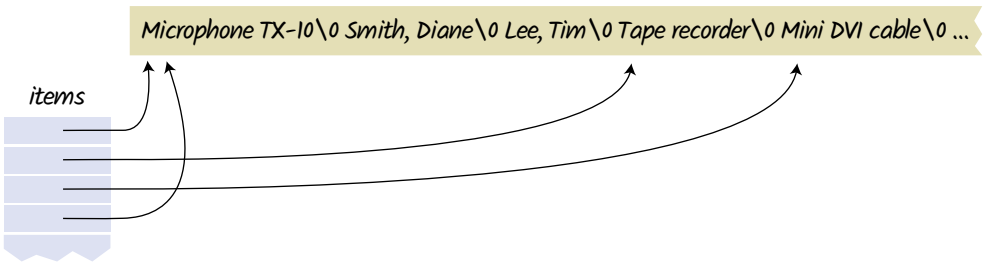
Finally, draw the pointers as arrows. If the pointers will vary as the program executes, draw a typical arrangement. It can also be useful to draw several diagrams that show how the pointers change.

Consider the following problem: The media center of a university loans out equipment, such as microphones, cables, and so on, to faculty and students. We want to track the name of each item, and the name of the user who checked it out.

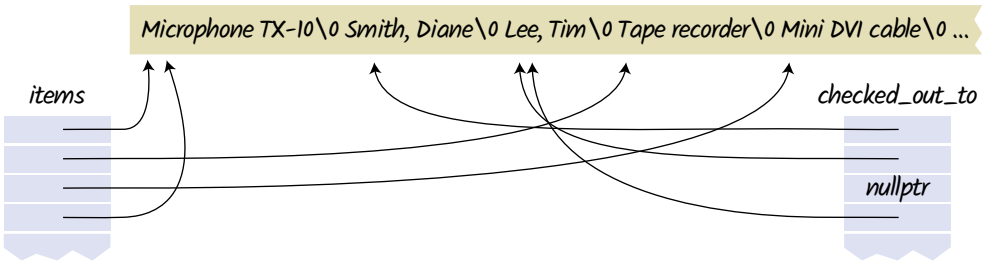
In this case, the data that are being accessed are item and user names. We will store all names in a long array of characters, adding new names to the end as needed. If a name is already present, we don't store it twice. This is an efficient way of storing strings, provided that few strings need to be removed. Here is a section of the character array:



Next, let us draw the pointers to the item names. They are stored in an array of pointers called `items`. Some items may have the same name. In the example below, we have two microphones with the same name.



Finally, there is a parallel array `checked_out_to` of pointers to user names. Sometimes, items can be checked out to the same user. Other items aren't checked out at all—the user name pointer is `nullptr`. When you draw a diagram, try to include examples of all scenarios.



Now that you have a pointer diagram, you can use it to visualize operations on the data. Suppose we want to print a report of all items that are currently checked out. This can be achieved by visiting all pointers in both arrays. The pointer `items[i]` gives the name of the item, and `checked_out_to[i]` is either `nullptr`, in which case we do not want to include this item, or it is the name of the user.

Because programming with pointers is complex, you should always draw diagrams whenever you use pointers.

HOW TO 7.1

Working with Pointers

You use pointers because you want flexibility in your program: the ability to change the relationships between data. This How To walks you through the decision-making process for using pointers.

Problem Statement We will illustrate the steps with the task of simulating a portion of the control logic for a departmental photocopier. A person making a copy enters a user number. There are 100 different users, with numbers 0 to 99. Each user is linked to a copy account, either the master account or one of ten project accounts. That linkage is maintained by the administrator, and it can change as users work on different projects. When copies are made, the appropriate account should be incremented.

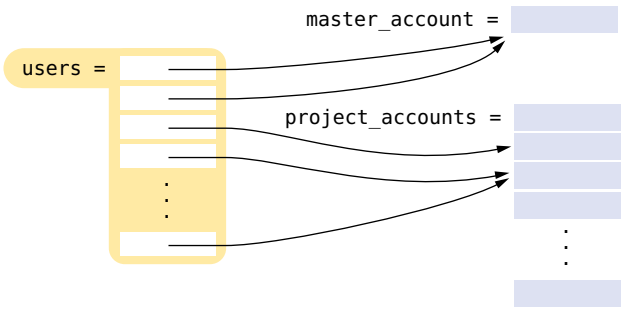
© Bartosz Liszkowski/iStockphoto.

Users identify themselves on the copier control panel. Using pointers, the relationships between users and copy accounts can be flexible.

Step 1 Draw a picture.

As described in Section 7.6, it is a good idea to draw a picture that shows the pointers in your program.

In our example, we need to track the copy accounts: a master account and ten project accounts. For each user, we need a pointer to the copy account:



Step 2 Declare pointer variables.

This step is usually easy. With numerical data, you will have pointers of type `int*` or `double*`. If you manipulate character arrays, you use `char*` pointers. In Chapter 10, you will use pointers to objects.

How many pointer variables do you have? If you only have one or two, just declare variables such as

```
double* account_pointer;
```

If you have a sequence of pointers, use a vector or array, such as

```
vector<char*> lines;
```

In our example, the purpose is to manipulate copy counters. Therefore, our data are integers and the pointers have type `int*`. We know that we will have exactly 100 pointers, one for each user. Therefore, we can choose an array

```
int* users[100];
```

Step 3 Initialize the pointers with variable addresses or free store memory.

Will you allocate variables and then take their addresses with the `&` operator? That is fine if you have a fixed amount of data that you want to reach through the pointers. Otherwise, use the `new` operator to allocate memory dynamically. Be sure to deallocate the memory when you are done, using the `delete` or `delete[]` operator.

In our example, there is no need for dynamic allocation, so we will just take addresses of variables.

```
int master_account;
int project_accounts[10];
for (int i = 0; i < 100; i++)
{
   users[i] = &master_account;
}
// Here we reassign several users to project accounts.
// The following code is a simulation of the actions that would
// occur in an administration interface, which we do not implement.
users[2] = project_accounts + 1
users[3] = project_accounts + 2
users[99] = project_accounts + 2
```

Step 4 Use the `*` or `[]` operator to access your data.

When you access a single variable through a pointer, use the `*` operator to read or write the variable. For example,

```
*account_pointer = *account_pointer + 20;
```

When you have a pointer to an array, use the `[]` notation. Simply think of the pointer as the array:

```
account_array[i] = account_array[i] + 20;
```

If you have an array or vector of pointers, then you need brackets to get at an individual pointer. Then supply another `*` or `[]` to get at the value. You saw an example in the Galton board simulator where a count was accessed as

```
counts[i][j]
```

Implement your algorithm, keeping these access rules in mind.

In our example, we read the user ID and number of copies. Then we increment the copy account:

```
cin >> id >> copies;
*users[id] = *users[id] + copies;
```

Exercise P7.2 asks you to complete this simulation.

WORKED EXAMPLE 7.1

Producing a Mass Mailing

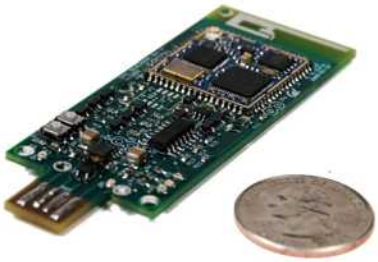
Learn how to use pointers to create a template for a mass mailing. See your E-Text or visit `wiley.com/go/bclo3`.

Computing & Society 7.1 Embedded Systems

An **embedded system** is a computer system that controls a device. The device contains a processor and other hardware and is controlled by a computer program. Unlike a personal computer, which has been designed to be flexible and run many different computer programs, the hardware and software of an embedded system are tailored to a specific device. Computer controlled devices are becoming increasingly common, ranging from washing machines to medical equipment, cell phones, automobile engines, and spacecraft.

Several challenges are specific to programming embedded systems. Most importantly, a much higher standard of quality control applies. Vendors are often unconcerned about bugs in personal computer software, because they can always make you install a patch or upgrade to the next version. But in an embedded system, that is not an option. Few consumers would feel comfortable upgrading the software in their washing machines or automobile engines. If you ever handed in a programming assignment that you believed to be correct, only to have the instructor or grader find bugs in it, then you know how hard it is to write software that can reliably do its task for many years without a chance of changing it. Quality standards are especially important in devices whose failure would destroy property or endanger human life. Many personal computer purchasers buy computers that are fast and have a lot of storage, because the investment is paid back over time when many programs are run on the same equipment. But the hardware for an embedded device is not shared—it is dedicated to one device. A separate processor, memory, and so on, are built for every copy of the device. If it is possible to shave a few pennies off the manufacturing cost of every unit, the savings can add up quickly for devices that are produced in large volumes. Thus, the programmer of an embedded system has a much larger economic incentive to conserve resources than the desktop software programmer. Unfortunately, trying to conserve resources usually makes it harder to write programs that work correctly.

C and C++ are commonly used languages for developing embedded systems.



© Courtesy of Professor Prabal Dutta.

The Controller of an Embedded System

7.7 Structures

In C++, you use arrays or vectors to collect values of the same type. If you want to group values of different types together, you use a structured type instead. In the following sections, you will learn how to define structured types and how to work with values of these types.

7.7.1 Structured Types

A structure combines member values into a single value.

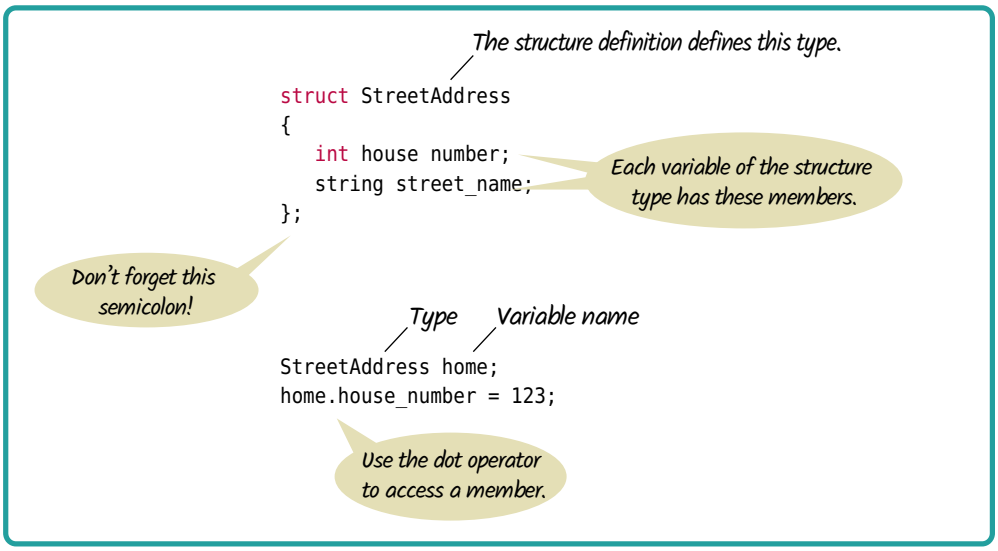
Consider a street address that is composed of a house number and a street name. A **structure** named `StreetAddress` can be defined to combine these two values into a single entity.

In C++, we define a structure with the `struct` reserved word:

```
struct StreetAddress
{
   int house_number;
   string street_name;
};
```

© Joel Carillet/iStockphoto.

Syntax 7.3 Defining a Structure



This definition yields a new type, `StreetAddress`, that you can use for declaring variables:

```
StreetAddress white_house;
```

You use the dot notation to access members of a structure.

The variable `white_house` has two named parts, called *members*, `house_number` and `street_name`. You use the "dot notation" to access each member, like this:

```
white_house.house_number = 1600;
white_house.street_name = "Pennsylvania Avenue";
```

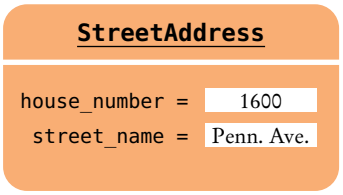


Figure 11 A Structure Value

7.7.2 Structure Assignment and Comparison

When you assign one structure value to another, all members are assigned.

You can use the = operator to assign one structure value to another. All members are assigned simultaneously. For example, if we declare a structure value

```
StreetAddress destination;
```

then the assignment

```
destination = white_house;
```

is equivalent to

```
destination.house_number = white_house.house_number;
destination.street_name = white_house.street_name;
```

However, you cannot compare two structures for equality. That is,

```
if (destination == white_house) // Error
```

is not legal. You need to compare each member separately:

```
if (destination.house_number == white_house.house_number
   && destination.street_name == white_house.street_name) // Ok
```

7.7.3 Functions and Structures

Structures can be function arguments and return values. You simply specify the structure type as the parameter variable or return type, as you would with any other type.

For example, the following function has a parameter variable of type `StreetAddress`:

```
void print_address(StreetAddress address)
{
   cout << address.house_number << " " << address.street_name;
}
```

A function can return a structure. For example, the following function returns a random address on Main Street:

```
StreetAddress make_random_address()
{
   StreetAddress result;
   result.house_number = 100 + rand() % 100;
   result.street_name = "Main Street";
   return result;
}
```

7.7.4 Arrays of Structures

If you need to collect many addresses, you can put them into an array. For example, here is a variable `delivery_route` that is an array of street addresses:

```
StreetAddress delivery_route[ROUTE_LENGTH];
```

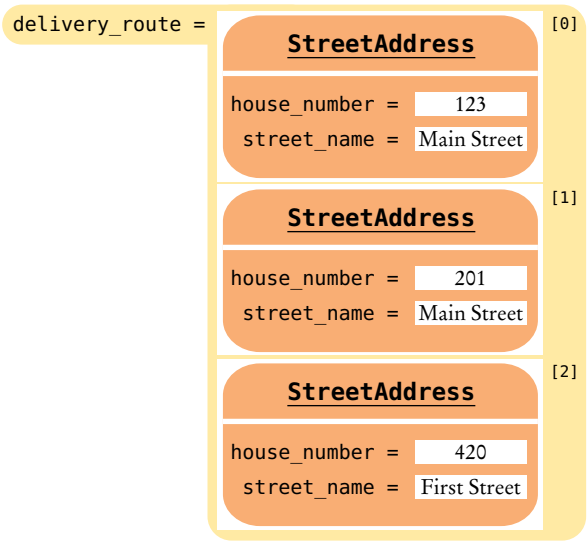


Figure 12
An Array of Structures

To access a member of the structure, you first access the element in the array with the bracket operator, and then use the dot operator:

```
delivery_route[0].house_number = 123;
delivery_route[0].street_name = "Main Street";
```

You can also access a structure value in its entirety, like this:

```
StreetAddress start = delivery_route[0];
```

Of course, you can also form vectors of structures:

```
vector<StreetAddress> tour_destinations;
tour_destinations.push_back(white_house);
```

7.7.5 Structures with Array Members

Structure members can contain arrays. For example, here is a structure that describes monthly temperatures at a given location. The temperature values are stored in an array.

```
struct MonthlyTemperatures
{
   string location;
   double values[12];
};
```

To access an array element, you first select the array member with the dot notation, then use brackets:

```
MonthlyTemperatures death_valley_noon;
death_valley_noon.values[2] = 82;
```

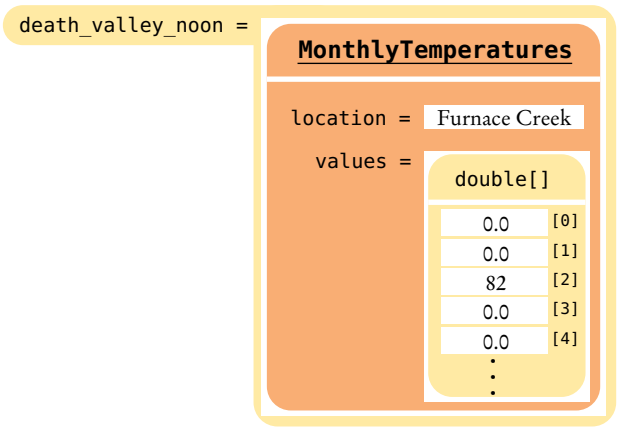


Figure 13 A Structure Containing an Array

7.7.6 Nested Structures

A structure can have a member that is another structure. For example, a person has a name and a street address. Because we already have a structure that describes street addresses, we can use it inside a `Person` structure.

```
struct Person
{
   string name;
   StreetAddress work_address;
};
```

You can access the nested member in its entirety, like this:

```
Person theodore;
theodore.work_address = white_house;
```

Alternatively, if you want to select a member of a member, use the dot operator twice:

```
theodore.work_address.street_name = "Pennsylvania Avenue";
```

A structure can have multiple members that are structures. For example, `Person` can also have a home address that again has type `StreetAddress`:

```
struct Person
{
   string name;
   StreetAddress home_address;
   StreetAddress work_address;
};
```

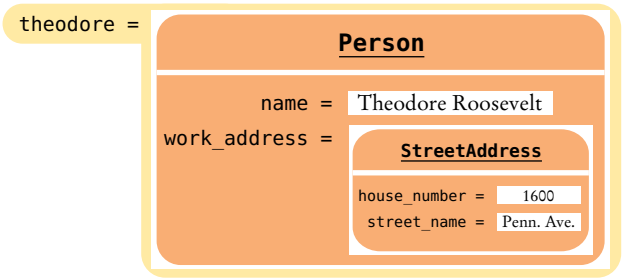


Figure 14 A Structure Containing Another Structure

EXAMPLE CODE See sec07 of your companion code for a program that shows structures in use.

7.8 Pointers and Structures

7.8.1 Pointers to Structures

It is common to allocate structure values dynamically. As with all dynamic allocations, you use the `new` operator:

```
StreetAddress* address_pointer = new StreetAddress;
```

Now suppose you want to set the house number of the structure to which `address_pointer` points. As always when you have a pointer, you can use the `*` operator for

accessing the data to which the pointer points. However, you have to be careful. Consider the following attempt:

```
*address_pointer.house_number = 1600; // Error
```

Unfortunately, that is a syntax error. The dot operator has a higher precedence than the * operator. That is, the compiler thinks that you mean

```
*(address_pointer.house_number) = 1600; // Error
```

However, `address_pointer` is a pointer, not a structure. You can't apply the dot (.) operator to a pointer, and the compiler reports an error. Instead, you must make it clear that you first want to apply the * operator, then the dot:

```
(*address_pointer).house_number = 1600; // OK
```

Use the -> operator to access a structure member through a pointer.

Because this is such a common situation, the designers of C++ supply an operator to abbreviate the "follow pointer and access member" operation. That operator is written -> and usually pronounced "arrow".

```
address_pointer->house_number = 1600; // OK
```

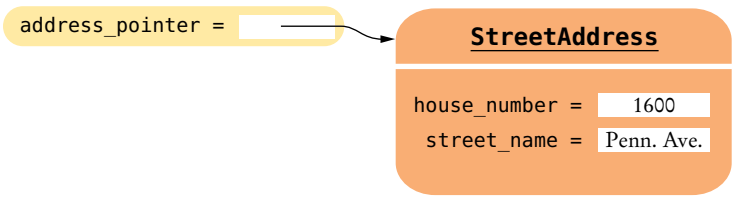


Figure 15 A Pointer to a Structure

7.8.2 Structures with Pointer Members

A member of a structure can be a pointer. This situation commonly arises when information is shared among structure values. Consider this example. In an organization with multiple offices, each employee has a name and an office location:

```
struct Employee
{
   string name;
   StreetAddress* office;
};
```

Here, we define two employees who both work for the accounting office:

```
StreetAddress accounting;
accounting.house_number = 1729;
accounting.street_name = "Park Avenue";

Employee harry;
harry.name = "Smith, Harry";
harry.office = &accounting;
Employee sally;
sally.name = "Lee, Sally";
sally.office = &accounting;
```

Figure 16 shows how these structures are related.

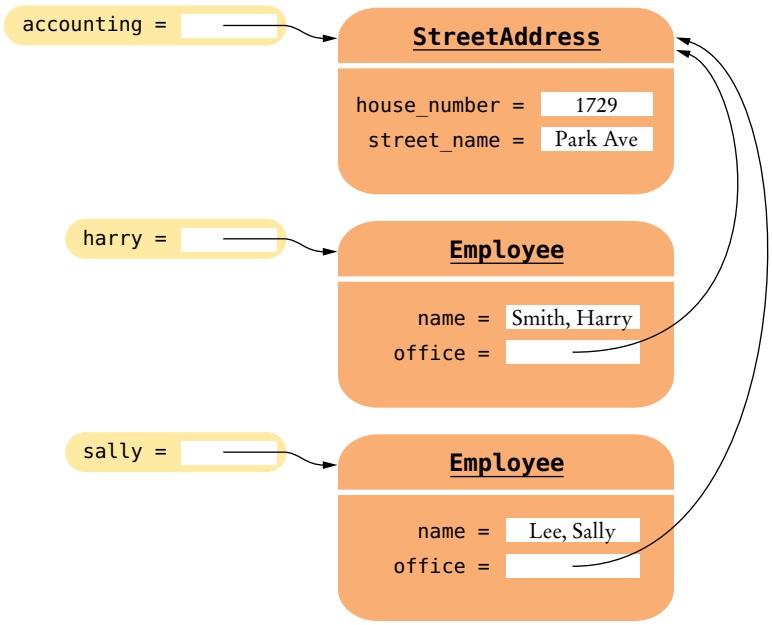


Figure 16 Two Pointers to a Shared Structure

This sharing of information has an important benefit. Suppose the accounting office moves across the street:

```
accounting.house_number = 1720;
```

Now both Harry's and Sally's office addresses are automatically updated.

EXAMPLE CODE See sec08 of your companion code for a program that shows dynamically-allocated structures.

Special Topic 7.5

Smart Pointers

As you saw in Section 7.4, it can be a signifcant burden for programmers to allocate and deallocate memory. C++ 11 introduces a `shared_ptr` type that can automatically reclaim memory that is no longer used. Consider, for example, the `Employee` structure from Section 7.8.2. We can change the `StreetAddress*` pointers to shared pointers:

```
shared_ptr<StreetAddress> accounting(new StreetAddress);
accounting->house_number = 1729;
accounting->street_name = "Park Avenue";
Employee sally;
sally.name = "Lee, Sally";
sally.office = accounting;
```

Now the `StreetAddress` structure describing the accounting office has two shared pointers pointing to it: `accounting` and `sally.office`. When both of these variables go away, then the structure memory is automatically deleted.

We will discuss additional strategies for memory management in C++ in Chapter 13.

EXAMPLE CODE See special_topic_5 of your companion code for a program that uses shared pointers.

CHAPTER SUMMARY

Define and use pointer variables.

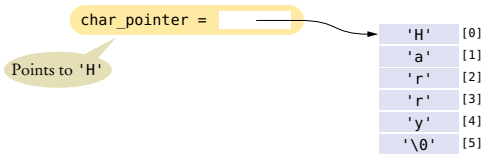
- A pointer denotes the location of a variable in memory.
- The type `T*` denotes a pointer to a variable of type `T`.
- The `&` operator yields the location of a variable.
- The `*` operator accesses the variable to which a pointer points.
- It is an error to use an uninitialized pointer.
- The `nullptr` pointer does not point to any object.

Understand the relationship between arrays and pointers in C++.

- The name of an array variable is a pointer to the starting element of the array.
- Pointer arithmetic means adding an integer offset to an array pointer, yielding a pointer that skips past the given number of elements.
- The array/pointer duality law states that `a[n]` is identical to `*(a + n)`, where `a` is a pointer into an array and `n` is an integer offset.
- When passing an array to a function, only the starting address is passed.

Use C++ string objects with functions that process character arrays.

- A value of type `char` denotes an individual character. Character literals are enclosed in single quotes.
- A literal string (enclosed in double quotes) is an array of `char` values with a zero terminator.
- Many library functions use pointers of type `char*`.
- The `c_str` member function yields a `char*` pointer from a `string` object.
- You can initialize C++ `string` variables with C strings.
- You can access characters in a C++ `string` object with the `[]` operator.



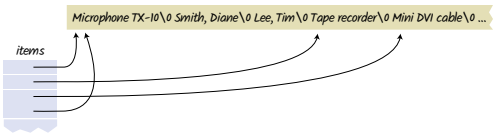
Allocate and deallocate memory in programs whose memory requirements aren't known until run time.

- Use dynamic memory allocation if you do not know in advance how many values you need.
- The `new` operator allocates memory from the free store.
- You must reclaim dynamically allocated objects with the `delete` or `delete[]` operator.
- Using a dangling pointer (a pointer that points to memory that has been deleted) is a serious programming error.
- Every call to `new` should have a matching call to `delete`.

Work with arrays and vectors of pointers.

Draw diagrams for visualizing pointers and the data to which they point.

- Draw the data that is being processed, then draw the pointer variables. When drawing the pointer arrows, illustrate a typical situation.



Use structures to aggregate data items.

- A structure combines member values into a single value.
- You use the dot notation to access members of a structure.
- When you assign one structure value to another, all members are assigned.

Work with pointers to structures.

- Use the -> operator to access a structure member through a pointer.

REVIEW EXERCISES

■ **R7.1** Trace the following code. Assume that `a` and `b` are stored at 20300 and 20308. Your trace table should have entries for `a`, `b`, and `p`.

```
double a = 1000;
double b = 2000;
double* p = &a;
*p = 3000;
p = &b;
a = *p * 2;
```

■ **R7.2** Trace the following code. Assume that `a` and `b` are stored at 20300 and 20308. Your trace table should have entries for `a`, `b`, `p`, and `q`.

```
double a = 1000;
double b = 2000;
double* p = &a;
double* q = &b;
*p = *q;
p = q;
*p = 3000;
```

■ **R7.3** What does the following code print?

```
double a = 1000;
double b = 2000;
double* p = &a;
double* q = p;
b = *q;
p = &b;
a = *p + *q;
cout << a << " " << b << endl;
```

■ **R7.4** Explain the mistakes in the following code. Not all lines contain mistakes. Each line depends on the lines preceding it.

```
1 double a = 1000;
2 double* p = a;
3 int* p = &a;
4 double* q;
5 *q = 2000;
6 int* r = nullptr;
7 *r = 3000;
```

■■ **R7.5** Suppose that a system allows the use of any string as a password, even the empty string. However, when a user connects to the system for the first time, no password has been assigned. Describe how you can use a `string*` variable and the `nullptr` pointer to distinguish unassigned passwords from empty ones.

■■ **R7.6** Given the definitions

```
double primes[] = { 2, 3, 5, 7, 11, 13 };
double* p = primes + 3;
```

draw a diagram that explains the meanings of the following expressions:

a. `primes[1]`

b. `primes + 1`

c. `*(primes + 1)`

d. `p[1]`

e. `p + 1`

■ **R7.7** Suppose the array `primes`, defined as

```
double primes[] = { 2, 3, 5, 7, 11, 13 };
```

starts at memory location 20300. What are the values of

a. `primes`
b. `*primes`
c. `primes + 4`
d. `*(primes + 4)`
e. `primes[4]`
f. `&primes[4]`

■ **R7.8** Suppose the array `primes` is defined as

```
double primes[] = { 2, 3, 5, 7, 11, 13 };
```

Consider the `sum` function discussed in Section 7.2.3. What are the values of

a. `sum(primes, 6);`
b. `sum(primes, 4);`
c. `sum(primes + 2, 4);`
d. `sum(primes, 0);`
e. `sum(nullptr, 4);`

■■ **R7.9** Suppose the array `primes`, defined as

```
double primes[] = { 2, 3, 5, 7, 11, 13 };
```

starts at memory location 20300. Trace the function call `sum(primes, 4)`, using the definition of `sum` from Special Topic 7.2. In your trace table, show the values for `a`, `size`, `total`, `p`, and `i`.

■■■ **R7.10** Pointers are addresses and have a numerical value. You can print out the value of a pointer as `cout << (unsigned)(p)`. Write a program to compare `p`, `p + 1`, `q`, and `q + 1`, where `p` is an `int*` and `q` is a `double*`. Explain the results.

■■ **R7.11** A pointer variable can contain a pointer to a single variable, a pointer to an array, `nullptr`, or a random value. Write code that creates and sets four pointer variables `a`, `b`, `c`, and `d` to show each of these possibilities.

■■ **R7.12** Implement a function `firstlast` that obtains the first and last values in an array of integers and stores the result in an array *parameter*.

■ **R7.13** Explain the meanings of the following expressions:

a. `"Harry" + 1`
b. `*("Harry" + 2)`
c. `"Harry"[3]`
d. `[4]"Harry"`

■ **R7.14** What is the difference between the following two variable definitions?

a. `char a[6] = "Hello";`
b. `char* b = "Hello";`

■ **R7.15** What is the difference between the following three variable definitions?

a. `char* p = nullptr;`
b. `char* q = "";`
c. `char r[1] = { '\0' };`

■■ **R7.16** Consider this program segment:

```
char a[] = "Mary had a little lamb";
char* p = a;
int count = 0;
while (*p != '\0')
{
   count++;
   while (*p != ' ' && *p != '\0') { p++; }
   while (*p == ' ') { p++; }
}
```

What is the value of `count` at the end of the outer `while` loop?

■■ **R7.17** Consider the following code that repeats a C++ string three times.

```
string a = "Hello";
string b = a + a + a;
```

Suppose `s` is a C string, and `t` is declared as

```
char t[100];
```

Write the equivalent code for C strings that stores the threefold repetition of `s` (or as much of it as will fit) into `t`.

■ **R7.18** Which of the following assignments are legal in C++?

```
void f(int p[])
{
   int* q;
   const int* r;
   int s[10];
   p = q;  // 1
   p = r;  // 2
   p = s;  // 3
   q = p;  // 4
   q = r;  // 5
   q = s;  // 6
   r = p;  // 7
   r = q;  // 8
   r = s;  // 9
   s = p;  // 10
   s = q;  // 11
   s = r;  // 12
}
```

■ **R7.19** What happens if you forget to delete an object that you obtained from the free store? What happens if you delete it twice?

■ **R7.20** Write a program that accesses a deleted pointer, an uninitialized pointer, and a `nullptr`. What happens when you run your program?

■■ **R7.21** Find the mistakes in the following code. Not all lines contain mistakes. Each line depends on the lines preceding it. Watch out for uninitialized pointers, `nullptr` pointers, pointers to deleted objects, and confusing pointers with objects.

```
1 int* p = new int;
2 p = 5;
3 *p = *p + 5;
4 string s = "Harry";
5 *s = "Sally";
6 delete &s;
```

```
 7  int* a = new int[10];
 8  *a = 5;
 9  a[10] = 5;
10  delete a;
11  int* q;
12  *q = 5;
13  q = p;
14  delete q;
15  delete p;
```

■■ **R7.22** How do you define a triangular two-dimensional array using just vectors, not arrays or pointers?

■ **R7.23** Rewrite the statements in Section 7.8.2 so that the street address and employee structures are allocated on the free store.

■ **R7.24** What is a key difference between a structure and an array?

■■ **R7.25** For each of the following, should you use an array of structures, a structure containing one or more arrays, or a structure containing another structure?

a. A course with multiple students

b. An assignment with a due date

c. A student with grades for each assignment

d. An instructor with multiple courses

■■■ **R7.26** How could you implement the delivery route from Section 7.7.4 as a structure containing arrays? Which implementation is better? Why?

■■ **R7.27** Design a structure type `Person` that contains the name of a person and pointers to the person's father and mother. Write statements that define a structure value for yourself and your parents, correctly establishing the pointer links. (Use `nullptr` for your parents' parents.)

■ **R7.28** Assuming the following declarations, write C++ expressions to set all integer values of `b`'s members to 42.

```
struct A { int m1[2]; }
struct B { A a1; A a2; int m2; }
B b;
```

■ **R7.29** Assuming the following declarations, write C++ expressions to set all integer values of `b` to 42.

```
struct A { int m1[2]; }
struct B { A a1; A* a2; int m2; }
B* b = new B;
```

■ **R7.30** Assuming the following declarations, find the mistakes in the statements below.

```
struct A { int m1[2]; int m2; }
struct B { A a1; A* a2; int m3; }
A a;
B b[2];
```

a. `b[0].a2 = a;`

b. `b[1].m2 = 42;`

c. `b[2].a2 = new A;`

d. `b[0].a1.m1 = 42;`

e. `b[1].a2->m2 = 42;`

■■■ **R7.31** Draw a figure showing the result of a call to the `maximum` function in Exercise E7.4.

■■■ **R7.32** Draw a figure showing the pointers in the `lines` array in Exercise P7.3 after reading a few lines.

■■ **R7.33** Section 7.6 described an arrangement where each item had a pointer to the user who had checked out the item. This makes it difficult to find out the items that a particular user checked out. To solve this problem, have an array of strings `user_names` and a parallel array `loaned_items`. `loaned_items[i]` points to an array of `char*` pointers, each of which is a name of an item that the `i`th user checked out. If the `i`th user didn't check out any items, then `loaned_items[i]` is `nullptr`. Draw a picture of this arrangement.

PRACTICE EXERCISES

■ **E7.1** Write a function

```
void sort2(double* p, double* q)
```

that receives two pointers and sorts the values to which they point. If you call

```
sort2(&x, &y)
```

then `x <= y` after the call.

■ **E7.2** Write a function

```
double replace_if_greater(double* p, double x)
```

that replaces the value to which `p` points with `x` if `x` is greater. Return the old value to which `p` pointed.

■ **E7.3** Write a function that computes the average value of an array of floating-point data:

```
double average(double* a, int size)
```

In the function, use a pointer variable, not an integer index, to traverse the array elements.

■ **E7.4** Write a function that returns a pointer to the maximum value of an array of floating-point data:

```
double* maximum(double* a, int size)
```

If `size` is 0, return `nullptr`.

■ **E7.5** Write a function that returns a pointer to the first occurrence of the character `c` in the string `s`, or `nullptr` if there is no match.

```
char* find(char s[], char c)
```

■ **E7.6** Write a function that returns a pointer to the last occurrence of the character `c` in the string `s`, or `nullptr` if there is no match.

```
char* find_last(char s[], char c)
```

■■ **E7.7** Write a function that returns a pointer to the `n`th occurrence of the character `c` in the string `s`, or `nullptr` if there is no match.

```
char* find_last(char s[], char c, int n)
```

■■ **E7.8** Write a function that returns a pointer to the first occurrence of the substring `t` in the string `s`, or `nullptr` if there is no match.

```
char* find(char s[], char t[])
```

■■ **E7.9** Write a function that, given strings `s`, `t`, and `u`, returns a string (allocated with the `new` operator) in which all occurrences of `t` in `s` are replaced with `u`:

```
char* replace_all(const char s[], const char t[], const char u[])
```

■■ **E7.10** Write a function that reverses the values of an array of floating-point data:

```
void reverse(double* a, int size)
```

In the function, use two pointer variables, not integer indexes, to traverse the array elements.

■■ **E7.11** Implement the `strncpy` function of the standard library.

■■■ **E7.12** Implement the standard library function

```
int strspn(const char s[], const char t[])
```

that returns the length of the initial portion of `s` consisting only of characters contained in `t` (in any order). For example, if all of the characters in `s` are in `t`, the function returns the length of `s`; if the first character in `s` is not in `t`, the function returns zero.

■■ **E7.13** Write a function

```
void reverse(char s[])
```

that reverses a character string. For example, `"Harry"` becomes `"yrraH"`.

■■■ **E7.14** Using the `strncpy` and `strncat` functions, implement a function

```
void safe_concat(const char a[], const char b[], char result[],
   int result_maxlength)
```

that concatenates the strings `a` and `b` to the buffer `result`. Be sure not to overrun the buffer. It can hold `result_maxlength` characters, not counting the `'\0'` terminator. (That is, the buffer has `result_maxlength + 1` bytes available.)

■■■ **E7.15** Write a function `int* read_data(int& size)` that reads data from `cin` until the user terminates input by entering `Q`. The function should set the `size` reference parameter to the number of numeric inputs. Return a pointer to an array on the free store. That array should have exactly `size` elements. Of course, you won't know at the outset how many elements the user will enter. Start with an array of 10 elements, and double the size whenever the array fills up. At the end, allocate an array of the correct size and copy all inputs into it. Be sure to delete any intermediate arrays.

■ **E7.16** Define a structure `Point`. A point has an *x*- and a *y*-coordinate. Write a function `double distance(Point a, Point b)` that computes the distance from `a` to `b`. Write a program that reads the coordinates of the points, calls your function, and displays the result.

■ **E7.17** Using the `Point` structure from Exercise E7.16, write a function `Point midpoint(Point a, Point b)` that computes the point that is halfway between `a` and `b`. Write a program that reads the coordinates of the points, calls your function, and displays the result.

■■ **E7.18** Define a structure `Triangle` that contains three `Point` members. Write a function that computes the perimeter of a `Triangle`. Write a program that reads the coordinates of the points, calls your function, and displays the result.

■■ **E7.19** Reimplement the `Triangle` structure and the perimeter function of Exercise E7.18 so that it contains two arrays of three `double` values each, one for the *x*-coordinates and one for the *y*-coordinates.

■ **E7.20** Define an `Employee` structure with a name and an `Employee*` pointer to the employee's manager. For the CEO, that pointer will be a `nullptr`. Write a program that defines several employees and their managers. For each employee, print the chain of superiors.

PROGRAMMING PROJECTS

■■ **P7.1** Enhance the Galton board simulation by printing a bar chart of the bottommost counters. Draw the bars vertically, below the last row of numbers.

```
 1  18  64 149 239 265 186  61  15   2
    *   *   *   *   *   *   *   *
        *   *   *   *   *   *
        *   *   *   *   *   *
            *   *   *   *
            *   *   *   *
            *   *   *   *
            *   *   *   *
                *   *   *
                *   *   *
                *   *
                *   *
                *   *
                    *
```

■■■ **P7.2** Complete the copier simulation of How To 7.1. Your program should first show the main menu:

```
U)ser A)dministrator Q)uit
```

For a user, prompt for the ID and the number of copies, increment the appropriate account, and return to the main menu.

For an administrator, show this menu:

```
B)alance M)aster P)roject
```

In the balance option, show the balances of the master account and the ten project accounts. In the master option, prompt for a user ID and link it to the master account. In the project option, prompt for user and project IDs. Afterward, return to the main menu.

■■ **P7.3** Write a program that reads lines of text and appends them to a `char buffer[1000]`. Read one character at a time by calling `cin.get(ch)`, where `ch` is a variable of type `char`. Use input redirection (Special Topic 4.3). Stop after reading 1,000 characters. As you read in the text, replace all newline characters `'\n'` with `'\0'` terminators. Establish an array `char* lines[100]`, so that the pointers in that array point to the beginnings of the lines in the text. Consider only 100 input lines if the input has more lines. Then display the lines in reverse order, starting with the last input line.

■■■ **P7.4** The program in Exercise P7.3 is limited by the fact that it can only handle inputs of 1,000 characters or 100 lines. Remove this limitation as follows. Concatenate the input in one long `string` object. Use the `c_str` member function to obtain a `char*` into the string's character buffer. Store the beginnings of the lines as a `vector<char*>`.

■■ **P7.5** Exercise P7.4 demonstrated how to use the `string` and `vector` classes to implement resizable arrays. In this exercise, you should implement that capability manually. Allocate a buffer of 1,000 characters from the free store (`new char[1000]`). Whenever

the buffer fills up, allocate a buffer of twice the size, copy the buffer contents, and delete the old buffer. Do the same for the array of `char*` pointers—start with a `new char*[100]` and keep doubling the size.

■■■ **P7.6** Modify Exercise P7.3 so that you first print the lines in the order that they were entered, then print them in sorted order. When you sort the lines, only rearrange the pointers in the `lines` array.

■■■ **P7.7** When you read a long document, there is a good chance that many words occur multiple times. Instead of storing each word, it may be beneficial to only store unique words, and to represent the document as a vector of pointers to the unique words. Write a program that implements this strategy. Read a word at a time from `cin`. Keep a `vector<char*>` of words. If the new word is not contained in this vector, allocate memory, copy the word into it, and append a pointer to the new memory. If the word is already present, then append a pointer to the existing word.

■ **P7.8** Define a structure `Student` with a first name, last name, and course grade (A, B, C, D, or F). Write a program that reads input in which each line has the first and last name and course grade, separated by spaces. Upon reading the input, your program should print all students with grade A, then all students with grade B, and so on.

■■ **P7.9** Enhance the program in Exercise P7.8 so that each student has ten quiz scores. The input contains the student names and quiz scores but no course grades. The program should compute the course grade. If the sum of the quiz scores is at least 90, the grade is an A. If the sum is at least 80, the grade is a B, and so on. Then print all students with grade A together with their individual quiz scores, followed by all students with grade B, and so on.

■■■ **P7.10** Define a structure `Student` with a name and a `vector<Course*>` of courses. Define a structure `Course` with a name and a `vector<Student*>` of enrolled students. Then define these functions:

- `void print_student(Student* s)` that prints the name of a student and the names of all courses that the student takes.
- `void print_course(Course* c)` that prints the name of a course and the names of all students in that course.
- `void enroll(Student* s, Course* c)` that enrolls the given student in the given course, updating both vectors.

In your `main` function, define several students and courses, and enroll students in the courses. Then call `print_student` for all students and `print_course` for all courses.

■■ **Engineering P7.11** Write a program that simulates a device that gathers measurements and processes them.

A `gather` function gathers data values (which you should simulate with random integers between 0 and 100) and places them in an array. When the array is full, the `gather` function calls a function `new_array` to request a new array to fill.

A `process` function processes a data value (for this exercise, it simply updates global variables for computing the maximum, minimum, and average). When it has reached the end of the array, it calls a function `next_array` to request a new array to process.

Because the `gather` function may fill arrays faster than the `process` function processes them, store the pointers to the filled arrays in another array. The `new_array` and `next_array` functions need to maintain that array of pointers.

In a real device, data gathering and processing happen in parallel. Simulate this by calling the `gather` and `process` functions randomly from `main`.

■■■ **Engineering P7.12** Write a program that simulates the control software for a "people mover" system, a set of driverless trains that move in two concentric circular tracks. A set of switches allows trains to switch tracks.

© TexPhoto/iStockphoto.

In your program, the outer and inner tracks should each be divided into ten segments. Each track segment can contain a train that moves either clockwise or counterclockwise. A train moves to an adjacent segment in its track or, if that segment is occupied, to the adjacent segment in the other track.

Define a `Segment` structure. Each segment has a pointer to the next and previous segments in its track, a pointer to the next and previous segments in the other track, and a train indicator that is `0` (empty), +1 (train moving clockwise), or –1 (train moving counterclockwise). Populate the system with four trains at random segments, two in each direction. Display the tracks and trains in each step, like this:

```
+-------------->--+
|  x  x  x  x  x  |
| ----------<---- |
| |            | |
| ->--<---------- |
|  x  x  x  x  x  |
+-----------------+
```

The two rectangles indicate the tracks. Each switch that allows a train to switch between the outer and inner track is indicated by an `x`. Each train is drawn as a `>` or `<`, indicating its current direction. Your program should show fifty rounds. In each round, all trains move once.

WORKED EXAMPLE 7.1

Producing a Mass Mailing

Problem Statement We want to automate the process of producing mass mailings. A typical letter might look as follows:

```
To: Ms. Sally Smith
123 Main Street
Anytown, NY 12345

Dear Ms. Smith:

The Smith family may be the lucky winner in the C++ sweepstakes.
Wouldn't it be exciting if you were the first Anytown residents
to use ACME's new C++ development environment? etc. etc.
```

Another letter with the same template but different values for the variable parts would look like this:

```
To: Mr. Harry Morgan
456 Park Ave
Everyville, KS 67890

Dear Mr. Morgan:

The Morgan family may be the lucky winner in the C++ sweepstakes.
Wouldn't it be exciting if you were the first Everyville residents
to use ACME's new C++ development environment? etc. etc.
```

To set up such a mailing, we start with a template string in which each of the variable parts is denoted by a digit 0 ... 9. (We assume that digits are not otherwise used in the template text.)

```
To: 0 1 2
3
4, 5 6

Dear 0 2:

The 2 family may be the lucky winner in the C++ sweepstakes.
Wouldn't it be exciting if you were the first 4 residents
to use ACME's new C++ development environment? etc. etc.
```

This template is set up in a string, like this:

```
char letter_template[] = "To: 0 1 2\n3\n4, 5 6\n\nDear 0 2: \n\n"
   "The 2 family may be the lucky winner in the C++ sweepstakes.\n"
   "Wouldn't it be exciting if you were the first 4 residents\n"
   "to use ACME's new C++ development environment? etc. etc. \n\n\n";
```

(To declare a literal string in C++ that does not fit on a single line, you write a sequence of literal strings. The compiler combines them into a single string.)

Of course, we could produce a separate string for each mailing, by replacing the digits with the values for a particular recipient. But there is a more efficient way. Your task will be to set up a vector of character pointers that point to successive fragments of the letter, either from the template string or the variable parts. For example, the body of the letter above is represented by pointers to the following strings:

- `": \n\nThe "`
- Variable 2
- `" family may be . . . were the first "`

- Variable 4
- `" residents . . . "`

The variable parts are stored in a two-dimensional array of characters:

```
char variable_parts[10][VAR_LENGTH];
```

Each row of this array contains a variable:

```
strcpy(variable_parts[0], "Ms.");
strcpy(variable_parts[1], "Sally");
strcpy(variable_parts[2], "Smith");
strcpy(variable_parts[3], "123 Main Street");
strcpy(variable_parts[4], "Anytown");
strcpy(variable_parts[5], "NY");
strcpy(variable_parts[6], "12345");
```

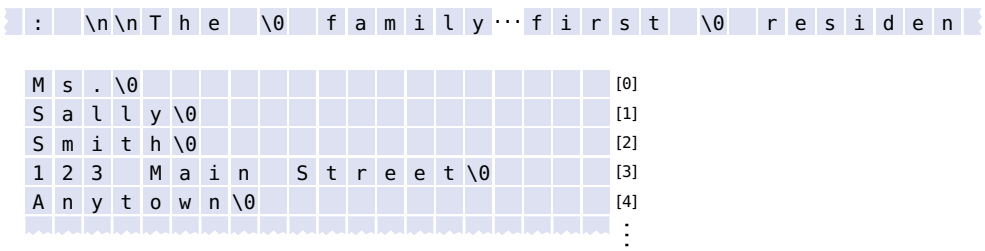
Printing a letter is achieved with the following loop:

```
vector<char*> fragments = prepare_mailing(letter_template, variable_parts);

for (int i = 0; i < fragments.size(); i++)
{
   cout << fragments[i];
}
```

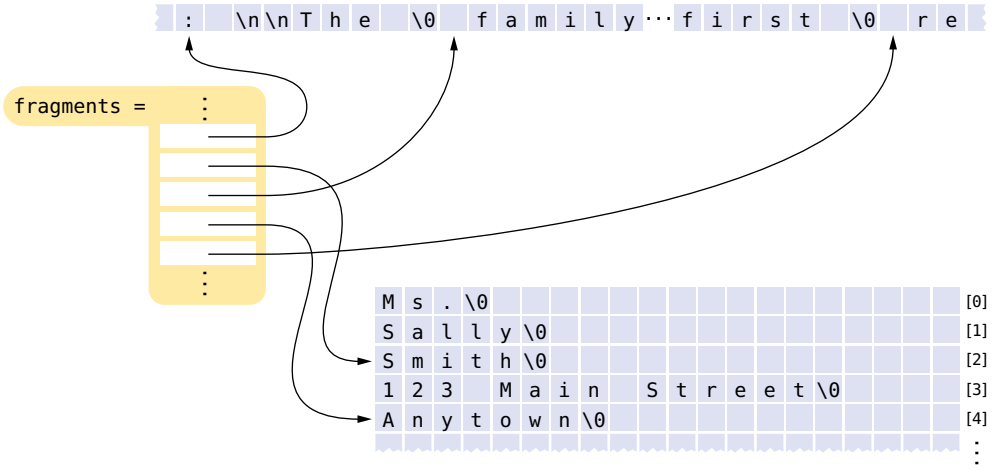
Your task will be to implement the `prepare_mailing` function, and to produce a `main` function that prints the two letters given above.

Step 1 Draw a picture.

We start with the data that is being accessed through pointers: the strings in the letter template (after replacing the variable placeholders with null terminators), and the strings in the variable array.



The pointers point into the template and variable strings, like this:



Step 2 Declare pointer variables.

We collect a sequence of `char*` pointers of unknown length. Therefore, we use a vector of pointers:

```
vector<char*> fragments;
```

Step 3 Initialize the pointers with variable addresses or free store memory.

This is the task of the `prepare_mailing` function. That function needs to scan the letter template for variable indicators, replace them with null terminators, and store pointers in the `fragments` vector. Here is the pseudocode:

```
Set the fragment start to the beginning of the letter template.
For each character in the letter template
   If the character is a digit
      Add the fragment start to the fragments vector.
      Add the row of the variable denoted by the digit to the fragments vector.
      Replace the digit with a null terminator.
      Set the fragment start to the following character.
```

You will find the C++ code in the code listing below.

Step 4 Use the `*` or `[]` operator to access your data.

This step has already been completed in the problem statement. It was a requirement that the fragments can be printed with the following loop:

```
for (int i = 0; i < fragments.size(); i++)
{
   cout << fragments[i];
}
```

All that remains is to complete a `main` program that prints two letters, as shown in the code listing. Note that the `prepare_mailing` function is only called once. Afterward, the variable array is updated for each mailing, but the `fragments` vector never changes.

Here is the complete program:

worked_example_1/mail.cpp

```
#include <cctype>
#include <iostream>
#include <string>
#include <vector>

using namespace std;

const int VAR_LENGTH = 20;

/**
   Prepares a letter template for a mass mailing.
   @param letter a string with placeholders 0 ... 9 (which will be replaced
   with null terminators)
   @param vars a two-dimensional array for holding the variable parts of the letter
   @return a vector of fragments, pointing alternately to strings in the letter
   template and in the variable array
 */
vector<char*> prepare_mailing(char* letter, char vars[][VAR_LENGTH])
{
   vector<char*> fragments;
   char* fragment_start = letter;
   for (char* p = letter; *p != '\0'; p++)
   {
```

```
      if (isdigit(*p))
      {
         fragments.push_back(fragment_start);
         fragment_start = p + 1;
         int var_index = *p - '0';
         fragments.push_back(vars[var_index]);
         *p = '\0';
      }
   }
   fragments.push_back(fragment_start);
   return fragments;
}


int main()
{
   char variable_parts[10][VAR_LENGTH];

   strcpy(variable_parts[0], "Ms.");
   strcpy(variable_parts[1], "Sally");
   strcpy(variable_parts[2], "Smith");
   strcpy(variable_parts[3], "123 Main Street");
   strcpy(variable_parts[4], "Anytown");
   strcpy(variable_parts[5], "NY");
   strcpy(variable_parts[6], "12345");

   char letter_template[] = "To: 0 1 2\n3\n4, 5 6\n\nDear 0 2: \n\n"
      "The 2 family may be the lucky winner in the C++ sweepstakes.\n"
      "Wouldn't it be exciting if you were the first 4 residents\n"
      "to use ACME's new C++ development environment? etc. etc. \n\n\n";

   vector<char*> fragments = prepare_mailing(letter_template, variable_parts);

   for (int i = 0; i < fragments.size(); i++)
   {
      cout << fragments[i];
   }

   strcpy(variable_parts[0], "Mr.");
   strcpy(variable_parts[1], "Harry");
   strcpy(variable_parts[2], "Morgan");
   strcpy(variable_parts[3], "456 Park Ave");
   strcpy(variable_parts[4], "Everyville");
   strcpy(variable_parts[5], "KS");
   strcpy(variable_parts[6], "67890");

   for (int i = 0; i < fragments.size(); i++)
   {
      cout << fragments[i];
   }

   return 0;
}
```

CHAPTER 8

STREAMS

James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

CHAPTER GOALS

To be able to read and write files

To convert between strings and numbers using string streams

To process command line arguments

To understand the concepts of sequential and random access

CHAPTER CONTENTS

In this chapter, you will learn how to read and write files using the C++ stream library—a very useful skill for processing real world data. As an application, you will learn how to encrypt data. (The Enigma machine shown in the photo is an encryption device used by Germany in World War II. Pioneering British computer scientists broke the code and were able to intercept encoded messages, which was a significant help in winning the war.) Later in the chapter, you will learn to process binary files, such as those that store image data.

8.1 Reading and Writing Text Files

The C++ input/output library is based on the concept of **streams**. An **input stream** is a source of data, and an **output stream** is a destination for data. The most common sources and destinations for data are the files on your hard disk.

To read or write files, you use variables of type `fstream`, `ifstream`, or `ofstream`.

To access a file, you use a file stream. There are three types of file streams: `ifstream` (for input), `ofstream` (for output), and `fstream` (for both input and output). Include the `<fstream>` header when you use any of these file streams.

In the following sections, you will learn how to process data from files. File processing is a very useful skill in many disciplines because it is exceedingly common to analyze large data sets stored in files.

© nullplus/iStockphoto.

Data arrive in an input stream just like items on a conveyor belt, one at a time.

8.1.1 Opening a Stream

When opening a file stream, you supply the name of the file stored on disk.

To read anything from a file stream, you need to *open* it. When you open a stream, you give the name of the file stored on disk. Suppose you want to read data from a file named `input.dat`, located in the same directory as the program. Then you use the following function call to open the file:

```
in_file.open("input.dat");
```

This statement associates the variable `in_file` with the file named `input.dat`.

Note that all streams are objects, and you use the dot notation for calling functions that manipulate them.

To open a file for writing, you use an `ofstream` variable. To open the same file for both reading and writing, you use an `fstream` variable.

File names can contain directory path information, such as

```
~/homework/input.dat (UNIX)
c:\homework\input.dat (Windows)
```

When you specify the file name as a string literal, and the name contains backslash characters (as in a Windows filename), you must supply each backslash *twice*:

```
in_file.open("c:\\homework\\input.dat");
```

Recall that a single backslash inside a string literal is an **escape character** that is combined with another character to form a special meaning, such as `\n` for a newline character. The `\\` combination denotes a single backslash.

In older versions of C++, you must be careful when opening a file with a name that is stored in a string variable. Use the `c_str` function to convert the C++ `string` to a C string:

```
cout << "Please enter the file name:";
string filename;
cin >> filename;
ifstream in_file;
in_file.open(filename.c_str()); // Before C++ 11
```

If it is not possible to open a stream, then the stream variable is set to a failed state. You can test for that condition:

```
in_file.open("input.dat");
if (in_file.fail())
{
   cout << "Cannot read from input.dat" << endl;
}
else
{
   Read input.
}
```

When the program ends, all streams that you have opened will be automatically closed. You can also manually close a stream with the `close` member function:

```
in_file.close();
```

Manual closing is only necessary if you want to use the stream variable again to process another file.

8.1.2 Reading from a File

Read from a file stream with the same operations that you use with `cin`.

Reading data from a file stream is completely straightforward: You simply use the same functions that you have always used for reading from `cin`:

```
string name;
double value;
in_file >> name >> value;
```

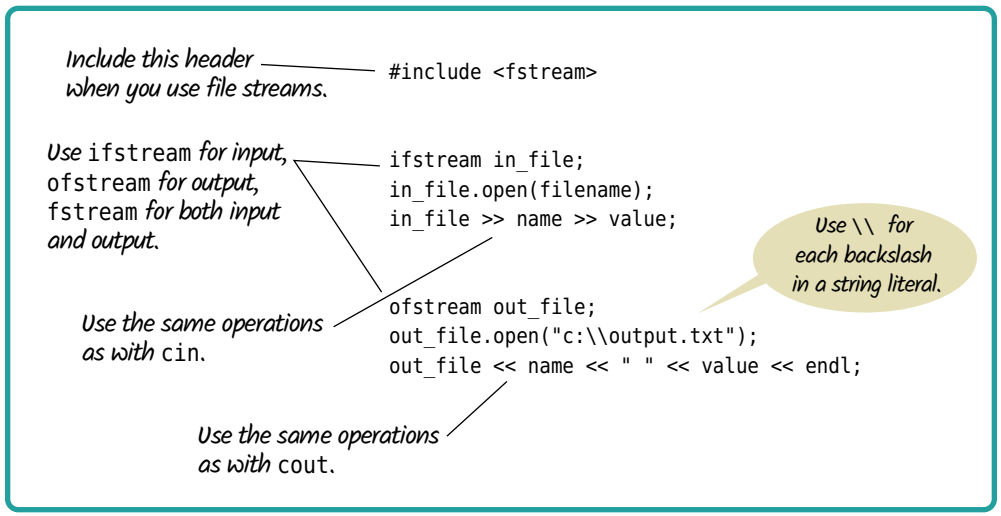
The `fail` function tells you whether input has failed. You have already used this function with `cin`, to check for errors in console input. File streams behave in the same way. When you try to read a number from a file, and the next data item is not a properly formatted number, then the stream fails. After reading data, you should test for success before processing:

```
if (!in_file.fail())
{
   Process input.
}
```

When you read input from a file, number format errors are not the only reason for failure. Suppose you have consumed all of the data contained in a file and try to read

more items. A file stream enters the failed state, whereas `cin` would just wait for more user input. Moreover, if you open a file and the name is invalid, or if there is no file of that name, then the file stream is also in a failed state. It is a good idea to test for failure immediately after calling `open`.

Syntax 8.1 Working with File Streams



8.1.3 Writing to a File

Write to a file stream with the same operations that you use with `cout`.

In order to write to a file, you define an `ofstream` or `fstream` variable and open it. Then you send information to the output file, using the same operations that you used with `cout`:

```
ofstream out_file;
out_file.open("output.txt");
out_file << name << " " << value << endl;
```

8.1.4 A File Processing Example

Here is a typical example of processing data in a file. The Social Security Administration publishes lists of the most popular baby names on their web site, `http://www.ssa.gov/OACT/babynames/`. If you query the 1,000 most popular names for a given decade, the browser displays the result on the screen (see Figure 1).

To save the data as text, simply select it and paste the result into a file. This book's companion code contains a file called `babynames.txt` with the data for the 1990s.

Each line in the file contains seven entries:

- The rank (from 1 to 1,000)
- The name, frequency, and percentage of the male name of that rank
- The name, frequency, and percentage of the female name of that rank

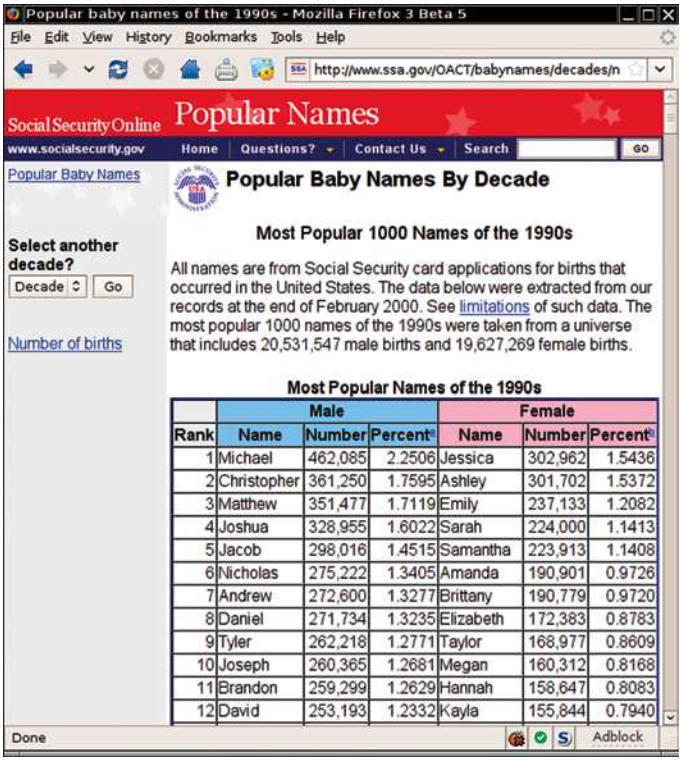


Figure 1 Querying Baby Names

For example, the line

```
10  Joseph  260365  1.2681  Megan  160312  0.8168
```

shows that the 10th most common boy's name was Joseph, with 260,365 births, or 1.2681 percent of all births during that period. The 10th most common girl's name was Megan. Why are there many more Josephs than Megans? Parents seem to use a wider set of girl's names, making each one of them less frequent.

Let us test that conjecture, by determining the names given to the top 50 percent of boys and girls in the list.

To process each line, we first read the rank:

```
int rank;
in_file >> rank;
```

We then read a set of three values for the boy's name:

```
string name;
int count;
double percent;
in_file >> name >> count >> percent;
```

Then we repeat that step for girls. Because the actions are identical, we supply a helper function `process_name` for that purpose. To stop

© Nancy Ross/iStockphoto.

Sellers of personalized items can find trends in popular names by processing data files from the Social Security Administration.

processing after reaching 50 percent, we can add up the frequencies and stop when they reach 50 percent. However, it turns out to be a bit simpler to initialize a total with 50 and subtract the frequencies. We need separate totals for boys and girls. When a total falls below 0, we stop printing. When both totals fall below 0, we stop reading.

Always use a reference parameter for a stream.

Note that the `in_file` parameter variable of the `process_name` function in the code below is a reference parameter. Reading or writing modifies a stream variable. The stream variable monitors how many characters have been read or written so far. Any read or write operation changes that data. For that reason, you must always make stream parameter variables reference parameters.

The complete program is shown below. As you can see, reading from a file is just as easy as reading keyboard input.

Have a look at the program output. Remarkably, only 69 boy names and 153 girl names account for half of all births. That's good news for those who are in the business of producing personalized doodads. Exercise P8.3 asks you to study how this distribution has changed over the years.

sec01/babynames.cpp

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

/**
   Reads name information, prints the name if total >= 0, and adjusts the total.
   @param in_file the input stream
   @param total the total percentage that should still be processed
*/
void process_name(ifstream& in_file, double& total)
{
   string name;
   int count;
   double percent;
   in_file >> name >> count >> percent;

   if (in_file.fail()) { return; } // Check for failure after each input
   if (total > 0) { cout << name << " "; }
   total = total - percent;
}

int main()
{
   ifstream in_file;
   in_file.open("babynames.txt");
   if (in_file.fail()) { return 0; } // Check for failure after opening

   double boy_total = 50;
   double girl_total = 50;

   while (boy_total > 0 || girl_total > 0)
   {
      int rank;
      in_file >> rank;
      if (in_file.fail()) { return 0; }

      cout << rank << " ";
```

```
40
41       process_name(in_file, boy_total);
42       process_name(in_file, girl_total);
43
44       cout << endl;
45    }
46
47    return 0;
48 }
```

Program Run

```
 1 Michael Jessica
 2 Christopher Ashley
 3 Matthew Emily
 4 Joshua Sarah
 5 Jacob Samantha
 6 Nicholas Amanda
 7 Andrew Brittany
 8 Daniel Elizabeth
 9 Tyler Taylor
10 Joseph Megan
. . .
68 Dustin Gabrielle
69 Noah Katie
70 Caitlin
71 Lindsey
. . .
150 Hayley
151 Rebekah
152 Jocelyn
153 Cassidy
```

8.2 Reading Text Input

In the following sections, you will learn how to process text with complex contents such as that which often occurs in real-life situations.

8.2.1 Reading Words

You already know how to read the next word from a stream, using the >> operator.

```
string word;
in_file >> word;
```

When reading a string with the >> operator, the white space between words is consumed.

Here is precisely what happens when that operation is executed. First, any input characters that are white space are removed from the stream, but they are not added to the word. White space includes spaces, tab characters, and the newline characters that separate lines. The first character that is not white space becomes the first character in the string `word`. More characters are added until either another white space character occurs, or the end of the file has been reached. The white space after the word is not removed from the stream.

8.2.2 Reading Characters

Instead of reading an entire word, you can read one character at a time by calling the `get` function:

```
char ch;
in_file.get(ch);
```

The `get` function returns the "not failed" condition. The following loop processes all characters in a file:

```
while (in_file.get(ch))
{
   Process the character ch.
}
```

The `get` function reads white space characters. This is useful if you need to process characters such as spaces, tabs, or newlines. On the other hand, if you are not interested in white space, use the `>>` operator instead.

```
in_file >> ch; // ch is set to the next non-white space character
```

You can get individual characters from a stream and unget the last one.

If you read a character and you regretted it, you can *unget* it, so that the next input operation can read it again. However, you can unget only the last character. This is called *one-character lookahead.* You get a chance to look at the next character in the input stream, and you can make a decision whether you want to consume it or put it back.

A typical situation for lookahead is to look for numbers:

```
char ch;
in_file.get(ch);
if (isdigit(ch))
{
   in_file.unget(); // Put the digit back so that it is part of the number
   int n;
   in_file >> n; // Read integer starting with ch
}
```

The `isdigit` function is one of several useful functions that categorize characters—see Table 1. All return `true` or `false` as to whether the argument passes the test. You must include the `<cctype>` header to use these functions.

Table 1 Character Functions in <cctype>

Function	Accepted Characters
`isdigit`	0 ... 9
`isalpha`	a ... z, A ... Z
`islower`	a ... z
`isupper`	A ... Z
`isalnum`	a ... z, A ... Z, 0 ... 9
`isspace`	White space (space, tab, newline, and the rarely used carriage return, form feed, and vertical tab)

If you read a character from a stream and you don't like what you get, you can unget it.

© altrendo images/Getty Images.

8.2.3 Reading Lines

When each line of a file is a data record, it is often best to read entire lines with the `getline` function:

```
string line;
getline(in_file, line);
```

You can read a line of input with the `getline` function and then process it further.

The next input line (without the newline character) is placed into the string `line`.

The `getline` function returns the "not failed" condition. You can use the following loop to process each line in a file:

```
while (getline(in_file, line))
{
   Process line.
}
```

Note that `getline` is not a member function, but an ordinary function that is not called with the dot notation.

Here is a typical example of processing lines in a file. A file with population data from the CIA World Factbook site (`https://www.cia.gov/library/publications/the-world-factbook/index.html`) contains lines such as the following:

```
China  1330044605
India  1147995898
United States 303824646
. . .
```

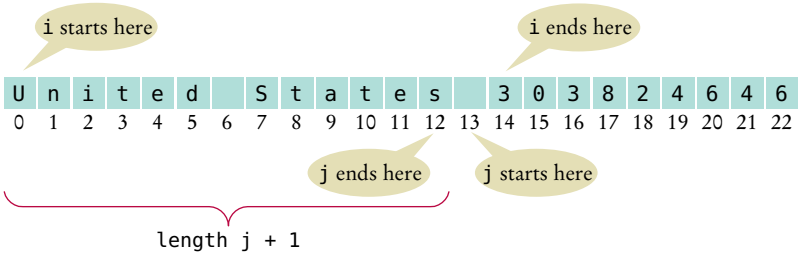
Because each line is a data record, it is natural to use the `getline` function for reading lines into a string variable. To extract the data from that string, you need to find out where the name ends and the number starts.

Locate the first digit:

```
int i = 0;
while (!isdigit(line[i])) { i++; }
```

Then go backward and skip white space:

```
int j = i - 1;
while (isspace(line[j])) { j--; }
```



Finally, extract the country name and population:

```
string country_name = line.substr(0, j + 1);
string population = line.substr(i);
```

There is just one problem. The population is stored in a string, not a number. You will see in Section 8.4 how to extract the population number as a number.

Common Error 8.1

Mixing >> and getline Input

It is tricky to mix >> and getline input. Suppose we place country names and populations on separate lines:

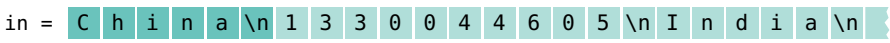
```
China
1330044605
India
1147995898
United States
303824646
```

Now we can use getline to read each country name without worrying about spaces:

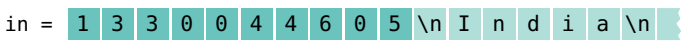
```
getline(in, country_name);
in >> population;
```

The getline function reads an entire line of input, including the newline character at the end of the line. It places all characters except for that newline character into the string country_name. The >> operator reads all white space (that is, spaces, tabs, and newlines) until it reaches a number. Then it reads only the characters in that number. It does not consume the character following the number, typically a newline. This is a problem when a call to getline immediately follows a call to >>. Then the call to getline reads only the newline, considering it as the end of an empty line.

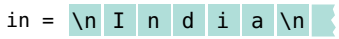
Perhaps an example will make this clearer. Consider the first input lines of the product descriptions. Calling getline consumes the darker-colored characters.



After the call to getline, the first line has been read completely, including the newline at the end. Next, the call to in >> population reads the digits.



After the call to in >> population, the digits of the number have been read, but the newline is still unread, because the >> operator never reads any more characters than absolutely necessary. Now we have a problem. The next call to getline reads a blank line.



This is a problem whenever an input with the >> operator is followed by a call to getline. The intention, of course, is to skip the rest of the current line and have getline read the next line. This purpose is achieved by the following statements, which must be inserted after the last call to the >> operator:

```
string remainder; // Read remainder of line
getline(in, remainder);
// Now you are ready to call getline again
```

Special Topic 8.1

Stream Failure Checking

Given a stream variable, for example,

```
istream in_file;
```

you know that you can use the `fail` member function to check whether the stream has failed:

```
if (in_file.fail())
{
   Report error.
}
else
{
   Process input.
}
```

Alternatively, you can use the stream variable itself as a condition. The C++ compiler converts the stream to a `bool` value that is `true` if the stream has *not* failed, and `false` if it has failed. Therefore, you can achieve the same effect with the following statement:

```
if (in_file)
{
   Process input.
}
else
{
   Report error.
}
```

This alternative is particularly attractive with input operations that return a stream. In Section 4.5.2, you saw that you can read a sequence of values with the loop

```
cout << "Enter values, Q to quit: ";
int value;
while (cin >> value)
{
   Process value.
}
```

When the user enters a `Q`, or any other input that is not an integer, the stream state fails.

You can chain the `>>` operators. Consider this example:

```
while (in_file >> name >> value)
{
   Process the name and value.
}
```

When evaluating the expression

```
in_file >> name >> value;
```

the subexpression `in_file >> name` executes first, and it returns `in_file`. Next, the expression `in_file >> value` executes and again returns `in_file`. The `in_file` object is then converted to a `bool` value and tested in the condition.

As you saw in Section 8.2.1 and Section 8.2.3, you can also use expressions

```
in_file.get(ch)
```

and

```
get_line(in_file, line)
```

as conditions. This works for the same reason. The `get` and `get_line` functions return the input stream, and it is converted to a `bool` value.

8.3 Writing Text Output

You use the << operator to send strings and numbers to a stream. To write a single character to a stream, use

```
out_file.put(ch);
```

To control how the output is formatted, you use stream *manipulators*. A manipulator is a value that affects the behavior of the stream. It is sent to a stream using the << operator. The `setw` manipulator, which you have already used, is a typical example. The statement

```
out_file << setw(10);
```

Use the `setw` manipulator to set the width of the next output.

does not cause any immediate output, but when the next item is written, it is padded with sufficient spaces so that the output spans ten characters. (If a value does not fit into the given width, it is not truncated.)

Occasionally, you need to pad numbers with leading zeroes, for example to print hours and minutes as 09:01. This is achieved with the `setfill` manipulator:

```
out_file << setfill('0') << setw(2) << hours
   << ":" << setw(2) << minutes << setfill(' ');
```

Now, a zero is used to pad the field. Afterward, the space is restored as the fill character.

By default, the fill characters appear before the item:

```
out_file << setw(10) << 123 << endl << setw(10) << 4567;
```

produces

```
       123
      4567
```

The numbers line up to the right. That alignment works well for numbers, but not for strings. Usually, you want strings to line up at the left. You use the `left` and `right` manipulators to set the alignment. The following example uses left alignment for a string and then switches back to right alignment for a number:

```
out_file << left << setw(10) << word << right << setw(10) << number;
```

The *default floating-point format* displays as many digits as are specified by the *precision* (6 by default), switching to scientific notation for large and small numbers. For example,

```
out_file << 12.3456789 << " " << 123456789.0 << " " << 0.0000123456789;
```

yields

```
12.3457 1.23457e+08 1.23457e-05
```

A manipulator is like a control button on a sound mixer. It doesn't produce an output, but it affects how the output looks.

The *fixed* format prints all values with the same number of digits after the decimal point. In the fixed format, the same numbers are displayed as

```
12.345679 123456789.000000 0.000012
```

Use the `fixed` and `setprecision` manipulators to format floating-point numbers with a fixed number of digits after the decimal point.

Use the `fixed` manipulator to select that format, and the `setprecision` manipulator to change the precision.

For example,

```
out_file << fixed << setprecision(2) << 1.2 << " " << 1.235
```

yields

```
1.20 1.24
```

Table 2 summarizes the stream manipulators. Note that all manipulators set the state of the stream object for all subsequent operations, with the exception of `setw`. After each output operation, the field width is reset to `0`. To use any of these manipulators, include the `<iomanip>` header.

Table 2 Stream Manipulators

Manipulator	Purpose	Example	Output

`setw`	Sets the field width of the next item only.	`out_file << setw(6) << 123 << endl`
		`  << 123 << endl`
		`  << setw(6) << 12345678;`
			`  123`
			`123`
			`12345678`

`setfill`	Sets the fill character for padding a field. (The default character is a space.)	`out_file << setfill('0') << setw(6)` `  << 123;`	`000123`
`left`	Selects left alignment.	`out_file << left << setw(6) << 123;`	`123`
`right`	Selects right alignment (default).	`out_file << right << setw(6) << 123;`	`  123`
`fixed`	Selects fixed format for floating-point numbers.	`double x = 123.4567;` `out_file << x << endl << fixed << x;`	`123.457` `123.456700`

`setprecision`	Sets the number of significant digits for the default floating-point format, the number of digits after the decimal point for fixed format.	`double x = 123.4567;`
		`out_file << fixed << x << endl`
		`  << setprecision(2) << x;`
			`123.456700`
			`123.46`

`scientific`	Selects scientific floating-point format, with one digit before the decimal point and an exponent.	`out_file << scientific` `  << setprecision(3) << 123.4567;`	`1.235e+02`

`defaultfloat`	(Since C++ 11) Switches back to the default floating-point format.	`double x = 123.4567;`
		`out_file << fixed << x << endl`
		`  << defaultfloat << x;`
			`123.456700`
			`123.457`

Special Topic 8.2

Unicode, UTF-8, and C++ Strings

As described in Computing & Society 2.2, the **Unicode** standard encodes alphabets from many languages. Each Unicode character has a unique 21-bit code that is written using the hexadecimal number system. For example, é (Latin small letter e with acute accent) has the code U+OOE9 and 🚄 (high speed train) has the code U+1F684. However, it would be inefficient to use these codes when characters are saved in files or transmitted over the Internet. Instead, a character encoding is used that represents each Unicode character as a sequence of one or more bytes.

If you process English text, you don't need to worry about character encodings because English characters are encoded with a single byte. The situation is more complex if you want to process text that contains characters such as é or 🚄. Nowadays, most data on the Internet uses the UTF-8 encoding. As it happens, the UTF-8 encoding of the string é takes up two bytes, and the UTF-8 encoding of 🚄 takes four bytes (see Appendix C). Therefore, the text

```
San José 🚄
```

with two spaces, six English characters and these two characters, consists of 14 bytes when encoded with UTF-8.

When you read this text into a C++ string value, you get a string object of length 14, even though it only contains ten Unicode characters. It is not fruitful to look at the individual `char` values in isolation. Instead, work with substrings that are made up of one or more Unicode characters.

For example, suppose you want to find the position of the é. Form a string containing a single Unicode character.

If your C++ compiler uses the UTF-8 encoding, you can make such a string as

```
string e_acute = u8"é";
string high_speed_train = u8"🚄";
```

The `u8` prefix indicates that you want to use the UTF-8 encoding. However, to make sure that your source file is portable to any system, it is a good idea to provide the Unicode value for the letter instead. You can use the `\U` prefix followed by eight hexadecimal digits:

```
string e_acute = u8"\U000000e9";
string high_speed_train = u8"\U0001f684";
```

Now that you know how to specify a string containing an arbitrary Unicode character, you can use the `find` member function to check whether the Unicode character occurs in a given string. We haven't covered this member function before because it is a bit fussy to use. Here is what you need to do:

```
string message = . . .;
size_t pos = message.find(e_acute);
if (pos != string::npos)
{
   // Message has e_acute starting at position pos
}
```

The `size_t` type indicates a non-negative position, and `string::npos` is a special value that denotes no position.

The `locale` library, which we don't cover in this book, gives you additional mechanisms to process UTF-8 strings. Exercise P8.11 provides a simple way of accessing individual characters.

EXAMPLE CODE See special_topic_2 of your companion code for a sample program using UTF-8 encoding.

8.4 Parsing and Formatting Strings

In the preceding sections, you saw how file streams read characters from a file and write characters to a file. The `istringstream` class reads characters from a string, and the `ostringstream` class writes characters to a string. That doesn't sound so exciting—we already know how to access and change the characters of a string. However, the string stream classes have the same **public interface** as the other stream classes. In particular, you can use the familiar `>>` and `<<` operators to read and write numbers that are contained in strings. For that reason, the `istringstream` and `ostringstream` classes are called *adapters*—they adapt strings to the stream interface. Include the `<sstream>` header when you use string streams.

© iStockphoto.

Like an adapter that converts your power plugs to international outlets, the string stream adapters allow you to access strings as streams.

Use an `istringstream` to convert the numbers inside a string to integers or floating-point numbers.

Here is a typical example. Suppose the string `date` contains a date such as `"January 24, 1973"`, and we want to separate it into month, day, and year. First, construct an `istringstream` object. Then use the `str` function to set the stream to the string that you want to read:

```
istringstream strm;
strm.str("January 24, 1973");
```

Next, simply use the `>>` operator to read the month name, the day, the comma separator, and the year:

```
string month;
int day;
string comma;
int year;
strm >> month >> day >> comma >> year;
```

Now `month` is `"January"`, `day` is `24`, and `year` is `1973`. Note that this input statement yields `day` and `year` as *integers*. Had we taken the string apart with `substr`, we would have obtained only strings, not numbers.

If you only need to convert a single string to its `int` or `double` value, you can instead use the functions `stoi` and `stod`:

```
string year = "1973";
int y = stoi(year); // Sets y to the integer 1973
```

These functions are available since C++ 11. If you have an older version of C++, use a helper function for this purpose:

```
int string_to_int(string s)
{
   istringstream strm;
   strm.str(s);
   int n = 0;
   strm >> n;
   return n;
}
```

Use an ostringstream to convert numeric values to strings.

By writing to a string stream, you can convert integers or floating-point numbers to strings. First construct an `ostringstream` object:

```
ostringstream strm;
```

Next, use the << operator to add a number to the stream. The number is converted into a sequence of characters:

```
strm << fixed << setprecision(5) << 10.0 / 3;
```

Now the stream contains the string `"3.33333"`. To obtain that string from the stream, call the `str` member function:

```
string output = strm.str();
```

You can build up more complex strings in the same way. Here we build a data string of the month, day, and year:

```
string month = "January";
int day = 4;
int year = 1973;
ostringstream strm;
strm << month << " " << setw(2) << setfill('0') << day << ", " << year;
string output = strm.str();
```

Now `output` is the string `"January 04, 1973"`. Note that we converted the *integers* `day` and `year` into a string.

If you don't need formatting, you can use the `to_string` function to convert numbers to strings. For example, `to_string(1973)` is the string `"1973"`.

You can produce a formatted date, without a leading zero for days less than ten, like this:

```
string output = month + " " + to_string(day) + ", " + to_string(year);
```

This function is available since C++ 11. Here is an implementation that you can use with older versions:

```
string int_to_string(int n)
{
   ostringstream strm;
   strm << n;
   return strm.str();
}
```

8.5 Command Line Arguments

Depending on the operating system and C++ development environment used, there are different methods of starting a program—for example, by selecting "Run" in the compilation environment, by clicking on an icon, or by typing the name of the program at a prompt in a command shell window. The latter method is called "invoking the program from the **command line**". When you use this method, you must type the name of the program, of course, but you can also type in additional information that the program can use. These additional strings are called **command line arguments**. For example, if you start a program with the command line

```
prog -v input.dat
```

then the program receives two command line arguments: the strings `"-v"` and `"input.dat"`. It is entirely up to the program what to do with these strings. It is customary to interpret strings starting with a hyphen (`-`) as options and other strings as file names.

Programs that start from the command line can receive the name of the program and the command line arguments in the `main` function.

To receive command line arguments, you need to define the `main` function in a different way. You define two parameter variables: an integer and an array of string literals of type `char*`.

```
int main(int argc, char* argv[])
{
   . . .
}
```

Here `argc` is the count of arguments, and `argv` contains the values of the arguments. In our example, `argc` is 3, and `argv` contains the three strings

```
argv[0]:   "prog"
argv[1]:   "-v"
argv[2]:   "input.dat"
```

Note that `argv[0]` is always the name of the program and that `argc` is always at least 1.

Let's write a program that *encrypts* a file—that is, scrambles it so that it is unreadable except to those who know the decryption method. Ignoring 2,000 years of progress in the field of encryption, we will use a method familiar to Julius Caesar, replacing an A with a D, a B with an E, and so on. That is, each character c is replaced with $c + 3$ (see Figure 2).

© xyno/iStockphoto.

The emperor Julius Caesar used a simple scheme to encrypt messages.

The program takes the following command line arguments:

- An optional `-d` flag to indicate decryption instead of encryption
- The input file name
- The output file name

For example,

```
caesar input.txt encrypt.txt
```

encrypts the file `input.txt` and places the result into `encrypt.txt`.

```
caesar -d encrypt.txt output.txt
```

decrypts the file `encrypt.txt` and places the result into `output.txt`.

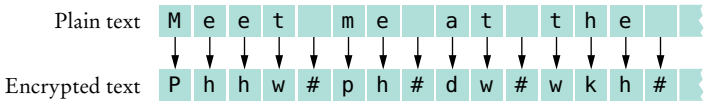


Figure 2 Caesar Cipher

sec05/caesar.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <sstream>
5
6  using namespace std;
7
8  /**
9     Encrypts a stream using the Caesar cipher.
```

```
   @param in the stream to read from
   @param out the stream to write to
   @param k the encryption key
*/
void encrypt_file(ifstream& in, ofstream& out, int k)
{
   char ch;
   while (in.get(ch))
   {
      out.put(ch + k);
   }
}

int main(int argc, char* argv[])
{
   int key = 3;
   int file_count = 0; // The number of files specified
   ifstream in_file;
   ofstream out_file;

   for (int i = 1; i < argc; i++) // Process all command-line arguments
   {
      string arg = argv[i]; // The currently processed argument
      if (arg == "-d") // The decryption option
      {
         key = -3;
      }
      else // It is a file name
      {
         file_count++;
         if (file_count == 1) // The first file name
         {
            in_file.open(arg);
            if (in_file.fail()) // Exit the program if opening failed
            {
               cout << "Error opening input file " << arg << endl;
               return 1;
            }
         }
         else if (file_count == 2) // The second file name
         {
            out_file.open(arg);
            if (out_file.fail())
            {
               cout << "Error opening output file " << arg << endl;
               return 1;
            }
         }
      }
   }

   if (file_count != 2) // Exit if the user didn't specify two files
   {
      cout << "Usage: " << argv[0] << " [-d] infile outfile" << endl;
      return 1;
   }

   encrypt_file(in_file, out_file, key);
   return 0;
}
```

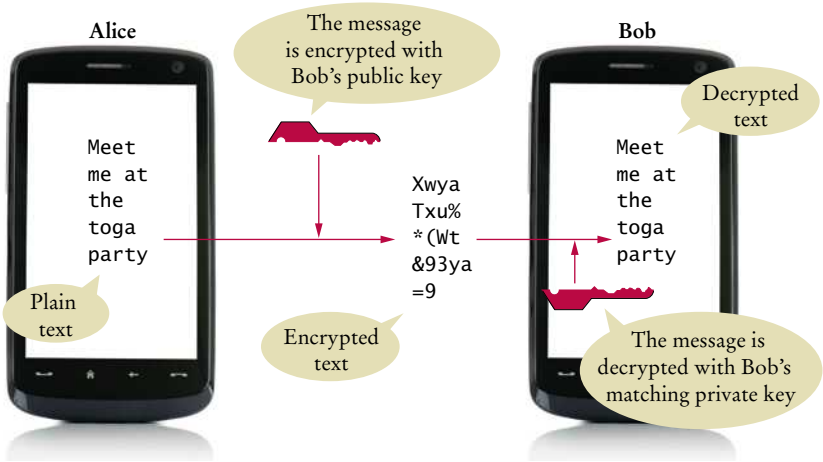
Computing & Society 8.1 Encryption Algorithms

The exercises at the end of this chapter give a few algorithms to encrypt text. Don't actually use any of those methods to send secret messages to your lover. Any skilled cryptographer can *break* these schemes in a very short time—that is, reconstruct the original text without knowing the secret keyword.

In 1978 Ron Rivest, Adi Shamir, and Leonard Adleman introduced an encryption method that is much more powerful. The method is called *RSA encryption*, after the last names of its inventors. The exact scheme is too complicated to present here, but it is not actually difficult to follow. You can find the details in `http://people.csail.mit.edu/rivest/Rsapaper.pdf`.

RSA is a remarkable encryption method. There are two keys: a public key and a private key. (See the figure.) You can print the public key on your business card (or in your e-mail signature block) and give it to anyone. Then anyone can send you messages that only you can decrypt. Even though everyone else knows the public key, and even if they intercept all the messages coming to you, they cannot break the scheme and actually read the messages. In 1994, hundreds of researchers, collaborating over the Internet, cracked an RSA message encrypted with a 129-digit key. Messages encrypted with a key of 230 digits or more are expected to be secure.

The inventors of the algorithm obtained a *patent* for it. A patent is a deal that society makes with an inventor. For a period of 20 years, the inventor has an exclusive right for its commercialization, may collect royalties from others wishing to manufacture the invention, and may even stop competitors from using it altogether. In return, the inventor must publish the invention, so that others may learn from it, and must relinquish all claim to it after the monopoly period ends. The presumption is that in the absence of patent law, inventors would be reluctant to go through the trouble of inventing, or they would try to cloak their techniques to prevent others from copying their devices.



(mobile phone) © Anna Khomulo/iStockphoto.

Public-Key Encryption

There has been some controversy about the RSA patent. Had there not been patent protection, would the inventors have published the method anyway, thereby giving the benefit to society without the cost of the 20-year monopoly? In this case, the answer is probably yes. The inventors were academic researchers, who live on salaries rather than sales receipts and are usually rewarded for their discoveries by a boost in their reputation and careers. Would their followers have been as active in discovering (and patenting) improvements? There is no way of knowing, of course. Is an algorithm even patentable, or is it a mathematical fact that belongs to nobody? The patent office did take the latter attitude for a long time. The RSA inventors and many others described their inventions in terms of imaginary electronic devices, rather than algorithms, to circumvent that restriction. Nowadays, the patent office will award software patents.

There is another interesting aspect to the RSA story. A programmer, Phil Zimmermann, developed a program called PGP (for *Pretty Good Privacy*) that is based on RSA. Anyone can use the program to encrypt messages, and decryption is not feasible even with the most powerful computers. You can get a copy of a free PGP implementation from the GNU project (`http://www.gnupg.org`). The existence of strong encryption methods bothers the United States government to no end. Criminals and foreign agents can send communications that the police and intelligence agencies cannot decipher. The government considered charging Zimmermann with breaching a law that forbids the unauthorized export of munitions, arguing that he should have known that his program would appear on the Internet. There have been serious proposals to make it illegal for private citizens to use these encryption methods, or to keep the keys secret from law enforcement.

HOW TO 8.1

Processing Text Files

Processing text files that contain real data can be surprisingly challenging. This How To gives you step-by-step guidance.

Problem Statement As an example, we will consider this task: Read two country data files, `worldpop.txt` and `worldarea.txt` (supplied with the book's companion code). Both files contain the same countries in the same order. Write a file `world_pop_density.txt` that contains country names and population densities (people per square km), with the country names aligned left and the numbers aligned right:

```
Afghanistan                    50.56
Akrotiri                      127.64
Albania                       125.91
Algeria                        14.18
American Samoa                288.92
. . .
```

© Oksana Perkins/iStockphoto.

Singapore is one of the most densely populated countries in the world.

Step 1 Understand the processing task.

As always, you need to have a clear understanding of the task before designing a solution. Can you carry out the task by hand (perhaps with smaller input files)? If not, get more information about the problem.

The following pseudocode describes our processing task:

```
While there are more lines to be read
   Read a line from each file.
   Extract the country name.
   population = number following the country name in the first line
   area = number following the country name in the second line
   If area != 0
      density = population / area
   Print country name and density.
```

Step 2 Determine which files you need to read and write.

This should be clear from the problem. In our example, there are two input files, the population data and the area data, and one output file.

Step 3 Choose a method for obtaining the file names.

There are three options:

- Hard-coding the file names (such as `"worldpop.txt"`)
- Asking the user:

```
cout << "Enter filename: ";
cin >> filename;
in_file.open(filename);
```

- Using command-line arguments for the file names

In our example, we use hard-coded file names for simplicity.

Step 4 Choose between line, word, and character-based input.

As a rule of thumb, read lines if the input data is grouped by lines. That is the case with tabular data, as in our example, or when you need to report line numbers.

When gathering data that can be distributed over several lines, then it makes more sense to read words. Keep in mind that you lose all white space when you read words.

Reading characters is mostly useful for tasks that require access to individual characters. Examples include analyzing character frequencies, changing tabs to spaces, or encryption.

Step 5 With line-oriented input, extract the required data.

It is simple to read a line of input with the `getline` function. Then you need to get the data out of that line. You can extract substrings, as described in Section 8.2. Alternatively, you can turn the line into an `istringstream` and extract its components with the `>>` operator. The latter approach is easier when the number of items on each line is constant. In our example, that is not the case—country names can consist of more than one string. Therefore, we choose to extract substrings from each input line.

If you need any of the substrings as numbers, you must convert them (see Section 8.4).

Step 6 Place repeatedly occurring tasks into functions.

Processing input files usually has repetitive tasks, such as skipping over white space or extracting numbers from strings. It really pays off to develop a set of functions to handle these tedious operations.

In our example, we have a common task that calls for a helper function: extracting the country name and the value that follows. This task can be implemented in a helper function

```
void read_line(string line, string& country, double& value)
```

We also need a helper function `string_to_double` to convert the population and area values to floating-point numbers. This function is similar to `string_to_int` that was developed in Section 8.4.

Step 7 If required, use manipulators to format the output.

If you are asked to format your output, use manipulators, as described in Section 8.3. Usually, you want to switch to fixed format for the output and set the precision. Then use `setw` before every value, and use `left` for aligning strings and `right` for aligning numbers:

```
out << setw(40) << left << country << setw(15) << right << density << endl;
```

Here is the complete program:

how_to_1/popdensity.cpp

```
#include <cctype>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>

using namespace std;

/**
   Converts a string to a floating-point number, e.g. "3.14" -> 3.14.
   @param s a string representing a floating-point number
   @return the equivalent floating-point number
*/
double string_to_double(string s)
{
   istringstream stream;
   stream.str(s);
```

```
19    double x = 0;
20    stream >> x;
21    return x;
22 }
23
24 /**
25    Extracts the country and associated value from an input line.
26    @param line a line containing a country name, followed by a number
27    @param country the string for holding the country name
28    @param value the variable for holding the associated value
29    @return true if a line has been read, false at the end of the stream
30 */
31 void read_line(string line, string& country, double& value)
32 {
33    int i = 0; // Locate the start of the first digit
34    while (!isdigit(line[i])) { i++; }
35    int j = i - 1; // Locate the end of the preceding word
36    while (isspace(line[j])) { j--; }
37
38    country = line.substr(0, j + 1); // Extract the country name
39    value = string_to_double(line.substr(i)); // Extract the number value
40 }
41
42 int main()
43 {
44    ifstream in1;
45    ifstream in2;
46    in1.open("worldpop.txt"); // Open input files
47    in2.open("worldarea.txt");
48
49    ofstream out;
50    out.open("world_pop_density.txt"); // Open output file
51    out << fixed << setprecision(2);
52
53    string line1;
54    string line2;
55
56    // Read lines from each file
57    while (getline(in1, line1) && getline(in2, line2))
58    {
59       string country;
60       double population;
61       double area;
62
63       // Split the lines into country and associated value
64       read_line(line1, country, population);
65       read_line(line2, country, area);
66
67       // Compute and print the population density
68       double density = 0;
69       if (area != 0) // Protect against division by zero
70       {
71          density = population * 1.0 / area;
72       }
73       out << setw(40) << left << country
74          << setw(15) << right << density << endl;
75    }
76
77    return 0;
78 }
```

WORKED EXAMPLE 8.1

Looking for for Duplicates

Learn how to process a file to locate lines that contain repeated words. See your E-Text or visit `wiley.com/go/bclo3`.

8.6 Random Access and Binary Files

In the following sections, you will learn how to read and write data at arbitrary positions in a file, and how to edit image files.

At a sit-down dinner, food is served sequentially. At a buffet, you have "random access" to all food items.

© iStockphoto.

8.6.1 Random Access

You can access any position in a random access file by moving the *file pointer* prior to a read or write operation.

So far, you've read from a file an item at a time and written to a file an item at a time, without skipping forward or backward. That access pattern is called **sequential access**. In many applications, we would like to access specific items in a file without first having to read all preceding items. This access pattern is called **random access** (see Figure 3). There is nothing "random" about random access—the term means that you can read and modify any item stored at any location in the file.

Only file streams support random access; the `cin` and `cout` streams, which are attached to the keyboard and the terminal, do not. Each file stream has two special positions: the *get* position and the *put* position (see Figure 4). These positions determine where the next character is read or written.

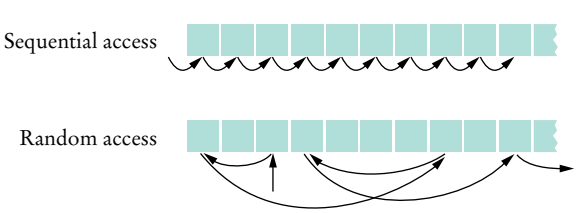


Figure 3 Sequential and Random Access

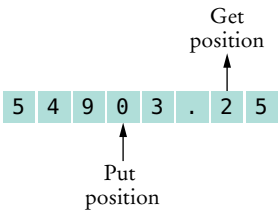


Figure 4 Get and Put Positions

The following function calls move the get and put positions to a given value, counted from the beginning of the stream.

```
strm.seekg(position);
strm.seekp(position);
```

To determine the current values of the get and put positions (counted from the beginning of the file), use

```
position = strm.tellg();
position = strm.tellp();
```

Whenever you put data to the stream, the `get` position becomes undefined. Call `seekg` when you switch back to reading. Call `seekp` when you switch from reading to writing.

8.6.2 Binary Files

Many files, in particular those containing images and sounds, do not store information as text but as binary numbers. The numbers are represented as sequences of **bytes**, just as they are in the memory of the computer. (Each byte is a value between `0` and 255.) In binary format, a floating-point number always occupies 8 bytes. We will study random access with a binary file format for images.

We have to cover a few technical issues about binary files. To open a binary file for reading and writing, use the following command:

```
fstream strm;
strm.open(filename, ios::in | ios::out | ios::binary);
```

You read a byte with the call

```
int input = strm.get();
```

This call returns a value between `0` and 255. To read an integer, read four bytes b_0, b_1, b_2, b_3 and combine them to $b_0 + b_1 \cdot 256 + b_2 \cdot 256^2 + b_3 \cdot 256^3$. We will supply a helper function for this task.

The `>>` operator cannot be used to read numbers from a binary file.

8.6.3 Processing Image Files

In this section, you will learn how to write a program for editing image files in the BMP format. Unlike the more common GIF, PNG, and JPEG formats, the BMP format is quite simple because it does not use data compression. As a consequence, BMP files are huge and you will rarely find them in the wild. However, image editors can convert any image into BMP format.

There are different versions of the BMP format; we will only cover the simplest and most common one, sometimes called the 24-bit true color format. In this format, each pixel is represented as a sequence of three bytes, one each for the blue, green, and red value. For example, the color cyan (a mixture of blue and green) is 255 255 `0`, red is `0 0` 255, and medium gray is 128 128 128.

A BMP file starts with a header that contains various pieces of information. We only need the following items:

Position	Item
2	The size of this file in bytes
10	Offset to the start of the image data
18	The width of the image in pixels
22	The height of the image in pixels

The image is stored as a sequence of pixel rows, starting with the pixels of the bottom-most row of the image. Each pixel row contains a sequence of blue/green/red triplets. The end of the row is padded with additional bytes so that the number of bytes in the row is divisible by 4. (See Figure 5.) For example, if a row consisted of merely three pixels, one cyan, one red, and one medium gray one, the row would be encoded as

```
255 255 0 0 0 255 128 128 128 x y z
```

where *x y z* are padding bytes to bring the row length up to 12, a multiple of 4. It is these little twists that make working with real-life file formats such a joyful experience.

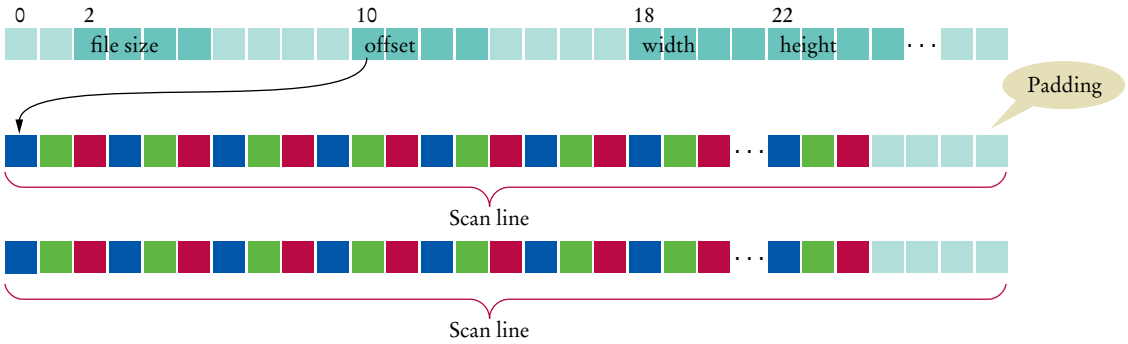


Figure 5 The BMP File Format for 24-bit True Color Images

The sample program at the end of this section reads every pixel of a BMP file and replaces it with its negative, turning white to black, cyan to red, and so on. The result is a negative image of the kind that old-fashioned film cameras used to produce (see Figure 6).

Figure 6
An Image and Its Negative

To try out this program, take one of your favorite images, use an image editor to convert to BMP format (or use `queen-mary.bmp` from the code files for this book), then run the program and view the transformed file in an image editor. Exercises P8.16 and P8.17 ask you to produce more interesting effects.

sec06/imagemod.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>

using namespace std;

/**
   Processes a pixel by forming the negative.
   @param blue the blue value of the pixel
   @param green the green value of the pixel
   @param red the red value of the pixel
*/
void process(int& blue, int& green, int& red)
{
   blue = 255 - blue;
   green = 255 - green;
   red = 255 - red;
}

/**
   Gets an integer from a binary stream.
   @param stream the stream
   @param offset the offset at which to read the integer
   @return the integer starting at the given offset
*/
int get_int(fstream& stream, int offset)
{
   stream.seekg(offset);
   int result = 0;
   int base = 1;
   for (int i = 0; i < 4; i++)
   {
      result = result + stream.get() * base;
      base = base * 256;
   }
   return result;
}

int main()
{
   cout << "Please enter the file name: ";
   string filename;
   cin >> filename;

   fstream stream;
   // Open as a binary file
   stream.open(filename, ios::in | ios::out | ios::binary);

```

```
50    int file_size = get_int(stream, 2); // Get the image dimensions
51    int start = get_int(stream, 10);
52    int width = get_int(stream, 18);
53    int height = get_int(stream, 22);
54
55    // Scan lines must occupy multiples of four bytes
56    int scanline_size = width * 3;
57    int padding = 0;
58    if (scanline_size % 4 != 0)
59    {
60       padding = 4 - scanline_size % 4;
61    }
62
63    if (file_size != start + (scanline_size + padding) * height)
64    {
65       cout << "Not a 24-bit true color image file." << endl;
66       return 1;
67    }
68
69    int pos = start;
70
71    for (int i = 0; i < height; i++) // For each scan line
72    {
73       for (int j = 0; j < width; j++) // For each pixel
74       {
75          stream.seekg(pos); // Go to the next pixel
76          int blue = stream.get(); // Read the pixel
77          int green = stream.get();
78          int red = stream.get();
79
80          process(blue, green, red); // Process the pixel
81
82          stream.seekp(pos); // Go back to the start of the pixel
83
84          stream.put(blue); // Write the pixel
85          stream.put(green);
86          stream.put(red);
87          pos = pos + 3;
88       }
89
90       stream.seekg(padding, ios::cur); // Skip the padding
91       pos = pos + padding;
92    }
93
94    return 0;
95 }
```

Computing & Society 8.2 Databases and Privacy

Most companies use computers to keep huge data files of customer records and other business information. Databases not only lower the cost of doing business; they improve the quality of service that companies can offer. Nowadays it is almost unimaginable how time-consuming it used to be to withdraw money from a bank branch or to make travel reservations.

Today most databases are organized according to the *relational model*. Suppose a company stores your orders and payments. They will probably not repeat your name and address on every order; that would take unnecessary space. Instead, they will keep one file of all their customer names and identify each customer by a unique customer number. Only that customer number, not the entire customer information, is kept with an order record.

To print an invoice, the database program must issue a *query* against both the customer and order files and pull the necessary information (name, address, articles ordered) from both. Frequently, queries involve more than two files. For example, the company may have a file of addresses of car owners and a file of people with good payment history and may want to find all of its customers who placed an order in the last month, drive an expensive car, and pay their bills, so they can send them another catalog. This kind of query is, of course, much faster if all customer files use the *same* key, which is why so many organizations in the United States try to collect the Social Security numbers of their customers.

Customers

Cust. #:	Name
11439	Doe, John

Orders

Order #:	Cust. #:	Item
59673	11439	DOS for Historians
59897	11439	Big C++
61013	11439	C++ for Everyone

Relational Database Files

The Social Security Act of 1935 provided that each contributor be assigned a Social Security number to track contributions into the Social Security Fund. These numbers have a distinctive format, such as 078-05-1120. (This particular number was printed on sample cards that were inserted in wallets. It actually was the Social Security number of the secretary of a vice president at the wallet manufacturer. When thousands of people used it as their own, the number was voided, and the secretary received a new number.) Although they had not originally been intended for use as a universal identification number, Social Security numbers have become just that.

Some people are very concerned about the fact that just about every organization wants to store their Social Security number and other personal information. There is the possibility that companies and the government can merge multiple databases and derive information about us that we may wish they did not have or that simply may be untrue. An insurance company may deny coverage, or charge a higher premium, if it finds that you have too many relatives with a certain disease. You may be denied a job because of an inaccurate credit or medical report, and you may not even know the reason. These are very disturbing developments that have had a very negative impact for a small but growing number of people.

In many industrialized countries (but not currently in the United States), citizens have a right to control what information about themselves should be communicated to others and under what circumstances.

Social Security Card

CHAPTER SUMMARY

Develop programs that read and write files.

- To read or write files, you use variables of type `fstream`, `ifstream`, or `ofstream`.
- When opening a file stream, you supply the name of the file stored on disk.
- Read from a file stream with the same operations that you use with `cin`.
- Write to a file stream with the same operations that you use with `cout`.
- Always use a reference parameter for a stream.

Be able to process text in files.

- When reading a string with the >> operator, the white space between words is consumed.
- You can get individual characters from a stream and unget the last one.
- You can read a line of input with the `getline` function and then process it further.

Write programs that neatly format their output.

- Use the `setw` manipulator to set the width of the next output.
- Use the `fixed` and `setprecision` manipulators to format floating-point numbers with a fixed number of digits after the decimal point.

Convert between strings and numbers.

- Use an `istringstream` to convert the numbers inside a string to integers or floating-point numbers.
- Use an `ostringstream` to convert numeric values to strings.

Process the command line arguments of a C++ program.

- Programs that start from the command line can receive the name of the program and the command line arguments in the `main` function.

Develop programs that read and write binary files.

- You can access any position in a random access file by moving the file pointer prior to a read or write operation.

REVIEW EXERCISES

■ **R8.1** When do you open a file as an `ifstream`, as an `ofstream`, or as an `fstream`? Could you simply open all files as an `fstream`?

■■ **R8.2** What happens if you write to a file that you only opened for reading? Try it out if you don't know.

■■ **R8.3** What happens if you try to open a file for reading that doesn't exist? What happens if you try to open a file for writing that doesn't exist?

■■ **R8.4** What happens if you try to open a file for writing, but the file or device is write-protected (sometimes called read-only)? Try it out with a short test program.

■ **R8.5** How do you open a file whose name contains a backslash, such as `c:\temp\output.dat`?

■ **R8.6** Why are the `in` and `out` parameter variables of the `encrypt_file` function in Section 8.5 reference parameters and not value parameters?

■■ **R8.7** Give an output statement to write a date and time in ISO 8601 format, such as

```
2011-03-01 09:35
```

Assume that the date and time are given in five integer variables year, month, day, hour, minute.

■■ **R8.8** Give an output statement to write one line of a table containing a product description, quantity, unit price, and total price in dollars and cents. You want the columns to line up, like this:

```
Item                Qty  Price     Total
Toaster             3    $29.92    $89.85
Hair Dryer          1    $24.95    $24.95
Car Vacuum          2    $19.99    $39.98
```

■■ **R8.9** How can you convert the string `"3.14"` into the floating-point number 3.14? How can you convert the floating-point number 3.14 into the string `"3.14"`? Give two ways.

■ **R8.10** What is a command line? How can a program read its command line?

■ **R8.11** If a program `woozle` is started with the command

```
woozle -DNAME=Piglet -I\eeyore -v heff.cpp a.cpp lump.cpp
```

what is the value of `argc`, and what are the values of `argv[0]`, `argv[1]`, and so on?

■ **R8.12** What is the difference between sequential access and random access?

■ **R8.13** What is the difference between a text file and a binary file?

■ **R8.14** What are the get and put positions in a file? How do you move them? How do you tell their current positions?

■■ **R8.15** What happens if you try to move the get or put position past the end of a file? What happens if you try to move the get or put position of `cin` or `cout`? Try it out and report your results.

PRACTICE EXERCISES

■ **E8.1** Write a program that carries out the following tasks:

Open a file with the name hello.txt.
Store the message "Hello, World!" in the file.
Close the file.
Open the same file again.
Read the message into a string variable and print it.

■ **E8.2** Write a program that reads a text file containing floating-point numbers. Print the average of the numbers in the file. Prompt the user for the file name.

■ **E8.3** Repeat Exercise E8.2, but allow the user to specify the file name on the command-line. If the user doesn't specify any file name, then prompt the user for the name.

■ **E8.4** Write a program that reads a file containing two columns of floating-point numbers. Prompt the user for the file name. Print the average of each column.

■■ **E8.5** Write a program that reads each line in a file, reverses its characters, and writes the resulting line to another file. Suppose the user specifies `input.txt` and `output.txt` when prompted for the file names, and `input.txt` contains the lines

```
Mary had a little lamb
Its fleece was white as snow
And everywhere that Mary went
The lamb was sure to go.
```

After the program is finished, `output.txt` should contain

```
bmal elttil a dah yraM
wons sa etihw saw eceelf stI
tnew yraM taht erehwyreve dnA
.og ot erus saw bmal ehT
```

■■ **E8.6** Write a program that reads each line in a file, reverses its characters, and writes the resulting line to the same file. Use the following pseudocode:

```
While the end of the file has not been reached
   pos1 = current get position
   Read a line.
   If the line was successfully read
      pos2 = current get position
      Set put position to pos1.
      Write the reversed line.
      Set get position to pos2.
```

■■ **E8.7** Write a program that reads each line in a file, reverses its lines, and writes them to another file. Suppose the user specifies `input.txt` and `output.txt` when prompted for the file names, and `input.txt` contains the lines

```
Mary had a little lamb
Its fleece was white as snow
And everywhere that Mary went
The lamb was sure to go.
```

After the program is finished, `output.txt` should contain

```
The lamb was sure to go.
And everywhere that Mary went
Its fleece was white as snow
Mary had a little lamb
```

■■ **E8.8** Write a program that asks the user for a file name and displays the number of characters, words, and lines in that file. Then have the program ask for the name of the next file. When the user enters a file that doesn't exist (such as the empty string), the program should exit.

■ **E8.9** Write a program `copyfile` that copies one file to another. The file names are specified on the command line. For example,

```
copyfile report.txt report.sav
```

■ **E8.10** Write a program that **concatenates** the contents of several files into one file. For example,

```
catfiles chapter1.txt chapter2.txt chapter3.txt book.txt
```

makes a long file `book.txt` that contains the contents of the files `chapter1.txt`, `chapter2.txt`, and `chapter3.txt`. The target file is always the last file specified on the command line.

■ **E8.11** Write a program that reads a file, removes any blank lines, and writes the non-blank lines back to the same file.

■■ **E8.12** Write a program that reads a file, removes any blank lines at the beginning or end of the file, and writes the remaining lines back to the same file.

PROGRAMMING PROJECTS

■■ **P8.1** Write a program `find` that searches all files specified on the command line and prints out all lines containing a keyword. For example, if you call

```
find Tim report.txt address.txt homework.cpp
```

then the program might print

```
report.txt: discussed the results of my meeting with Tim T
address.txt: Torrey, Tim|11801 Trenton Court|Dallas|TX
address.txt: Walters, Winnie|59 Timothy Circle|Detroit|MI
homework.cpp: Time now;
```

The keyword is always the first command-line argument.

■■ **P8.2** Write a program that checks the spelling of all words in a file. It should read each word of a file and check whether it is contained in a word list. A word list is available on most UNIX systems (including Linux and Mac OS X) in the file `/usr/share/dict/words`. (If you don't have access to a UNIX system, you can find a copy of the file on the Internet by searching for `/usr/share/dict/words`.) The program should print out all words that it cannot find in the word list. Follow this pseudocode:

```
Open the dictionary file.
Define a vector of strings called words.
For each word in the dictionary file
   Append the word to the words vector.
Open the file to be checked.
For each word in that file
   If the word is not contained in the words vector
      Print the word.
```

■■ **P8.3** Get the data for names in prior decades from the Social Security Administration. Paste the table data in files named `babynames80s.txt`, etc. Modify the `sec01/babynames.cpp`

program so that it prompts the user for a file name. The numbers in the files have comma separators, so modify the program to handle them. Can you spot a trend in the frequencies?

■■ **P8.4** Write a program that reads in `sec01/babynames.txt` and produces two files, `boynames.txt` and `girlnames.txt`, separating the data for the boys and girls.

■■■ **P8.5** Write a program that reads a file in the same format as `sec01/babynames.txt` and prints all names that are both boy and girl names (such as Alexis or Morgan).

■■ **P8.6** Write a program that reads the country data in the file `how_to_1/worldpop.txt` (included with the book's source code). Do not edit the file. Use the following algorithm for processing each line. Add characters that are not white space to the country name. When you encounter a white space, locate the next non-white space character. If it is not a digit, add a space and that character to the country name, and keep adding characters until the next white space. If a white space is followed by a digit, unget it and read the number. Print the total of all country populations (excepting the entry for "European Union").

■■ **P8.7** *Random monoalphabet cipher*. The Caesar cipher, which shifts all letters by a fixed amount, is far too easy to crack. Here is a better idea. As the key, don't use numbers but words. Suppose the key word is `FEATHER`. Then first remove duplicate letters, yielding `FEATHR`, and append the other letters of the alphabet in reverse order:

F	E	A	T	H	R	Z	Y	X	W	V	U	S	Q	P	O	N	M	L	K	J	I	G	D	C	B

Now encrypt the letters as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
F	E	A	T	H	R	Z	Y	X	W	V	U	S	Q	P	O	N	M	L	K	J	I	G	D	C	B

Write a program that encrypts or decrypts a file using this cipher. For example,

```
crypt -d -kFEATHER encrypt.txt output.txt
```

decrypts a file using the keyword `FEATHER`. It is an error not to supply a keyword.

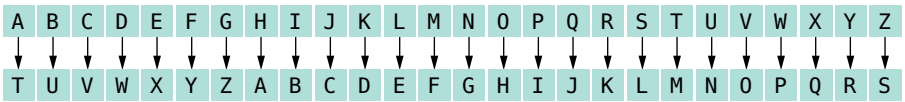
■ **P8.8** *Letter frequencies*. If you encrypt a file using the cipher of Exercise P8.7, it will have all of its letters jumbled up, and will look as if there is no hope of decrypting it without knowing the keyword. Guessing the keyword seems hopeless too. There are just too many possible keywords. However, someone who is trained in decryption will be able to break this cipher in no time at all. The average letter frequencies of English letters are well known. The most common letter is E, which occurs about 13 percent of the time. Here are the average frequencies of the letters.

A	8%	H	4%	O	7%	U	3%
B	<1%	I	7%	P	3%	V	<1%
C	3%	J	<1%	Q	<1%	W	2%
D	4%	K	<1%	R	8%	X	<1%
E	13%	L	4%	S	6%	Y	2%
F	3%	M	3%	T	9%	Z	<1%
G	2%	N	8%				

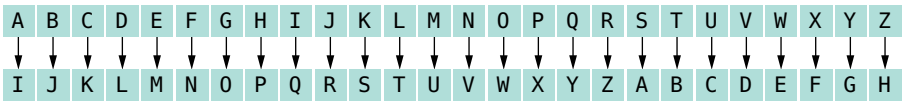
Write a program that reads an input file and displays the letter frequencies in that file. Such a tool will help a code breaker. If the most frequent letters in an encrypted file are H and K, then there is an excellent chance that they are the encryptions of E and T.

Show the result in a table such as the one above, and make sure the columns line up.

■■ **P8.9** *Vigenère cipher*. In order to defeat a simple letter frequency analysis, the Vigenère cipher encodes a letter into one of several cipher letters, depending on its position in the input document. Choose a keyword, for example `TIGER`. Then encode the first letter of the input text like this:



The encoded alphabet is just the regular alphabet shifted to start at `T`, the first letter of the keyword `TIGER`. The second letter is encrypted according to the following map.



The third, fourth, and fifth letters in the input text are encrypted using the alphabet sequences beginning with characters `G`, `E`, and `R`, and so on. Because the key is only five letters long, the sixth letter of the input text is encrypted in the same way as the first.

Write a program that encrypts or decrypts an input text according to this cipher.

■■ **P8.10** *Playfair cipher*. Another way of thwarting a simple letter frequency analysis of an encrypted text is to encrypt *pairs* of letters together. A simple scheme to do this is the Playfair cipher. You pick a keyword and remove duplicate letters from it. Then you fill the keyword, and the remaining letters of the alphabet, into a 5×5 square. (Since there are only 25 squares, I and J are considered the same letter.)

Here is such an arrangement with the keyword `PLAYFAIR`.

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

To encrypt a letter pair, say `AM`, look at the rectangle with corners `A` and `M`:

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

The encoding of this pair is formed by looking at the other two corners of the rectangle, in this case, `FH`. If both letters happen to be in the same row or column, such as `GO`, simply swap the two letters. Decryption is done in the same way.

Write a program that encrypts or decrypts an input text according to this cipher.

■■ **P8.11** Special Topic 8.2 explains how Unicode strings are commonly encoded with the UTF-8 encoding. This exercise explores the encoding in more detail. In UTF-8, a Unicode character is either a byte < 128 (the traditional English alphabet, punctuation marks, and so on, known as ASCII), or a sequence of multiple bytes, where the first byte is ≥ 192, and subsequent bytes are between 128 and 191. Unfortunately, the `char` type may be signed or unsigned in any particular C++ implementation, which means that you cannot reliably compare `char` values to integers above 127. Instead, use the following tests:

```
if ((c & 0x80) == 0) // c is an ASCII character
else if ((c & 0xC0) == 0xC0) // c is the first byte of a multi-byte sequence
else if ((c & 0xC0) == 0x80) // c is a continuation byte of a multi-byte sequence
```

Write a program that reads a UTF-encoded file and prints the number of bytes of each Unicode character in the file. For example, if the file contains

```
San José
```

followed by a newline, your program should print

```
1 1 1 1 1 1 2 1 4 1
```

■■ **P8.12** Write a program that reads a UTF-8 encoded text file (see Special Topic 8.2) and prints all strings enclosed in curly quotes “ ” (U+201C and U+201D).

■■ **P8.13** Write a program that reads a text file and replaces all straight quotes "..." with curly quotes “ ” (U+201C and U+201D). Also replace double dashes -- with a Unicode em dash (U+2014).

■■■ **P8.14** The CSV (or *comma-separated values*) format is commonly used for tabular data. Each table row is a line, with columns separated by commas. Items may be enclosed in quotation marks, and they must be if they contain commas or quotation marks. Quotation marks inside quoted fields are doubled. Here is a line with four fields:

```
1729, San Francisco, "Hello, World", "He asked: ""Quo vadis?"""
```

Provide functions

```
int number_of_fields(string row)
string field(string row, int column)
```

and write a program that tests your functions with a given CSV file.

■■ **P8.15** Find an interesting data set in CSV format (or in spreadsheet format, then use a spreadsheet to save the data as CSV). Using the helper functions from Exercise P8.14, read the data and compute a summary, such as the maximum, minimum, or average of one of the columns.

■■ **P8.16** Write a program that edits an image file and reduces the blue and green values by 30 percent, giving it a “sunset” effect.

© Cay Horstmann.

■ **P8.17** Write a program that edits an image file, turning it into grayscale.

© Cay Horstmann.

Replace each pixel with a pixel that has the same grayness level for the blue, green, and red component. The grayness level is computed by adding 30 percent of the red level, 59 percent of the green level, and 11 percent of the blue level. (The color-sensing cone cells in the human eye differ in their sensitivity for red, green, and blue light.)

■■■ **P8.18** *Junk mail*. Write a program that reads in two files: a *template* and a *database*. The template file contains text and tags. The tags have the form `|1| |2| |3|`... and need to be replaced with the first, second, third, ... field in the current database record. Your program should print out copies of the template with the tag fields replaced with database entries.

A typical database looks like this:

```
Mr.|Harry|Morgan|1105 Torre Ave.|Cupertino|CA|95014
Dr.|John|Lee|702 Ninth Street Apt. 4|San Jose|CA|95109
Miss|Evelyn|Garcia|1101 S. University Place|Ann Arbor|MI|48105
```

And here is a typical template:

```
To:
|1| |2| |3|
|4|
|5|, |6| |7|

Dear |1| |3|:

You and the |3| family may be the lucky winners of $10,000,000 in the C++ compiler
clearinghouse sweepstakes! . . .
```

■■■ **P8.19** Write a program that manipulates three database files. The first file contains the names and telephone numbers of a group of people. The second file contains the names and Social Security numbers of a group of people. The third file contains the Social Security numbers and annual income of a group of people. The groups of people should overlap but need not be completely identical. Your program should ask the user for a telephone number and then print the name, Social Security number, and annual income, if it can determine that information.

■■■ **P8.20** Write a program that prints out a student grade report. There is a file, `classes.txt`, that contains the names of all classes taught at a college, such as

classes.txt

```
CSC1
CSC2
CSC46
CSC151
MTH121
. . .
```

For each class, there is a file with student ID numbers and grades:

csc2.txt

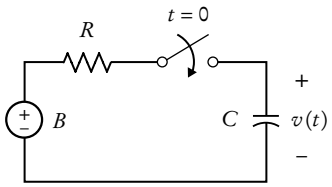
```
11234 A-
12547 B
16753 B+
21886 C
. . .
```

Write a program that asks for a student ID and prints out a grade report for that student, by searching all class files. Here is a sample report

```
Student ID 16753
CSC2 B+
MTH121 C+
CHN1 A
PHY50 A-
```

■■■ Engineering P8.21 After the switch in the figure below closes, the voltage (in volts) across the capacitor is represented by the equation

$$v(t) = B\left(1 - e^{-t/(RC)}\right)$$



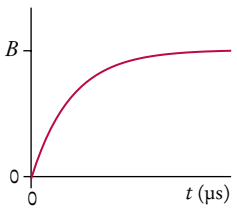
Suppose the parameters of the electric circuit are $B = 12$ volts, $R = 500\ \Omega$, and $C = 0.25\ \mu\text{F}$. Consequently

$$v(t) = 12\left(1 - e^{-0.008t}\right)$$

where t has units of μs. Read a file `params.txt` (provided with your companion code) containing the values for B, R, C, and the starting and ending values for t. Write a file `rc.txt` of values for the time t and the corresponding capacitor voltage $v(t)$, where t goes from the given starting value to the given ending value in 100 steps. In our example, if t goes from 0 to 1,000 μs, the twelfth entry in the output file would be:

```
110  7.02261
```

■■■ Engineering P8.22 The figure below shows a plot of the capacitor voltage from the circuit shown in Exercise P8.21. The capacitor voltage increases from 0 volts to B volts. The "rise time" is defined as the time required for the capacitor voltage to change from $v_1 = 0.05 \times B$ to $v_2 = 0.95 \times B$.



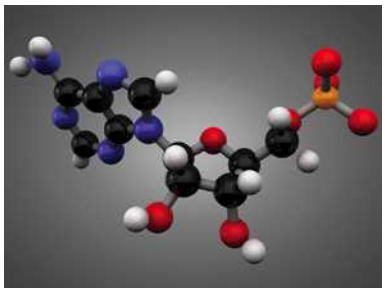
The file `rc.txt` contains a list of values of time t and the corresponding capacitor voltage $v(t)$. A time in μs and the corresponding voltage in volts are printed on the same line. For example, the line

```
110  7.02261
```

indicates that the capacitor voltage is 7.02261 volts when the time is 110 μs. The time is increasing in the data file.

Write a program that reads the file `rc.txt` and uses the data to calculate the rise time. Approximate B by the voltage in the last line of the file, and find the data points that are closest to $0.05 \times B$ and $0.95 \times B$.

■■ Engineering P8.23 Suppose a file contains bond energies and bond lengths for covalent bonds in the following format:



© Chris Dascher/iStockphoto.

Single, double, or triple bond	Bond energy (kJ/mol)	Bond length (nm)
C|C	370	0.154
C||C	680	0.13
C|||C	890	0.12
C|H	435	0.11
C|N	305	0.15
C|O	360	0.14
C|F	450	0.14
C|Cl	340	0.18
O|H	500	0.10
O|O	220	0.15
O|Si	375	0.16
N|H	430	0.10
N|O	250	0.12
F|F	160	0.14
H|H	435	0.074

Write a program that prompts the user to enter a data item in the format of any of the columns, and prints the corresponding data from the other columns in the stored file. The program should print "No matching entries" and exit when the user types an item that does not appear in the table. If input data matches different rows, then return all matching row data. For example, a bond length input of 0.12 should return triple bond C|||C and bond energy 890 kJ/mol *and* single bond N|O and bond energy 250 kJ/mol.

WORKED EXAMPLE 8.1
Looking for for Duplicates

Problem Statement Your task is to write a program that reads a file and prints all lines that contain a repeated word (such as an accidental "the the"), together with their line numbers.

Step 1 Understand the processing task.

Whenever we find a line containing a repeated word, we are to print it like this:

```
 360:bat?' when suddenly, thump! thump! down she came upon a heap of
2103:'Twinkle, twinkle, twinkle, twinkle--' and went on so long that
```

A word is only counted as repeated when it is the same as its predecessor. For example, a line that contains two "the" that are not adjacent would not be reported. The words must be exactly the same. For example, "Twinkle" and "twinkle" don't match.

Step 2 Determine which files you need to read and write.

We only need to read one file, the one with the words. The result is displayed in the console window; no output file is required.

Step 3 Choose a method for obtaining the file names.

This is a student program with console output; we'll ask the user through the console.

Step 4 Choose between line, word, and character-based input.

We definitely want to use line-based input because we need to count line numbers and print the entire line if it contains repeating words.

Step 5 With line-oriented input, extract the required data.

When we have an input line, we still need to extract the words. The easiest approach is to use a string stream, and read words off that stream. We will keep a variable that holds the previous word.

```
For each word in the line
   If word equals previous word
      Found a duplicate.
   Else
      previous word = word
```

Step 6 Place repeatedly occurring tasks into functions.

In this program, there are no repeated tasks. But let's take the bigger view. Scanning lines and printing out the ones that match a particular criterion is a fairly common task. Therefore, let's put the checking for repeated words into a separate function,

```
bool has_repeated_words(string line)
```

Then the basic processing loop becomes very simple:

```
string line;
int line_number = 0;
while (getline(in_file, line))
{
   line_number++;
   if (has_repeated_words(line))
   {
      cout << setw(7) << line_number << ":" << line << endl;
   }
}
```

Step 7 If required, use manipulators to format the output.

There is only one formatting job: to print the line numbers so that the lines line up. Since an integer has no more than 7 digits, we use

```
cout << setw(7) << line_number << ":" << line << endl;
```

Here's the complete program:

worked_example_1/repeated.cpp

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <sstream>
#include <string>

using namespace std;

/**
   Checks whether a given line has repeated words (such as "the the")
   @param line a line of text
   @return true if the line contains repeated words
*/
bool has_repeated_words(string line)
{
   istringstream strm;
   strm.str(line); // This string stream reads the contents of the line
   string previous_word = "";
   string word;
   while (strm >> word) // For each word in the line
   {
      if (word == previous_word) // Found a duplicate
      {
         return true;
      }
      else // Remember this word for the next iteration
      {
         previous_word = word;
      }
   }
   return false;
}

int main()
{
   string filename;
   cout << "Enter filename: ";
   cin >> filename;
   ifstream in_file;
   in_file.open(filename);

   int line_number = 0;
   string line;
   while (getline(in_file, line)) // For each line in the file
   {
      line_number++;
      // Print line if it has repeated words
      if (has_repeated_words(line))
      {
         cout << setw(7) << line_number << ":" << line << endl;
```

```
51          }
52      }
53      return 0;
54  }
```

CHAPTER 9

CLASSES

© Cameron Strathdee/iStockphoto.

CHAPTER GOALS

To understand the concept of encapsulation

To master the separation of interface and implementation

To be able to implement your own classes

To understand how constructors and member functions act on objects

To discover appropriate classes for solving programming problems

To distribute a program over multiple source files

CHAPTER CONTENTS

This chapter introduces you to object-oriented programming, an important technique for writing complex programs. In an object-oriented program, you don't simply manipulate numbers and strings, but you work with objects that are meaningful for your application. Objects with the same behavior (such as the windmills in the photo) are grouped into classes. A programmer provides the desired behavior by specifying and implementing functions for these classes. In this chapter, you will learn how to discover, specify, and implement your own classes, and how to use them in your programs.

9.1 Object-Oriented Programming

You have learned how to structure your programs by decomposing tasks into functions. This is an excellent practice, but experience shows that it does not go far enough. As programs get larger, it becomes increasingly difficult to maintain a large collection of functions.

To overcome this problem, computer scientists invented **object-oriented programming**, a programming style in which tasks are solved by collaborating objects. Each object has its own set of data, together with a set of functions that can act upon the data. (These functions are called **member functions**).

You have already experienced the object-oriented programming style when you used `string` objects or streams such as `cin` and `cout`. For example, you use the `length` and `substr` member functions to work with `string` objects. The `>>` and `<<` operators that you use with streams are also implemented as member functions—see Special Topic 9.1.

A class describes a set of objects with the same behavior.

In C++, a programmer doesn't implement a single object. Instead, the programmer provides a **class**. A class describes a set of objects with the same behavior. For example, the `string` class describes the behavior of all strings. The class specifies how a string stores its characters, which member functions can be used with strings, and how the member functions are implemented.

A class describes a set of objects with the same behavior. For example, a `Car` *class describes all passenger vehicles that have a certain capacity and shape.*

You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its member functions. The implementation is hidden.

When you develop an object-oriented program, you create your own classes that describe what is important in your application. For example, in a student database you might work with `Student` and `Course` classes. Of course, then you must supply member functions for these classes.

Every class has a public interface: a collection of member functions through which the objects of the class can be manipulated.

When you work with an object, you do not know how it is implemented. You need not know how a `string` organizes a character sequence, or how the `cin` object reads input from the console. All you need to know is the **public interface**: the specifications for the member functions that you can invoke. The process of providing a public interface, while hiding the implementation details, is called **encapsulation**.

Encapsulation is the act of providing a public interface and hiding implementation details.

You will want to use encapsulation for your own classes. When you define a class, you will specify the behavior of the public member functions, but you will hide the implementation details. Encapsulation benefits the programmers who use your classes. They can put your classes to work without having to know their implementations, just as you are able to make use of the `string` and stream classes without knowing their internal details.

Encapsulation enables changes in the implementation without affecting users of a class.

Encapsulation is also a benefit for the implementor of a class. When working on a program that is being developed over a long period of time, it is common for implementation details to change, usually to make objects more efficient or more capable. Encapsulation is crucial to enabling these changes. When the implementation is hidden, the implementor is free to make improvements. Because the implementation is hidden, these improvements do not affect the programmers who use the objects.

A driver of an electric car doesn't have to learn new controls even though the car engine is very different. Neither does the programmer who uses an object with an improved implementation—as long as the same member functions are used.

In this chapter, you will learn how to design and implement your own classes in C++, and how to structure your programs in an object-oriented way, using the principle of encapsulation.

9.2 Implementing a Simple Class

In this section, we look at the implementation of a very simple class. You will see how objects store their data, and how member functions access the data of an object. Knowing how a very simple class operates will help you design and implement more complex classes later in this chapter.

Our first example is a class that models a *tally counter*, a mechanical device that is used to count people—for example, to find out how many people attend a concert or board a bus (see Figure 1).

© Jasmin Awad/iStockphoto.

Figure 1 A Tally Counter

Whenever the operator pushes a button, the counter value advances by one. We model this operation with a `count` function. A physical counter has a display to show the current value. In our simulation, we use a `get_value` function instead.

Here is an example of using the `Counter` class. As you know from using classes such as `string` and `ifstream`, you use the class name and a variable name to define an object of the class:

```
Counter tally;
```

In Section 9.6, you will learn how to ensure that objects are properly initialized. For now, we will call a member function to ensure that the object has the correct state:

```
tally.reset();
```

Next, we invoke member functions on our object. First, we invoke the `count` member function twice, simulating two button pushes. Then we invoke the `get_value` member function to check how many times the button was pushed.

```
tally.count();
tally.count();
int result = tally.get_value(); // Sets result to 2
```

We can invoke the member functions again, and the result will be different.

```
tally.count();
tally.count();
result = tally.get_value(); // Sets result to 4
```

As you can see, the `tally` object remembers the effect of prior function calls.

The member functions of a class define the behavior of its objects.

By specifying the member functions of the `Counter` class, we have now specified the **behavior** of `Counter` objects. In order to produce this behavior, each object needs internal data, called the **state** of the object. In this simple example, the object state is very simple. It is the value that keeps track of how many times the counter has been advanced.

An object's data members represent the state of the object.

An object stores its state in **data members**. A data member is a storage location that is present in each object of the class.

You specify data members in the class definition:

```
class Counter
{
public:
   . . .
private:
   int value;
   . . .
};
```

A class definition has a public and private section. As you will see in the following section, we always place data members in the private section.

Each object of a class has its own set of data members.

Each object of a class has its own set of data members. For example, if `concert_counter` and `boarding_counter` are two objects of the `Counter` class, then each object has its own `value` variable (see Figure 2).

The member functions are declared in the public section of the class.

```
class Counter
{
public:
   void reset();
   void count();
   int get_value() const;
private:
   int value;
};
```

The `const` reserved word denotes the fact that calling the `get_value` member function, unlike the other member functions, does not change the object. We will discuss this distinction in more detail in the following section.

Next, let us have a quick look at the implementation of the member functions of the `Counter` class. The `count` member function advances the counter value by 1.

```
void Counter::count()
{
   value = value + 1;
}
```

The function definition can be placed anywhere below the definition of the `Counter` class. The `Counter::` prefix indicates that we are defining the `count` function of the `Counter` class.

Now have another look at the implementation of the `Counter::count` member function. Note how the body of the function increments the data member `value`. Which

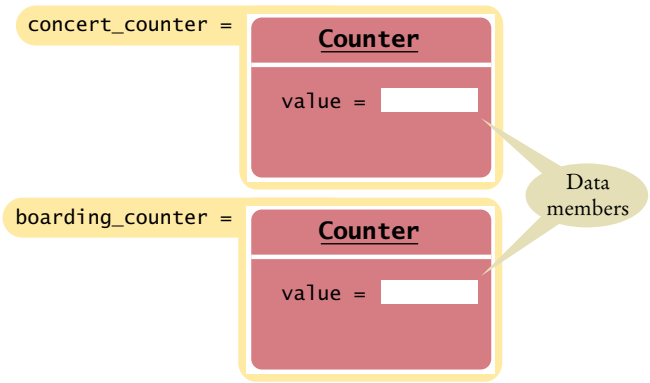


Figure 2 Data Members

data member? The one belonging to the object on which the function is called. For example, consider the call

```
concert_counter.count();
```

A member function can access the data members of the object on which it acts.

This call advances the `value` member of the `concert_counter` object.

The `reset` member function simply sets the value to zero:

```
void Counter::reset()
{
   value = 0;
}
```

Finally, the `get_value` member function returns the current value:

```
int Counter::get_value() const
{
   return value;
}
```

A private data member can only be accessed by the member functions of its own class.

This member function is required so that users of the `Counter` class can find out how often a particular counter has been clicked. A user cannot simply access the `value` data member. That variable has been declared in the private section. The `private` reserved word restricts access to the member functions of the *same class*. For example, the `value` variable can be accessed by the `count` and `get_value` member functions of the `Counter` class but not by a member function of another class. Those other functions need to use the `get_value` member function if they want to find out the counter's value, or the `count` member function if they want to change it.

Private data members are an essential part of encapsulation. They allow a programmer to hide the implementation of a class from a class user.

EXAMPLE CODE See sec02 of your companion code for a program that uses the `Counter` class.

9.3 Specifying the Public Interface of a Class

To define a class, we first need to specify its **public interface**. The public interface of a class consists of all member functions that a user of the class may want to apply to its objects.

Let's consider a simple example. We want to use objects that simulate cash registers. A cashier who rings up a sale presses a key to start the sale, then rings up each item. A display shows the amount owed as well as the total number of items purchased.

In our simulation, we want to carry out the following operations:

- Add the price of an item.
- Get the total amount of all items, and the count of items purchased.
- Clear the cash register to start a new sale.

© James Richey/iStockphoto.

Our first example of a class simulates a cash register.

The interface is specified in the **class definition**, summarized in Syntax 9.1. We will call our class `CashRegister`. (We follow the convention that the name of a programmer-

defined class starts with an uppercase letter, as does each word within the name. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)

Here is the C++ syntax for the `CashRegister` class definition:

```
class CashRegister
{
public:
   void clear();
   void add_item(double price);

   double get_total() const;
   int get_count() const;

private:
   data members—see Section 9.4
};
```

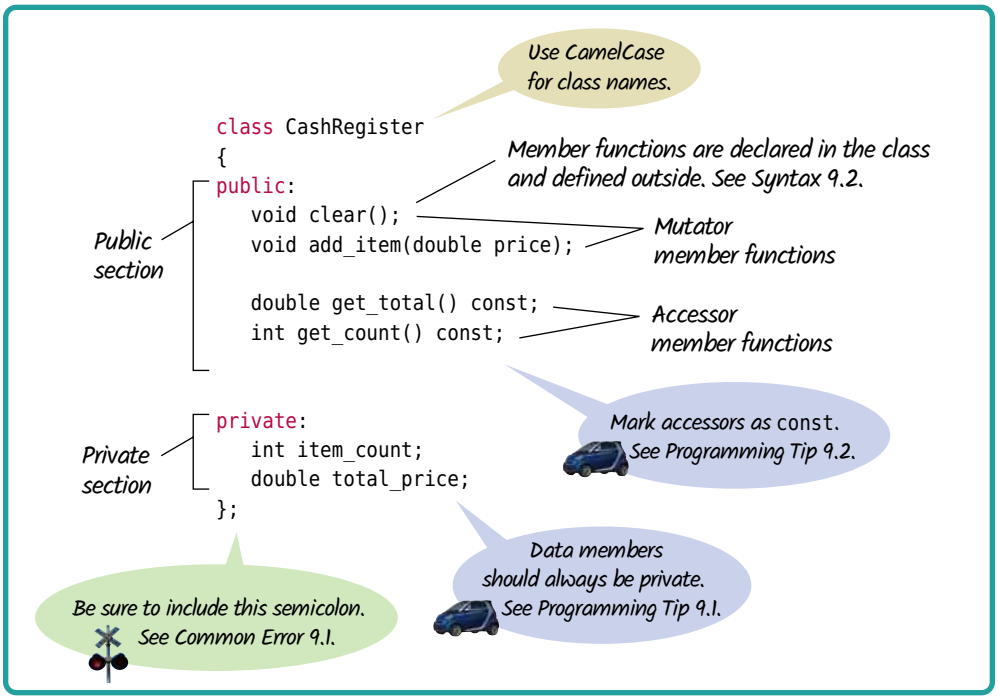
You can use member function declarations and function comments to specify the public interface of a class.

The member functions are declared in the *public section* of the class. Any part of the program can call the member functions. The data members are defined in the *private section* of the class. Only the member functions of the class can access those data members; they are hidden from the remainder of the program.

It is legal to declare the private members before the public section, but in this book, we place the public section first. After all, most programmers reading a class are class users, not implementors, and they are more interested in the public interface than in the private implementation.

The member function declarations look similar to the declarations of regular functions. These declarations do not provide any implementation. You will see in Section 9.5 how to implement the member functions.

Syntax 9.1 Class Definition



A mutator member function changes the object on which it operates.

There are two kinds of member functions, called **mutators** and **accessors**. A mutator is a function that modifies the data members of the object. The `CashRegister` class has two mutators: `clear` and `add_item`. After you call either of these functions, the total amount and item count are changed.

An accessor member function does not change the object on which it operates. Use `const` with accessors.

Accessors just query the object for some information without changing it. The `CashRegister` class has two accessors: `get_total` and `get_count`. Applying either of these functions to a `CashRegister` object simply returns a value and does not modify the object. In C++, you should use the `const` reserved word to mark accessor functions (see Programming Tip 9.2), like this:

```
double get_total() const;
```

Member functions are invoked using the dot notation that you have already seen with string and stream functions:

```
CashRegister register1; // Defines a CashRegister object
register1.clear(); // Invokes a member function
```

Now we know *what* a `CashRegister` object can do, but not *how* it does it. Of course, to use `CashRegister` objects in our programs, we don't need to know. We simply use the public interface. Figure 3 shows the interface of the `CashRegister` class. The mutator functions are shown with arrows pointing inside the private data to indicate that they modify the data. The accessor functions are shown with arrows pointing the other way to indicate that they just read the data.

Figure 3 The Interface of the `CashRegister` Class

EXAMPLE CODE See sec03 of your companion code for the public interface of the `CashRegister` class.

Common Error 9.1

Forgetting a Semicolon

Braces { } are common in C++ code, and usually you do not place a semicolon after the closing brace. However, class definitions always end in };. A common error is to forget that semicolon:

```
class CashRegister
{
public:
   . . .
private:
   . . .
} // Forgot semicolon

int main()
{
```

```
   // Many compilers report the error in this line
   . . .
}
```

This error can be extremely confusing to many compilers. There is syntax, now obsolete but supported for compatibility with old code, to define class types and variables of that type simultaneously. Because the compiler doesn't know that you don't use that obsolete construction, it tries to analyze the code wrongly and ultimately reports an error. Unfortunately, it may report the error *several lines away* from the line in which you forgot the semicolon.

If the compiler reports bizarre errors in lines that you are sure are correct, check that each of the preceding class definitions is terminated by a semicolon.

9.4 Designing the Data Representation

An object holds data members that are accessed by member functions.

An object stores its data in **data members**. These are variables that are declared inside the class.

When implementing a class, you have to determine which data each object needs to store. The object needs to have all the information necessary to carry out any member function call.

Go through all member functions and consider their data requirements. It is a good idea to start with the accessor functions. For example, a `CashRegister` object must be able to return the correct value for the `get_total` function. That means, it must either store all entered prices and compute the total in the function call, or it must store the total.

© migin/iStockphoto.

Like a wilderness explorer who needs to carry all items that may be needed, an object needs to store the data required for any function calls.

Now apply the same reasoning to the `get_count` function. If the cash register stores all entered prices, it can count them in the `get_count` function. Otherwise, you need to have a variable for the count.

The `add_item` function receives a price as an argument, and it must record the price. If the `CashRegister` object stores an array of entered prices, then the `add_item` function appends the price. On the other hand, if we decide to store just the item total and count, then the `add_item` function updates these two variables.

Finally, the `clear` function must prepare the cash register for the next sale, either by emptying the array of prices or by setting the total and count to zero.

We have now discovered two different ways of representing the data that the object needs. Either of them will work, and we have to make a choice. We will choose the simpler one: variables for the total price and the item count. (Other options are explored in Exercises P9.6 and P9.7.)

The data members are defined in the private section of the class definition:

```
class CashRegister
{
```

```
public:
   // See Section 9.3
private:
   int item_count;
   double total_price;
};
```

Every object has its own set of data members.

Every `CashRegister` object has a separate copy of these data members (see Figure 4).

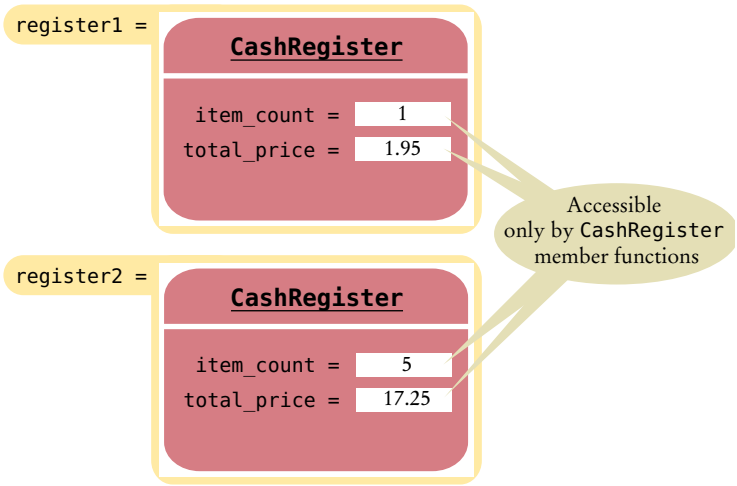


Figure 4 Data Members of `CashRegister` Objects

Because the data members are defined to be private, only the member functions of the class can access them. Programmers using the `CashRegister` class cannot access the data members directly:

```
int main()
{
   . . .
   cout << register1.item_count; // Error—use get_count() instead
   . . .
}
```

These clocks have common behavior, but each of them has a different state. Similarly, objects of a class can have their data members set to different values.

Private data members can only be accessed by member functions of the same class.

All data access must occur through the public interface. Thus, the data members of an object are effectively hidden from the programmer using the class. While it is theoretically possible in C++ to leave data members unencapsulated (by placing them into the public section), this is very uncommon in practice. We will always make all data members private in this book.

EXAMPLE CODE See sec04 of your companion code for the `CashRegister` class with its data members defined.

9.5 Member Functions

The definition of a class declares its member functions. Each member function is defined separately, after the class definition. The following sections show how to define member functions.

9.5.1 Implementing Member Functions

Here is the implementation of the `add_item` function of the `CashRegister` class.

```
void CashRegister::add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}
```

Use the *ClassName::* prefix when defining member functions.

The `CashRegister::` prefix makes it clear that we are defining the `add_item` function of the `CashRegister` class. In C++ it is perfectly legal to have `add_item` functions in other classes as well, and it is important to specify exactly which `add_item` function we are defining. (See Syntax 9.2.) You use the *ClassName*`::add_item` syntax only when *defining* the function, not when calling it. When you call the `add_item` member function, the call has the form *object*`.add_item(...)`.

When defining an accessor member function, supply the reserved word `const` following the closing parenthesis of the parameter list. Here is the `get_count` member function:

```
int CashRegister::get_count() const
{
   return item_count;
}
```

You will find the other member functions with the example program at the end of this section.

9.5.2 Implicit and Explicit Parameters

Whenever you refer to a data member, such as `item_count` or `total_price`, in a member function, it denotes the data member *of the object on which the member function was invoked*. For example, consider the call

```
register1.add_item(1.95);
```

The first statement in the `CashRegister::add_item` function is

```
item_count++;
```

Which `item_count` is incremented? In this call, it is the `item_count` of the `register1` object. (See Figure 5.)

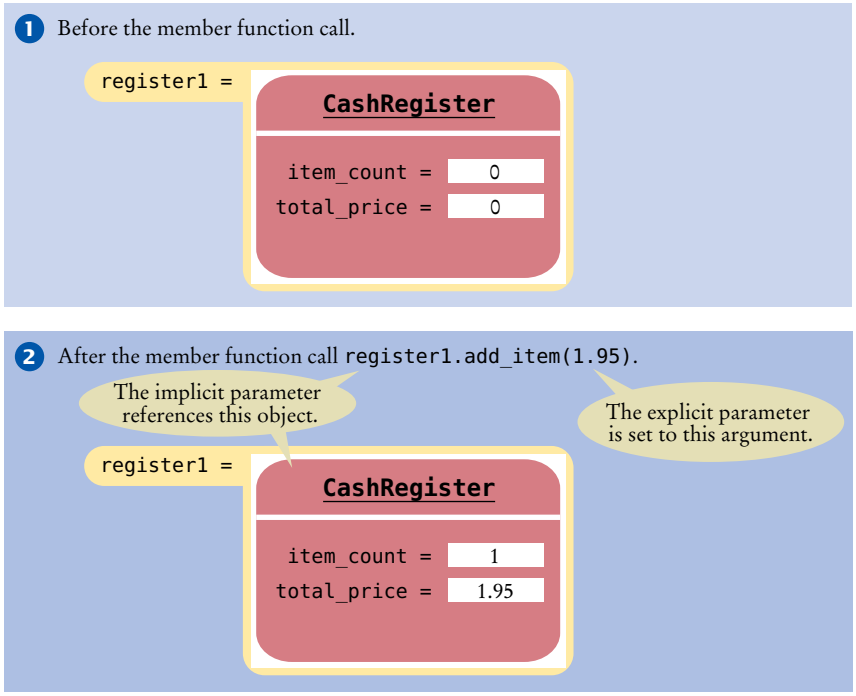


Figure 5 Implicit and Explicit Parameters

The implicit parameter is a reference to the object on which a member function is applied.

When a member function is called on an object, the **implicit parameter** is a reference to that object.

You can think of the code of the `add_item` function like this:

```
void CashRegister::add_item(double price)
{
   implicit parameter.item_count++;
   implicit parameter.total_price = implicit parameter.total_price + price;
}
```

In C++, you do not actually write the implicit parameter in the function definition. For that reason, the parameter is called "implicit".

When an item is added, it affects the data members of the cash register object on which the function is invoked.

Explicit parameters of a member function are listed in the function definition.

In contrast, parameter variables that are explicitly mentioned in the function definition, such as the `price` parameter variable, are called **explicit parameters**. Every member function has exactly one implicit parameter and zero or more explicit parameters.

9.5.3 Calling a Member Function from a Member Function

When calling another member function on the same object, do not use the dot notation.

When one member function calls another member function *on the same object*, you do not use the dot notation. Instead, you simply use the name of the other function. Here is an example. Suppose we want to implement a member function to add multiple instances of the same item. An easy way to implement this function is to repeatedly call the `add_item` function:

```
void CashRegister::add_items(int quantity, double price)
{
   for (int i = 1; i <= quantity; i++)
   {
      add_item(price);
   }
}
```

Here, the `add_item` member function is invoked on the implicit parameter.

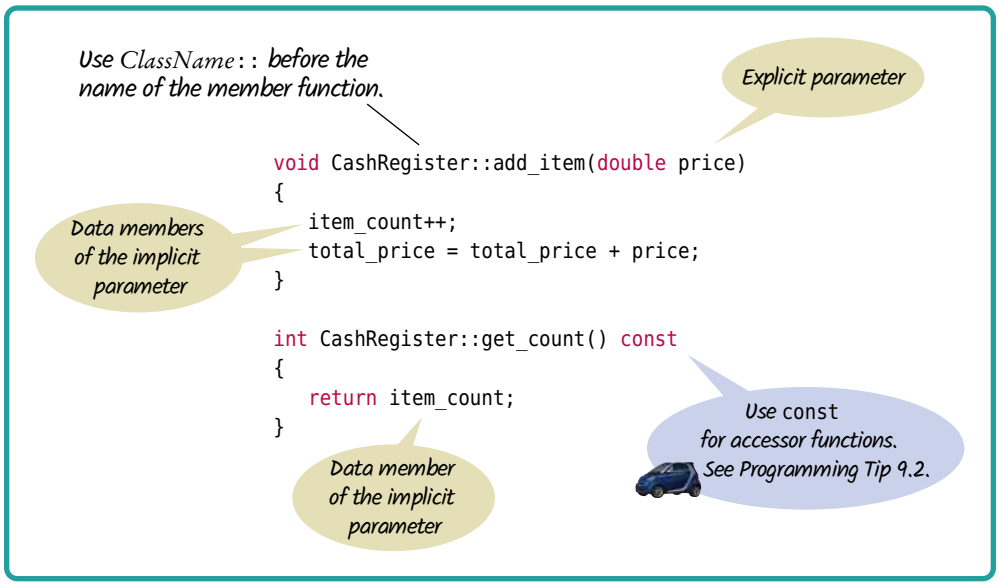
```
for (int i = 1; i <= quantity; i++)
{
   implicit parameter.add_item(price);
}
```

That is the object on which the `add_items` function is invoked. For example, in the call

```
register1.add_items(6, 0.95);
```

the `add_item` function is invoked six times on `register1`.

Syntax 9.2 Member Function Definition



sec05/cashregister.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;

/**
   A simulated cash register that tracks the item count and
   the total amount due.
*/
class CashRegister
{
public:
   /**
      Clears this cash register.
   */
   void clear();
   /**
      Adds an item to this cash register.
      @param price the price of the added item
   */
   void add_item(double price);
   /**
      Gets the price of all items in the current sale.
      @return the total amount
   */
   double get_total() const;
   /**
      Gets the number of items in the current sale.
      @return the item count
   */
   int get_count() const;
private:
   int item_count;
   double total_price;
};

void CashRegister::clear()
{
   item_count = 0;
   total_price = 0;
}

void CashRegister::add_item(double price)
{
   item_count++;
   total_price = total_price + price;
}

double CashRegister::get_total() const
{
   return total_price;
}

int CashRegister::get_count() const
{
   return item_count;
}

```

```
59 /**
60    Displays the item count and total price of a cash register.
61    @param reg the cash register to display
62 */
63 void display(CashRegister reg)
64 {
65    cout << "Item " << reg.get_count() << ": $"
66       << fixed << setprecision(2) << reg.get_total() << endl;
67 }
68
69 int main()
70 {
71    CashRegister register1;
72    register1.clear();
73    register1.add_item(1.95);
74    display(register1);
75    register1.add_item(0.95);
76    display(register1);
77    register1.add_item(2.50);
78    display(register1);
79    return 0;
80 }
```

Program Run

```
Item 1: $1.95
Item 2: $2.90
Item 3: $5.40
```

Programming Tip 9.1

All Data Members Should Be Private; Most Member Functions Should Be Public

It is possible to define data members in the public section of a class, but you should not do that in your own code. Always use encapsulation, with private data members that are manipulated with member functions.

Generally, member functions should be public. However, sometimes you have a member function that is only used as a helper function by other member functions. In that case, you can make the helper function private. Simply declare it in the private section of the class.

Programming Tip 9.2

`const` Correctness

You should declare all accessor functions in C++ with the `const` reserved word. (Recall that an accessor function is a member function that does not modify its implicit parameter.)

For example, suppose you design the following class:

```
class CashRegister
{
   void display(); // Bad style—no const
   . . .
};
```

When you compile your code, no error is reported. But now suppose that another programmer uses your `CashRegister` class in a function:

```
void display_all(const CashRegister registers[])
{
   for (int i = 0; i < NREGISTERS; i++) { registers[i].display(); }
}
```

That programmer is conscientious and declares the `registers` parameter variable as `const`. But then the call `registers[i].display()` will not compile. Because `CashRegister::display` is not tagged as `const`, the compiler suspects that the call `registers[i].display()` may modify `registers[i]`. But the function promised not to modify the `registers` array.

If you write a program with other team members who are conscientious about `const`, it is very important that you do your part as well. You should therefore get into the habit of using `const` with all accessor member functions.

9.6 Constructors

A constructor is called automatically whenever an object is created.

A **constructor** is a member function that initializes the data members of an object. The constructor is automatically called whenever an object is created. By supplying a constructor, you can ensure that all data members are properly set before any member functions act on an object.

To understand the importance of constructors, consider the following statements:

```
CashRegister register1;
register1.add_item(1.95);
int count = register1.get_count(); // May not be 1
```

Here, the programmer forgot to call `clear` before adding items. Therefore, the data members of the `register1` object were initialized with random values. Constructors guarantee that an object is always fully initialized when it is defined.

The name of a constructor is the same as the class name.

The name of a constructor is identical to the name of its class. You declare constructors in the class definition, for example:

```
class CashRegister
{
public:
   CashRegister(); // A constructor
   . . .
};
```

A constructor is like a set of assembly instructions for an object.

Constructors never return values, but you do not use the `void` reserved word when declaring them.

Here is the definition of that constructor:

```
CashRegister::CashRegister()
{
   item_count = 0;
   total_price = 0;
}
```

In the constructor definition, the first `CashRegister` (before the `::`) indicates that we are about to define a member function of the `CashRegister` class. The second `CashRegister` is the name of that member function.

A default constructor has no arguments.

The constructor that you just saw has no arguments. Such a constructor is called a **default constructor**. It is used whenever you define an object and do not specify any parameters for the construction. For example, if you define

```
CashRegister register1;
```

then the default constructor is called. It sets `register1.item_count` and `register1.total_price` to zero.

A class can have multiple constructors.

Many classes have more than one constructor. This allows you to define objects in different ways. Consider for example a `BankAccount` class that has two constructors:

```
class BankAccount
{
public:
   BankAccount(); // Sets balance to 0
   BankAccount(double initial_balance); // Sets balance to initial_balance
   // Member functions omitted
private:
   double balance;
};
```

The compiler picks the constructor that matches the construction arguments.

Both constructors have the same name as the class, `BankAccount`. But the default constructor has no parameter variables, whereas the second constructor has a `double` parameter variable. (This is an example of *overloading*—see Special Topic 9.1.)

When you construct an object, the compiler chooses the constructor that matches the arguments that you supply. For example,

```
BankAccount joes_account;
   // Uses default constructor
BankAccount lisas_account(499.95);
   // Uses BankAccount(double) constructor
```

Be sure to initialize all number and pointer data members in a constructor.

When implementing a constructor, you need to pay particular attention to all data members that are numbers or pointers. These types are not classes and therefore have no constructors. If you have a data member that is an object of a class (such as a `string` object), then that class has a constructor, and the object will be initialized. For example, all `string` objects are automatically initialized to the empty string.

Consider this class:

```
class Item
{
public:
   Item();
   // Additional member functions omitted
private:
   string description;
   double price;
```

```
};
```

In the `Item` constructor, you need to set `price` to `0`, but you need not initialize the `description` data member. It is automatically initialized to the empty string.

As of C++ 11, you can specify default values for data members:

```
class Item
{
public:
   // No constructor
   // Member functions
private:
   string description;
   double price = 0;
};
```

If you do not initialize `price` in a constructor, it is set to zero.

If you do not supply any constructor for a class, the compiler automatically generates a default constructor. The automatically generated default constructor initializes the data members that have default constructors or default values, and leaves the other data members uninitialized.

EXAMPLE CODE See sec06 of your companion code for the complete `CashRegister` class with a single constructor, a `BankAccount` class with two constructors, and an `Item` class with three constructors.

Common Error 9.2

Trying to Call a Constructor

The constructor is invoked only when an object is first created. You cannot invoke it again. For example, you cannot call the constructor to clear an object:

```
CashRegister register1;
. . .
register1.CashRegister(); // Error
```

It is true that the default constructor sets a *new* `CashRegister` object to the cleared state, but you cannot invoke a constructor on an *existing* object.

In this case, you can simply invoke the `clear` member function:

```
register1.clear();
```

Alternatively, you can construct a second cash register and assign it to this one:

```
CashRegister cleared_register;
register = cleared_register;
```

Special Topic 9.1

Overloading

When the same function name is used for more than one function, then the name is **overloaded**. In C++ you can overload function names provided the types of the parameter variables are different. For example, you can define two functions, both called `print`:

```
void print(CashRegister r)
void print(Item i)
```

When the `print` function is called,

```
print(x);
```

the compiler looks at the type of `x`. If `x` is a `CashRegister` object, the first function is called. If `x` is an `Item` object, the second function is called. If `x` is neither, the compiler generates an error.

It is always possible to avoid overloading by giving each function a unique name, such as `print_register` or `print_item`. However, we have no choice with constructors. C++ demands that the name of a constructor equal the name of the class. If a class has more than one constructor, then that name must be overloaded.

In addition to name overloading, C++ also supports *operator overloading*. It is possible to give new meanings to the familiar C++ operators such as `+`, `==`, and `<<`. This is an advanced technique that we will discuss in Chapter 13.

Special Topic 9.2

Initializer Lists

When you construct an object whose data members are themselves objects, those objects are constructed by their class's default constructor. However, if a data member belongs to a class without a default constructor, you need to invoke the data member's constructor explicitly. Here is an example.

This `Item` class has no default constructor:

```
class Item
{
public:
   Item(string item_description, double item_price);
   // No other constructors
   . . .
};
```

This `Order` class has a data member of type `Item`:

```
class Order
{
public:
   Order(string customer_name, string item_description, double item_price);
   . . .
private:
   Item article;
   string customer;
};
```

The `Order` constructor must call the `Item` constructor. That is achieved with an *initializer list*. The initializer list is placed before the opening brace of the constructor. The list starts with a colon and contains names of data members with their construction arguments.

```
Order::Order(string customer_name, string item_description, double item_price)
   : article(item_description, item_price)
{
   customer = customer_name;
}
```

Initializers are separated by commas. They can also initialize a data member as a copy of a value of the same type. Here is another way of writing the `Order` constructor:

```
Order::Order(string customer_name, string item_description, double item_price)
   : article(item_description, item_price), customer(customer_name)
{
}
```

Special Topic 9.3

Universal and Uniform Initialization Syntax

In C++, there are several syntactic variations to initialize a variable, such as

```
double price = 19.25;
int squares[] = { 1, 4, 9, 16 };
BankAccount lisas_account(499.95);
```

C++ 11 introduces a uniform syntax, using braces and no equal sign, like this:

```
double price { 19.25 };
int squares[] { 1, 4, 9, 16 };
BankAccount lisas_account { 499.95 };
```

Use empty braces for default initialization.

```
double balance {}; // Initialized with zero
BankAccount joes_account {}; // Uses default constructor
```

The syntax is universal: it can be used in every context in which variables can be initialized. For example,

```
double* price_pointer = new double { 19.15 };
int result = sum({ 1, 4, 9, 16});
   // where sum has a parameter variable of type vector<int>
```

You can also use the uniform syntax to initialize data members in a constructor. For example, the `Order` constructor of Special Topic 9.2 can be written as

```
Order::Order(string customer_name, string item_description, double item_price)
  : article{item_description, item_price}, customer{customer_name}
{}
```

There is much to like about this universal and uniform syntax. However, as it is quite different from the traditional syntax, it will likely take some time before it is commonly used.

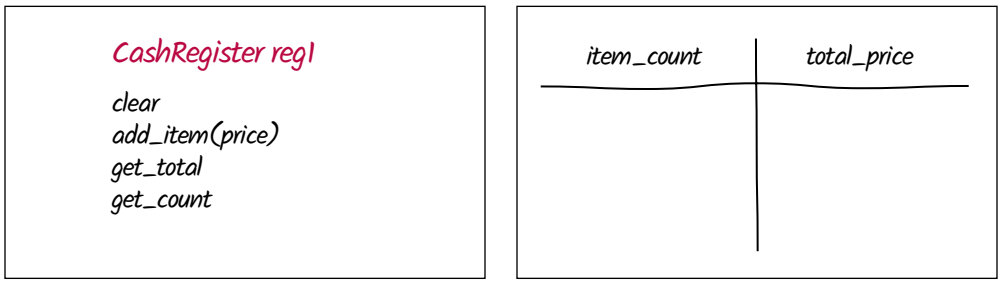
9.7 Problem Solving: Tracing Objects

You have seen how the technique of hand-tracing is useful for understanding how a program works. When your program contains objects, it is useful to adapt the technique so that you gain a better understanding about object data and encapsulation.

Use an index card or a sticky note for each object. On the front, write the member functions that the object can execute. On the back, make a table for the values of the data members.

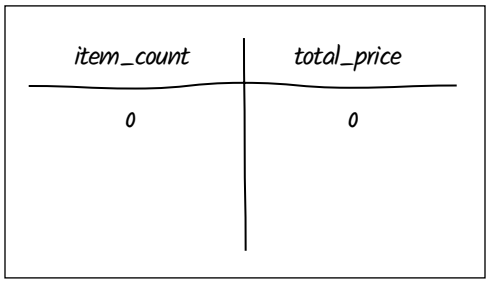
Write the member functions on the front of a card, and the data member values on the back.

Here is a card for a `CashRegister` object:



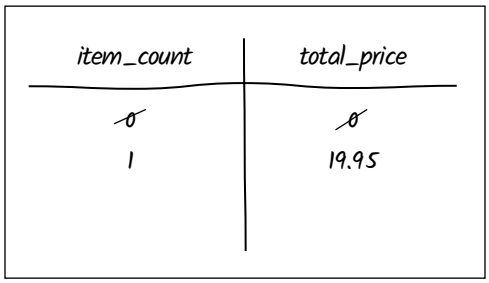
In a small way, this gives you a feel for encapsulation. An object is manipulated through its public interface (on the front of the card), and the data members are hidden in the back.

When an object is constructed, fill in the initial values of the data members.

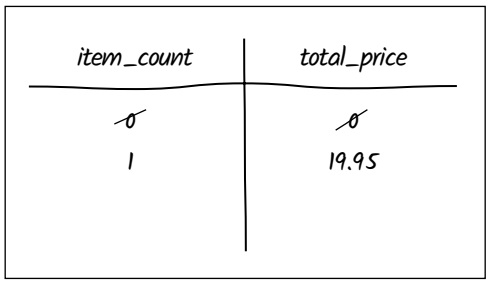


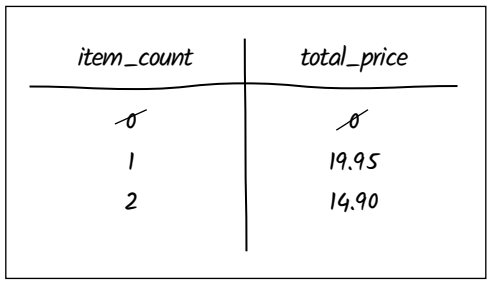
Update the values of the data members when a mutator member function is called.

Whenever a mutator member function is executed, cross out the old values and write the new ones below. Here is what happens after a call to the `add_item` member function:



If you have more than one object in your program, you will have multiple cards, one for each object:





These diagrams are also useful when you design a class. Suppose you are asked to enhance the `CashRegister` class to compute the sales tax. Add a function `get_sales_tax` to the front of the card. Now turn the card over, look over the data members, and ask yourself whether the object has sufficient information to compute the answer. Remember that each object is an autonomous unit. Any data value that can be used in a computation must be

- A data member.
- A function argument.
- A global constant or variable.

To compute the sales tax, we need to know the tax rate and the total of the taxable items. (Food items are usually not subject to sales tax.) We don't have that information available. Let us introduce additional data members for the tax rate and the taxable total. The tax rate can be set in the constructor (assuming it stays fixed for the lifetime of the object). When adding an item, we need to be told whether the item is taxable. If so, we add its price to the taxable total.

For example, consider the following statements.

```
CashRegister reg2(7.5); // 7.5 percent sales tax
reg2.add_item(3.95, false); // not taxable
reg2.add_item(19.95, true); // taxable
```

When you record the effect on a card, it looks like this:

item_count	total_price	taxable_total	tax_rate
~~0~~	~~0~~	~~0~~	7.5
~~1~~	~~3.95~~		
2	23.90	19.95	

With this information, it becomes easy to compute the tax. It is *taxable_total* x *tax_rate* / 100. Tracing the object helped us understand the need for additional data members.

HOW TO 9.1

Implementing a Class

A very common task is to implement a class whose objects can carry out a set of specified actions. This How To walks you through the necessary steps.

Problem Statement As an example, consider a class `Menu`. An object of this class can display a menu such as

```
1) Open new account
2) Log into existing account
3) Help
4) Quit
```

Then the menu waits for the user to supply a value. If the user does not supply a valid value, the menu is redisplayed, and the user can try again.

© Mark Evans/iStockphoto.

Step 1 Get an informal list of the responsibilities of your objects.

Be careful that you restrict yourself to features that are actually required in the problem. With real-world items, such as cash registers or bank accounts, there are potentially dozens of features that might be worth implementing. However, your job is not to faithfully model the real world. You need to determine only those responsibilities that you need for solving your specific problem.

In the case of the menu, you need to

Display the menu.
Get user input.

Now look for hidden responsibilities that aren't part of the problem description. How do objects get created? Which mundane activities need to happen, such as clearing the cash register at the beginning of each sale?

In the menu example, consider how a menu is produced. The programmer creates an empty menu object and then adds options "Open new account", "Help", and so on. There is a hidden responsibility:

Add an option.

Step 2 Specify the public interface.

Turn the list in Step 1 into a set of member functions, with specific types for the parameter variables and the return values. Be sure to mark accessors as `const`. Many programmers find this step simpler if they write out member function calls that are applied to a sample object, like this:

```
Menu main_menu;
main_menu.add_option("Open new account");
// Add more options
int input = main_menu.get_input();
```

Now we have a specific list of member functions.

- `void add_option(string option)`
- `int get_input() const`

What about displaying the menu? There is no sense in displaying the menu without also asking the user for input. However, `get_input` may need to display the menu more than once if the user provides a bad input. Thus, `display` is a good candidate for a private member function.

To complete the public interface, you need to specify the constructors. Ask yourself what information you need in order to construct an object of your class. Sometimes you will want two constructors: one that sets all data members to a default and one that sets them to user-supplied values.

In the case of the menu example, we can get by with a single constructor that creates an empty menu.

Here is the public interface:

```
class Menu
{
public:
   Menu();
   void add_option(string option);
   int get_input() const
private:
   . . .
};
```

Step 3 Document the public interface.

Supply a **documentation comment** for the class, then comment each member function.

```
/**
   A menu that is displayed on a console.
*/
class Menu
{
public:
   /**
      Constructs a menu with no options.
   */
   Menu();
```

```
   /**
      Adds an option to the end of this menu.
      @param option the option to add
   */
   void add_option(string option);

   /**
      Displays the menu, with options numbered starting with 1,
      and prompts the user for input. Repeats until a valid input
      is supplied.
      @return the number that the user supplied
   */
   int get_input() const;
private:
   . . .
};
```

Step 4 Determine data members.

Ask yourself what information an object needs to store to do its job. Remember, the member functions can be called in any order! The object needs to have enough internal memory to be able to process every member function using just its data members and the member function arguments. Go through each member function, perhaps starting with a simple one or an interesting one, and ask yourself what you need to carry out the member function's task. Make data members to store the information that the member function needs.

In the menu example, we clearly need to store the menu options so that the menu can be displayed. How should we store them? As a vector of strings? As one long string? Both approaches can be made to work. We will use a vector here. Exercise E9.6 asks you to implement the other approach.

```
class Menu
{
   . . .
private:
   . . .
   vector<string> options;
};
```

When checking for user input, we need to know the number of menu items. Because we store them in a vector, the number of menu items is simply obtained as the size of the vector. If you stored the menu items in one long string, you might want to keep another data member that stores the menu item count.

Step 5 Implement constructors and member functions.

Implement the constructors and member functions in your class, one at a time, starting with the easiest ones. For example, here is the implementation of the `add_option` member function:

```
void Menu::add_option(string option)
{
   options.push_back(option);
}
```

Here is the `get_input` member function. This member function is a bit more sophisticated. It loops until a valid input has been obtained, and it calls the private `display` member function to display the menu.

```
int Menu::get_input() const
{
   int input = 0;
   do
   {
      display();
      cin >> input;
```

```
   }
   while (input < 1 || input > options.size());
   return input;
}
```

Finally, here is the `display` member function:

```
void Menu::display() const
{
   for (int i = 0; i < options.size(); i++)
   {
      cout << i + 1 << ") " << options[i] << endl;
   }
}
```

The `Menu` constructor is a bit odd. We need to construct a menu with no options. A vector is a class, and it has a default constructor. That constructor does exactly what we want, namely to construct an empty vector. Nothing else needs to be done:

```
Menu::Menu()
{
}
```

If you find that you have trouble with the implementation of some of your member functions, you may need to rethink your choice of data members. It is common for a beginner to start out with a set of data members that cannot accurately describe the state of an object. Don't hesitate to go back and rethink your implementation strategy.

Once you have completed the implementation, compile your class and fix any compiler errors.

Step 6 Test your class.

Write a short tester program and execute it. The tester program should carry out the member function calls that you found in Step 2.

```
int main()
{
   Menu main_menu;
   main_menu.add_option("Open new account");
   main_menu.add_option("Log into existing account");
   main_menu.add_option("Help");
   main_menu.add_option("Quit");
   int input = main_menu.get_input();
   cout << "Input: " << input << endl;
   return 0;
}
```

Program Run

```
1) Open new account
2) Log into existing account
3) Help
4) Quit
5
1) Open new account
2) Log into existing account
3) Help
4) Quit
3
Input: 3
```

EXAMPLE CODE See how_to_1 of your companion code for the complete menu program.

WORKED EXAMPLE 9.1

Implementing a Bank Account Class

Learn how to develop a class that simulates a bank account. See your E-Text or visit `wiley.com/go/bclo3`.

Computing & Society 9.1 Electronic Voting Machines

In the 2000 presidential elections in the United States, votes were tallied by a variety of machines. Some machines processed cardboard ballots into which voters punched holes to indicate their choices (see photo below). When voters were not careful, remains of paper—the now infamous "chads"—were partially stuck in the punch cards, causing votes to be miscounted. A manual recount was necessary, but it was not carried out everywhere due to time constraints and procedural wrangling. The election was very close, and there remain doubts in the minds of many people whether the election outcome would have been different if the voting machines had accurately counted the intent of the voters.

© Peter Nguyen/iStockphoto.

Punch Card Ballot

Subsequently, voting machine manufacturers have argued that electronic voting machines would avoid the problems caused by punch cards or optically scanned forms. In an electronic voting machine, voters indicate their preferences by pressing buttons or touching icons on a computer screen. Typically, each voter is presented with a summary screen for review before casting the ballot. The process is very similar to using an automatic bank teller machine.

It seems plausible that these machines make it more likely that a vote is counted in the same way that the voter intends. However, there has been significant controversy surrounding some types of electronic voting machines. If a machine simply records the votes and prints out the totals after the election has been completed, then how do you know that the machine worked correctly? Inside the machine is a computer that executes a program, and, as you may know from your own experience, programs can have bugs.

In fact, some electronic voting machines do have bugs. There have been isolated cases where machines reported tallies that were impossible. When a machine reports far more or far fewer votes than voters, then it is clear that it malfunctioned. Unfortunately, it is then impossible to find out the actual votes. Over time, one would expect these bugs to be fixed in the software. More insidiously, if the results are plausible, nobody may ever investigate.

Many computer scientists have spoken out on this issue and confirmed that it is impossible, with today's technology, to tell that software is error free and has not been tampered with. Many of them recommend that electronic voting machines should employ a *voter verifiable audit trail*. (A good source of information is `http://verifiedvoting.org`.) Typically, a voter-verifiable machine prints out a ballot. Each voter has a chance to review the printout, and then deposits it in an old-fashioned ballot box. If there is a problem with the electronic equipment, the printouts can be scanned or counted by hand.

© Lisa F. Young/iStockphoto.

Touch Screen Voting Machine

As this book is written, this concept is strongly resisted both by manufacturers of electronic voting machines and by their customers, the cities and counties that run elections. Manufacturers are reluctant to increase the cost of the machines because they may not be able to pass the cost increase on to their customers, who tend to have tight budgets. Election officials fear problems with malfunctioning printers, and some of them have publicly stated that they actually prefer equipment that eliminates bothersome recounts.

What do you think? You probably use an automatic bank teller machine to get cash from your bank account. Do you review the paper record that the machine issues? Do you check your bank statement? Even if you don't, do you put your faith in other people who double-check their balances, so that

the bank won't get away with widespread cheating?

At any rate, is the integrity of banking equipment more important or less important than that of voting machines? Won't every voting process have some room for error and fraud anyway? Is the added cost for equipment, paper, and staff time reasonable to combat a potentially slight risk of malfunction and fraud? Computer scientists cannot answer these questions—an informed society must make these tradeoffs. But, like all professionals, they have an obligation to speak out and give accurate testimony about the capabilities and limitations of computing equipment.

9.8 Problem Solving: Discovering Classes

To discover classes, look for nouns in the problem description.

When you solve a problem using objects and classes, you need to determine the classes required for the implementation. You may be able to reuse existing classes, or you may need to implement new ones. One simple approach for discovering classes and member functions is to look for the nouns and verbs in the problem description. Often, nouns correspond to classes, and verbs correspond to member functions.

Concepts from the problem domain, be it science, business, or a game, often make good classes. Examples are

- `Cannonball`
- `CashRegister`
- `Monster`

Concepts from the problem domain are good candidates for classes.

The name for such a class should be a noun that describes the concept. Other frequently used classes represent system services such as files or menus.

What might not be a good class? If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track. For example, your homework assignment might ask you to write a program that prints paychecks. Suppose you start by trying to design a class `PaycheckProgram`. What would an object of this class do? An object of this class would have to do everything that the homework needs to do. That doesn't simplify anything. A better class would be `Paycheck`. Then your program can manipulate one or more `Paycheck` objects.

In a class scheduling system, potential classes from the problem domain include Class, LectureHall, Instructor, and Student.

Another common mistake, particularly by students who are used to writing programs that consist of functions, is to turn an action into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful member functions of the `Paycheck` class, such as `compute_taxes`, that help you solve the assignment.

When you analyze a problem description, you often find that you need multiple classes. It is then helpful to consider how these classes are related. One of the fundamental relationships between classes is the "aggregation" relationship (which is informally known as the "has-a" relationship).

A car has a motor and tires. In object-oriented design, this "has-a" relationship is called aggregation.

A class aggregates another if its objects contain objects of the other class.

The **aggregation** relationship states that objects of one class contain objects of another class. Consider a quiz that is made up of questions. Since each quiz has one or more questions, we say that the class `Quiz` aggregates the class `Question`. There is a standard notation, called a UML (**Unified Modeling Language**) class diagram, to describe class relationships. In the UML notation, aggregation is denoted by a line with a diamond-shaped symbol (see Figure 6).

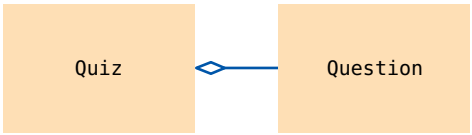


Figure 6 Class Diagram

Finding out about aggregation is very helpful for deciding how to implement classes. For example, when you implement the `Quiz` class, you will want to store the questions of a quiz as a data member. Since a quiz can have any number of questions, you will choose a vector:

```
class Quiz
{
   . . .
private:
   vector<Question> questions;
};
```

In summary, when you analyze a problem description, you will want to carry out these tasks:

- Find the concepts that you need to implement as classes. Often, these will be nouns in the problem description.
- Find the responsibilities of the classes. Often, these will be verbs in the problem description.
- Find relationships between the classes that you have discovered. In this section, we described the aggregation relationship. In the next chapter, you will learn about another important relationship between classes, called inheritance.

Programming Tip 9.3

Make Parallel Vectors into Vectors of Objects

Sometimes, you find yourself using vectors of the same length, each of which stores a part of what conceptually should be an object. In that situation, it is a good idea to reorganize your program and use a single vector whose elements are objects.

For example, suppose an invoice contains a series of item descriptions and prices. One solution is to keep two vectors:

```
vector<string> descriptions;
vector<double> prices;
```

Each of the vectors will have the same length, and the `i`th *slice*, consisting of `descriptions[i]` and `prices[i]`, contains data that needs to be processed together. These vectors are called **parallel vectors** (see Figure 7).

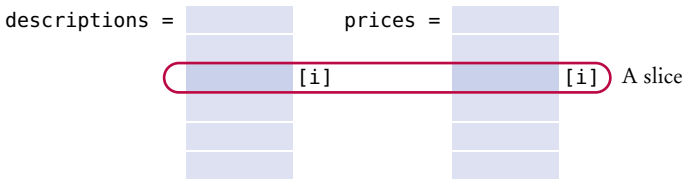


Figure 7 Parallel Vectors

Parallel vectors become a headache in larger programs. The programmer must ensure that the vectors always have the same length and that each slice is filled with values that actually belong together. Moreover, any function that operates on a slice must get all of the vectors as arguments, which is tedious to program.

Avoid parallel vectors by changing them into vectors of objects.

The remedy is simple. Look at the slice and find the *concept* that it represents. Then make the concept into a class. In this example, each slice contains the description and price of an *item;* turn this into a class.

```
class Item
{
public:
   . . .
private:
   string description;
   double price;
};
```

You can now eliminate the parallel vectors and replace them with a single vector:

```
vector<Item> items;
```

Each slot in the resulting vector corresponds to a slice in the set of parallel vectors (see Figure 8).

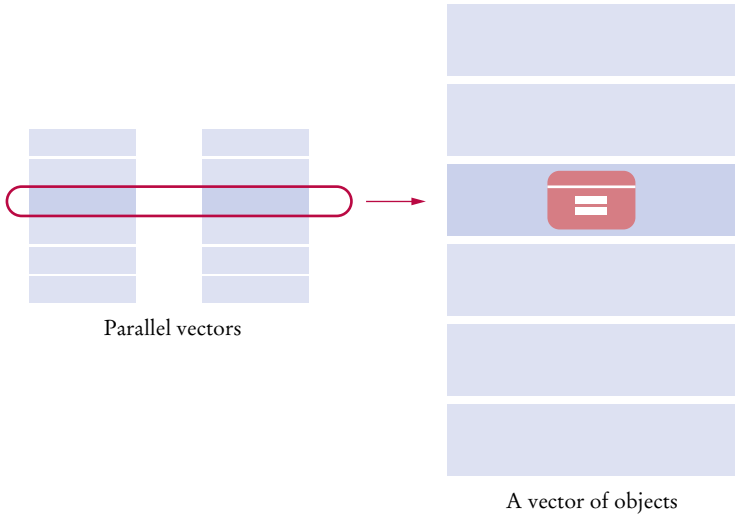


Figure 8 Eliminating Parallel Vectors

9.9 Separate Compilation

The code of complex programs is distributed over multiple files.

When you write and compile small programs, you can place all your code into a single source file. When your programs get larger or you work in a team, that situation changes. You will want to split your code into separate source files. There are two reasons why this split becomes necessary. First, it takes time to compile a file, and it seems silly to wait for the compiler to keep translating code that doesn't change. If your code is distributed over several source files, then only those files that you change need to be recompiled. The second reason becomes apparent when you work with other programmers in a team. It would be very difficult for multiple programmers to edit a single source file simultaneously. Therefore, the program code is broken up so that each programmer is solely responsible for a separate set of files.

If your program is composed of multiple files, some of these files will define data types or functions that are needed in other files. There must be a path of communication between the files. In C++, that communication happens through the inclusion of **header files**.

A header file contains

- Definitions of classes.
- Definitions of constants.
- Declarations of nonmember functions.

The source file contains

- Definitions of member functions.
- Definitions of nonmember functions.

For the `CashRegister` class, you create a pair of files, `cashregister.h` and `cashregister.cpp`, that contain the interface and the implementation, respectively.

Header files contain the definitions of classes and declarations of nonmember functions.

The header file contains the class definition:

sec09/cashregister.h

```
#ifndef CASHREGISTER_H
#define CASHREGISTER_H

/**
   A simulated cash register that tracks the item count and
   the total amount due.
*/
class CashRegister
{
public:
   /**
      Constructs a cash register with cleared item count and total.
   */
   CashRegister();

   /**
      Clears the item count and the total.
   */
   void clear();

   /**
      Adds an item to this cash register.
      @param price the price of this item
   */
   void add_item(double price);

   /**
      @return the total amount of the current sale
   */
   double get_total() const;

   /**
      @return the item count of the current sale
   */
   int get_count() const;

private:
   int item_count;
   double total_price;
};

#endif
```

You include this header file whenever the definition of the `CashRegister` class is required. Because this file is not a standard header file, you must enclose its name in quotes, not <. . .>, when you include it, like this:

```
#include "cashregister.h"
```

Note the set of directives that bracket the header file:

```
#ifndef CASHREGISTER_H
#define CASHREGISTER_H
. . .
#endif
```

Suppose a file includes two header files: `cashregister.h`, and another header file that itself includes `cashregister.h`. The effect of the directives is to skip the file when it is encountered the second time. If we did not have that check, the compiler would complain when it saw the definition for the `CashRegister` class twice. (Sadly, it doesn't check whether the definitions are identical.)

Source files contain function implementations.

The source file for the `CashRegister` class simply contains the definitions of the member functions (including constructors).

Note that the source file `cashregister.cpp` includes its own header file, `cashregister.h`. The compiler needs to know how the `CashRegister` class is defined in order to compile the member functions.

sec09/cashregister.cpp

```
1  #include "cashregister.h"
2
3  CashRegister::CashRegister()
4  {
5     clear();
6  }
7
8  void CashRegister::clear()
9  {
10    item_count = 0;
11    total_price = 0;
12 }
13
14 void CashRegister::add_item(double price)
15 {
16    item_count++;
17    total_price = total_price + price;
18 }
19
20 double CashRegister::get_total() const
21 {
22    return total_price;
23 }
24
25 int CashRegister::get_count() const
26 {
27    return item_count;
28 }
```

Note that the function comments are in the header file, because comments are a part of the interface, not the implementation.

The `cashregister.cpp` file does *not* contain a `main` function. There are many potential programs that might make use of the `CashRegister` class. Each of these programs will need to supply its own `main` function, as well as other functions and classes.

Here is a simple test program that puts the `CashRegister` class to use. Its source file includes the `cashregister.h` header file.

sec09/registerdemo2.cpp

```
#include <iostream>
#include <iomanip>
#include "cashregister.h"

using namespace std;

/**
   Displays the item count and total price of a cash register.
   @param reg the cash register to display
*/
void display(CashRegister reg)
{
   cout << reg.get_count() << " $" << fixed << setprecision(2)
      << reg.get_total() << endl;
}

int main()
{
   CashRegister register1;
   register1.clear();
   register1.add_item(1.95);
   display(register1);
   register1.add_item(0.95);
   display(register1);
   register1.add_item(2.50);
   display(register1);
   return 0;
}
```

To build the complete program, you need to compile both the `registerdemo2.cpp` and `cashregister.cpp` source files (see Figure 9). The details depend on your compiler. For example, with the GNU compiler, you issue the command

```
g++ -o registerdemo registerdemo2.cpp cashregister.cpp
```

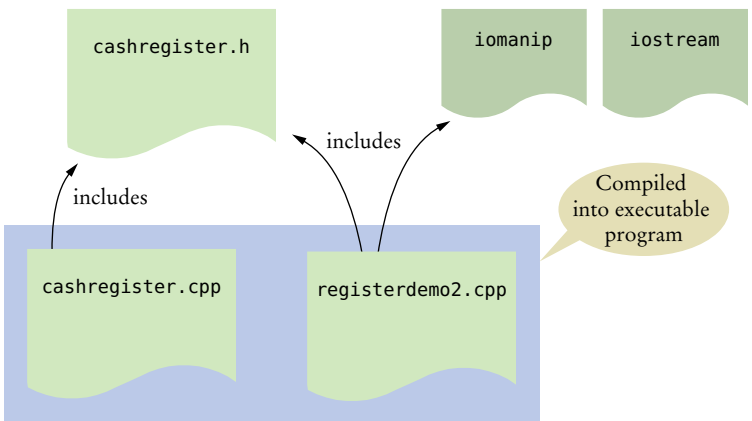


Figure 9 Compiling a Program from Multiple Source Files

You have just seen the simplest and most common case for designing header and source files. There are a few additional technical details that you should know.

- A header file should include all headers that are necessary for defining the class. For example, if a class uses the `string` class, include the `<string>` header as well. Anytime you include a header from the standard library, also include the directive

  ```
  using namespace std;
  ```

  item.h

  ```
  1  #include <string>
  2  using namespace std;
  3
  4  class Item
  5  {
  6     . . .
  7  private:
  8     string description
  9  };
  ```

- Place shared constants into a header file. For example,

  volumes.h

  ```
  6  const double CAN_VOLUME = 0.355;
  ```

- To share a nonmember function, place the function declaration into a header file and the definition of the function into the corresponding source file.

  cube.h

  ```
  8  double cube_volume(double side_length);
  ```

  cube.cpp

  ```
  1  #include "cube.h"
  2
  3  double cube_volume(double side_length)
  4  {
  5     double volume = side_length * side_length * side_length;
  6     return volume;
  7  }
  ```

9.10 Pointers to Objects

It is often desirable to provide shared access to objects. This is achieved by **pointers**. A pointer provides the location of an object. When you know where an object is, you can update it without having to receive and return it. In the following sections, you will see how to use pointers for object sharing. In Chapter 10, you will see another use of pointers: for referring to objects of related classes.

9.10.1 Dynamically Allocating Objects

Suppose that we want to model two store employees who share a cash register, or two friends who share a bank account. First, we allocate the shared object on the **free**

Use the new operator to obtain an object that is located on the free store.

store, a storage area that provides memory on demand. Use the `new` operator together with a constructor invocation:

```
CashRegister* register_pointer = new CashRegister;
BankAccount* lisas_account_pointer = new BankAccount(1000);
```

The new operator returns a pointer to the allocated object.

The type `BankAccount*` denotes a pointer to a `BankAccount` object.

Now we can copy the pointer, without copying the object:

```
BankAccount* joes_account_pointer = lisas_account_pointer;
```

Figure 10 shows the result: two pointers that point to the same object.

When you no longer need an object that is allocated on the free store, be sure to delete it:

```
delete register_pointer;
delete lisas_account_pointer;
```

When an object allocated on the free store is no longer needed, use the delete operator to reclaim its memory.

Now the objects are no longer allocated. Be sure not to use any of the pointers afterwards.

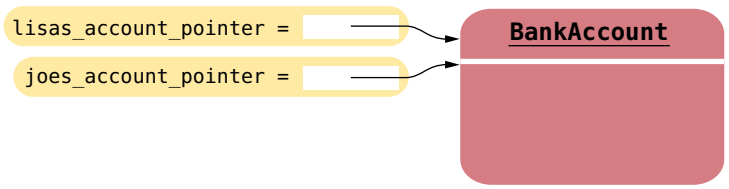


Figure 10 Two Pointers to the Same Object

9.10.2 The -> Operator

Suppose we have a pointer to a `CashRegister` object:

```
CashRegister* register_pointer = new CashRegister;
```

To refer to the cash register, you can use the `*` operator: `*register_pointer` is the `CashRegister` object to which `register_pointer` points. To invoke a member function on that object, you might call

```
(*register_pointer).add_item(1.95);
```

The parentheses are necessary because in C++ the dot operator takes precedence over the `*` operator. The expression without the parentheses would be a compile-time error:

```
*register_pointer.add_item(1.95); // Error—bad syntax
```

Because the dot operator has higher precedence than `*`, the dot would be applied to `register_pointer`, and `*` would be applied to the result of the member function call. Both of these would be syntax errors.

Use the -> operator to invoke a member function through a pointer.

Because calling a member function through a pointer is very common, there is an operator to abbreviate the "follow pointer and call member function" operation. That operator is written `->` and usually pronounced as "arrow". Here is how you use the "arrow" operator:

```
register_pointer->add_item(1.95);
```

This call means: When invoking the `add_item` member function, set the implicit parameter to `*register_pointer` and the explicit parameter to 1.95.

9.10.3 The this Pointer

In a member function, the this pointer points to the implicit parameter.

Each member function has a special parameter variable, called this, which is a pointer to the implicit parameter. For example, consider the CashRegister::add_item function. If you call

```
register1.add_item(1.95)
```

then the this pointer has type CashRegister* and points to the register1 object.

You can use the this pointer inside the definition of a member function. For example, you can implement the add_item function as

```
void CashRegister::add_item(double price)
{
   this->item_count++;
   this->total_price = this->total_price + price;
}
```

Here, the expression this->item_count refers to the item_count data member of the implicit parameter (which is register1.item_count in our example). Some programmers like to use the this pointer in this fashion to make it clear that item_count is a data member and not a variable.

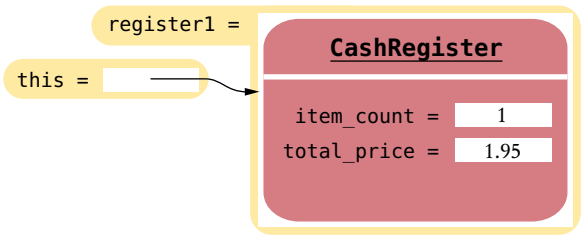


Figure 11 The this Pointer

EXAMPLE CODE See sec10 of your companion code for a program that uses pointers to objects.

9.11 Problem Solving: Patterns for Object Data

When you design a class, you first consider the needs of the programmers who use the class. You provide the member functions that the users of your class will call when they manipulate objects. When you implement the class, you need to come up with the data members for the class. It is not always obvious how to do this. Fortunately, there is a small set of recurring patterns that you can adapt when you design your own classes. We introduce these patterns in the following sections.

9.11.1 Keeping a Total

Many classes need to keep track of a quantity that can go up or down as certain functions are called. Examples:

- A bank account has a balance that is increased by a deposit, decreased by a withdrawal.

- A cash register has a total that is increased when an item is added to the sale, cleared after the end of the sale.
- A car has gas in the tank, which is increased when fuel is added and decreased when the car drives.

A data member for the total is updated in member functions that increase or decrease the total amount.

In all of these cases, the implementation strategy is similar. Keep a data member that represents the current total. For example, for the cash register:

```
double total_price;
```

Locate the member functions that affect the total. There is usually a member function to increase it by a given amount.

```
void add_item(double price)
{
   total_price = total_price + price;
}
```

Depending on the nature of the class, there may be a member function that reduces or clears the total. In the case of the cash register, there is a `clear` member fucntion:

```
void clear()
{
   total_price = 0;
}
```

There is usually a member function that yields the current total. It is easy to implement:

```
double get_total()
{
   return total_price;
}
```

All classes that manage a total follow the same basic pattern. Find the member functions that affect the total and provide the appropriate code for increasing or decreasing it. Find the member functions that report or use the total, and have those member functions read the current total.

9.11.2 Counting Events

A counter that counts events is incremented in member functions that correspond to the events.

You often need to count how often certain events occur in the life of an object. For example:

- In a cash register, you want to know how many items have been added in a sale.
- A bank account charges a fee for each transaction; you need to count them.

Keep a counter, such as

```
int item_count;
```

Increment the counter in those member functions that correspond to the events that you want to count.

```
void add_item(double price)
{
   total_price = total_price + price;
   item_count++;
}
```

You may need to clear the counter, for example at the end of a sale or a statement period.

```
void clear()
{
   total_price = 0;
   item_count = 0;
}
```

There may or may not be a member function that reports the count to the class user. The count may only be used to compute a fee or an average. Find out which member functions in your class make use of the count, and read the current value in those member functions.

9.11.3 Collecting Values

An object can collect other objects in an array or vector.

Some objects collect numbers, strings, or other objects. For example, each multiple-choice question has a number of choices. A cash register may need to store all prices of the current sale.

Use a vector or an array to store the values. (A vector is usually simpler because you won't need to track the number of values.) For example,

```
class Question
{
   . . .
private:
   vector<string> choices;
   . . .
};
```

© paul prescott/iStockphoto.

A shopping cart object needs to manage a collection of items.

In the constructor, we want to initialize the data member to an empty collection. The default constructor of the `vector` class does that automatically. Therefore, the constructor has no explicit actions:

```
Question::Question()
{
}
```

You need to supply some mechanism for adding values. It is common to provide a member function for appending a value to the collection:

```
void Question::add(string choice)
{
   choices.push_back(choice);
}
```

The user of a `Question` object can call this member function multiple times to add the various choices.

9.11.4 Managing Properties of an Object

A *property* is a value of an object that an object user can set and retrieve. For example, a `Student` object may have a name and an ID.

An object property can be accessed with a getter member function and changed with a setter member function.

Provide a data member to store the property's value and member functions to get and set it:

```
class Student
{
public:
   string get_name() const;
   void set_name(string new_name);
private:
   string name;
   . . .
};
```

It is common to add error checking to the setter member function. For example, we may want to reject a blank name:

```
void Student::set_name(string new_name)
{
   if (new_name.length() > 0) { name = new_name; }
}
```

Some properties should not change after they have been set in the constructor. For example, a student's ID may be fixed (unlike the student's name, which may change). In that case, don't supply a setter member function.

```
class Student
{
public:
   Student(int an_id);
   string get_id() const;
   // No set_id member function
   . . .
private:
   int id;
   . . .
};
```

EXAMPLE CODE See sec11_04 of your companion code for a class with getter and setter member functions.

9.11.5 Modeling Objects with Distinct States

If your object can have one of several states that affect the behavior, supply a data member for the current state.

Some objects have behavior that varies depending on what has happened in the past. For example, a `Fish` object may look for food when it is hungry and ignore food after it has eaten. Such an object would need to remember whether it has recently eaten.

Supply a data member that models the state, together with some constants for the state values:

```
class Fish
{
public:
   const int NOT_HUNGRY = 0;
   cont int SOMEWHAT_HUNGRY = 1;
   const int VERY_HUNGRY = 2;
private:
   int hungry;
   . . .
};
```

© John Alexander/iStockphoto.

If a fish is in a hungry state, its behavior changes.

Determine which member functions change the state. In this example, a fish that has just eaten food won't be hungry. But as the fish moves, it will get hungrier.

```
void Fish::eat()
{
   hungry = NOT_HUNGRY;
   . . .
}

void Fish::move()
{
   . . .
   if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first.

```
void Fish::move()
{
   if (hungry == VERY_HUNGRY)
   {
      Look for food.
   }
   . . .
}
```

EXAMPLE CODE See sec11_05 of your companion code for the Fish class.

9.11.6 Describing the Position of an Object

To model a moving object, you need to store and update its position.

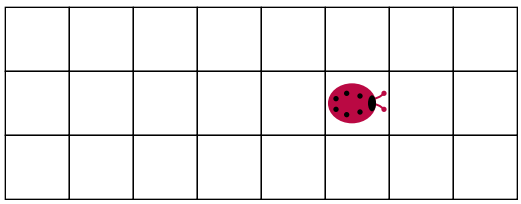
Some objects move around during their lifetime, and they remember their current position. For example,

- A train drives along a track and keeps track of the distance from the terminus.
- A simulated bug living on a grid crawls from one grid location to the next, or makes 90 degree turns to the left or right.
- A cannonball is shot into the air, then descends as it is pulled by the gravitational force.

Such objects need to store their position. Depending on the nature of their movement, they may also need to store their orientation or velocity.

If the object moves along a line, you can represent the position as a distance from a fixed point.

```
double distance_from_terminus;
```



A bug in a grid needs to store its row, column, and direction.

If the object moves in a grid, remember its current location and direction in the grid:

```
int row;
int column;
int direction; // 0 = North, 1 = East, 2 = South, 3 = West
```

When you model a physical object such as a cannonball, you need to track both the position and the velocity, possibly in two or three dimensions. Here we model a cannonball that is shot upward into the air:

```
double z_position;
double z_velocity;
```

There will be member functions that update the position. In the simplest case, you may be told by how much the object moves:

```
void move(double distance_moved)
{
   distance_from_terminus = distance_from_terminus + distance_moved;
}
```

If the movement happens in a grid, you need to update the row or column, depending on the current orientation.

```
void move_one_unit()
{
   if (direction == NORTH) { row--; }
   else if (direction == EAST) { column++; }
   . . .
}
```

Exercise P9.21 shows you how to update the position of a physical object with known velocity.

Whenever you have a moving object, keep in mind that your program will simulate the actual movement in some way. Find out the rules of that simulation, such as movement along a line or in a grid with integer coordinates. Those rules determine how to represent the current position. Then locate the member functions that move the object, and update the positions according to the rules of the simulation.

EXAMPLE CODE See sec11_06 of your companion code for two classes that update position data.

Computing & Society 9.2 Open Source and Free Software

Most companies that produce software regard the source code as a trade secret. After all, if customers or competitors had access to the source code, they could study it and create similar programs without paying the original vendor. For the same reason, customers dislike secret source code. If a company goes out of business or decides to discontinue support for a computer program, its users are left stranded. They are unable to fix bugs or adapt the program to a new operating system. Fortunately, many software packages are distributed as "open source software", giving its users the right to see, modify, and redistribute the source code of a program.

Having access to source code is not sufficient to ensure that software serves the needs of its users. Some companies have created software that spies on users or restricts access to previously purchased books, music, or videos. If that software runs on a server or in an embedded device, the user cannot change its behavior. In the article `http://www.gnu.org/philosophy/free-software-even-more-important.en.html`, Richard Stallman, a famous computer scientist and winner of a MacArthur "genius" grant, describes the "free software movement" that champions the right of users to control what their software does. This is an ethical position that goes beyond using open source for reasons of convenience or cost savings.

Stallman is the originator of the GNU project (`http://gnu.org/gnu/the-gnu-project.html`) that has produced

an entirely free version of a UNIX-compatible operating system: the GNU operating system. All programs of the GNU project are licensed under the GNU General Public License (GNU GPL). The license allows you to make as many copies as you wish, make any modifications to the source, and redistribute the original and modified programs, charging nothing at all or whatever the market will bear. In return, you must agree that your modifications also fall under the license. You must give out the source code to any changes that you distribute, and anyone else can distribute them under the same conditions. The GNU GPL forms a social contract. Users of the software enjoy the freedom to use and modify the software, and in return they are obligated to share any improvements that they make available.

Some commercial software vendors have attacked the GPL as "viral" and "undermining the commercial software sector". Other companies have a more nuanced strategy, producing free or open source software, but charging for support or proprietary extensions. For example, the Java Development Kit is available under the GPL, but companies that need security updates for old versions or other support must pay Oracle.

Open source software sometimes lacks the polish of commercial software because many of the programmers are volunteers who are interested in solving their own problems, not in making a product that is easy to use by everyone. Open source software has been particularly successful in areas that are of interest to programmers, such as the Linux kernel, Web servers, and programming tools.

The open source software community can be very competitive and creative. It is quite common to see several competing projects that take ideas from each other, all rapidly becoming more capable. Having many programmers involved, all reading the source code, often means that bugs tend to get squashed quickly. Eric Raymond describes open source development in his famous article "The Cathedral and the Bazaar" (`http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html`). He writes "Given enough eyeballs, all bugs are shallow".

Courtesy of Richard Stallman published under a CC license.

Richard Stallman, a pioneer of the free source movement.

CHAPTER SUMMARY

Understand the concepts of objects and classes.

- A class describes a set of objects with the same behavior.
- Every class has a public interface: a collection of member functions through which the objects of the class can be manipulated.
- Encapsulation is the act of providing a public interface and hiding implementation details.
- Encapsulation enables changes in the implementation without affecting users of a class.

Understand data members and member functions of a simple class.

- The member functions of a class define the behavior of its objects.
- An object's data members represent the state of the object.
- Each object of a class has its own set of data members.
- A member function can access the data members of the object on which it acts.
- A private data member can only be accessed by the member functions of its own class.

Formulate the public interface of a class in C++.

- You can use member function declarations and function comments to specify the public interface of a class.
- A mutator member function changes the object on which it operates.
- An accessor member function does not change the object on which it operates. Use `const` with accessors.

Choose data members to represent the state of an object.

- An object holds data members that are accessed by member functions.
- Every object has its own set of data members.
- Private data members can only be accessed by member functions of the same class.

Implement member functions of a class.

- Use the *ClassName*:: prefix when defining member functions.
- The implicit parameter is a reference to the object on which a member function is applied.
- Explicit parameters of a member function are listed in the function definition.
- When calling another member function on the same object, do not use the dot notation.

Design and implement constructors.

- A constructor is called automatically whenever an object is created.
- The name of a constructor is the same as the class name.
- A default constructor has no arguments.
- A class can have multiple constructors.
- The compiler picks the constructor that matches the construction arguments.
- Be sure to initialize all number and pointer data members in a constructor.

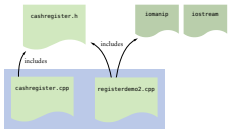
Use the technique of object tracing for visualizing object behavior.

- Write the member functions on the front of a card, and the data member values on the back.
- Update the values of the data members when a mutator member function is called.

Discover classes that are needed for solving a programming problem.

- To discover classes, look for nouns in the problem description.
- Concepts from the problem domain are good candidates for classes.
- A class aggregates another if its objects contain objects of the other class.
- Avoid parallel vectors by changing them into vectors of objects.

Separate the interface and implementation of a class in header and source files.



- The code of complex programs is distributed over multiple files.
- Header files contain the definitions of classes and declarations of nonmember functions.
- Source files contain function implementations.

Use pointers to objects and manage dynamically allocated objects.

- Use the `new` operator to obtain an object that is located on the free store.
- The `new` operator returns a pointer to the allocated object.
- When an object allocated on the free store is no longer needed, use the `delete` operator to reclaim its memory.
- Use the `->` operator to invoke a member function through a pointer.
- In a member function, the `this` pointer points to the implicit parameter.

Use patterns to design the data representation of a class.

- An data member for the total is updated in member functions that increase or decrease the total amount.
- A counter that counts events is incremented in member functions that correspond to the events.
- An object can collect other objects in an array or vector.
- An object property can be accessed with a getter member function and changed with a setter member function.
- If your object can have one of several states that affect the behavior, supply a data member for the current state.
- To model a moving object, you need to store and update its position.

REVIEW EXERCISES

■ **R9.1** List all classes in the C++ library that you have encountered in this book up to this point.

■■ **R9.2** Consider a `Date` class that stores a calendar date such as November 20, 2011. Consider two possible implementations of this class: one storing the day, month, and year, and another storing the number of days since January 1, 1900. Why might an implementor prefer the second version? How does the choice affect the user of the class?

■■ **R9.3** Write a partial C++ class definition that contains the public interface of the `Date` class described in Exercise R9.2. Supply member functions for setting the date to a particular year, month, and day, for advancing the date by a given number of days, and for finding the number of days between this date and another. Pay attention to `const`.

■ **R9.4** What value is returned by the calls `reg1.get_count()`, `reg1.get_total()`, `reg2.get_count()`, and `reg2.get_total()` after these statements?

```
CashRegister reg1;
reg1.clear();
reg1.add_item(0.90);
reg1.add_item(1.95);
CashRegister reg2;
reg2.clear();
reg2.add_item(1.90);
```

■ **R9.5** Consider the `Menu` class in How To 9.1. What is displayed when the following calls are executed?

```
Menu simple_menu;
simple_menu.add_option("Ok");
simple_menu.add_option("Cancel");
int response = simple_menu.get_input();
```

■ **R9.6** What is the *interface* of a class? How does it differ from the *implementation* of a class?

■ **R9.7** What is a member function, and how does it differ from a nonmember function?

■ **R9.8** What is a mutator function? What is an accessor function?

■■ **R9.9** What happens if you forget the `const` in an accessor function? What happens if you accidentally supply a `const` in a mutator function?

■ **R9.10** What is an implicit parameter? How does it differ from an explicit parameter?

■ **R9.11** How many implicit parameters can a member function have? How many implicit parameters can a nonmember function have? How many explicit parameters can a function have?

■ **R9.12** What is a constructor?

■ **R9.13** What is a default constructor? What is the consequence if a class does not have a default constructor?

■ **R9.14** How many constructors can a class have? Can you have a class with no constructors? If a class has more than one constructor, which of them gets called?

■ **R9.15** What is encapsulation? Why is it useful?

■■■ **R9.16** Data members are hidden in the private section of a class, but they aren't hidden very well at all. Anyone can read the private section. Explain to what extent the `private` reserved word hides the private members of a class.

■■■ **R9.17** You can read the `item_count` data member of the `CashRegister` class with the `get_count` accessor function. Should there be a `set_count` mutator function to change it? Explain why or why not.

■ **R9.18** Suppose you implement a `ChessBoard` class. Provide data members to store the pieces on the board and the player who has the next turn.

■■■ **R9.19** In a nonmember function, it is easy to differentiate between calls to member functions and calls to nonmember functions. How do you tell them apart? Why is it not as easy for functions that are called from a member function?

■ **R9.20** Using the object tracing technique described in Section 9.7, trace the program at the end of Section 9.5.

■■ **R9.21** Using the object tracing technique described in Section 9.7, trace the program in Worked Example 9.1.

■■■ **R9.22** Design a modification of the `BankAccount` class in Worked Example 9.1 in which the first five transactions per month are free and a $1 fee is charged for every additional transaction. Provide a member function that deducts the fee at the end of a month. What additional data members do you need? Using the object tracing technique described in Section 9.7, trace a scenario that shows how the fees are computed over two months.

■■ **R9.23** Consider the following problem description:

> *Users place coins in a vending machine and select a product by pushing a button. If the inserted coins are sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the inserted coins are returned to the user.*

What classes should you use to implement it?

■■ **R9.24** Consider the following problem description:

> *Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; however, if they worked more than 40 hours per week, they are paid overtime at 150 percent of their regular wage.*

What classes should you use to implement it?

■■ **R9.25** Consider the following problem description:

> *Customers order products from a store. Invoices are generated to list the items and quantities ordered, payments received, and amounts still due. Products are shipped to the shipping address of the customer, and invoices are sent to the billing address.*

What classes should you use to implement it?

■■ **R9.26** Suppose a vending machine contains products, and users insert coins into the vending machine to purchase products. Draw a UML diagram showing the aggregation relationships between the classes `VendingMachine`, `Coin`, and `Product`.

■■ **R9.27** Suppose an `Invoice` object contains descriptions of the products ordered and the billing and shipping address of the customer. Draw a UML diagram showing the aggregation relationships between the classes `Invoice`, `Address`, `Customer`, and `Product`.

■■ **R9.28** Consider the implementation of a program that plays TicTacToe, with two classes `Player` and `TicTacToeBoard`. Each class implementation is placed in its own C++ source file and each class interface is placed in its own header file. In addition, a source file `game.cpp` contains the code for playing the game and displaying the scores of the players. Describe the contents of each header file, and determine in which files each of them is included.

■ **R9.29** What would happen if the `display` function was moved from `registerdemo2.cpp` to `cashregister.h`? Try it out if you are not sure.

■■ **R9.30** In Exercise P9.17, a `MenuItem` optionally contains a `Menu`. Generally, there are two ways for implementing an "optional" relationship. You can use a pointer that may be set to `nullptr`. Or you may use an indicator, such as a `bool` value, that specifies whether the optional item is present. Why are pointers required in this case?

■■ **R9.31** What is the `this` reference? Why would you use it?

PRACTICE EXERCISES

■ **E9.1** We want to add a button to the tally counter in Section 9.2 that allows an operator to undo an accidental button click. Provide a member function

```
void undo()
```

that simulates such a button. As an added precaution, make sure that the operator cannot click the undo button more often than the count button.

■ **E9.2** Simulate a tally counter that can be used to admit a limited number of people. First, the limit is set with a call

```
void set_limit(int maximum)
```

If the count button was clicked more often than the limit, simulate an alarm by printing out a message "Limit exceeded".

■■ **E9.3** Simulate a circuit for controlling a hallway light that has switches at both ends of the hallway. Each switch can be up or down, and the light can be on or off. Toggling either switch turns the lamp on or off. Provide member functions

```
int get_first_switch_state() // 0 for down, 1 for up
int get_second_switch_state()
int get_lamp_state() // 0 for off, 1 for on
void toggle_first_switch()
void toggle_second_switch()
```

■ **E9.4** Change the public interface of the circuit class of Exercise E9.3 so that it has the following member functions:

```
int get_switch_state(int switch_num)
int get_lamp_state()
void toggle_switch(int switch_num)
```

■ **E9.5** Implement a class `Rectangle`. Provide a constructor to construct a rectangle with a given width and height, member functions `get_perimeter` and `get_area` that compute the perimeter and area, and a member function `void resize(double factor)` that resizes the rectangle by multiplying the width and height by the given factor.

■■■ **E9.6** Reimplement the `Menu` class so that it stores all menu items in one long string. *Hint:* Keep a separate counter for the number of options. When a new option is added, append the option count, the option, and a newline character.

■■ **E9.7** Implement a class `StreetAddress`. An address has a house number, a street, an optional apartment number, a city, a state, and a postal code. Supply two constructors: one with an apartment number and one without. Supply a `print` member function that prints the address with the street (and optional apartment number) on one line and the city, state, and postal code on the next line. Supply a member function `comes_before` that tests whether one address comes before another when the addresses are compared by postal code.

■ **E9.8** Implement a class `Student`. For the purpose of this exercise, a student has a name and a total quiz score. Supply an appropriate constructor and functions `get_name()`, `add_quiz(int score)`, `get_total_score()`, and `get_average_score()`. To compute the latter, you also need to store the *number of quizzes* that the student took.

■■ **E9.9** Modify the `Student` class of Exercise E9.8 to compute grade point averages. Member functions are needed to add a grade and get the current GPA. Specify grades as elements of a class `Grade`. Supply a constructor that constructs a grade from a string, such as `"B+"`. You will also need a function that translates grades into their numeric values (for example, `"B+"` becomes 3.3).

■■ **E9.10** Write a class `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, its position changes by one unit in the current direction. Provide a constructor

```
Bug(int initial_position)
```

and member functions

```
void turn()
void move()
int get_position()
```

Sample usage:

```
Bug bugsy(10);
bugsy.move(); // Now the position is 11
bugsy.turn();
bugsy.move(); // Now the position is 10
```

Your `main` function should construct a bug, make it move and turn a few times, and print the actual and expected positions.

■■ **E9.11** Implement a class `Moth` that models a moth flying along the *x*-axis. The moth has a position, the distance from a fixed origin. When the moth moves toward a point of light, its new position is halfway between its old position and the position of the light source. Supply a constructor

```
Moth(double initial_position)
```

and member functions

```
void move_to_light(double light_position)
double get_position()
```

Your `main` function should construct a moth, move it toward a couple of light sources, and check that the moth's position is as expected.

■■ **E9.12** Implement classes `Person` and `StreetAddress`. Each person has a street address. Provide `display` functions in each class for displaying their contents. Distribute your code over three source files, one for each class, and one containing the `main` function. Construct two `Person` objects and display them.

■■ **E9.13** Modify the `Person` class in Exercise E9.12 so that it contains a pointer to the street address. Construct and display two `Person` objects that share the same `StreetAddress` object.

PROGRAMMING PROJECTS

■■ **P9.1** A microwave control panel has four buttons: one for increasing the time by 30 seconds, one for switching between power levels 1 and 2, a reset button, and a start button. Implement a class that simulates the microwave, with a member function for each button. The member function for the start button should print a message "Cooking for ... seconds at level ...".

■■ **P9.2** A `Person` has a name (just a first name for simplicity) and friends. Store the names of the friends in a string, separated by spaces. Provide a constructor that constructs a person with a given name and no friends. Provide member functions

```
void befriend(Person p)
void unfriend(Person p)
string get_friend_names()
```

■ **P9.3** Add a member function

```
int get_friend_count()
```

to the `Person` class of Exercise P9.2.

■■ **P9.4** Reimplement the `Person` class from Exercise P9.2 so that the collection of a person's friends is stored in a `vector<Person*>`.

■ **P9.5** Write a class `Battery` that models a rechargeable battery. A battery has a constructor

```
Battery(double capacity)
```

where `capacity` is a value measured in milliampere hours. A typical AA battery has a capacity of 2000 to 3000 mAh. The member function

```
void drain(double amount)
```

drains the capacity of the battery by the given amount. The member function

```
void charge()
```

charges the battery to its original capacity.

The member function

```
double get_remaining_capacity()
```

gets the remaining capacity of the battery.

■■ **Business P9.6** Reimplement the `CashRegister` class so that it keeps track of the price of each added item in a `vector<double>`. Remove the `item_count` and `total_price` data members. Reimplement the `clear`, `add_item`, `get_total`, and `get_count` member functions. Add a member function `display_all` that displays the prices of all items in the current sale.

■■ **Business P9.7** Reimplement the `CashRegister` class so that it keeps track of the total price as an integer: the total cents of the price. For example, instead of storing 17.29, store the integer 1729. Such an implementation is commonly used because it avoids the accumulation of roundoff errors. Do not change the public interface of the class.

■■ **Business P9.8** Add a feature to the `CashRegister` class for computing sales tax. The tax rate should be supplied when constructing a `CashRegister` object. Add `add_taxable_item` and `get_total_tax` member functions. (Items added with `add_item` are not taxable.)

■■ **Business P9.9** After closing time, the store manager would like to know how much business was transacted during the day. Modify the `CashRegister` class to enable this functionality. Supply member functions `get_sales_total` and `get_sales_count` to get the total amount of all sales and the number of sales. Supply a member function `reset_sales` that resets any counters and totals so that the next day's sales start from zero.

■ **P9.10** Implement a class `SodaCan` with member functions `get_surface_area()` and `get_volume()`. In the constructor, supply the height and radius of the can.


© Miklos Voros/iStockphoto.

■■ **Business P9.11** Implement a class `Portfolio`. This class has two data members, `checking` and `savings`, of the type `BankAccount` that was developed in Worked Example 9.1 (`worked_example_1/account.cpp` in your code files). Implement four member functions:

```
deposit(double amount, string account)
withdraw(double amount, string account)
transfer(double amount, string account)
print_balances()
```

Here the account string is `"S"` or `"C"`. For the deposit or withdrawal, it indicates which account is affected. For a transfer, it indicates the account from which the money is taken; the money is automatically transferred to the other account.

■■ **P9.12** Define a class `Country` that stores the name of the country, its population, and its area. Using that class, write a program that reads in a set of countries and prints

- The country with the largest area.
- The country with the largest population.
- The country with the largest population density (people per square kilometer or mile).

■■ **Business P9.13** Design a class `Message` that models an e-mail message. A message has a recipient, a sender, and a message text. Support the following member functions:

- A constructor that takes the sender and recipient and sets the time stamp to the current time
- A member function `append` that appends a line of text to the message body
- A member function `to_string` that makes the message into one long string like this: `"From: Harry Hacker\nTo: Rudolf Reindeer\n . . ."`
- A member function `print` that prints the message text. *Hint:* Use `to_string`.

Write a program that uses this class to make a message and print it.

■■ **Business P9.14** Design a class `Mailbox` that stores e-mail messages, using the `Message` class of Exercise P9.13. Implement the following member functions:

```
void Mailbox::add_message(Message m)
Message Mailbox::get_message(int i) const
void Mailbox::remove_message(int i)
```

■■ **P9.15** Implement a `VotingMachine` class that can be used for a simple election. Have member functions to clear the machine state, to vote for a Democrat, to vote for a Republican, and to get the tallies for both parties.

■■ **P9.16** Provide a class for authoring a simple letter. In the constructor, supply the names of the sender and the recipient:

```
Letter(string from, string to)
```

Supply a member function

```
void add_line(string line)
```

to add a line of text to the body of the letter. Supply a member function

```
string get_text()
```

that returns the entire text of the letter. The text has the form:

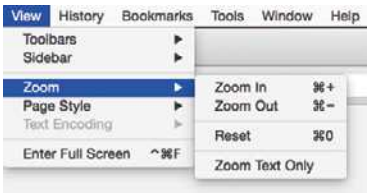
```
Dear recipient name:
blank line
first line of the body
second line of the body
. . .
last line of the body
blank line
Sincerely,
blank line
sender name
```

Also supply a `main` function that prints this letter.

```
Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary
```

Construct an object of the `Letter` class and call `add_line` twice.

■■■ **P9.17** Reimplement the `Menu` class from How To 9.1 to support submenus similar to the ones in a graphical user interface.



A menu contains a sequence of menu items. Each menu item has a name and, optionally, a submenu.

Implement classes `Menu` and `MenuItem`. Supply a function to display a menu and get user input. Simply number the displayed items and have the user enter the number of the selected item. When an item with a submenu is selected, display the submenu. Otherwise simply print a message with the name of the selected item.

Note that there is a circular dependency between the two classes. To break it, first provide a declaration

```
class Menu;
```

Then define the `MenuItem` class and finally define the `Menu` class and the `main` function.

■■■ **Engineering P9.18** Define a class `ComboLock` that works like the combination lock in a gym locker, as shown here. The lock is constructed with a combination—three numbers between `0` and 39. The `reset` function resets the dial so that it points to `0`. The `turn_left` and `turn_right` functions turn the dial by a given number of ticks to the left or right. The

`open` function attempts to open the lock. The lock opens if the user first turned it right to the first number in the combination, then left to the second, and then right to the third.

```
class ComboLock
{
public:
   ComboLock(int secret1, int secret2, int secret3);
   void reset();
   void turn_left(int ticks);
   void turn_right(int ticks);
   bool open() const;
private:
   . . .
};
```

▪▪ Engineering P9.19 Implement a class `Car` with the following properties. A car has a certain fuel efficiency (measured in miles/gallon) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is `0`. Supply a member function `drive` that simulates driving the car for a certain distance, reducing the fuel level in the gas tank, and member functions `get_gas_level`, to return the current fuel level, and `add_gas`, to tank up. Sample usage:

```
Car my_hybrid(50); // 50 miles per gallon
my_hybrid.add_gas(20); // Tank 20 gallons
my_hybrid.drive(100); // Drive 100 miles
cout << my_hybrid.get_gas_level() << endl; // Print fuel remaining
```

▪▪ Engineering P9.20 Write a program that prints all real solutions to the quadratic equation $ax^2 + bx + c = 0$. Read in a, b, c and use the quadratic formula. You may assume that $a \neq 0$. If the *discriminant* $b^2 - 4ac$ is negative, display a message stating that there are no real solutions.

Implement a class `QuadraticEquation` whose constructor receives the coefficients `a`, `b`, `c` of the quadratic equation. Supply member functions `get_solution1` and `get_solution2` that get the solutions, using the quadratic formula, or `0` if no solution exists. The `get_solution1` function should return the smaller of the two solutions.

Supply a function

```
bool has_solutions() const
```

that returns `false` if the discriminant is negative.

▪▪▪ Engineering P9.21 Design a class `Cannonball` to model a cannonball that is fired into the air. A ball has

- An x- and a y-position.
- An x- and a y-velocity.

Supply the following member functions:

- A constructor with an x-position (the y-position is initially `0`)
- A member function `move(double sec)` that moves the ball to the next position (First compute the distance traveled in `sec` seconds, using the current velocities, then update the x- and y-positions; then update the y-velocity by taking into account the gravitational acceleration of –9.81 m/sec^2; the x-velocity is unchanged.) (See Exercise P4.23 for additional details.)
- A member function `shoot` whose parameters are the angle α and initial velocity v (Compute the x-velocity as $v \cos \alpha$ and the y-velocity as $v \sin \alpha$; then keep

calling `move` with a time interval of 0.1 seconds until the y-position is ≤ 0; display the (x, y) position after every move.)

Use this class in a program that prompts the user for the starting angle and the initial velocity. Then call `shoot`.

■■ Engineering P9.22 The colored bands on the top-most resistor shown in the photo below indicate a resistance of 6.2 kΩ ±5%. The resistor tolerance of ±5% indicates the acceptable variation in the resistance. A 6.2 kΩ ±5% resistor could have a resistance as small as 5.89 kΩ or as large as 6.51 kΩ. We say that 6.2 kΩ is the *nominal value* of the resistance and that the actual value of the resistance can be any value between 5.89 kΩ and 6.51 kΩ.

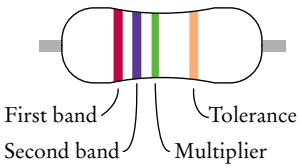
© Maria Toutoudaki/iStockphoto.

Write a C++ program that represents a resistor as a class. Provide a single constructor that accepts values for the nominal resistance and tolerance and then determines the actual value randomly. The class should provide public member functions to get the nominal resistance, tolerance, and the actual resistance.

Write a `main` function for the C++ program that demonstrates that the class works properly by displaying actual resistances for ten 330 Ω ±10% resistors.

■■ Engineering P9.23 In the `Resistor` class from Exercise P9.22, supply a member function that returns a string consisting of the four colors of the "color bands" for the resistance and tolerance. A resistor has four color bands:

- The first band is the first significant digit of the resistance value.
- The second band is the second significant digit of the resistance value.
- The third band is the decimal multiplier.
- The fourth band indicates the tolerance.



Color	Digit	Multiplier	Tolerance
Black	0	$\times 10^0$	—
Brown	1	$\times 10^1$	±1%
Red	2	$\times 10^2$	±2%
Orange	3	$\times 10^3$	—
Yellow	4	$\times 10^4$	—
Green	5	$\times 10^5$	±0.5%

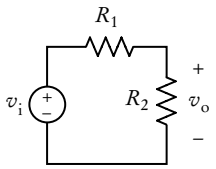
Color	Digit	Multiplier	Tolerance
Blue	6	$\times 10^6$	$\pm 0.25\%$
Violet	7	$\times 10^7$	$\pm 0.1\%$
Gray	8	$\times 10^8$	$\pm 0.05\%$
White	9	$\times 10^9$	—
Gold	—	$\times 10^{-1}$	$\pm 5\%$
Silver	—	$\times 10^{-2}$	$\pm 10\%$
None	—	—	$\pm 20\%$

For example (using the values from the table as a key), a resistor with red, violet, green, and gold bands (left to right) will have 2 as the first digit, 7 as the second digit, a multiplier of 10^5, and a tolerance of $\pm 5\%$, for a resistance of 2,700 kΩ, plus or minus 5%.

■■■ Engineering P9.24 The figure below shows a frequently used electric circuit called a "voltage divider". The input to the circuit is the voltage v_i. The output is the voltage v_o. The output of a voltage divider is proportional to the input, and the constant of proportionality is called the "gain" of the circuit. The voltage divider is represented by the equation

$$G = \frac{v_o}{v_i} = \frac{R_2}{R_1 + R_2}$$

where G is the gain and R_1 and R_2 are the resistances of the two resistors that comprise the voltage divider.



Manufacturing variations cause the actual resistance values to deviate from the nominal values, as described in Exercise P9.22. In turn, variations in the resistance values cause variations in the values of the gain of the voltage divider. We calculate the *nominal value of the gain* using the nominal resistance values and the *actual value of the gain* using actual resistance values.

Write a C++ program that contains two classes, `VoltageDivider` and `Resistor`. The `Resistor` class is described in Exercise P9.22. The `VoltageDivider` class should have two data members that are objects of the `Resistor` class. Provide a single constructor that accepts two `Resistor` objects, nominal values for their resistances, and the resistor tolerance. The class should provide public member functions to get the nominal and actual values of the voltage divider's gain.

Write a `main` function for the program that demonstrates that the class works properly by displaying nominal and actual gain for ten voltage dividers each consisting of 5% resistors having nominal values $R_1 = 250\ \Omega$ and $R_2 = 750\ \Omega$.

WORKED EXAMPLE 9.1

Implementing a Bank Account Class

Problem Statement Write a class that simulates a bank account. Customers can deposit and withdraw funds. If sufficient funds are not available for withdrawal, a $10 overdraft penalty is charged. At the end of the month, interest is added to the account. The interest rate can vary every month.

Step 1 Get an informal list of the responsibilities of your objects.

The following responsibilities are mentioned in the problem statement:

Deposit funds.
Withdraw funds.
Add interest.

There is a hidden responsibility as well. We need to be able to find out how much money is in the account.

Get balance.

Step 2 Specify the public interface.

We need to specify parameter variables and determine which member functions are accessors. To deposit or withdraw money, one needs to know the amount of the deposit or withdrawal:

```
void deposit(double amount);
void withdraw(double amount);
```

To add interest, one needs to know the interest rate that is to be applied:

```
void add_interest(double rate);
```

Clearly, all these member functions are mutators because they change the balance.

Finally, we have

```
double get_balance() const;
```

This function is an accessor because inquiring about the balance does not change it.

Now we move on to constructors. A default constructor makes an account with a zero balance. It can also be useful to supply a constructor with an initial balance.

Here is the complete public interface:

```
class BankAccount
{
public:
   BankAccount();
   BankAccount(double initial_balance);

   void deposit(double amount);
   void withdraw(double amount);
   void add_interest(double rate);

   double get_balance() const;
private:
   . . .
};
```

Step 3 Document the public interface.

```
/**
   A bank account whose balance can be changed by deposits and withdrawals.
*/
class BankAccount
{
public:
   /**
      Constructs a bank account with zero balance.
   */
   BankAccount();

   /**
      Constructs a bank account with a given balance.
      @param initial_balance the initial balance
   */
   BankAccount(double initial_balance);

   /**
      Makes a deposit into this account.
      @param amount the amount of the deposit
   */
   void deposit(double amount);

   /**
      Makes a withdrawal from this account, or charges a penalty if
      sufficient funds are not available.
      @param amount the amount of the withdrawal
   */
   void withdraw(double amount);

   /**
      Adds interest to this account.
      @param rate the interest rate in percent
   */
   void add_interest(double rate);

   /**
      Gets the current balance of this bank account.
      @return the current balance
   */
   double get_balance() const;
private:
   . . .
};
```

Step 4 Determine data members.

Clearly we need to store the bank balance:

```
class BankAccount
{
   . . .
private:
   double balance;
};
```

Do we need to store the interest rate? No—it varies every month, and is supplied as an argument to `add_interest`. What about the withdrawal penalty? The problem description states that it is a fixed $10, so we need not store it. If the penalty could vary over time, as is the case with

most real bank accounts, we would need to store it somewhere (perhaps in a `Bank` object), but it is not our job to model every aspect of the real world.

Step 5 Implement constructors and member functions.

Let's start with a simple one:

```
double BankAccount::get_balance() const
{
   return balance;
}
```

The `deposit` member function is a bit more interesting:

```
void BankAccount::deposit(double amount)
{
   balance = balance + amount;
}
```

The `withdraw` member function needs to charge a penalty if sufficient funds are not available:

```
void BankAccount::withdraw(double amount)
{
   const double PENALTY = 10;
   if (amount > balance)
   {
      balance = balance - PENALTY;
   }
   else
   {
      balance = balance - amount;
   }
}
```

Finally, here is the `add_interest` member function. We compute the interest and then simply call the `deposit` member function to add the interest to the balance:

```
void BankAccount::add_interest(double rate)
{
   double amount = balance * rate / 100;
   deposit(amount);
}
```

The constructors are once again fairly simple:

```
BankAccount::BankAccount()
{
   balance = 0;
}

BankAccount::BankAccount(double initial_balance)
{
   balance = initial_balance;
}
```

This finishes the implementation (see `ch09/worked_example_1/account.cpp` in your companion code).

Step 6 Test your class.

Here is a simple test program that exercises all member functions:

```
int main()
{
   BankAccount harrys_account(1000);
   harrys_account.deposit(500); // Balance is now $1500
   harrys_account.withdraw(2000); // Balance is now $1490
```

```
   harrys_account.add_interest(1); // Balance is now $1490 + 14.90
   cout << fixed << setprecision(2)
      << harrys_account.get_balance() << endl;
   return 0;
}
```

Program Run

```
1504.90
```

CHAPTER 10

INHERITANCE

© Lisa Thornberg/iStockphoto.

CHAPTER GOALS

To understand the concepts of inheritance and polymorphism

To learn how to inherit and override member functions

To be able to implement constructors for derived classes

To be able to design and use virtual functions

CHAPTER CONTENTS

Objects from related classes usually share common behavior. For example, shovels, rakes, and clippers all perform gardening tasks. In this chapter, you will learn how the notion of inheritance expresses the relationship between specialized and general classes. By using inheritance, you will be able to share code between classes and provide services that can be used by multiple classes.

10.1 Inheritance Hierarchies

A derived class inherits data and behavior from a base class.

In object-oriented design, **inheritance** is a relationship between a more general class (called the **base class**) and a more specialized class (called the **derived class**). The derived class inherits data and behavior from the base class. For example, consider the relationships between different kinds of vehicles depicted in Figure 1.

Cars share the common traits of all vehicles, such as the ability to transport people from one place to another. We say that the class `Car` *inherits* from the class `Vehicle`. In this relationship, the `Vehicle` class is the base class and the `Car` class is the derived class.

Informally, the inheritance relationship is called the *is-a* relationship. Contrast this relationship with the *has-a* relationship that we discussed in Section 9.8. Every car *is a* vehicle. Every vehicle *has an* engine.

The inheritance relationship is very powerful because it allows us to reuse algorithms with objects of different classes. Suppose we have an algorithm that

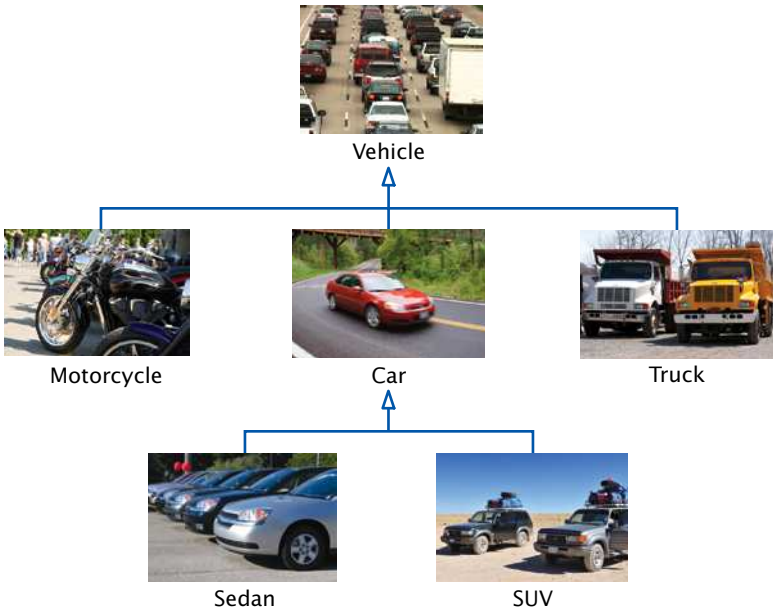


Figure 1 An Inheritance Hierarchy of Vehicle Classes

You can always use a derived-class object in place of a base-class object.

manipulates a `Vehicle` object. Because a car is a special kind of vehicle, we can supply a `Car` object to such an algorithm, and it will work correctly. This is an example of the **substitution principle** that states that you can always use a derived-class object when a base-class object is expected.

The inheritance relationship can give rise to hierarchies where classes get ever more specialized, as shown in Figure 1. The C++ stream classes, shown in Figure 2, are another example of such a hierarchy. Figure 2 uses the UML notation for inheritance where the base and derived class are joined with an arrow that points to the base class.

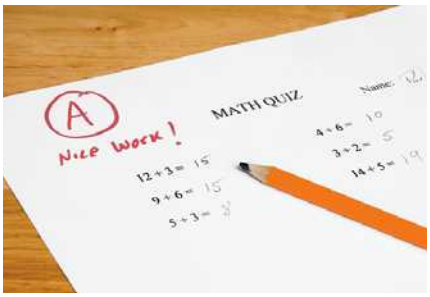
As you can see, an `ifstream` (an input stream that reads from a file) is a special case of an `istream` (an input stream that reads data from any source). If you have an `ifstream`, it can be the argument for a function that expects an `istream`.

```
void process_input(istream& in) // Can call with an ifstream object
```

Why provide a function that processes `istream` objects instead of `ifstream` objects? That function is more useful because it can handle *any* kind of input stream (such as an `istringstream`, which is convenient for testing). This again is the substitution principle at work.

In this chapter, we will consider a simple hierarchy of classes. Most likely, you have taken computer-graded quizzes. A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok; e.g., 1.33 when the actual answer is 4/3)
- Free response



© paul kline/iStockphoto.

We will develop a simple but flexible quiz-taking program to illustrate inheritance.

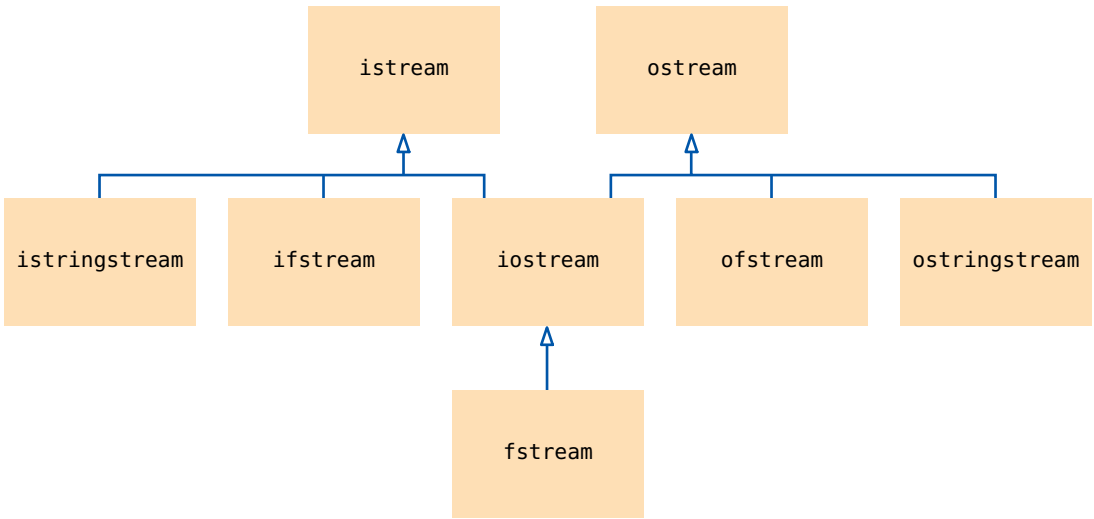


Figure 2 The Inheritance Hierarchy of Stream Classes

Figure 3 shows an inheritance hierarchy for these question types.

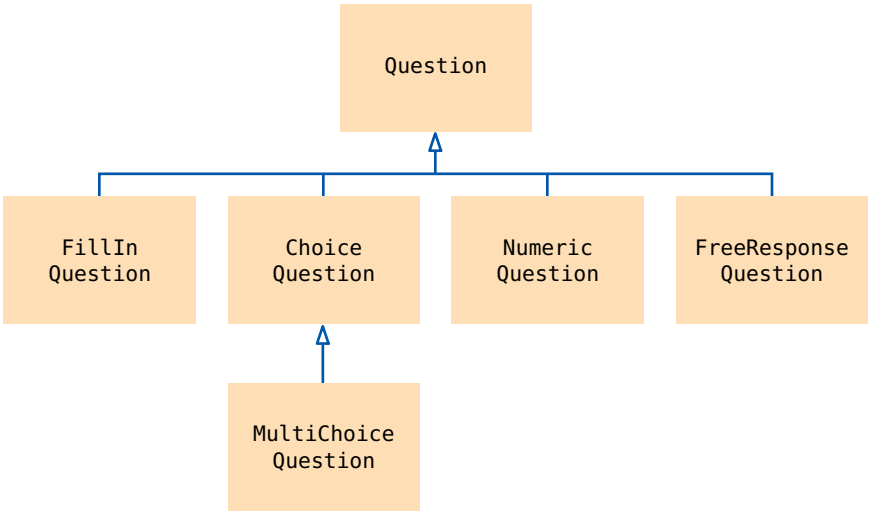


Figure 3 Inheritance Hierarchy of Question Types

At the root of this hierarchy is the `Question` type. A question can display its text, and it can check whether a given response is a correct answer:

```
class Question
{
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   bool check_answer(string response) const;
   void display() const;
private:
   string text;
   string answer;
};
```

How the text is displayed depends on the question type. Later in this chapter, you will see some variations, but the base class simply sends the question text to `cout`. How the response is checked also depends on the question type. As already mentioned, a numeric question might accept approximate answers (see Exercise E10.1). In Exercise E10.3, you will see another way of checking the response. But in the base class, we will simply require that the response match the correct answer exactly.

In the following sections, you will see how to form derived classes that inherit the member functions and data members of this base class.

Here is the implementation of the `Question` class and a simple test program. Note that the `Question` class constructor needs to do no work because the `text` and `answer` data members are automatically set to the empty string. The `boolalpha` stream manipulator in the `main` function causes Boolean values to be displayed as `true` and `false` instead of the default 1 and 0.

sec01/demo.cpp

```
1  #include <iostream>
2  #include <sstream>
```

```
#include <string>

using namespace std;

class Question
{
public:
   /**
      Constructs a question with empty text and answer.
   */
   Question();

   /**
      @param question_text the text of this question
   */
   void set_text(string question_text);

   /**
      @param correct_response the answer for this question
   */
   void set_answer(string correct_response);

   /**
      @param response the response to check
      @return true if the response was correct, false otherwise
   */
   bool check_answer(string response) const;

   /**
      Displays this question.
   */
   void display() const;

private:
   string text;
   string answer;
};

Question::Question()
{
}

void Question::set_text(string question_text)
{
   text = question_text;
}

void Question::set_answer(string correct_response)
{
   answer = correct_response;
}

bool Question::check_answer(string response) const
{
   return response == answer;
}

void Question::display() const
{
   cout << text << endl;
```

```
63 }
64
65 int main()
66 {
67    string response;
68    cout << boolalpha; // Show Boolean values as true, false
69
70    Question q1;
71    q1.set_text("Who was the inventor of C++?");
72    q1.set_answer("Bjarne Stroustrup");
73
74    q1.display();
75    cout << "Your answer: ";
76    getline(cin, response);
77    cout << q1.check_answer(response) << endl;
78
79    return 0;
80 }
```

Program Run

```
Who was the inventor of C++?
Your answer: Bjarne Stroustrup
true
```

10.2 Implementing Derived Classes

In C++, you form a derived class from a base class by specifying what makes the derived class different. You define the member functions that are new to the derived class. The derived class inherits all member functions from the base class, but you can change the implementation if the inherited behavior is not appropriate.

The derived class automatically inherits all data members from the base class. You only define the added data members.

Here is the syntax for the definition of a derived class:

```
class ChoiceQuestion : public Question
{
public:
   New and changed member functions
private:
   Additional data members
};
```

The : symbol denotes inheritance. The reserved word `public` is required for a technical reason (see Common Error 10.1).

Like the manufacturer of a stretch limo, who starts with a regular car and modifies it, a programmer makes a derived class by modifying another class.

Media Bakery.

A `ChoiceQuestion` object differs from a `Question` object in three ways:

- Its objects store the various choices for the answer.
- There is a member function for adding another choice.
- The `display` function of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

When the `ChoiceQuestion` class inherits from the `Question` class, it needs only to spell out these three differences:

```
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   void display() const;
private:
   vector<string> choices;
};
```

Figure 4 shows the layout of a `ChoiceQuestion` object. It inherits the `text` and `answer` data members from the `Question` base object, and it adds an additional data member: the `choices` vector.

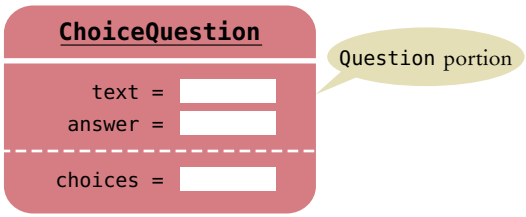


Figure 4 Data Layout of a Derived-Class Object

A derived class can override a base-class function by providing a new implementation.

The derived class inherits all data members and all functions that it does not override.

The `add_choice` function is specific to the `ChoiceQuestion` class. You can only apply it to `ChoiceQuestion` objects, not general `Question` objects. However, the `display` function is a redefinition of a function that exists in the base class; it is redefined to take into account the special needs of the derived class. We say that the derived class **overrides** this function. You will see how in Section 10.3.

In the `ChoiceQuestion` class definition you specify only new member functions and data members. All other member functions and data members of the `Question` class are automatically inherited by the `Question` class. For example, each `ChoiceQuestion` object still has `text` and `answer` data members, and `set_text`, `set_answer`, and `check_answer` member functions.

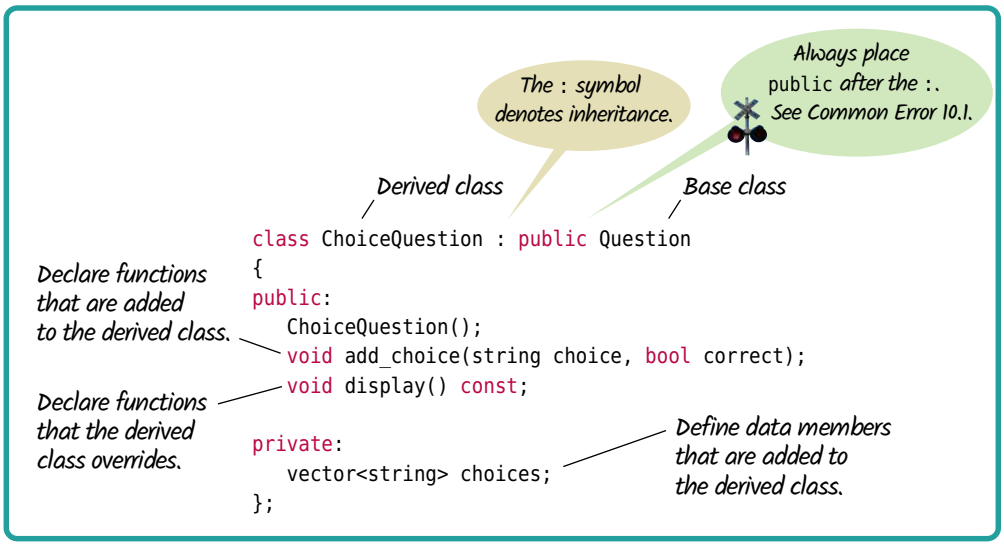
You can call the inherited member functions on a derived-class object:

```
choice_question.set_answer("2");
```

However, the inherited data members are inaccessible. Because these members are private data of the base class, only the base class has access to them. The derived class has no more access rights than any other class.

In particular, the `ChoiceQuestion` member functions cannot directly access the `answer` member. These member functions must use the public interface of the `Question` class to access its private data, just like every other function.

Syntax 10.1 Derived-Class Definition



To illustrate this point, let us implement the `add_choice` member function. The function has two parameters: the choice to be added (which is appended to the vector of choices), and a Boolean value to indicate whether this choice is correct. If it is `true`, set the answer to the current choice number. (We use the `to_string` function to convert the number to a string—see Section 8.4 for details.)

```
void ChoiceQuestion::add_choice(string choice, bool correct)
{
   choices.push_back(choice);
   if (correct)
   {
      // Convert choices.size() to string
      string num_str = to_string(choices.size());
      // Set num_str as the answer
      . . .
   }
}
```

You can't just access the `answer` member in the base class. Fortunately, the `Question` class has a `set_answer` member function. You can call that member function. On which object? The question that you are currently modifying—that is, the implicit parameter of the `ChoiceQuestion::add_choice` function. As you saw in Chapter 9, if you invoke a member function on the implicit parameter, you don't specify the parameter but just write the member function name:

```
set_answer(num_str);
```

The compiler interprets this call as

```
implicit parameter.set_answer(num_str);
```

Common Error 10.1

Private Inheritance

It is a common error to forget the reserved word `public` that must follow the colon after the derived-class name.

```
class ChoiceQuestion : Question // Error
{
   . . .
};
```

The class definition will compile. The `ChoiceQuestion` still inherits from `Question`, but it inherits *privately*. That is, only the member functions of `ChoiceQuestion` get to call member functions of `Question`. Whenever another function invokes a `Question` member function on a `ChoiceQuestion` object, the compiler will flag this as an error:

```
int main()
{
   ChoiceQuestion q;
   . . .
   cout << q.check_answer(response); // Error
}
```

This private inheritance is rarely useful. In fact, it violates the spirit of using inheritance in the first place—namely, to create objects that are usable just like the base-class objects. You should always use public inheritance and remember to supply the `public` reserved word in the definition of the derived class.

Common Error 10.2

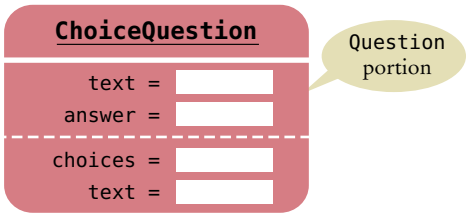
Replicating Base-Class Members

A derived class has no access to the data members of the base class.

```
ChoiceQuestion::ChoiceQuestion(string question_text)
{
   text = question_text; // Error—tries to access private base-class member
}
```

When faced with a compiler error, beginners commonly "solve" this issue by adding *another* data member with the same name to the derived class:

```
class ChoiceQuestion : public Question
{
   . . .
private:
   vector<string> choices;
   string text; // Don't!
}
```



Sure, now the constructor compiles, but it doesn't set the correct text! Such a `ChoiceQuestion` object has two data members, both named `text`. The constructor sets one of them, and the `display` member function displays the other.

Instead of uselessly replicating a base-class data member, you need to call a member function that updates the base-class member, such as the `set_text` function in our example.

Programming Tip 10.1

Use a Single Class for Variation in Values, Inheritance for Variation in Behavior

The purpose of inheritance is to model objects with different *behavior*. When students first learn about inheritance, they have a tendency to overuse it, by creating multiple classes even though the variation could be expressed with a simple data member.

Consider a program that tracks the fuel efficiency of a fleet of cars by logging the distance traveled and the refueling amounts. Some cars in the fleet are hybrids. Should you create a derived class `HybridCar`? Not in this application. Hybrids don't behave any differently than other cars when it comes to driving and refueling. They just have a better fuel efficiency. A single `Car` class with a data member

```
double miles_per_gallon;
```

is entirely sufficient.

However, if you write a program that shows how to repair different kinds of vehicles, then it makes sense to have a separate class `HybridCar`. When it comes to repairs, hybrid cars behave differently from other cars.

Special Topic 10.1

Calling the Base-Class Constructor

Consider the process of constructing a derived-class object. A derived-class constructor can only initialize the data members of the derived class. But the base-class data members also need to be initialized. Unless you specify otherwise, the base-class data members are initialized with the default constructor of the base class.

Unless specified otherwise, the base-class data members are initialized with the default constructor.

In order to specify another constructor, you use an *initializer list*, as described in Special Topic 9.2. Specify the name of the base class and the construction arguments in the initializer list. For example, suppose the `Question` base class had a constructor for setting the question text. Here is how a derived-class constructor could call that base-class constructor:

```
ChoiceQuestion::ChoiceQuestion(string question_text)
   : Question(question_text)
{
}
```

The constructor of a derived class can supply arguments to a base-class constructor.

The derived-class constructor calls the base-class constructor before executing the code inside the `{ }`.

In our example program, we used the default constructor of the base class. However, if a base class has no default constructor, you must use the initializer list syntax.

Syntax 10.2 Constructor with Base-Class Initializer

```
ChoiceQuestion::ChoiceQuestion(string question_text)
   : Question(question_text)
{

}
```

The base-class constructor is called first.

This block can contain additional statements.

If you omit the base-class constructor call, the default constructor is invoked.

10.3 Overriding Member Functions

The derived class inherits the member functions from the base class. If you are not satisfied with the behavior of the inherited member function, you can **override** it by specifying a new implementation in the derived class.

A derived class can inherit a function from the base class, or it can override it by providing another implementation.

Consider the `display` function of the `ChoiceQuestion` class. It needs to override the base-class `display` function in order to show the choices for the answer. Specifically, the derived-class function needs to

- *Display the question text.*
- *Display the answer choices.*

The second part is easy because the answer choices are a data member of the derived class.

```
void ChoiceQuestion::display() const
{
   // Display the question text
   . . .
   // Display the answer choices
   for (int i = 0; i < choices.size(); i++)
   {
      cout << i + 1 << ": " << choices[i] << endl;
   }
}
```

But how do you get the question text? You can't access the `text` member of the base class directly because it is private.

Instead, you can call the `display` function of the base class.

```
void ChoiceQuestion::display() const
{
   // Display the question text
   display(); // Invokes implicit parameter.display()
   // Display the answer choices
   . . .
}
```

However, this won't quite work. Because the implicit parameter of `ChoiceQuestion::display` is of type `ChoiceQuestion`, and there is a function named `display` in the `ChoiceQuestion` class, that function will be called—but that is just the function you are currently writing! The function would call itself over and over.

To display the question text, you must be more specific about which function named `display` you want to call. You want `Question::display`:

```
void ChoiceQuestion::display() const
{
   // Display the question text
   Question::display(); // OK
   // Display the answer choices
   . . .
}
```

Use *BaseClass*::*function* notation to explicitly call a base-class function.

When you override a function, you usually want to *extend* the functionality of the base-class version. Therefore, you often need to invoke the base-class version before extending it. To invoke it, you need to use the *BaseClass*::*function* notation. However, you have no obligation to call the base-class function. Occasionally, a derived class overrides a base-class function and specifies an entirely different functionality.

Here is the complete program that displays a plain `Question` object and a `ChoiceQuestion` object. (The definition of the `Question` class, which you have already seen, is placed into `question.h`, and the implementation is in `question.cpp`.) This example shows how you can use inheritance to form a more specialized class from a base class.

sec03/demo.cpp

```
#include <iostream>
#include <sstream>
#include <vector>
#include "question.h"

class ChoiceQuestion : public Question
{
public:
   /**
      Constructs a choice question with no choices.
   */
   ChoiceQuestion();

   /**
      Adds an answer choice to this question.
      @param choice the choice to add
      @param correct true if this is the correct choice, false otherwise
   */
   void add_choice(string choice, bool correct);

   void display() const;
private:
   vector<string> choices;
};

ChoiceQuestion::ChoiceQuestion()
{
}

void ChoiceQuestion::add_choice(string choice, bool correct)
{
   choices.push_back(choice);
   if (correct)
   {
      // Convert choices.size() to string
      string num_str = to_string(choices.size());
      set_answer(num_str);
   }
}

void ChoiceQuestion::display() const
{
   // Display the question text
   Question::display();
   // Display the answer choices
   for (int i = 0; i < choices.size(); i++)
   {
      cout << i + 1 << ": " << choices[i] << endl;
   }
}

```

```
52 int main()
53
54    string response;
55    cout << boolalpha;
56
57    // Ask a basic question
58
59    Question q1;
60    q1.set_text("Who was the inventor of C++?");
61    q1.set_answer("Bjarne Stroustrup");
62
63    q1.display();
64    cout << "Your answer: ";
65    getline(cin, response);
66    cout << q1.check_answer(response) << endl;
67
68    // Ask a choice question
69
70    ChoiceQuestion q2;
71    q2.set_text("In which country was the inventor of C++ born?");
72    q2.add_choice("Australia", false);
73    q2.add_choice("Denmark", true);
74    q2.add_choice("Korea", false);
75    q2.add_choice("United States", false);
76
77    q2.display();
78    cout << "Your answer: ";
79    getline(cin, response);
80    cout << q2.check_answer(response) << endl;
81
82    return 0;
83 }
```

Program Run

```
Who was the inventor of C++?
Your answer: Bjarne Stroustrup
true
In which country was the inventor of C++ born?
1: Australia
2: Denmark
3: Korea
4: United States
Your answer: 2
true
```

Common Error 10.3

Forgetting the Base-Class Name

A common error in extending the functionality of a base-class function is to forget the base-class name. For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
double Manager::get_salary() const
{
   double base_salary = get_salary();
      // Error—should be Employee::get_salary()
```

```
   return base_salary + bonus;
}
```

Here `get_salary()` refers to the `get_salary` function applied to the implicit parameter of the member function. The implicit parameter is of type `Manager`, and there is a `Manager::get_salary` function, so that function is called. Of course, that is a recursive call to the function that we are writing. Instead, you must specify which `get_salary` function you want to call. In this case, you need to call `Employee::get_salary` explicitly.

Whenever you call a base-class function from a derived-class function with the same name, be sure to give the full name of the function, including the base-class name.

10.4 Virtual Functions and Polymorphism

In the preceding sections you saw one important use of inheritance: to form a more specialized class from a base class. In the following sections you will see an even more powerful application of inheritance: to work with objects whose type and behavior can vary at run time. This variation of behavior is achieved with **virtual functions**. When you invoke a virtual function on an object, the C++ run-time system determines which actual member function to call, depending on the class to which the object belongs.

In the following sections, you will see why you need to use pointers to access objects whose class can vary at run-time, and how a virtual function selects the member function that is appropriate for a given object.

10.4.1 The Slicing Problem

In this section, we will discuss a problem that commonly arises when you work with a collection of objects that belong to different classes in a class hierarchy.

If you look into the `main` function of `sec03/demo.cpp`, you will find that there is some repetitive code to display each question and check the responses. It would be nicer if all questions were collected in an array and one could use a loop to present them to the user:

```
const int QUIZZES = 2;
Question quiz[QUIZZES];
quiz[0].set_text("Who was the inventor of C++?");
quiz[0].set_answer("Bjarne Stroustrup");
ChoiceQuestion cq;
cq.set_text("In which country was the inventor of C++ born?");
cq.add_choice("Australia", false);
. . .
quiz[1] = cq;

for (int i = 0; i < QUIZZES; i++)
{
   quiz[i].display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << quiz[i].check_answer(response) << endl;
}
```

The array `quiz` holds objects of type `Question`. The compiler realizes that a `ChoiceQuestion` is a special case of a `Question`. Thus it permits the assignment from a choice question to a question:

```
quiz[1] = cq;
```

However, a `ChoiceQuestion` object has three data members, whereas a `Question` object has just two. There is no room to store the derived-class data. That data simply gets *sliced away* when you assign a derived-class object to a base-class variable (see Figure 5).

If you run the resulting program, the options are not displayed:

```
Who was the inventor of C++?
Your answer: Bjarne Stroustrup
true
In which country was the inventor of C++ born?
Your answer:
```

When converting a derived-class object to a base class, the derived-class data is sliced away.

This problem is very typical of code that needs to manipulate objects from a mixture of classes in an inheritance hierarchy. Derived-class objects are usually bigger than base-class objects, and objects of different derived classes have different sizes. An array of objects cannot deal with this variation in sizes.

Instead, you need to store the actual objects elsewhere and collect their locations in an array by storing pointers. We will discuss the use of pointers in the next section.

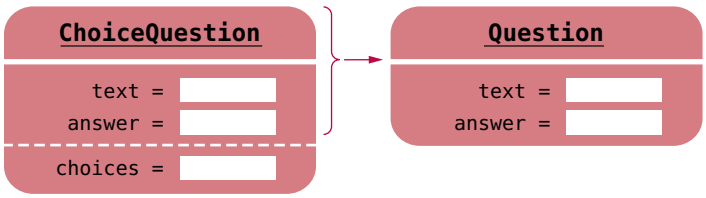


Figure 5 Slicing Away Derived-Class Data

10.4.2 Pointers to Base and Derived Classes

To access objects from different classes in a class hierarchy, use pointers. Pointers to the various objects all have the same size—namely, the size of a memory address—even though the objects themselves may have different sizes.

Here is the code to set up the array of pointers (see Figure 6):

```
Question* quiz[2];
quiz[0] = new Question;
quiz[0]->set_text("Who was the inventor of C++?");
quiz[0]->set_answer("Bjarne Stroustrup");
ChoiceQuestion* cq_pointer = new ChoiceQuestion;
cq_pointer->set_text("In which country was the inventor of C++ born?");
cq_pointer->add_choice("Australia", false);
. . .
quiz[1] = cq_pointer;
```

As the highlighted code shows, you simply define the array to hold pointers, allocate all objects by calling `new`, and use the `->` operator instead of the dot operator.

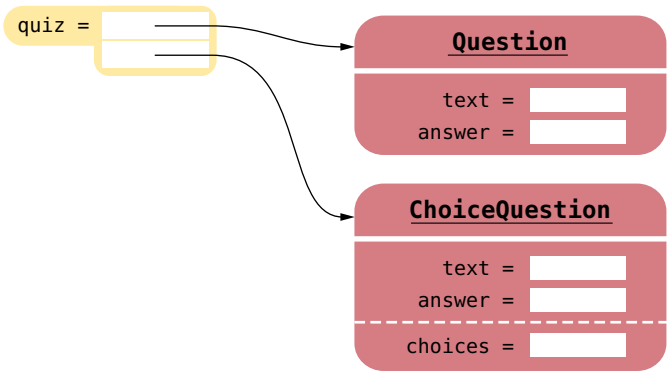


Figure 6 An Array of Pointers Can Store Objects from Different Classes

A derived-class pointer can be converted to a base-class pointer.

Note that the last assignment assigns a derived-class pointer of type `ChoiceQuestion*` to a base-class pointer of type `Question*`. This is perfectly legal. A pointer is the starting address of an object. Because every `ChoiceQuestion` is a special case of a `Question`, the starting address of a `ChoiceQuestion` object is, in particular, the starting address of a `Question` object. The reverse assignment—from a base-class pointer to a derived-class pointer—is an error.

The code to present all questions is

```
for (int i = 0; i < QUIZZES; i++)
{
   quiz[i]->display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << quiz[i]->check_answer(response) << endl;
}
```

Again, note the use of the `->` operator because `quiz[i]` is a pointer.

10.4.3 Virtual Functions

When you collect objects of different classes in a class hierarchy, and then invoke a member function, you want the appropriate member function to be applied. For example, when you call the `display` member function on a `Question*` pointer that happens to point to a `ChoiceQuestion`, you want the choices to be displayed.

For reasons of efficiency, this is not the default in C++. By default, a call

```
quiz[i]->display();
```

always calls `Question::display` because the type of `quiz[i]` is `Question*`.

However, in this case you really want to determine the actual type of the object to which `quiz[i]` points, which can be either a `Question` or a `ChoiceQuestion` object, and then call the appropriate function. In C++, you must alert the compiler that the function call needs to be preceded by the appropriate function selection, which can be a different one for every iteration in the loop. You use the `virtual` reserved word for this purpose:

```
class Question
{
```

```
public:
   Question();
   void set_text(string question_text);
   void set_answer(string correct_response);
   virtual bool check_answer(string response) const;
   virtual void display() const;
private:
   . . .
};
```

The `virtual` reserved word must be used in the *base class*. All functions with the same name and parameter variable types in derived classes are then automatically virtual. However, it is considered good taste to supply the `virtual` reserved word for the derived-class functions as well.

```
class ChoiceQuestion : public Question
{
public:
   ChoiceQuestion();
   void add_choice(string choice, bool correct);
   virtual void display() const;
private:
   . . .
};
```

You do not supply the reserved word `virtual` in the function definition:

```
void Question::display() const // No virtual reserved word
{
   cout << text << endl;
}
```

When a virtual function is called, the version belonging to the actual type of the implicit parameter is invoked.

Whenever a virtual function is called, the compiler determines the type of the implicit parameter in the particular call at run time. The appropriate function for that object is then called. For example, when the `display` function is declared virtual, the call

```
quiz[i]->display();
```

always calls the function belonging to the actual type of the object to which `quiz[i]` points—either `Question::display` or `ChoiceQuestion::display`.

10.4.4 Polymorphism

Polymorphism (literally, "having multiple shapes") describes objects that share a set of tasks and execute them in different ways.

The `quiz` array collects a mixture of both kinds of questions. Such a collection is called **polymorphic** (literally, "of multiple shapes"). Objects in a polymorphic collection have some commonality but are not necessarily of the same type. Inheritance is used to express this commonality, and virtual functions enable variations in behavior.

Virtual functions give programs a great deal of flexibility. The question presentation loop describes only the general mechanism: "Display the question, get a response, and check it". Each object knows on its own how to carry out the specific tasks: "Display the question" and "Check a response".

Using virtual functions makes programs *easily extensible*. Suppose we want to have a new kind of question for calculations, where we are willing to accept an approximate answer. All we need to do is to define a new class `NumericQuestion`, with its own `check_answer` function. Then we can populate the `quiz` array with a mixture of plain questions, choice questions, and numeric questions. The code that presents the questions need not be changed at all! The calls to the virtual functions automatically select the correct member functions of the newly defined classes.

In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

Here is the final version of the quiz program, using pointers and virtual functions. When you run the program, you will find that the appropriate versions of the virtual functions are called. (The files `question.cpp` and `choicequestion.cpp` are included in your book's companion code.)

sec04/question.h

```
#ifndef QUESTION_H
#define QUESTION_H

#include <string>

using namespace std;

class Question
{
public:
   /**
      Constructs a question with empty question and answer.
   */
   Question();

   /**
      @param question_text the text of this question
   */
   void set_text(string question_text);

   /**
      @param correct_response the answer for this question
   */
   void set_answer(string correct_response);

   /**
      @param response the response to check
      @return true if the response was correct, false otherwise
   */
   virtual bool check_answer(string response) const;

   /**
      Displays this question.
```

```
34       */
35       virtual void display() const;
36    private:
37       string text;
38       string answer;
39    };
40
41    #endif
```

sec04/choicequestion.h

```
1     #ifndef CHOICEQUESTION_H
2     #define CHOICEQUESTION_H
3
4     #include <vector>
5     #include "question.h"
6
7     class ChoiceQuestion : public Question
8     {
9     public:
10       /**
11          Constructs a choice question with no choices.
12       */
13       ChoiceQuestion();
14
15       /**
16          Adds an answer choice to this question.
17          @param choice the choice to add
18          @param correct true if this is the correct choice, false otherwise
19       */
20       void add_choice(string choice, bool correct);
21
22       virtual void display() const;
23    private:
24       vector<string> choices;
25    };
26
27    #endif
```

sec04/demo.cpp

```
1     #include <iostream>
2     #include "question.h"
3     #include "choicequestion.h"
4
5     int main()
6     {
7        string response;
8        cout << boolalpha;
9
10       // Make a quiz with two questions
11       const int QUIZZES = 2;
12       Question* quiz[QUIZZES];
13       quiz[0] = new Question;
14       quiz[0]->set_text("Who was the inventor of C++?");
15       quiz[0]->set_answer("Bjarne Stroustrup");
16
17       ChoiceQuestion* cq_pointer = new ChoiceQuestion;
18       cq_pointer->set_text(
19          "In which country was the inventor of C++ born?");
```

```
20    cq_pointer->add_choice("Australia", false);
21    cq_pointer->add_choice("Denmark", true);
22    cq_pointer->add_choice("Korea", false);
23    cq_pointer->add_choice("United States", false);
24    quiz[1] = cq_pointer;
25
26    // Check answers for all questions
27    for (int i = 0; i < QUIZZES; i++)
28    {
29       quiz[i]->display();
30       cout << "Your answer: ";
31       getline(cin, response);
32       cout << quiz[i]->check_answer(response) << endl;
33    }
34
35    return 0;
36 }
```

Programming Tip 10.2

Don't Use Type Tags

Some programmers build inheritance hierarchies in which each object has a tag that indicates its type, commonly a string. They then query that string:

```
if (q->get_type() == "Question")
{
   // Do something
}
else if (q->get_type() == "ChoiceQuestion")
{
   // Do something else
}
```

This is a poor strategy. If a new class is added, then all these queries need to be revised. In contrast, consider the addition of a class `NumericQuestion` to our quiz program. *Nothing* needs to change in that program because it uses virtual functions, not type tags.

Whenever you find yourself adding a type tag to a hierarchy of classes, reconsider and use virtual functions instead.

Common Error 10.4

Slicing an Object

In C++ it is legal to copy a derived-class object into a base-class variable. However, any derived-class information is lost in the process. For example, when a `Manager` object is assigned to a variable of type `Employee`, the result is only the employee portion of the manager data:

```
Manager m;
. . .
Employee e = m; // Holds only the Employee base data of m
```

Any information that is particular to managers is sliced off, because it would not fit into a variable of type `Employee`. To avoid **slicing**, you can use pointers.

The slicing problem commonly occurs when a function has a polymorphic parameter (that is, a parameter that can belong to a base class or a derived class). In that case, the parameter variable must be a pointer or a reference. Consider this example:

```
void ask(Question q) // Error
{
```

```
   q.display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << q.check_answer(response) << endl;
}
```

If you call this function with a `ChoiceQuestion` object, then the parameter variable `q` is initialized with a copy of that object. But `q` is a `Question` object; the derived-class information is sliced away. The simplest remedy is to use a reference:

```
void ask(const Question& q)
```

Now only the *address* is passed to the function. A reference is really a pointer in disguise. No slicing occurs, and virtual functions work correctly.

Common Error 10.5

Failing to Override a Virtual Function

In C++, two functions can have the same name, provided they differ in their parameter types. For example, you can define two member functions called `display` in the `Question` class:

```
class Question
{
public:
   virtual void display() const;
   virtual void display(ostream& out) const;
   . . .
};
```

These are different functions, each with its own implementation. The C++ compiler considers them to be completely unrelated. We say that the `display` name is *overloaded*. This is different from overriding, where a derived class function provides an implementation of a base class function with the same name and the same parameter types.

It is a common error to accidentally provide an overloaded function when you actually mean to override a function. Consider this scary example:

```
class ChoiceQuestion : public Question
{
public:
   void display(); // Does not override Question::display() const
   . . .
};
```

The `display` member function in the `Question` class has subtly different parameter types: the `this` pointer that points to the implicit parameter (see Section 9.10.3) has type `const Question*`, whereas in the `ChoiceQuestion` class, the `this` pointer is not `const`.

In C++ 11, you can use the reserved word `override` to tag any member function that should override a virtual function:

```
class ChoiceQuestion : public Question
{
public:
   void display() override;
   . . .
};
```

If the member function does not actually override a virtual function, the compiler generates an error. This is good, because you can then have a closer look and add the missing `const` reserved word.

The compiler also generates an error if you forget to declare the base class member function as virtual. If you use C++ 11, it is a good idea to take advantage of the `override` reserved word.

Special Topic 10.2

Virtual Self-Calls

Suppose we add the following function to the `Question` class:

```
void Question::ask() const
{
   display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << check_answer(response) << endl;
}
```

Now consider the call

```
ChoiceQuestion cq;
cq.set_text("In which country was the inventor of C++ born?");
. . .
cq.ask();
```

Which `display` and `check_answer` function will the `ask` function call? If you look inside the code of the `Question::ask` function, you can see that these functions are executed on the implicit parameter:

```
void Question::ask() const
{
   implicit parameter.display();
   cout << "Your answer: ";
   getline(cin, response);
   cout << implicit parameter.check_answer(response) << endl;
}
```

The implicit parameter in our call is `cq`, an object of type `ChoiceQuestion`. Because the `display` and `check_answer` functions are virtual, the `ChoiceQuestion` versions of the functions are called automatically. This happens even though the `ask` function is defined in the `Question` class, which has *no knowledge* of the `ChoiceQuestion` class.

As you can see, virtual functions are a very powerful mechanism. The `Question` class supplies an `ask` function that specifies the common nature of asking a question, namely to display it and check the response. How the displaying and checking are carried out is left to the derived classes.

HOW TO 10.1

Developing an Inheritance Hierarchy

When you work with a set of classes, some of which are more general and others more specialized, you want to organize them into an inheritance hierarchy. This enables you to process objects of different classes in a uniform way.

Problem Statement As an example, we will consider a bank that offers its customers the following account types:

- *A savings account that earns interest. The interest compounds monthly and is computed on the minimum monthly balance.*
- *A checking account that has no interest, gives you three free withdrawals per month, and charges a $1 transaction fee for each additional withdrawal.*

The program will manage a set of accounts of both types, and it should be structured so that other account types can be added without affecting the main processing loop. Supply a menu

```
D)eposit  W)ithdraw  M)onth end  Q)uit
```

For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.

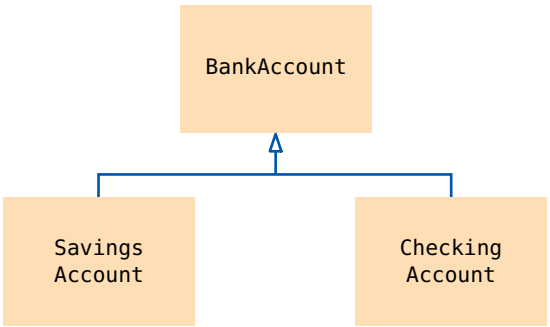
In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description yields two classes: `SavingsAccount` and `CheckingAccount`. To express the commonality between them, we will introduce a class `BankAccount`.

Step 2 Organize the classes into an inheritance hierarchy.

Draw a UML diagram that shows base and derived classes. Here is the diagram for our example:



Step 3 Determine the common responsibilities.

In Step 2, you will have identified a class at the base of the hierarchy. That class needs to have sufficient responsibilities to carry out the tasks at hand. To find out what those tasks are, write pseudocode for processing the objects:

```
For each user command
   If it is a deposit or withdrawal
      Deposit or withdraw the amount from the specified account.
      Print the balance.
   If it is month end processing
      For each account
         Call month end processing.
         Print the balance.
```

From the pseudocode, we obtain the following list of common responsibilities that every bank account must carry out:

```
Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
```

Step 4 Decide which functions are overridden in derived classes.

For each derived class and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden. Declare any functions that are overridden as `virtual` in the root of the hierarchy.

Getting the balance is common to all account types. Withdrawing and end of month processing are different for the derived classes, so they need to be declared `virtual`. Because it is entirely possible that some future account type will levy a fee for deposits, it seems prudent to declare the `deposit` member function virtual as well.

```
class BankAccount
{
public:
   virtual void deposit(double amount);
   virtual void withdraw(double amount);
   virtual void month_end();
   double get_balance() const;
private:
   . . .
};
```

Step 5 Define the public interface of each derived class.

Typically, derived classes have responsibilities other than those of the base class. List those, as well as the member functions that need to be overridden. You also need to specify how the objects of the derived classes should be constructed.

In this example, we need a way of setting the interest rate for the savings account. In addition, we need to specify constructors and overridden functions.

```
class SavingsAccount : public BankAccount
{
public:
   /**
      Constructs a savings account with a zero balance.
   */
   SavingsAccount();

   /**
      Sets the interest rate for this account.
      @param rate the monthly interest rate in percent
   */
   void set_interest_rate(double rate);

   virtual void withdraw(double amount);
   virtual void month_end();
private:
   . . .
};

class CheckingAccount : public BankAccount
{
public:
   /**
      Constructs a checking account with a zero balance.
   */
   CheckingAccount();

   virtual void withdraw(double amount);
   virtual void month_end();
private:
   . . .
};
```

Step 6 Identify data members.

List the data members for each class. If you find a data member that is common to all classes, be sure to place it in the base of the hierarchy.

All accounts have a balance. We store that value in the `BankAccount` base class:

```
class BankAccount
{
   . . .
private:
   double balance;
};
```

The `SavingsAccount` class needs to store the interest rate. It also needs to store the minimum monthly balance, which must be updated by all withdrawals:

```
class SavingsAccount : public BankAccount
{
   . . .
private:
   double interest_rate;
   double min_balance;
};
```

The `CheckingAccount` class needs to count the withdrawals, so that the charge can be applied after the free withdrawal limit is reached:

```
class CheckingAccount : public BankAccount
{
   . . .
private:
   int withdrawals;
};
```

Step 7 Implement constructors and member functions.

The member functions of the `BankAccount` class update or return the balance:

```
BankAccount::BankAccount()
{
   balance = 0;
}

void BankAccount::deposit(double amount)
{
   balance = balance + amount;
}

void BankAccount::withdraw(double amount)
{
   balance = balance - amount;
}

double BankAccount::get_balance() const
{
   return balance;
}
```

At the level of the `BankAccount` base class, we can say nothing about end of month processing. We choose to make that function do nothing:

```
void BankAccount::month_end()
{
}
```

In the `withdraw` member function of the `SavingsAccount` class, the minimum balance is updated. Note the call to the base-class member function:

```
void SavingsAccount::withdraw(double amount)
{
   BankAccount::withdraw(amount);
   double balance = get_balance();
   if (balance < min_balance)
   {
      min_balance = balance;
   }
}
```

In the `month_end` member function of the `SavingsAccount` class, the interest is deposited into the account. We must call the `deposit` member function because we have no direct access to the `balance` data member. The minimum balance is reset for the next month:

```
void SavingsAccount::month_end()
{
   double interest = min_balance * interest_rate / 100;
   deposit(interest);
   min_balance = get_balance();
}
```

The `withdraw` function of the `CheckingAccount` class needs to check the withdrawal count. If there have been too many withdrawals, a charge is applied. Again, note how the function invokes the base-class function, using the `BankAccount::` syntax:

```
void CheckingAccount::withdraw(double amount)
{
   const int FREE_WITHDRAWALS = 3;
   const int WITHDRAWAL_FEE = 1;

   BankAccount::withdraw(amount);
   withdrawals++;
   if (withdrawals > FREE_WITHDRAWALS)
   {
      BankAccount::withdraw(WITHDRAWAL_FEE);
   }
}
```

End of month processing for a checking account simply resets the withdrawal count:

```
void CheckingAccount::month_end()
{
   withdrawals = 0;
}
```

Step 8 Allocate objects on the free store and process them.

For polymorphism (that is, variation of behavior) to work in C++, you need to call virtual functions through pointers. The easiest strategy is to allocate all polymorphic objects on the free store, using the `new` operator.

In our sample program, we allocate 5 checking accounts and 5 savings accounts and store their addresses in an array of bank account pointers. Then we accept user commands and execute deposits, withdrawals, and monthly processing.

```
int main()
{
   cout << fixed << setprecision(2);

   // Create accounts
   const int ACCOUNTS_SIZE = 10;
   BankAccount* accounts[ACCOUNTS_SIZE];
```

```
   for (int i = 0; i < ACCOUNTS_SIZE / 2; i++)
   {
      accounts[i] = new CheckingAccount;
   }
   for (int i = ACCOUNTS_SIZE / 2; i < ACCOUNTS_SIZE; i++)
   {
      SavingsAccount* account = new SavingsAccount;
      account->set_interest_rate(0.75);
      accounts[i] = account;
   }

   // Execute commands
   bool more = true;
   while (more)
   {
      cout << "D)eposit  W)ithdraw  M)onth end  Q)uit: ";
      string input;
      cin >> input;
      if (input == "D" || input == "W")  // Deposit or withdrawal
      {
         cout << "Enter account number and amount: ";
         int num;
         double amount;
         cin >> num >> amount;

         if (input == "D") { accounts[num]->deposit(amount); }
         else { accounts[num]->withdraw(amount); }

         cout << "Balance: " << accounts[num]->get_balance() << endl;
      }
      else if (input == "M")  // Month end processing
      {
         for (int n = 0; n < ACCOUNTS_SIZE(); n++)
         {
            accounts[n]->month_end();
            cout << n << " " << accounts[n]->get_balance() << endl;
         }
      }
      else if (input == "Q")
      {
         more = false;
      }
   }

   return 0;
}
```

EXAMPLE CODE See how_to_1 of your companion code for the complete program.

WORKED EXAMPLE 10.1

Implementing an Employee Hierarchy for Payroll Processing

Learn how to implement payroll processing that works for different kinds of employees. See your E-Text or visit wiley.com/go/bclo3.

Jose Luis Pelaez Inc./Getty Images, Inc.

Computing & Society 10.1 Who Controls the Internet?

In 1962, J.C.R. Licklider was head of the first computer research program at DARPA, the Defense Advanced Research Projects Agency. He wrote a series of papers describing a "galactic network" through which computer users could access data and programs from other sites. This was well before computer networks were invented. By 1969, four computers—three in California and one in Utah—were connected to the ARPANET, the precursor of the Internet. The network grew quickly, linking computers at many universities and research organizations. It was originally thought that most network users wanted to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent that remote execution was not what the network was actually used for. Instead, the "killer application" was electronic mail: the transfer of messages between computer users at different locations.

In 1972, Bob Kahn proposed to extend ARPANET into the *Internet:* a collection of interoperable networks. All networks on the Internet share common *protocols* for data transmission. Kahn and Vinton Cerf developed a protocol, now called TCP/IP (Transmission Control Protocol/Internet Protocol). On January 1, 1983, all hosts on the Internet simultaneously switched to the TCP/IP protocol (which is used to this day).

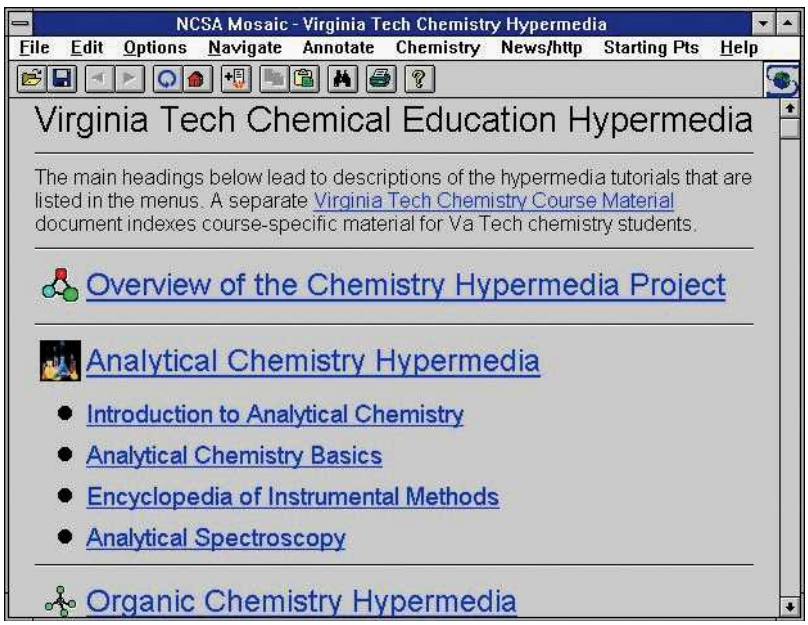
Over time, researchers, computer scientists, and hobbyists published increasing amounts of information on the Internet. For example, Project Gutenberg makes available the text of important classical books, whose copyright has expired, in computer-readable form (`www.gutenberg.org`). In 1989, Tim Berners-Lee, a computer scientist at CERN (the European organization for nuclear research) started work on hyperlinked documents, allowing users to browse by following links to related documents. This infrastructure is now known as the World Wide Web.

The first interfaces to retrieve this information were, by today's standards, unbelievably clumsy and hard to use. In March 1993, WWW traffic was 0.1 percent of all Internet traffic. All that changed when Marc Andreesen, then a graduate student working for the National Center for Supercomputing Applications (NCSA), released Mosaic. Mosaic displayed web pages in graphical form, using images, fonts, and colors (see the figure). Andreesen went on to fame and fortune at Netscape, and Microsoft licensed the Mosaic code to create Internet Explorer. By 1996, WWW traffic accounted for more than half of the data transported on the Internet.

The Internet has a very democratic structure. Anyone can publish anything, and anyone can read whatever has been published. This does not always sit well with governments and corporations.

Many governments control the Internet infrastructure in their country. For example, an Internet user in China, searching for the Tiananmen Square massacre or air pollution in their hometown, may find nothing. Vietnam blocks access to Facebook, perhaps fearing that anti-government protesters might use it to organize themselves. The U.S. government has required publicly funded libraries and schools to install filters that block sexually-explicit and hate speech, and its security organizations have spied on the Internet usage of citizens.

When the Internet is delivered by phone or TV cable companies, those companies sometimes interfere with competing Internet offerings. Cell phone companies refused to carry Voice-over-IP services, and cable companies slowed down movie streaming. The Internet has become a powerful force for delivering information—both good and bad. It is our responsibility as citizens to demand of our government that we can control which information to access.

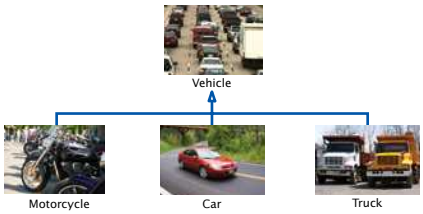


The NCSA Mosaic Browser

CHAPTER SUMMARY

Explain the notions of inheritance, base class, and derived class.

- A derived class inherits data and behavior from a base class.
- You can always use a derived-class object in place of a base-class object.



Implement derived classes in C++.

- A derived class can override a base-class function by providing a new implementation.
- The derived class inherits all data members and all functions that it does not override.
- Unless specified otherwise, the base-class data members are initialized with the default constructor.
- The constructor of a derived class can supply arguments to a base-class constructor.

Describe how a derived class can override functions from a base class.

- A derived class can inherit a function from the base class, or it can override it by providing another implementation.
- Use *BaseClass*::*function* notation to explicitly call a base-class function.

Describe virtual functions and polymorphism.

- When converting a derived-class object to a base class, the derived-class data is sliced away.
- A derived-class pointer can be converted to a base-class pointer.
- When a virtual function is called, the version belonging to the actual type of the implicit parameter is invoked.
- Polymorphism (literally, "having multiple shapes") describes objects that share a set of tasks and execute them in different ways.

REVIEW EXERCISES

■ **R10.1** Identify the base class and the derived class in each of the following pairs of classes.

a. `Employee, Manager`
b. `Polygon, Triangle`
c. `GraduateStudent, Student`
d. `Person, Student`
e. `Employee, Professor`
f. `BankAccount, CheckingAccount`
g. `Vehicle, Car`
h. `Vehicle, Minivan`
i. `Car, Minivan`
j. `Truck, Vehicle`

■ **R10.2** An object-oriented traffic simulation system has the classes listed below. Draw a UML diagram that shows the inheritance relationships between these classes.

- `Vehicle`
- `Car`
- `Truck`
- `Sedan`
- `Coupe`
- `PickupTruck`
- `SportUtilityVehicle`
- `Minivan`
- `Bicycle`
- `Motorcycle`

■ **R10.3** What inheritance relationships would you establish among the following classes?

- `Student`
- `Employee`
- `Person`
- `Professor`
- `Secretary`
- `Janitor`
- `DepartmentChair`

■ **R10.4** Draw a UML diagram that shows the inheritance and aggregation relationships between the classes:

- `Person`
- `Instructor`
- `Student`
- `Lecture`
- `Course`
- `Lab`

■ **R10.5** Consider a program for managing inventory in a small appliance store. Why isn't it useful to have a base class `SmallAppliance` and derived classes `Toaster`, `CarVacuum`, `TravelIron`, and so on?

■ **R10.6** Which data members does the `CheckingAccount` class in How To 10.1 inherit from its base class? Which data members does it add?

■ **R10.7** Which functions does the `SavingsAccount` class in How To 10.1 inherit from its base class? Which functions does it override? Which functions does it add?

■■■ **R10.8** Design an inheritance hierarchy for geometric shapes: rectangles, squares, and circles. Draw a UML diagram. Provide a virtual function to compute the area of a shape. Provide appropriate constructors for each class. Write the class definitions but do not provide implementations of the member functions.

■■ **R10.9** Continue Exercise R10.8 by writing a `main` function that executes the following steps:

Fill a vector of shape pointers with a rectangle, a square, and a circle.
Print the area of each shape.
Deallocate all objects that were allocated on the free store.

■ **R10.10** Can you convert a base-class object into a derived-class object? A derived-class object into a base-class object? A base-class pointer into a derived-class pointer? A derived-class pointer into a base-class pointer? If so, give examples. If not, explain why not.

■ **R10.11** Consider a function `process_file(ostream& str)`. Objects from which of the classes in Figure 2 can be passed as parameters to this function?

■■■ **R10.12** What does the following program print?

```
class B
{
public:
   void print(int n) const;
};

void B::print(int n) const
{
   cout << n << endl;
}

class D : public B
{
public:
   void print(int n) const;
};

void D::print(int n) const
{
   if (n <= 1) { B::print(n); }
   else if (n % 2 == 0) { print(n / 2); }
   else { print(3 * n + 1); }
}

int main()
{
```

```
   D d;
   d.print(3);
   return 0;
}
```

Determine the answer by hand, not by compiling and running the program.

■■ **R10.13** Suppose the class `D` inherits from `B`. Which of the following assignments are legal?

```
B b;
D d;
B* pb;
D* pd;
```

a. `b = d;`

b. `d = b;`

c. `pd = pb;`

d. `pb = pd;`

e. `d = pd;`

f. `b = *pd;`

g. `*pd = *pb;`

■■ **R10.14** Suppose the class `Sub` is derived from the class `Sandwich`. Which of the following assignments are legal?

```
Sandwich* x = new Sandwich;
Sub* y = new Sub;
```

a. `x = y;`

b. `y = x;`

c. `y = new Sandwich;`

d. `x = new Sub;`

e. `*x = *y;`

f. `*y = *x;`

■■■ **R10.15** What does the program print? Explain your answers by tracing the flow of each call.

```
class B
{
public:
   B();
   virtual void p() const;
   void q() const;
};

B::B() {}
void B::p() const { cout << "B::p\n"; }
void B::q() const { cout << "B::q\n"; }

class D : public B
{
public:
   D();
   virtual void p() const;
   void q() const;
};

D::D() {}
void D::p() const { cout << "D::p\n"; }
void D::q() const { cout << "D::q\n"; }
```

```
int main()
{
   B b;
   D d;
   B* pb = new B;
   B* pd = new D;
   D* pd2 = new D;

   b.p(); b.q();
   d.p(); d.q();
   pb->p(); pb->q();
   pd->p(); pd->q();
   pd2->p(); pd2->q();
   return 0;
}
```

■ **R10.16** In the `accounts.cpp` program of How To 10.1, would it be reasonable to make the `get_balance` function virtual? Explain your reasoning.

■ **R10.17** What is the effect of declaring the `display` member function virtual only in the `ChoiceQuestion` class?

PRACTICE EXERCISES

■■ **E10.1** Add a class `NumericQuestion` to the question hierarchy of Section 10.1. If the response and the expected answer differ by no more than 0.01, then accept it as correct.

■■ **E10.2** Add a class `FillInQuestion` to the question hierarchy of Section 10.1. Such a question is constructed with a string that contains the answer, surrounded by _ _, for example, `"The inventor of C++ was _Bjarne Stroustrup_"`. The question should be displayed as

```
The inventor of C++ was _____
```

Provide a `main` function that demonstrates your class.

■ **E10.3** Modify the `check_answer` member function of the `Question` class so that it does not take into account different spaces or upper/lowercase characters. For example, the response `" bjarne stroustrup"` should match an answer of `"Bjarne Stroustrup"`.

■■ **E10.4** Add a class `MultiChoiceQuestion` to the question hierarchy of Section 10.1 that allows multiple correct choices. The respondent should provide any one of the correct choices. The answer string should contain all of the correct choices, separated by spaces.

■■ **E10.5** Add a class `ChooseAllCorrect` to the question hierarchy of Section 10.1 that allows multiple correct choices. The respondent should provide all correct choices, separated by spaces.

■■ **E10.6** Add a member function `add_text` to the `Question` base class and provide a different implementation of `ChoiceQuestion` that calls `add_text` rather than storing a vector of choices.

■■ **E10.7** Implement a base class `Person`. Derive classes `Student` and `Instructor` from `Person`. A person has a name and a birthday. A student has a major, and an instructor has a salary. Write the class definitions, the constructors, and the member functions `display` for all classes.

■■ **E10.8** Using the `Employee` class from Worked Example 10.1 (`worked_example_1/salaries.cpp` in your companion code), form a derived class `Volunteer` of `Employee` and provide a constructor `Volunteer(string name)` that sets the salary to 0.

■■ **E10.9** Derive a class `Programmer` from `Employee`. Supply a constructor `Programmer(string name, double salary)` that calls the base-class constructor. Supply a function `get_name` that returns the name in the format `"Hacker, Harry (Programmer)"`.

■■ **E10.10** Derive a class `Manager` from `Employee`. Add a data member named `department` of type `string`. Supply a function `display` that displays the manager's name, department, and salary. Derive a class `Executive` from `Manager`. Supply a function `display` that displays the string `Executive`, followed by the information stored in the `Manager` base object.

■ **E10.11** Derive a class `BasicAccount` from `BankAccount` in How To 10.1 whose `withdraw` member function will not withdraw more money than is currently in the account.

■ **E10.12** Derive a class `BasicAccount` from `BankAccount` in How To 10.1 whose `withdraw` member function charges a penalty of $30 for each withdrawal that results in an overdraft.

■■ **E10.13** Reimplement the `CheckingAccount` class from How To 10.1 so that the first overdraft in any given month incurs a $20 penalty, and any further overdrafts in the same month result in a $30 penalty.

■■ **E10.14** Change the `CheckingAccount` class in How To 10.1 so that a $1 fee is levied for deposits or withdrawals in excess of three free monthly transactions. Place the code for computing the fee into a private member function that you call from the `deposit` and `withdraw` member functions.

PROGRAMMING PROJECTS

■■ **P10.1** Implement a class `Clock` whose `get_hours` and `get_minutes` member functions return the current time at your location. To get the current time, use the following code, which requires that you include the `<ctime>` header:

```
time_t current_time = time(0);
tm* local_time = localtime(&current_time);
int hours = local_time->tm_hour;
int minutes = local_time->tm_min;
```

Also provide a `get_time` member function that returns a string with the hours and minutes by calling the `get_hours` and `get_minutes` functions. Provide a derived class `WorldClock` whose constructor accepts a time offset. For example, if you live in California, a `new WorldClock(3)` should show the time in New York, three time zones ahead. Which member functions did you override? (You should not override `get_time`.)

■■ **P10.2** Add an alarm feature to the `Clock` class of Exercise P10.1. When `set_alarm(hours, minutes)` is called, the clock stores the alarm. When you call `get_time`, and the alarm time has been reached or exceeded, return the time followed by the string `"Alarm"` (or, if you prefer, the string `"\u23F0"`) and clear the alarm. What do you need to do to make the `set_alarm` function work for `WorldClock` objects?

■■ **Business P10.3** Implement a base class `Appointment` and derived classes `Onetime`, `Daily`, `Weekly`, and `Monthly`. An appointment has a description (for example, "see the dentist") and a date and time. Write a virtual function `occurs_on(int year, int month, int day)` that checks

whether the appointment occurs on that date. For example, for a monthly appointment, you must check whether the day of the month matches. Then fill a vector of `Appointment*` with a mixture of appointments. Have the user enter a date and print out all appointments that happen on that date.

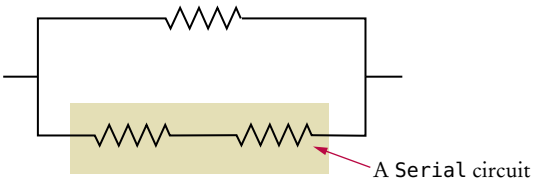
■■ Business P10.4 Improve the appointment book program of Exercise P10.3. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date and time.

■■■ Business P10.5 Improve the appointment book program of Exercises P10.3 and P10.4 by letting the user save the appointment data to a file and reload the data from a file. The saving part is straightforward: Make a virtual function `save`. Save out the type, description, date, and time. The loading part is not so easy. You must first determine the type of the appointment to be loaded, create an object of that type with its default constructor, and then call a virtual `load` function to load the remainder.

■■ P10.6 Use polymorphism to carry out image manipulations such as those described in Exercises P8.16 and P8.17. Design a base class `Effect` and derived classes `Sunset` and `Grayscale`. Use a virtual function `process`. The file processing part of your program should repeatedly call `process` on an `Effect*` pointer, without having to know which effect is applied.

■■■ Engineering P10.7 In this problem, you will model a circuit consisting of an arbitrary configuration of resistors. Provide a base class `Circuit` with a member function `get_resistance`. Provide a derived class `Resistor` representing a single resistor. Provide derived classes `Serial` and `Parallel`, each of which contains a `vector<Circuit*>`. A `Serial` circuit models a series of circuits, each of which can be a single resistor or another circuit. Similarly, a `Parallel` circuit models a set of circuits in parallel. For example, the following circuit is a `Parallel` circuit containing a single resistor and one `Serial` circuit.

© Maria Toutoudaki/iStockphoto.

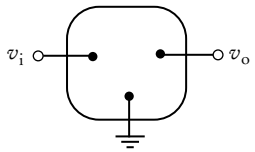


Use Ohm's law to compute the combined resistance.

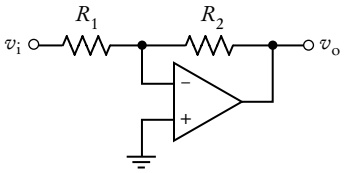
■■ Engineering P10.8 Part (a) of the figure below shows a symbolic representation of an electric circuit called an *amplifier*. The input to the amplifier is the voltage v_i and the output is the voltage v_o. The output of an amplifier is proportional to the input. The constant of proportionality is called the "gain" of the amplifier.

Parts (b), (c), and (d) show schematics of three specific types of amplifier: the *inverting amplifier*, *noninverting amplifier*, and *voltage divider amplifier*. Each of these three amplifiers consists of two resistors and an op amp. The value of the gain of each amplifier depends on the values of its resistances. In particular, the gain, g, of

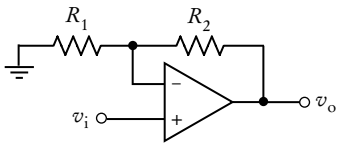
the inverting amplifier is given by $g = -\frac{R_2}{R_1}$. Similarly the gains of the noninverting amplifier and voltage divider amplifier are given by $g = 1 + \frac{R_2}{R_1}$ and $g = \frac{R_2}{R_1 + R_2}$, respectively.



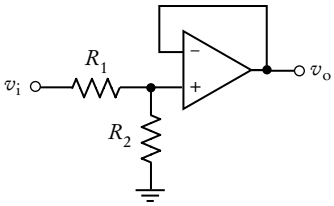
(a) Amplifier



(b) Inverting amplifier



(c) Noninverting amplifier



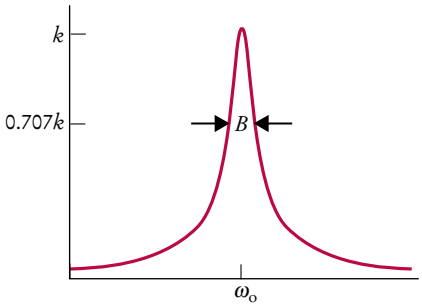
(d) Voltage divider amplifier

Write a C++ program that represents the amplifier as a base class and represents the inverting, noninverting, and voltage divider amplifiers as derived classes. Give the base class two virtual functions, `get_gain` and a `get_description` function that returns a string identifying the amplifier. Each derived class should have a constructor with two arguments, the resistances of the amplifier.

The derived classes need to override the `get_gain` and `get_description` functions of the base class.

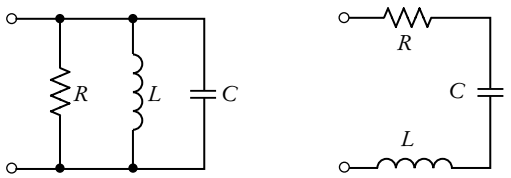
Write a `main` function for the C++ program that demonstrates that the derived classes all work properly for sample values of the resistances.

■■ Engineering P10.9 Resonant circuits are used to select a signal (e.g., a radio station or TV channel) from among other competing signals. Resonant circuits are characterized by the frequency response shown in the figure below. The resonant frequency response is completely described by three parameters: the resonant frequency, ω_o, the bandwidth, B, and the gain at the resonant frequency, k.



Frequency (rad/s, log scale)

Two simple resonant circuits are shown in the figure below. The circuit in (a) is called a *parallel resonant circuit*. The circuit in (b) is called a *series resonant circuit*. Both resonant circuits consist of a resistor having resistance R, a capacitor having capacitance C, and an inductor having inductance L.



(a) Parallel resonant circuit (b) Series resonant circuit

These circuits are designed by determining values of R, C, and L that cause the resonant frequency response to be described by specified values of ω_o, B, and k. The design equations for the parallel resonant circuit are:

$$R = k, \quad C = \frac{1}{BR}, \text{ and } \quad L = \frac{1}{\omega_o^2 C}$$

Similarly, the design equations for the series resonant circuit are:

$$R = \frac{1}{k}, \quad L = \frac{R}{B}, \text{ and } \quad C = \frac{1}{\omega_o^2 L}$$

Write a C++ program that represents `ResonantCircuit` as a base class and represents the `SeriesResonantCircuit` and `ParallelResonantCircuit` as derived classes. Give the base class three private data members representing the parameters ω_o, B, and k of the resonant frequency response. The base class should provide public member functions to get and set each of these members. The base class should also provide a `display` function that prints a description of the resonant frequency response.

Each derived class should provide a function that designs the corresponding resonant circuit. The derived classes should also override the `display` function of the base class to print descriptions of both the frequency response (the values of ω_o, B, and k) and the circuit (the values of R, C, and L).

All classes should provide appropriate constructors.

Write a `main` function for the C++ program that demonstrates that the derived classes all work properly.

WORKED EXAMPLE 10.1

Implementing an Employee Hierarchy for Payroll Processing

Jose Luis Pelaez Inc./Getty Images, Inc.

Problem Statement Your task is to implement payroll processing for different kinds of employees.

- *Hourly employees get paid an hourly rate, but if they work more than 40 hours per week, the excess is paid at "time and a half".*
- *Salaried employees get paid their salary, no matter how many hours they work.*
- *Managers are salaried employees who get paid a salary and a bonus.*

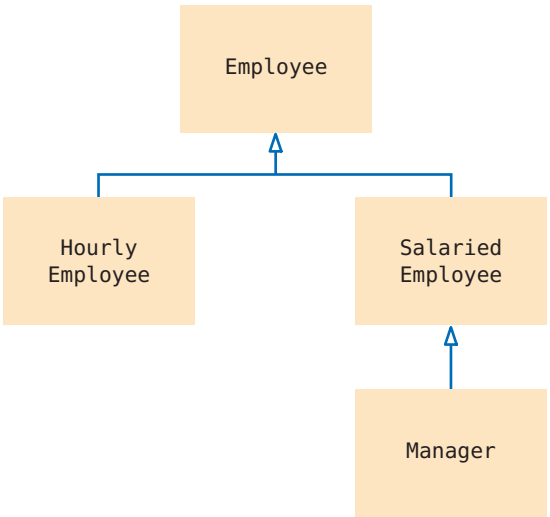
Your program should compute the pay for a collection of employees. For each employee, ask for the number of hours worked in a given week, then display the wages earned.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description lists three classes: `HourlyEmployee`, `SalariedEmployee`, and `Manager`. We need a class that expresses the commonality among them: `Employee`.

Step 2 Organize the classes into an inheritance hierarchy.

Here is the UML diagram for our classes.



Step 3 Determine the common responsibilities of the classes.

In order to discover the common responsibilities, write pseudocode for processing the objects.

```
For each employee
   Print the name of the employee.
   Read the number of hours worked.
   Compute the wages due for those hours.
```

We conclude that the `Employee` base class has these responsibilities:

```
Get the name.
Compute the wages due for a given number of hours.
```

Step 4 Decide which functions are overridden in derived classes.

In our example, there is no variation in getting the employee's name, but the salary is computed differently in each derived class. Therefore, we will declare the `weekly_pay` member function as `virtual` in the `Employee` class.

```
class Employee
{
public:
   Employee();
   string get_name() const;
   virtual double weekly_pay(int hours_worked) const;
   . . .
private:
   . . .
};
```

Step 5 Define the public interface of each class.

We will construct employees by supplying their name and salary information.

```
HourlyEmployee(string name, double wage);
SalariedEmployee(string name, double salary);
Manager(string name, double salary, double bonus);
```

These constructors need to set the name of the `Employee` base object. We will supply an `Employee` member function `set_name` for this purpose. In this simple example, no further member functions are required.

Step 6 Identify data members.

List the data members for each class. If you find a data member that is common to all classes, be sure to place it in the base of the hierarchy.

All employees have a name. Therefore, the `Employee` class should have a data member `name`. (See Figure 7.) What about the salaries? Hourly employees have an hourly wage, whereas salaried employees have an annual salary. While it would be possible to store these values in a data member of the base class, it would not be a good idea. The resulting code, which would need to make sense of what that number means, would be complex and error-prone.

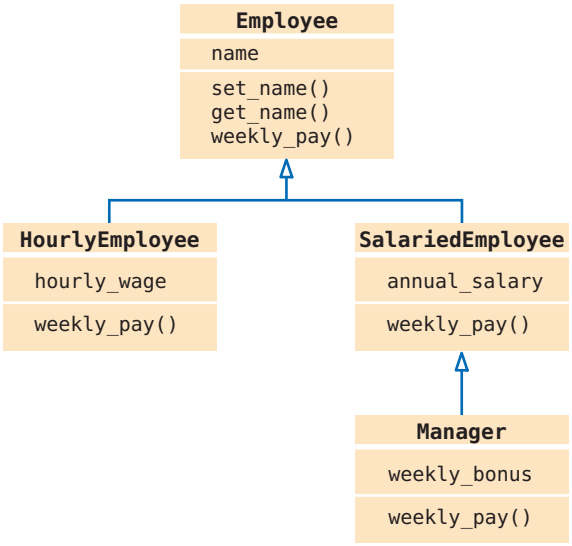


Figure 7 Employee Payroll Hierarchy

Step 7 Implement constructors and member functions.

In a derived-class constructor, we need to remember to set the data members of the base class.

```
SalariedEmployee::SalariedEmployee(string name, double salary)
{
   set_name(name);
   annual_salary = salary;
}
```

Here we use a member function. Special Topic 10.1 shows how to invoke a base-class constructor. We use that technique in the `Manager` constructor:

```
Manager::Manager(string name, double salary, double bonus)
   : SalariedEmployee(name, salary)
{
   weekly_bonus = bonus;
}
```

The weekly pay needs to be computed as specified in the problem description:

```
double HourlyEmployee::weekly_pay(int hours_worked) const
{
   double pay = hours_worked * hourly_wage;
   if (hours_worked > 40)
   {
      pay = pay + ((hours_worked - 40) * 0.5) * hourly_wage;
   }
   return pay;
}

double SalariedEmployee::weekly_pay(int hours_worked) const
{
   const int WEEKS_PER_YEAR = 52;
   return annual_salary / WEEKS_PER_YEAR;
}
```

In the case of the `Manager`, we need to call the version of `weekly_pay` from the `SalariedEmployee` base class:

```
double Manager::weekly_pay(int hours) const
{
   return SalariedEmployee::weekly_pay(hours) + weekly_bonus;
}
```

Step 8 Allocate objects on the free store and process them.

In our sample program, we populate a vector of pointers and compute the weekly salaries:

```
vector<Employee*> staff;
staff.push_back(new HourlyEmployee("Morgan, Harry", 30));
. . .
for (int i = 0; i < staff.size(); i++)
{
   cout << "Hours worked by " << staff[i]->get_name() << ": ";
   int hours;
   cin >> hours;
   cout << "Salary: " << staff[i]->weekly_pay(hours) << endl;
}
```

worked_example_1/salaries.cpp

```
1 #include <iomanip>
2 #include <iostream>
3 #include <vector>
```

```
 4
 5  using namespace std;
 6
 7  /**
 8     An employee with a name and a mechanism for computing weekly pay.
 9  */
10  class Employee
11  {
12  public:
13     /**
14        Constructs an employee with an empty name.
15     */
16     Employee();
17
18     /**
19        @param employee_name the name of this employee
20     */
21     void set_name(string employee_name);
22
23     /**
24        @return the name of this employee
25     */
26     string get_name() const;
27
28     /**
29        Computes the pay for one week of work.
30        @param hours_worked the number of hours worked in the week
31        @return the pay for the given number of hours
32     */
33     virtual double weekly_pay(int hours_worked) const;
34  private:
35     string name;
36  };
37
38  Employee::Employee()
39  {
40  }
41
42  void Employee::set_name(string employee_name)
43  {
44     name = employee_name;
45  }
46
47  string Employee::get_name() const
48  {
49     return name;
50  }
51
52  double Employee::weekly_pay(int hours_worked) const
53  {
54     return 0;
55  }
56
57  //.....................................................................
58
59  class HourlyEmployee : public Employee
60  {
61  public:
62     /**
63        @param name the name of this employee
```

```
64          @param wage the hourly wage
65       */
66       HourlyEmployee(string name, double wage);
67
68       virtual double weekly_pay(int hours_worked) const;
69    private:
70       double hourly_wage;
71    };
72
73    HourlyEmployee::HourlyEmployee(string name, double wage)
74    {
75       set_name(name);
76       hourly_wage = wage;
77    }
78
79    double HourlyEmployee::weekly_pay(int hours_worked) const
80    {
81       double pay = hours_worked * hourly_wage;
82       if (hours_worked > 40)
83       {
84          pay = pay + ((hours_worked - 40) * 0.5) * hourly_wage;
85       }
86       return pay;
87    }
88
89    //.....................................................................
90
91    class SalariedEmployee : public Employee
92    {
93    public:
94       /**
95          @param name the name of this employee
96          @param salary the annual salary
97        */
98       SalariedEmployee(string name, double salary);
99
100      virtual double weekly_pay(int hours_worked) const;
101   private:
102      double annual_salary;
103   };
104
105   SalariedEmployee::SalariedEmployee(string name, double salary)
106   {
107      set_name(name);
108      annual_salary = salary;
109   }
110
111   double SalariedEmployee::weekly_pay(int hours_worked) const
112   {
113      const int WEEKS_PER_YEAR = 52;
114      return annual_salary / WEEKS_PER_YEAR;
115   }
116
117   //.....................................................................
118
119   class Manager : public SalariedEmployee
120   {
121   public:
122      /**
123         @param name the name of this employee
```

```
124       @param salary the annual salary
125       @param bonus the weekly bonus
126    */
127    Manager(string name, double salary, double bonus);
128
129    virtual double weekly_pay(int hours) const;
130 private:
131    double weekly_bonus;
132 };
133
134 Manager::Manager(string name, double salary, double bonus)
135    : SalariedEmployee(name, salary)
136 {
137    weekly_bonus = bonus;
138 }
139
140 double Manager::weekly_pay(int hours) const
141 {
142    return SalariedEmployee::weekly_pay(hours) + weekly_bonus;
143 }
144
145 //.....................................................................
146
147 int main()
148 {
149    vector<Employee*> staff;
150    staff.push_back(new HourlyEmployee("Morgan, Harry", 30));
151    staff.push_back(new SalariedEmployee("Lin, Sally", 52000));
152    staff.push_back(new Manager("Smith, Mary", 104000, 50));
153
154    for (int i = 0; i < staff.size(); i++)
155    {
156       cout << "Hours worked by " << staff[i]->get_name() << ": ";
157       int hours;
158       cin >> hours;
159       cout << "Salary: " << staff[i]->weekly_pay(hours) << endl;
160    }
161
162    return 0;
163 }
```

CHAPTER 11

RECURSION

© Nicolae Popovici/iStockphoto.

CHAPTER GOALS

To learn to "think recursively"

To be able to use recursive helper functions

To understand the relationship between recursion and iteration

To understand when the use of recursion affects the efficiency of an algorithm

To analyze problems that are much easier to solve by recursion than by iteration

To process data with recursive structures using mutual recursion

CHAPTER CONTENTS

The method of recursion is a powerful technique for breaking up complex computational problems into simpler, often smaller, ones. The term "recursion" refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you both simple and complex examples of recursion and teaches you how to "think recursively".

11.1 Triangle Numbers

Chapter 5 contains a simple introduction to writing recursive functions—functions that call themselves with simpler inputs. In that chapter, you saw how to print triangle patterns such as this one:

```
[]
[][]
[][][]
[][][][]
```

© David Mantel/iStockphoto.

Using the same method as the one in this section, you can compute the volume of a Mayan pyramid.

The key observation is that you can print a triangle pattern of a given side length, provided you know how to print the smaller triangle pattern that is shown in color.

In this section, we will modify the example slightly and use recursion to compute the area of a triangle shape of side length *n*, assuming that each `[]` square has area 1. This value is sometimes called the *nth triangle number*. For example, as you can tell from looking at the above triangle, the third triangle number is 6 and the fourth triangle number is 10.

If the side length of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let's take care of this case first.

```
int triangle_area(int side_length)
{
   if (side_length == 1) { return 1; }
   . . .
}
```

To deal with the general case, suppose you knew the area of the smaller, blue triangle. Then you could easily compute the area of the larger triangle as

```
smaller_area + side_length
```

How can you get the smaller area? Call the `triangle_area` function!

```
int smaller_side_length = side_length - 1;
int smaller_area = triangle_area(smaller_side_length);
```

Now we can complete the `triangle_area` function:

```
int triangle_area(int side_length)
{
```

```
   if (side_length == 1) { return 1; }
   int smaller_side_length = side_length - 1;
   int smaller_area = triangle_area(smaller_side_length);
   return smaller_area + side_length;
}
```

If you prefer, you can implement this function more compactly:

```
int triangle_area(int side_length)
{
   if (side_length == 1) { return 1; }
   else
   {
      return triangle_area(side_length - 1) + side_length;
   }
}
```

We will look at the longer, more explicit form in this section.

Here is a trace of the function call `triangle_area(4)`.

- The `triangle_area` function executes with the parameter variable `side_length` set to 4.
- It sets `smaller_side_length` to 3 and calls `triangle_area` with argument `smaller_side_length`.
  - That function call has its own set of parameter and local variables. Its `side_length` parameter variable is 3, and it sets its `smaller_side_length` variable to 2.
  - The `triangle_area` function is called again, now with argument 2.
    - In that function call, `side_length` is 2 and `smaller_side_length` is 1.
    - The `triangle_area` function is called with argument 1.
      - That function call returns 1.
    - The function call sets `smaller_area` to 1 and returns `smaller_area + side_length` = 1 + 2 = 3.
  - The function call sets `smaller_area` to 3 and returns `smaller_area + side_length` = 3 + 3 = 6.
- The function call sets `smaller_area` to 6 and returns `smaller_area + side_length` = 6 + 4 = 10.

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

As you can see, the function calls itself multiple times, with ever simpler arguments, until a very simple case is reached. Then the recursive function calls return, one by one.

While it is good to understand this pattern of recursive calls, most people don't find it very helpful to think about the call pattern when designing or understanding a recursive solution. Instead, look at the `triangle_area` function one more time. The first part is very easy to understand. If the side length is 1, then of course the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle. Don't worry how that works—treat the function as a black box and simply assume that you will get the correct answer. Then the area of the larger triangle is clearly the sum of the smaller area and the side length.

When a function keeps calling itself, you may wonder how you know that the calls will eventually come to an end. Two conditions need to be fulfilled:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

For a recursion to terminate, there must be special cases for the simplest inputs.

The `triangle_area` function calls itself with a smaller side length. Eventually the side length must reach 1, and there is a special case for computing the area of a triangle with side length 1. Thus, the `triangle_area` function always succeeds.

Actually, you have to be careful. What happens when you call the area of a triangle with side length –1? It computes the area of a triangle with side length –2, which computes the area of a triangle with side length –3, and so on. To avoid this, you should add a condition to the `triangle_area` function:

```
if (side_length <= 0) { return 0; }
```

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

```
1 + 2 + 3 + ... + side_length
```

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= side_length; i++) { area = area + i; }
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first n integers can be computed as

$$1 + 2 + \cdots + n = n \times (n + 1)/2$$

Thus, the area equals

```
side_length * (side_length + 1) / 2
```

Therefore, neither recursion nor a loop are required to solve this problem. The recursive solution is intended as a "warm-up" for the sections that follow.

sec01/triangle.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Computes the area of a triangle with a given side length.
7     @param side_length the side length of the triangle base
8     @return the area
9  */
10 int triangle_area(int side_length)
11 {
12    if (side_length <= 0) { return 0; }
13    if (side_length == 1) { return 1; }
14    int smaller_side_length = side_length - 1;
15    int smaller_area = triangle_area(smaller_side_length);
16    return smaller_area + side_length;
17 }
18
19 int main()
20 {
21    cout << "Enter the side length: ";
22    int input;
23    cin >> input;
24    cout << "Area: " << triangle_area(input) << endl;
25    return 0;
26 }
```

Program Run

```
Enter the side length: 4
Area: 10
```

Common Error 11.1

Tracing Through Recursive Functions

Debugging a recursive function can be somewhat challenging. When you set a breakpoint in a recursive function, the program stops as soon as that program line is encountered in *any call to the recursive function*. Suppose you want to debug the recursive `triangle_area` function. Run until the beginning of the `triangle_area` function (Figure 1). Inspect the `side_length` data member. It is 4.

Remove the breakpoint and now run until the statement

```
return smaller_area + side_length;
```

When you inspect `side_length` again, its value is 2! That makes no sense. There was no instruction that changed the value of `side_length`! Is that a bug with the debugger?

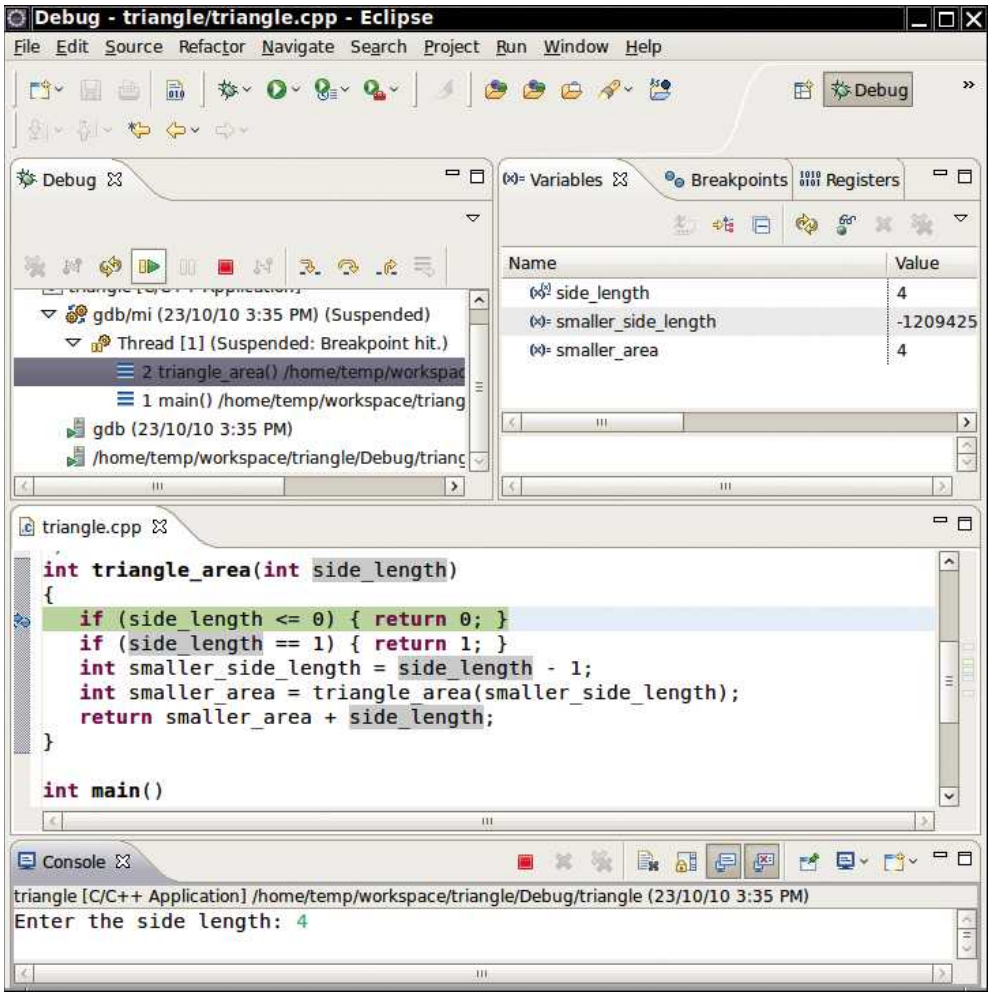


Figure 1 Debugging a Recursive Function

No. The program stopped in the first *recursive* call to `triangle_area` that reached the `return` statement. If you are confused, look at the **call stack** (Figure 2). You will see that three calls to `triangle_area` are pending.

You can debug recursive functions with the debugger. You just need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

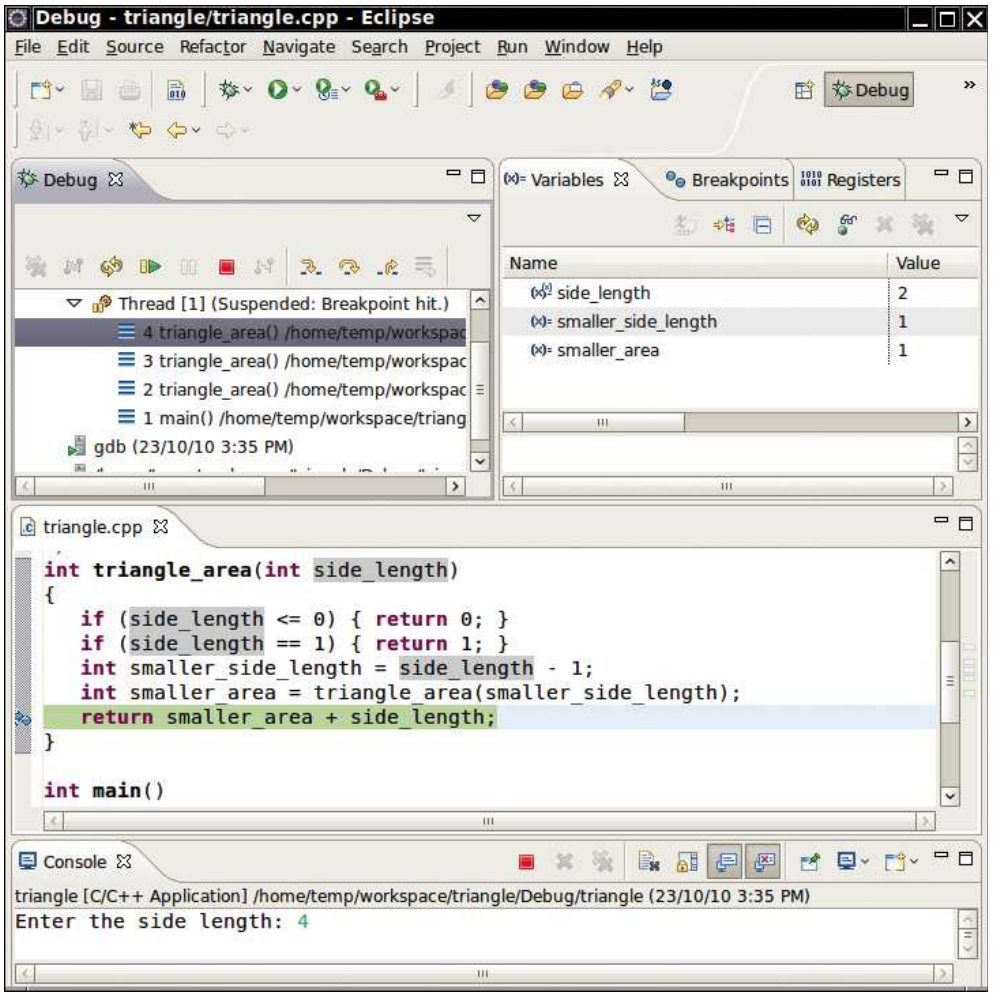


Figure 2
Three Calls to `triangle_area` Are Pending

Common Error 11.2

Infinite Recursion

A common programming error is an *infinite recursion*: a function calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is available for this purpose is exhausted. Your program shuts down and reports a "stack fault".

Infinite recursion happens either because the arguments don't get simpler or because a terminating case is missing. For example, suppose the `triangle_area` function computes the area of a triangle with side length 0. If you do not include a test for this situation, the function calls itself with side length –1, –2, –3, and so on.

HOW TO 11.1

Thinking Recursively

Solving a problem recursively requires a different mindset than solving it by programming a loop. In fact, it helps if you pretend to be a bit lazy, asking others to do most of the work for you. If you need to solve a complex problem, pretend that "someone else" will do most of the heavy lifting and solve the problem for simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem. To illustrate the technique of recursion, let us consider the following problem.

© Nikada/iStockphoto.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

Problem Statement Test whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters.

Typical examples of palindromes are:

- rotor
- A man, a plan, a canal—Panama!
- Go hang a salami, I'm a lasagna hog

and, of course, the oldest palindrome of all:

- Madam, I'm Adam

Our goal is to implement the function

```
bool is_palindrome(string s)
```

For simplicity, assume for now that the string has only lowercase letters and no punctuation marks or spaces. Exercise P11.6 asks you to generalize the function to arbitrary strings.

Step 1 Break the input into parts that can themselves be inputs to the problem.

In your mind, focus on a particular input or set of inputs for the problem that you want to solve. Think how you can simplify the inputs in such a way that the same problem can be applied to the simpler input.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and the last character.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

Step 2 Combine solutions with simpler inputs to a solution of the original problem.

In your mind, consider the solutions of your problem for the simpler inputs that you discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the

solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

Now think how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, related to the quantity that you lopped off to arrive at the simpler input. Maybe you cut the original input in half and have solutions for each half. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem a good idea. If you cut

```
"rotor"
```

in half, you get two strings:

```
"rot"
```

and

```
"or"
```

Neither of them is a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters, provided they match. Removing the `r` at the front and back of `"rotor"` yields

```
"oto"
```

Suppose you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match, and
- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

Step 3 Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them, which is usually very easy.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character
- The empty string

You don't have to come up with a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But you do need to worry about strings of length `0` and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string `"oo"`. According to the rule discovered in Step 2, this string is a palindrome if the first and last character of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

A string with a single character, such as `"I"`, is a palindrome. Thus, all strings of length `0` or 1 are palindromes.

Step 4 Implement the solution by combining the simple cases and the reduction step.

Usually, you want to get the simplest cases out of the way. In the case of the palindrome test, those are the strings with length 0 or 1. If the input isn't one of the simplest cases, then make one or more recursive calls with simpler inputs and use the results of those calls to complete your function.

The following program shows the complete `is_palindrome` function:

how_to_1/palindrome.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/**
   Tests whether a string is a palindrome. A palindrome
   is equal to its reverse, for example "rotor" or "racecar".
   @param s a string
   @return true if s is a palindrome
*/
bool is_palindrome(string s)
{
   // Separate case for shortest strings
   if (s.length() <= 1) { return true; }

   // Get first and last character, converted to lowercase
   char first = s[0];
   char last = s[s.length() - 1];

   if (first == last)
   {
      string shorter = s.substr(1, s.length() - 2);
      return is_palindrome(shorter);
   }
   else
   {
      return false;
   }
}

int main()
{
   cout << "Enter a string: ";
   string input;
   getline(cin, input);
   cout << input << " is ";
   if (!is_palindrome(input)) { cout << "not "; }
   cout << "a palindrome." << endl;
   return 0;
}
```

Program Run

```
Enter a string: aibohphobia
aibohphobia is a palindrome.
```

WORKED EXAMPLE 11.1

Finding Files

Learn how to use recursion to find all files with a given extension in a directory tree. See your E-Text or visit `wiley.com/go/bclo3`.

11.2 Recursive Helper Functions

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper function.

Here is a typical example. Consider the palindrome test of How To 11.1. It is a bit inefficient to construct new `string` objects in every step. Now consider the following change in the problem. Rather than testing whether the entire string is a palindrome, check whether a substring is a palindrome:

© gerenme/iStockphoto.

Sometimes, a task can be solved by handing it off to a recursive helper function.

```
/*
   Tests whether a substring of a string is a palindrome.
   @param s the string to test
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
bool substring_is_palindrome(string s, int start, int end)
```

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

This function turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the `start` and `end` arguments to skip over matching letter pairs. There is no need to construct new string objects to represent the shorter strings.

```
bool substring_is_palindrome(string s, int start, int end)
{
   // Separate case for substrings of length 0 and 1
   if (start >= end) { return true; }

   if (s[start] == s[end])
   {
      // Test substring that doesn't contain the first and last letters
      return substring_is_palindrome(s, start + 1, end - 1);
   }
   else
   {
      return false;
   }
}
```

You should supply a function to solve the whole problem—the user of your function shouldn't have to know about the trick with the substring positions. Simply call the helper function with positions that test the entire string:

```
bool is_palindrome(string s)
{
   return substring_is_palindrome(s, 0, s.length() - 1);
}
```

Note that the `is_palindrome` function is *not* recursive. It just calls a recursive helper function.

Use the technique of recursive helper functions whenever it is easier to solve a recursive problem that is slightly different from the original problem.

EXAMPLE CODE See sec02 of your companion code for a program that uses the recursive helper function.

11.3 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool for implementing complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

In most cases, iterative and recursive approaches have comparable efficiency.

The Fibonacci sequence is a sequence of numbers defined by the equations

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is 34 + 55 = 89.

We would like to write a function that computes f_n for any value of n. Here is a program that translates the definition directly into a recursive function:

sec03/fibdemo.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Computes a Fibonacci number.
7     @param n an integer
8     @return the nth Fibonacci number
9  */
10 int fib(int n)
11 {
```

```
12    if (n <= 2) { return 1; }
13    else { return fib(n - 1) + fib(n - 2); }
14 }
15
16 int main()
17 {
18    cout << "Enter n: ";
19    int n;
20    cin >> n;
21    int f = fib(n);
22    cout << "fib(" << n << ") = " << f << endl;
23    return 0;
24 }
```

Program Run

```
Enter n: 6
fib(6) = 8
```

That is certainly simple, and the function will work correctly. But watch the output closely as you run the test program. For small input values, the program is quite fast. Even for moderately large values, though, the program pauses an amazingly long time between outputs. Try out some numbers between 30 and 50 to see this effect.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer long.

To determine the problem, insert trace messages into the function:

sec03/fibtrace.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Computes a Fibonacci number.
7     @param n an integer
8     @return the nth Fibonacci number
9  */
10 int fib(int n)
11 {
12    cout << "Entering fib: n = " << n << endl;
13    int f;
14    if (n <= 2) { f = 1; }
15    else { f = fib(n - 1) + fib(n - 2); }
16    cout << "Exiting fib: n = " << n
17       << " return value = " << f << endl;
18    return f;
19 }
20
21 int main()
22 {
23    cout << "Enter n: ";
24    int n;
25    cin >> n;
26    int f = fib(n);
27    cout << "fib(" << n << ") = " << f << endl;
28    return 0;
29 }
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Figure 3 shows the call tree.

Now it is becoming apparent why the function takes so long. It is computing the same values over and over. For example, the computation of `fib(6)` calls `fib(4)` twice and `fib(3)` three times. That is very different from the computation you would do with pencil and paper. There you would just write down the values as they were

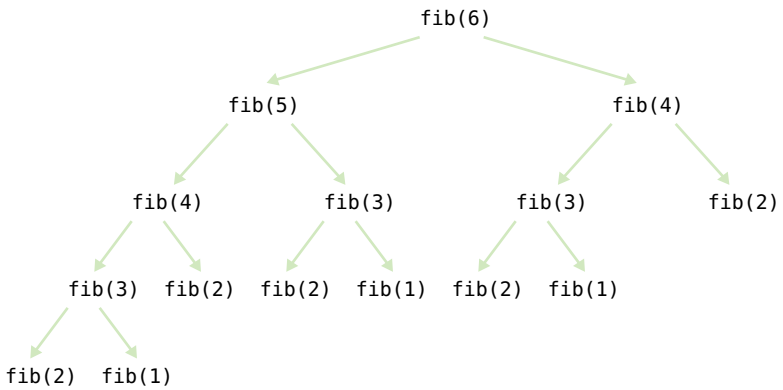


Figure 3 Call Pattern of the Recursive `fib` Function

computed and add up the last two to get the next one until you reached the desired entry; no sequence value would ever be computed twice.

If you imitate the pencil-and-paper process, then you get the following program:

sec03/fibloop.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Computes a Fibonacci number.
7     @param n an integer
8     @return the nth Fibonacci number
9  */
10 int fib(int n)
11 {
12    if (n <= 2) { return 1; }
13    int fold = 1;
14    int fold2 = 1;
15    int fnew;
16    for (int i = 3; i <= n; i++)
17    {
18       fnew = fold + fold2;
19       fold2 = fold;
20       fold = fnew;
21    }
22    return fnew;
23 }
24
25 int main()
26 {
27    cout << "Enter n: ";
28    int n;
29    cin >> n;
30    int f = fib(n);
31    cout << "fib(" << n << ") = " << f << endl;
32    return 0;
33 }
```

The `fib` function in this program runs *much* faster than the recursive version.

Occasionally, a recursive solution runs much more slowly than its iterative counterpart. However, in most cases, the recursive solution runs at about the same speed.

In this example of the `fib` function, the recursive solution was easy to program because it exactly followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test:

```
bool is_palindrome(string s)
{
   int start = 0;
   int end = s.length() - 1;
   while (start < end)
   {
      if (s[start] != s[end]) { return false; }
      start++;
      end--;
```

```
      }
      return true;
   }
```

This solution keeps two index variables: `start` and `end`. The `start` index starts at the beginning of the string, and the `end` index at the end of the string. Whenever the characters at `start` and `end` match, the `start` index is incremented and the `end` index is decremented. When the two index variables meet, then the iteration stops.

The iteration and the recursion do essentially the same work. If a palindrome has n characters, the iteration executes the loop $n/2$ times, comparing a pair of characters in each iteration. Similarly, the recursive solution calls itself $n/2$ times, because two characters are compared and removed in each step.

In such a situation, the iterative solution tends to be a bit faster, because each recursive function call takes a certain amount of processor time. In principle, it is possible for a smart compiler to avoid recursive function calls if they follow simple patterns, but most compilers don't do that. From that point of view, an iterative solution is preferable.

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Often, recursive solutions are easier to understand and implement correctly than their iterative counterparts. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing. As the computer scientist (and creator of the GhostScript interpreter for the PostScript graphics description language) L. Peter Deutsch put it: "To iterate is human, to recurse divine."

11.4 Permutations

In this section, we consider a more complex example of recursion that would be difficult to program with a simple loop. Our task is to generate all **permutations** of a string. A permutation is simply a rearrangement of the letters. For example, the string `"eat"` has six permutations (including the original string itself):

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

© Jeanine Groenwald/iStockphoto.

Using recursion, you can find all arrangements of a set of objects.

We will develop a function

```
vector<string> generate_permutations(string word)
```

that generates all permutations of a word.

Here is how you would use the function. The following code displays all permutations of the string `"eat"`:

```
vector<string> v = generate_permutations("eat");
for (int i = 0; i < v.size(); i++)
{
   cout << v[i] << endl;
}
```

Now you need a way to generate the permutations recursively. Consider the string `"eat"` and simplify the problem. First, generate all permutations that start with the letter `'e'`, then those that start with `'a'`, and finally those that start with `'t'`. How do you generate the permutations that start with `'e'`? You need to know the permutations of

the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Using recursion generates the permutations of the substring "at". You will get the strings

```
"at"
"ta"
```

For each result of the simpler problem, add the letter 'e' in front. Now you have all permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```

Next, turn your attention to the permutations of "eat" that start with 'a'. You must create the permutations of the remaining letters, "et", namely

```
"et"
"te"
```

Add the letter 'a' to the front of the strings and obtain

```
"aet"
"ate"
```

Generate the permutations that start with 't' in the same way.

That's the idea. To carry it out, you must implement a loop that iterates through the character positions of the word. Each loop iteration creates a shorter word that omits the current position:

```
vector<string> generate_permutations(string word)
{
   vector<string> result;
   . . .
   for (int i = 0; i < word.length(); i++)
   {
      string shorter_word = word.substr(0, i) + word.substr(i + 1);
      . . .
   }
   return result;
}
```

The next step is to compute the permutations of the shorter word.

```
vector<string> shorter_permutations = generate_permutations(shorter_word);
```

For each of the shorter permutations, add the omitted letter:

```
for (int j = 0; j < shorter_permutations.size(); j++)
{
   string longer_word = word[i] + shorter_permutations[j];
   result.push_back(longer_word);
}
```

The permutation generation algorithm is recursive—it uses the fact that we can generate the permutations of shorter words. When does the recursion stop? The simplest possible string is the empty string, which has a single permutation—itself.

```
vector<string> generate_permutations(string word)
{
   vector<string> result;
   if (word.length() == 0)
   {
      result.push_back(word); // The empty string has only itself as a permutation
      return result;
```

```
   }
   . . .
}
```

For generating permutations, it is much easier to use recursion than iteration.

Could you generate the permutations without recursion? There is no obvious way of writing a loop that iterates through all permutations. Exercise P11.5 shows that there is an iterative solution, but it is far more difficult to understand than the recursive algorithm.

Here is the complete permutation program:

sec04/permute.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/**
   Generates all permutations of the characters in a string.
   @param word a string
   @return a vector that is filled with all permutations
   of the word
*/
vector<string> generate_permutations(string word)
{
   vector<string> result;
   if (word.length() == 0)
   {
      result.push_back(word); // The empty string has only itself as a permutation
      return result;
   }

   for (int i = 0; i < word.length(); i++)
   {
      string shorter_word = word.substr(0, i)
         + word.substr(i + 1); // The word without the ith letter
      vector<string> shorter_permutations
         = generate_permutations(shorter_word);

      // Add the ith letter to the front of all permutations of the shorter word
      for (int j = 0; j < shorter_permutations.size(); j++)
      {
         string longer_word = word[i] + shorter_permutations[j];
         result.push_back(longer_word);
      }
   }
   return result;
}

int main()
{
   cout << "Enter a string: ";
   string input;
   getline(cin, input);
   vector<string> v = generate_permutations(input);
   for (int i = 0; i < v.size(); i++)
   {
      cout << v[i] << endl;
```

```
48     }
49     return 0;
50  }
```

Program Run

```
Enter a string: arm
arm
amr
ram
rma
mar
mra
```

11.5 Mutual Recursion

In a mutual recursion, a set of cooperating functions calls each other repeatedly.

In the preceding examples, a function called itself to solve a simpler problem. Sometimes, a set of cooperating functions calls each other in a recursive fashion. In this section, we will explore a typical situation of such a **mutual recursion**.

We will develop a program that can compute the values of arithmetic expressions such as

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

Computing such an expression is complicated by the fact that * and / bind more strongly than + and -, and that parentheses can be used to group subexpressions.

Figure 4 shows a set of *syntax diagrams* that describes the syntax of these expressions. An expression is either a term, or a sum or difference of terms. A term is either a factor, or a product or quotient of factors. Finally, a factor is either a number or an expression enclosed in parentheses.

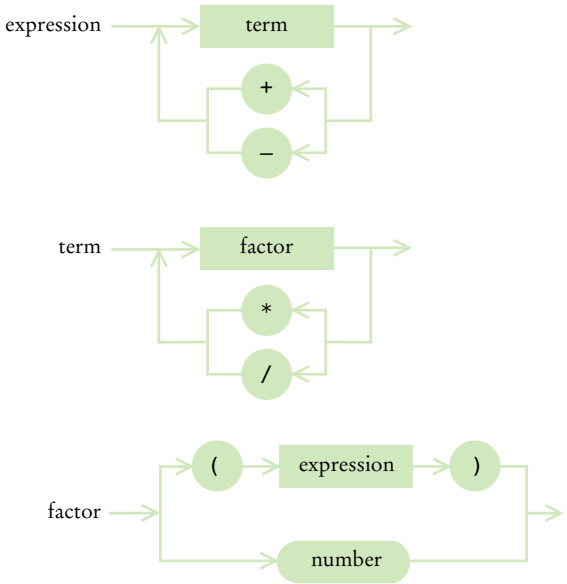


Figure 4 Syntax Diagrams for Evaluating an Expression

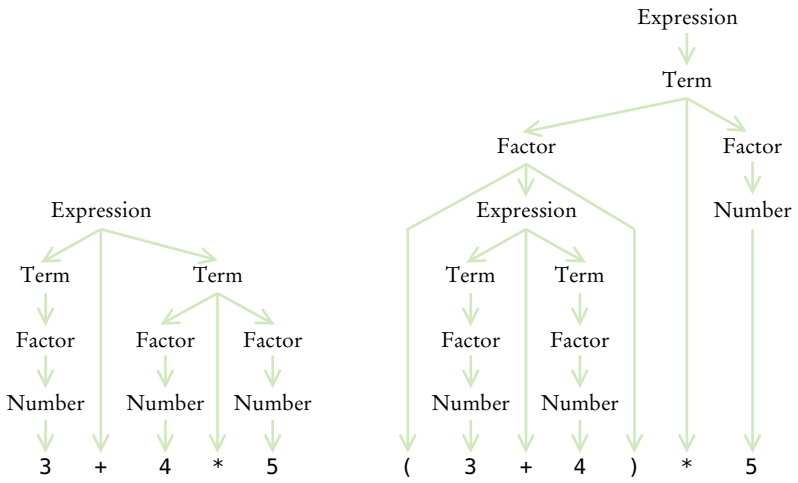


Figure 5 Syntax Trees for Two Expressions

Figure 5 shows how the expressions `3 + 4 * 5` and `(3 + 4) * 5` are derived from the syntax diagram.

Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first, starting with the bottom of the tree. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

To compute the value of an expression, we implement three functions: `expression_value`, `term_value`, and `factor_value`. The `expression_value` function first calls `term_value` to get the value of the first term of the expression. Then it checks whether the next input character is one of `+` or `-`. If so, it calls `term_value` again and adds or subtracts it:

```
int expression_value()
{
   int result = term_value();
   bool done = false;
   while (!done)
   {
      char op = cin.get();
      if (op == '+' || op == '-')
      {
         int value = term_value();
         if (op == '+') { result = result + value; }
         else { result = result - value; }
      }
      else { cin.unget(); done = true; }
   }
   return result;
}
```

The `term_value` function calls `factor_value` in the same way, multiplying or dividing the factor values.

Finally, the `factor_value` function checks whether the next input character is a `'('` or a digit. In the latter case, the value is simply the value of the number. However, if the function sees a parenthesis, the `factor_value` function makes a recursive call to `expression_value`. Thus, the three functions are mutually recursive.

```
int factor_value()
{
   int result = 0;
   char c = cin.get();
   if (c == '(')
   {
      result = expression_value();
      cin.get(); // Read ")"
   }
   else
   {
      cin.unget();
      cin >> result;
   }
   return result;
}
```

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see. If `expression_value` calls itself, the second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the characters of the input are consumed, so eventually the recursion must come to an end.

sec05/eval.cpp

```
1  #include <iostream>
2  #include <cctype>
3
4  using namespace std;
5
6  int term_value();
7  int factor_value();
8
9  /**
10    Evaluates the next expression found in cin.
11    @return the value of the expression
12 */
13 int expression_value()
14 {
15    int result = term_value();
16    bool done = false;
17    while (!done)
18    {
19       char op = cin.get();
20       if (op == '+' || op == '-')
21       {
22          int value = term_value();
23          if (op == '+') { result = result + value; }
24          else { result = result - value; }
25       }
26       else { cin.unget(); done = true; }
27    }
28    return result;
29 }
30
31 /**
32    Evaluates the next term found in cin.
33    @return the value of the term
34 */
```

```
35 int term_value()
36 {
37    int result = factor_value();
38    bool done = false;
39    while (!done)
40    {
41       char op = cin.get();
42       if (op == '*' || op == '/')
43       {
44          int value = factor_value();
45          if (op == '*') { result = result * value; }
46          else { result = result / value; }
47       }
48       else { cin.unget(); done = true; }
49    }
50    return result;
51 }
52
53 /**
54    Evaluates the next factor found in cin.
55    @return the value of the factor
56 */
57 int factor_value()
58 {
59    int result = 0;
60    char c = cin.get();
61    if (c == '(')
62    {
63       result = expression_value();
64       cin.get(); // Read ")"
65    }
66    else
67    {
68       cin.unget();
69       cin >> result;
70    }
71    return result;
72 }
73
74 int main()
75 {
76    cout << "Enter an expression: ";
77    cout << expression_value() << endl;
78    return 0;
79 }
```

Program Run

```
Enter an expression: 1+12*12*12
1729
```

11.6 Backtracking

Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal. If a partial solution cannot be completed, one abandons it and returns to examining the other candidates.

Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

Backtracking can be used to solve crossword puzzles, escape from mazes, or find solutions to systems that are constrained by rules. In order to employ backtracking for a particular problem, we need two characteristic properties:

1. A procedure to examine a partial solution and determine whether to
   - Accept it as an actual solution.
   - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
   - Continue extending it.
2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

In a backtracking algorithm, one explores all paths toward a solution. When one path is a dead end, one needs to backtrack and try another choice.

Backtracking can then be expressed with the following recursive algorithm:

```
Solve(partialSolution)
   Examine(partialSolution).
   If accepted
      Add partialSolution to the list of solutions.
   Else if continuing
      For each p in extend(partialSolution)
         Solve(p).
```

Of course, the processes of examining and extending a partial solution depend on the nature of the problem.

As an example, we will develop a program that finds all solutions to the eight queens problem: the task of positioning eight queens on a chess board so that none of them attacks another according to the rules of chess. In other words, there are no two queens on the same row, column, or diagonal. Figure 6 shows a solution.

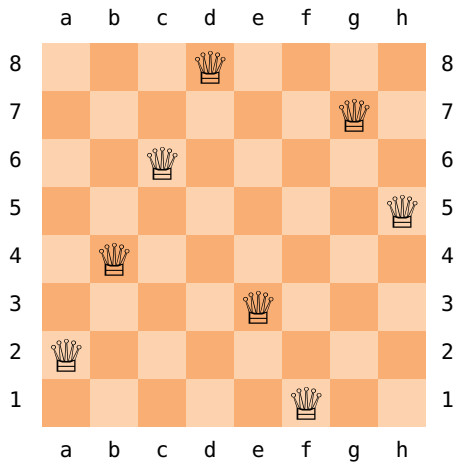


Figure 6 A Solution to the Eight Queens Problem

In this problem, it is easy to examine a partial solution. If two queens attack another, reject it. Otherwise, if it has eight queens, accept it. Otherwise, continue.

It is also easy to extend a partial solution. Simply add another queen on an empty square.

However, in the interest of efficiency, we will be a bit more systematic about the extension process. We will place the first queen in row 1, the next queen in row 2, and so on.

We provide a class `PartialSolution` that collects the queens in a partial solution, and that has member functions to examine and extend the solution:

```
class PartialSolution
{
public:
   int examine() const;
   vector<PartialSolution> extend() const;
   void print() const;

private:
   vector<Queen> queens;
};
```

The `examine` function simply checks whether two queens attack each other:

```
int PartialSolution::examine() const
{
   for (int i = 0; i < queens.size(); i++)
   {
      for (int j = i + 1; j < queens.size(); j++)
      {
         if (queens[i].attacks(queens[j])) { return ABANDON; }
      }
   }
   if (queens.size() == NQUEENS) { return ACCEPT; }
   else { return CONTINUE; }
}
```

The `extend` function takes a given solution and makes eight copies of it. Each copy gets a new queen in a different column.

```
vector<PartialSolution> PartialSolution::extend() const
{
   vector<PartialSolution> result;
   // The next row to populate
   int row = queens.size();
   // Generate a new solution for each column
   for (int column = 0; column < NQUEENS; column++)
   {
      PartialSolution extended;
      // Copy all queens from this solution
      extended.queens = queens;
      // Add the new queen
      extended.queens.push_back(Queen(row, column));
      result.push_back(extended);
   }
   return result;
}
```

You will find the `Queen` class at the end of the section. The only challenge is to determine when two queens attack each other diagonally. Here is an easy way of checking

that. Compute the slope and check whether it is ±1. This condition can be simplified as follows:

$$\begin{aligned}(\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) &= \pm 1 \\ \text{row}_2 - \text{row}_1 &= \pm(\text{column}_2 - \text{column}_1) \\ |\text{row}_2 - \text{row}_1| &= |\text{column}_2 - \text{column}_1|\end{aligned}$$

Have a close look at the `solve` function in the `eightqueens.cpp` program below. The function is a straightforward translation of the pseudocode for backtracking. Note how there is nothing specific about the eight queens problem in this function—it works for any partial solution with an `examine` and `extend` function (see Exercise E11.24).

Figure 7 shows the `solve` function in action for a four queens problem. Starting from a blank board, there are four partial solutions with a queen in row 1 ❶. When the queen is in column 1, there are four partial solutions with a queen in row 2 ❷. Two of them are abandoned immediately. The other two lead to partial solutions with three queens ❸ and ❹, all but one of which are abandoned. One partial solution is extended to four queens, but all of those are abandoned as well ❺. Then the algorithm backtracks, giving up on a queen in position `a1`, instead extending the solution with the queen in position `b1` (not shown).

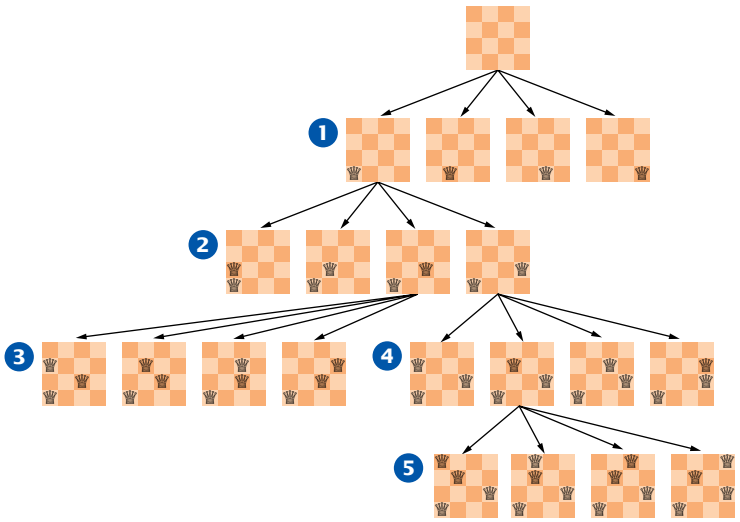


Figure 7 Backtracking in the Four Queens Problem

When you run the program, it lists 92 solutions, including the one in Figure 6. Exercise E11.25 asks you to remove those that are rotations or reflections of another.

sec06/eightqueens.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <vector>
4  #include <string>
5
```

```
using namespace std;

/**
   A queen in the eight queens problem.
*/
class Queen
{
public:
   /**
      Constructs a queen at a given position.
      @param r the row
      @param c the column
   */
   Queen(int r, int c);

   /**
      Checks whether this queen attacks another.
      @param other the other queen
      @return true if this and the other queen are in the same
      row, column, or diagonal
   */
   bool attacks(Queen other) const;

   /**
      Prints this queen.
   */
   void print() const;

private:
   int row;
   int column;
};

Queen::Queen(int r, int c)
{
   row = r;
   column = c;
}

bool Queen::attacks(Queen other) const
{
   return row == other.row
      || column == other.column
      || abs(row - other.row) == abs(column - other.column);
}

void Queen::print() const
{
   cout << "abcdefgh"[column] << row + 1;
}

const int NQUEENS = 8;

const int ACCEPT = 1;
const int ABANDON = 2;
const int CONTINUE = 3;

/**
   A partial solution to the eight queens puzzle.
*/
```

```
66  class PartialSolution
67  {
68  public:
69     /**
70        Examines a partial solution.
71        @return one of ACCEPT, ABANDON, CONTINUE
72     */
73     int examine() const;
74
75     /**
76        Yields all extensions of this partial solution.
77        @return a vector of partial solutions that extend this solution
78     */
79     vector<PartialSolution> extend() const;
80
81     /**
82        Prints this partial solution.
83     */
84     void print() const;
85
86  private:
87     vector<Queen> queens;
88  };
89
90  int PartialSolution::examine() const
91  {
92     for (int i = 0; i < queens.size(); i++)
93     {
94        for (int j = i + 1; j < queens.size(); j++)
95        {
96           if (queens[i].attacks(queens[j])) { return ABANDON; }
97        }
98     }
99     if (queens.size() == NQUEENS) { return ACCEPT; }
100    else { return CONTINUE; }
101 }
102
103 vector<PartialSolution> PartialSolution::extend() const
104 {
105    vector<PartialSolution> result;
106    // The next row to populate
107    int row = queens.size();
108    // Generate a new solution for each column
109    for (int column = 0; column < NQUEENS; column++)
110    {
111       PartialSolution extended;
112       // Copy all queens from this solution
113       extended.queens = queens;
114       // Add the new queen
115       extended.queens.push_back(Queen(row, column));
116       result.push_back(extended);
117    }
118    return result;
119 }
120
121 void PartialSolution::print() const
122 {
123    for (int i = 0; i < queens.size(); i++)
124    {
```

```
125       if (i > 0) { cout << ", "; }
126       queens[i].print();
127    }
128    cout << endl;
129 }
130
131 /**
132    Prints all solutions to the problem that can be extended from
133    a given partial solution.
134    @param sol the partial solution
135 */
136 void solve(PartialSolution sol)
137 {
138    int exam = sol.examine();
139    if (exam == ACCEPT)
140    {
141       sol.print();
142    }
143    else if (exam == CONTINUE)
144    {
145       vector<PartialSolution> extended = sol.extend();
146       for (int i = 0; i < extended.size(); i++)
147       {
148          solve(extended[i]);
149       }
150    }
151 }
152
153 int main()
154 {
155    PartialSolution initial;
156    solve(initial);
157    return 0;
158 }
```

Program Run

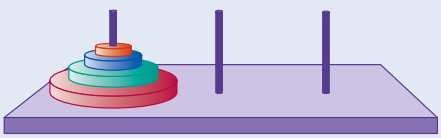
```
a1, e2, h3, f4, c5, g6, b7, d8
a1, f2, h3, c4, g5, d6, b7, e8
a1, g2, d3, f4, h5, b6, e7, c8
. . .
f1, a2, e3, b4, h5, c6, g7, d8
. . .
h1, c2, a3, f4, b5, e6, g7, d8
h1, d2, a3, c4, f5, b6, g7, e8
```

(92 solutions)

WORKED EXAMPLE 11.2

Towers of Hanoi

No discussion of recursion would be complete without the "Towers of Hanoi". Learn how to solve this classic puzzle with an elegant recursive solution. See your E-Text or visit wiley.com/go/bclo3.



Computing & Society 11.1 The Limits of Computation

Have you ever wondered how your instructor or grader makes sure your programming homework is correct? In all likelihood, they look at your solution and perhaps run it with some test inputs. But usually they have a correct solution available. That suggests that there might be an easier way. Perhaps they could feed your program and their correct program into a "program comparator", a computer program that analyzes both programs and determines whether they both compute the same results. Of course, your solution and the program that is known to be correct need not be identical—what matters is that they produce the same output when given the same input.

How could such a program comparator work? Well, the C++ compiler knows how to read a program and make sense of the classes, functions, and statements. So it seems plausible that someone could, with some effort, write a program that reads two C++ programs, analyzes what they do, and determines whether they solve the same task. Of course, such a program would be very attractive to instructors, because it could automate the grading process. Thus, even though no such program exists today, it might be tempting to try to develop one and sell it to universities around the world.

However, before you start raising venture capital for such an effort, you should know that theoretical computer scientists have proven that it is impossible to develop such a program, *no matter how hard you try*.

There are quite a few of these unsolvable problems. The first one, called the *halting problem*, was discovered by the British researcher Alan Turing in 1936. Because his research occurred before the first actual computer was constructed, Turing had to devise a theoretical device, the **Turing machine**, to explain how computers could work. The Turing machine consists of a long magnetic tape, a read/write head, and a program that has numbered instructions of the form: "If the current symbol under the head is *x*, then replace it with *y*, move the head one unit left or right, and continue with instruction *n*" (see figure at the end of this article). Interestingly enough, with just these instructions, you can program just as much as with C++, even though it is incredibly tedious to do so. Theoretical computer scientists like Turing machines because they can be described using nothing more than the laws of mathematics.

Science Photo Library/Photo Researchers.

Alan Turing

Expressed in terms of C++, the halting problem states: "It is impossible to write a program with two inputs, namely the source code of an arbitrary C++ program *P* and a string *I*, that decides whether the program *P*, when executed with the input *I*, will halt without getting into an infinite loop". Of course, for some kinds of programs and inputs, it is possible to decide whether the programs halt with the given input. The halting problem asserts that it is impossible to come up with a single decision-making algorithm that works with all programs and inputs. Note that you can't simply run the program *P* on the input *I* to settle this question. If the program runs for 1,000 days, you don't know that the program is in an infinite loop. Maybe you just have to wait another day for it to stop.

Such a "halt checker", if it could be written, might also be useful for grading homework. An instructor could use it to screen student submissions to see if they get into an infinite loop with a particular input, and then not check them any further. However, as Turing demonstrated, such a program cannot be written. His argument is ingenious and quite simple.

Suppose a "halt checker" program existed. Let's call it *H*. From *H*, we will develop another program, the "killer" program *K*. *K* does the following computation. Its input is a string containing the source code for a program *R*. It then applies the halting checker on the input program *R* and the input string *R*. That is, it checks whether the program *R* halts if its input is its own source code. It sounds bizarre to feed a program to itself, but it isn't impossible. For example, the C++ compiler is written in C++, and you can use it to compile itself. Or, as a simpler example, you can use a word count program to count the words in its own source code.

When *K* gets the answer from *H* that *R* halts when applied to itself, it is programmed to enter an infinite loop. Otherwise *K* exits. In C++, the program might look like this:

```
int main()
{
   string r = read program input;
   HaltChecker checker;
   if (checker.check(r, r))
   {
      while (true) { }
      // Infinite loop
   }
   else
   {
      return 0;
   }
}
```

Now ask yourself: What does the halt checker answer when asked if *K* halts when given *K* as the input? Maybe it finds out that *K* gets into an infinite loop with such an input. But wait, that can't be right. That would mean that `checker.check(r, r)` returns `false` when `r` is the program code of *K*. As you can plainly see, in that case, the `main` function returns, so *K* didn't get into an infinite loop. That shows that *K*

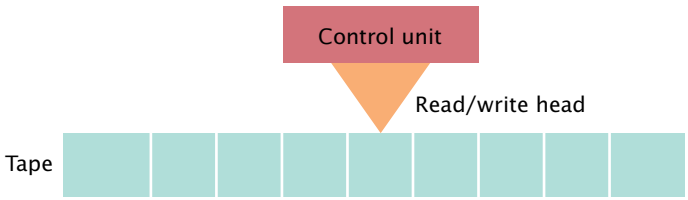
must halt when analyzing itself, so `checker.check(r, r)` should return `true`. But then the `main` function doesn't terminate—it goes into an infinite loop. That shows that it is logically impossible to implement a program that can check whether *every* program halts on a particular input.

It is sobering to know that there are *limits* to computing. There are problems that no computer program, no matter how ingenious, can answer.

Theoretical computer scientists are working on other research involving the nature of computation. One important question that remains unsettled to this day deals with problems that in practice are very time-consuming to solve. It may be that these problems are intrinsically hard, in which case it would be pointless to try to look for better algorithms. Such theoretical research can have important practical applications. For example, right now, nobody knows whether the most common encryption schemes used today could be broken by discovering a new algorithm (see Computing & Society 8.1 for more information on encryption algorithms). Knowing that no fast algorithms exist for breaking a particular code could make us feel more comfortable about the security of encryption.

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
	1	1	left	4
2	0	0	right	2
	1	1	right	2
	2	0	left	3
3	0	0	left	3
	1	1	left	3
	2	2	right	1
4	1	1	right	5
	2	0	left	4



A Turing Machine

CHAPTER SUMMARY

Understand the control flow in a recursive computation.

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.

Identify recursive helper functions for solving a problem.

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Contrast the efficiency of recursive and non-recursive algorithms.

- Occasionally, a recursive solution runs much more slowly than its iterative counterpart. However, in most cases, the recursive solution runs at about the same speed.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Review a complex recursion example that cannot be solved with a simple loop.

- For generating permutations, it is much easier to use recursion than iteration.

Recognize the phenomenon of mutual recursion in an expression evaluator.

- In a mutual recursion, a set of cooperating functions calls each other repeatedly.

Use backtracking to solve problems that require trying out multiple paths.

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

REVIEW EXERCISES

■ **R11.1** Define the terms

a. recursion

b. iteration

c. infinite recursion

d. mutual recursion

■ **R11.2** Give pseudocode for a recursive algorithm that replaces all digits with value 8 in a number with zeroes.

■■ **R11.3** Outline, but do not implement, a recursive solution for finding the smallest value in an array.

■■■ **R11.4** Outline, but do not implement, a recursive solution for finding the kth smallest element in an array. *Hint:* Look at the elements that are less than the initial element. Suppose there are m of them. How should you proceed if $k \leq m$? If $k > m$?

■■ **R11.5** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.

■ **R11.6** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First sort the subarray without the initial element.

■■ **R11.7** Outline, but do not implement, a recursive solution for generating all subsets of the set $\{1, 2, \ldots, n\}$.

■■■ **R11.8** Exercise P11.5 shows an iterative way of generating all permutations of the sequence $(0, 1, \ldots, n - 1)$. Explain why the algorithm produces the right result.

■ **R11.9** Write a recursive definition of x^n, where $n \geq 0$, similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute x^n from x^{n-1}? How does the recursion terminate?

■■ **R11.10** Improve upon Exercise R11.9 by computing x^n as $(x^{n/2})^2$ if n is even. Why is this approach significantly faster? *Hint:* Compute x^{1023} and x^{1024} both ways.

■ **R11.11** Write a recursive definition of $n! = 1 \times 2 \times \cdots \times n$, similar to the recursive definition of the Fibonacci numbers.

■■ **R11.12** Find out how often the recursive version of `fib` calls itself. Keep a global variable `fib_count` and increment it once in every call to `fib`. What is the relationship between `fib(n)` and `fib_count`?

■■■ **R11.13** Let moves(n) be the number of moves required to solve the Towers of Hanoi problem (see Worked Example 11.2). Find a formula that expresses moves(n) in terms of moves($n - 1$). Then show that moves(n) = $2^n - 1$.

■■ **R11.14** Trace the expression evaluator program from Section 11.5 with inputs `3 - 4 + 5`, `3 - (4 + 5)`, `(3 - 4) * 5`, and `3 * 4 + 5 * 6`.

PRACTICE EXERCISES

■ **E11.1** Given a class `Rectangle` with data members `width` and `height`, provide a recursive `get_area` member function. Construct a rectangle whose width is one less than the original and call its `get_area` function.

■■ **E11.2** Given a class `Square` with a data member `width`, provide a recursive `get_area` member function. Construct a square whose width is one less than the original and call its `get_area` function.

■ **E11.3** Write a recursive function for factoring an integer n. First, find a factor f, then recursively factor n/f.

■ **E11.4** Write a recursive function for computing a string with the binary digits of a number. If n is even, then the last digit is `0`. If n is odd, then the last digit is 1. Recursively obtain the remaining digits.

■ **E11.5** If a string has n letters, then the *number* of permutations is given by the factorial function:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

For example, $3! = 1 \times 2 \times 3 = 6$ and there are six permutations of the three-character string `"eat"`. Implement a recursive `factorial` function, using the definitions

$$n! = (n - 1)! \times n$$

and

$$0! = 1$$

■■ **E11.6** Write a recursive function that prints an integer with decimal separators. For example, 12345678 should be printed as `12,345,678`.

■ **E11.7** Write a recursive member function `void reverse()` that reverses a `Sentence` object, which has a `string` data member. For example:

```
Sentence greeting("Hello!");
greeting.reverse();
cout << greeting.get_text() << endl;
```

prints the string `"!olleH"`. Implement a recursive solution by removing the first character, reversing a sentence consisting of the remaining text, and combining the two.

■■ **E11.8** Redo Exercise E11.7 with a recursive helper function that reverses a substring of the sentence text.

■ **E11.9** Implement the `reverse` function of Exercise E11.7 as an iteration.

■■ **E11.10** Use recursion to implement a function `bool find(string s, string t)` that tests whether a string `t` is contained in a string `s`:

```
bool b = find("Mississippi!", "sip"); // Returns true
```

Hint: If the text starts with the string you want to match, then you are done. If not, consider the string that you obtain by removing the first character.

■■ **E11.11** Use recursion to implement a function `int index_of(string s, string t)` that returns the starting position of the first substring of the string `s` that matches `t`. Return –1 if `t` is not a substring of `s`. For example,

```
int n = index_of("Mississippi!", "sip"); // Returns 6
```

Hint: This is a bit trickier than Exercise E11.10, because you need to keep track of how far the match is from the beginning of the string. Make that value an argument for a helper function.

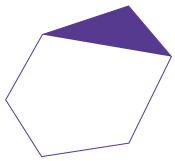
■ **E11.12** Using recursion, find the largest element in a vector of integer values:

```
int maximum(vector<int> values)
```

Hint: Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

■ **E11.13** Using recursion, compute the sum of all values in an array.

■■ **E11.14** Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners (x_1, y_1), (x_2, y_2), (x_3, y_3) has area



$$\frac{\left|x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1\right|}{2}$$

■■ **E11.15** The following method was known to the ancient Greeks for computing square roots. Given a value $x > 0$ and a guess g for the square root, a better guess is $(g + x/g) / 2$. Write a recursive helper function

```
double squareRootGuess(double x, double g)
```

If g^2 is approximately equal to x, return g, otherwise, return `squareRootGuess` with the better guess. Then write a function `double squareRoot(double x)` that uses the helper function.

■■■ **E11.16** Implement a function

```
vector<string> generate_substrings(string s)
```

that generates all substrings of a string. For example, the substrings of the string `"rum"` are the seven strings

```
"r", "ru", "rum", "u", "um", "m", ""
```

Hint: First enumerate all substrings that start with the first character. There are n of them if the string has length n. Then enumerate the substrings of the string that you obtain by removing the first character.

■■■ **E11.17** Implement a function

```
vector<string> generate_subsets(string s)
```

that generates all subsets of characters of a string. For example, the subsets of characters of the string `"rum"` are the eight strings

```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```

Note that the subsets don't have to be substrings—for example, `"rm"` isn't a substring of `"rum"`.

■■■ **E11.18** Recursively generate all ways in which a vector can be split up into a sequence of nonempty subvectors. For example, if you are given the vector `{1, 7, 2, 9}`, return the following vectors of vectors:

```
{{1}, {7}, {2}, {9}}, {{1, 7}, {2}, {9}}, {{1}, {7, 2}, {9}}, {{1, 7, 2}, {9}},
{{1}, {7}, {2, 9}}, {{1, 7}, {2, 9}}, {{1}, {7, 2, 9}}, {{1, 7, 2, 9}}
```

Hint: First generate all subvectors of the vector with the last element removed. The last element can either be a subsequence of length 1, or it can be added to the last subsequence.

■■ **E11.19** Given a vector `a` of integers, recursively find all vectors of elements of `a` whose sum is a given integer `n`.

■■ **E11.20** Suppose you want to climb a staircase with n steps and you can take either one or two steps at a time. Recursively enumerate all paths. For example, if n is 5, the possible paths are:

```
{1, 2, 3, 4, 5}, {1, 3, 4, 5}, {1, 2, 4, 5}, {1, 2, 3, 5}, {1, 4, 5}
```

■■■ **E11.21** Repeat Exercise E11.20, where the climber can take up to k steps at a time.

■■ **E11.22** Given an integer price, list all possible ways of paying for it with $100, $20, $5, and $1 bills, using recursion. Don't list duplicates.

■■ **E11.23** Extend the expression evaluator in Section 11.5 so that it can handle the `%` operator as well as a "raise to a power" operator `^`. For example, `2 ^ 3` should evaluate to 8. As in mathematics, raising to a power should bind more strongly than multiplication: `5 * 2 ^ 3` is 40.

■■ **E11.24** The backtracking algorithm will work for any problem whose partial solutions can be examined and extended. Provide a class

```
class PartialSolution
{
public:
   virtual int examine() const;
   virtual vector<PartialSolution*> extend() const;
   virtual void print() const;
   PartialSolution* solve() const;
};
```

and a derived class `EightQueensPartialSolution`.

■■■ **E11.25** Refine the program for solving the eight queens problem so that rotations and reflections of previously displayed solutions are not shown. Your program should display twelve unique solutions.

■■■ **E11.26** Refine the program for solving the eight queens problem so that the solutions are written to an HTML file, using tables with black and white background for the board and the Unicode character ♕ `'\u2655'` for the white queen.

■■ **E11.27** Generalize the program for solving the eight queens problem to the n queens problem. Your program should prompt for the value of n and display the solutions.

■■ **E11.28** Using backtracking, write a program that solves summation puzzles in which each letter should be replaced by a digit, such as

```
send + more = money
```

Your program should find the solution 9567 + 1085 = 10652. Other examples are `base + ball = games` and `kyoto + osaka = tokyo`.

Hint: In a partial solution, some of the letters have been replaced with digits. In the third example, you would consider all partial solutions where `k` is replaced by `0`, `1`, ... `9`: `0yoto + osa0a = to0yo`, `1yoto + osa1a = to1yo`, and so on. To extend a partial solution, find the first letterand replace all instances with a digit that doesn't yet occur in the partial solution. If a partial solution has no more letters, check whether the sum is correct.

■■ **E11.29** The recursive computation of Fibonacci numbers can be speeded up significantly by keeping track of the values that have already been computed. Provide an implementation of the `fib` function that uses this strategy. Whenever you return a new value, also store it in an auxiliary array. However, before embarking on a computation, consult the array to find whether the result has already been computed. Compare the running time of your improved implementation with that of the original recursive implementation and the loop implementation.

PROGRAMMING PROJECTS

■■ **P11.1** Phone numbers and PIN codes can be easier to remember when you find words that spell out the number on a standard phone keypad. For example, instead of remembering the combination 2633, you can just think of CODE.

Write a recursive function that, given a number, yields all possible spellings (which may or may not be real words).

■■ **P11.2** Continue Exercise P11.1, checking the words against the `/usr/share/dict/words` file on your computer, or the `words.txt` file in the companion code for this book. For a given number, return only actual words.

■■■ **P11.3** With a longer number, you may need more than one word to remember it on a phone pad. For example, 846-386-2633 is TIME TO CODE. Using your work from Exercise P11.2, write a program that, given any number, lists all word sequences that spell the number on a phone pad.

■■■ **P11.4** Change the `permutations` function of Section 11.4 (which computed all permutations at once) to a `PermutationIterator` (which computes them one at a time).

```
class PermutationIterator
{
public:
   PermutationIterator(string s) { . . . }
   string next_permutation() { . . . }
   bool has_more_permutations() { . . . }
};
```

Here is how you would print out all permutations of the string `"eat"`:

```
PermutationIterator iter("eat");
while (iter.has_more_permutations())
{
   cout << iter.next_permutation() << endl;
}
```

Now we need a way to iterate through the permutations recursively. Consider the string `"eat"`. As before, we'll generate all permutations that start with the letter `'e'`,

then those that start with `'a'`, and finally those that start with `'t'`. How do we generate the permutations that start with `'e'`? Make another `PermutationIterator` object (called `tail_iterator`) that iterates through the permutations of the substring `"at"`. In the `next_permutation` function, simply ask `tail_iterator` what *its* next permutation is, and then add the `'e'` at the front. However, there is one special case. When the tail iterator runs out of permutations, all permutations that start with the current letter have been enumerated. Then

- Increment the current position.
- Compute the tail string that contains all letters except for the current one.
- Make a new permutation iterator for the tail string.

You are done when the current position has reached the end of the string.

■■■ **P11.5** The following program generates all permutations of the numbers 0, 1, 2, ... , $n - 1$, *without* using recursion.

```
#include <iostream>
#include <vector>
using namespace std;

void swap(int& x, int& y)
{
   int temp = x;
   x = y;
   y = temp;
}

void reverse(vector<int>& a, int i, int j)
{
   while (i < j)
   {
      swap(a[i], a[j]); i++; j--;
   }
}

bool next_permutation(vector<int>& a)
{
   for (int i = a.size() - 1; i > 0; i--)
   {
      if (a[i - 1] < a[i])
      {
         int j = a.size() - 1;
         while (a[i - 1] > a[j]) { j--; }
         swap(a[i - 1], a[j]);
         reverse(a, i, a.size() - 1);
         return true;
      }
   }
   return false;
}

void print(const vector<int>& a)
{
   for (int i = 0; i < a.size(); i++)
   {
      cout << a[i] << " ";
   }
   cout << endl;
```

```
}

int main()
{
   const int n = 4;
   vector<int> a(n);
   for (int i = 0; i < a.size(); i++) { a[i] = i; }
   print(a);
   while (next_permutation(a)) { print(a); }
   return 0;
}
```

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this technique to get all permutations of the character positions and then compute a string whose ith character is `s[a[i]]`. Use this approach to reimplement the `generate_permutations` function without recursion.

■■ **P11.6** Refine the `is_palindrome` function to work with arbitrary strings, by ignoring non-letter characters and the distinction between upper- and lowercase letters. For example, if the input string is

```
"Madam, I'm Adam!"
```

then you'd first strip off the last character because it isn't a letter, and recursively check whether the shorter string

```
"Madam, I'm Adam"
```

is a palindrome.

■■■ **P11.7** *Escaping a Maze*. You are currently located inside a maze. The walls of the maze are indicated by asterisks (*).

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

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return `true`. Recursively check whether you can escape from one of the empty neighboring locations without visiting the current location.

You need to exclude the current location so that the problem is simplified. Make a class `Maze` and a class `Location`, and provide a member function

```
bool Maze::can_escape(Location from, vector<Location> excluded) const
```

that checks whether one can escape the maze from the given location without visiting the excluded locations. This member function merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.

■■■ **P11.8** Implement an iterator that produces the moves for the Towers of Hanoi puzzle described in Worked Example 11.2. Provide functions `has_more_moves` and `next_move`. The `next_move` function should yield a string describing the next move. For example, the following code prints all moves needed to move five disks from peg 1 to peg 3:

```
DiskMover mover(5, 1, 3);
while (mover.has_more_moves())
{
   cout << mover.next_move() << endl;
}
```

Hint: A disk mover that moves a single disk from one peg to another simply has a `next_move` function that returns a string

```
Move disk from peg source to target
```

A disk mover with more than one disk to move must work harder. It needs another `DiskMover` to help it move the first $d - 1$ disks. Then `next_move` asks that disk mover for its next move until it is done. Then the `next_move` function issues a command to move the dth disk. Finally, it constructs another disk mover that generates the remaining moves.

It helps to keep track of the state of the disk mover:

- `BEFORE_LARGEST`: A helper mover moves the smaller pile to the other peg.
- `LARGEST`: Move the largest disk from the source to the destination.
- `AFTER_LARGEST`: The helper mover moves the smaller pile from the other peg to the target.
- `DONE`: All moves are done.

■■ **P11.9** Using the `PartialSolution` class and `solve` function from Exercise E11.24, provide a class `MazePartialSolution` for solving the maze escape problem of Exercise P11.7.

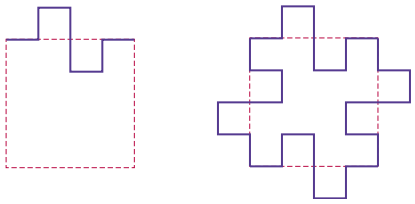
■■■ **P11.10** The expression evaluator in Section 11.5 returns the value of an expression. Modify the evaluator so that it returns a pointer to an object of a class that is derived from the `Expression` class. There are five such classes: `Constant`, `Sum`, `Difference`, `Product`, and `Quotient`. The `Expression` class has a member function `int value()`.

■■■ **P11.11** Refine the expression evaluator of Exercise P11.10 so that expressions can contain the variable `x`. For example, `3*x*x+4*x+5` is a valid expression. Change the `Expression` class so that its `value` function has as parameter the value that `x` should take. Add a class `Variable` that denotes an `x`. Write a program that reads an expression string and a value for `x`, translates the expression string into an `Expression` object, and prints the result of calling `value(x)`.

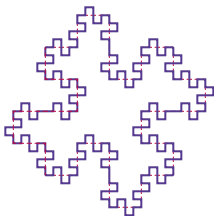
■■ **P11.12** Add a `to_string` function to the `Expression` class (as described in Exercise P11.10 and Exercise P11.11) that returns a string representation of the expression. It is ok to use more parentheses than required in mathematical notation. For example, for the expression `3*x*x+5`, you can print `(((3*x)*x)+5)`.

■■■ **P11.13** Write a program that reads an expression involving integers and the variable `x` into an `Expression` object, and then computes the derivative. Add a function `Expression derivative()` to the `Expression` class. Use the rules from calculus for computing the derivative of a sum, difference, product, quotient, constant, or variable. Don't simplify the result. Print the resulting expression. For example, when reading `x * x`, you should print `((1*x)+(x*1))`.

■■■ **Media P11.14** *The Koch Square.* Start with a square. Replace each straight line with six line segments:



Repeat the process:



Write a program that draws the iterations of the shape. Supply a button that, when clicked, produces the next iteration. Use the `Picture` class from the `media` directory in your companion code.

WORKED EXAMPLE 11.1

Finding Files

Problem Statement Your task is to print the names of all files in a directory tree that end in a given extension.

You are given a class `DirectoryEntry` (in the book's companion code). An object of this class can represent either a directory or a regular file, and the member function

```
boolean is_directory()
```

tells you which it is.

The member function

```
vector<DirectoryEntry> children()
```

yields the entries in a given directory. These can be directories or files.

For example, consider the directory tree at right, and suppose the `DirectoryEntry` object entry represents the `code` directory. Then `entry.is_directory()` returns true, and `entry.children()` returns a vector containing `DirectoryEntry` objects describing `code/ch01`, `code/ch02`, and `code/ch03`.

Step 1 Break the input into parts that can themselves be inputs to the problem.

Our problem has two inputs: A `DirectoryEntry` object representing a directory tree, and an extension. Clearly, nothing is gained from manipulating the extension. However, there is an obvious way of chopping up the directory tree:

- Consider all files in the root level of the directory tree.
- Then consider each tree formed by a subdirectory.

This leads to a valid strategy. Find matching files in the root directory, and then recursively find them in each child subdirectory.

```
For each DirectoryEntry object in the root
   If the DirectoryEntry object is a directory
      Recursively find files in that directory.
   Else if the name ends in the extension
      Print the name.
```

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

We are asked to simply print the files that we find, so there aren't any results to combine.

Had we been asked to produce a vector of the found files, we would place all matches of the root directory into a vector and add all results from the subdirectories into the same vector.

Step 3 Find solutions to the simplest inputs.

The simplest input is a file that isn't a directory. In that case, we simply check whether it ends in the given extension, and if so, print it.

Step 4 Implement the solution by combining the simple cases and the reduction step.

In our case, the reduction step is simply to look at the files and subdirectories:

```
For each child in children
   If the child is a directory
      Recursively find files in the child.
   Else
      If the name of child ends in extension
         Print the name.
```

Here is a recursive function that carries out this plan:

```
void find(DirectoryEntry directory, string extension)
{
   vector<DirectoryEntry> entries = directory.children();
   for (int i = 0; i < entries.size(); i++)
   {
      DirectoryEntry entry = entries[i];
      if (entry.is_directory())
      {
         find(entry, extension);
      }
      else if (entry.extension() == extension)
      {
         cout << entry.name() << endl;
      }
   }
}
```

The complete solution is in the `worked_example_1` folder of your companion code.

worked_example_1/filefinder.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <vector>
#include "direntry.h"

using namespace std;

void find(DirectoryEntry directory, string extension)
{
   vector<DirectoryEntry> entries = directory.children();
   for (int i = 0; i < entries.size(); i++)
   {
      DirectoryEntry entry = entries[i];
      if (entry.is_directory())
      {
         find(entry, extension);
      }
      else if (entry.extension() == extension)
      {
         cout << entry.name() << endl;
      }
   }
}

int main()
{
   find(DirectoryEntry(".."), "cpp");
}
```

WORKED EXAMPLE 11.2

Towers of Hanoi

Problem Statement The "Towers of Hanoi" puzzle has a board with three pegs and a stack of disks of decreasing size, initially on the first peg (see Figure 8).

The goal is to move all disks to the third peg. One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

Legend has it that a temple (presumably in Hanoi) contains such an assembly, with sixty-four golden disks, which the priests move in the prescribed fashion. When they have arranged all disks on the third peg, the world will come to an end.

Let us help out by writing a program that prints instructions for moving the disks.

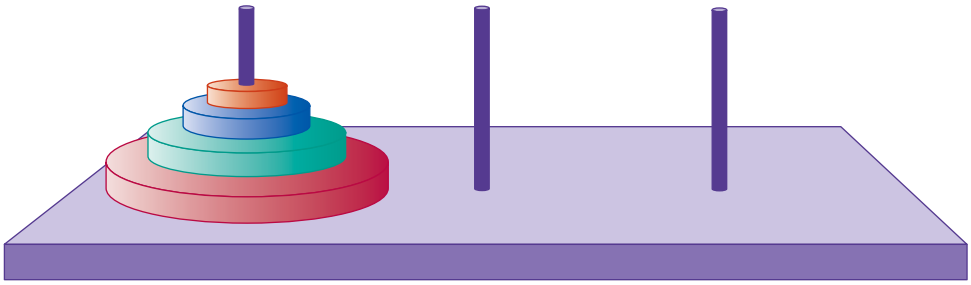


Figure 8 Towers of Hanoi

Consider the problem of moving d disks from peg p_1 to peg p_2, where p_1 and p_2 are 1, 2, or 3, and $p_1 \neq p_2$. Because $1 + 2 + 3 = 6$, we can get the index of the remaining peg as $p_3 = 6 - p_1 - p_2$.

Now we can move the disks as follows:

- Move the top $d - 1$ disks from p_1 to p_3
- Move one disk (the one on the bottom of the pile of d disks) from p_1 to p_2
- Move the $d - 1$ disks that were parked on p_3 to p_2

The first and third step need to be handled recursively, but because we move one fewer disk, the recursion will eventually terminate.

It is very straightforward to translate the algorithm into C++. For the second step, we simply print out the instruction for the priest, something like

```
Move disk from peg 1 to 3
```

worked_example_2/hanoi_instr.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  /**
6     Print instructions for moving a pile of disks from one peg to another.
7     @param disks the number of disks to move
8     @param from the peg from which to move the disks
9     @param to the peg to which to move the disks
10 */
11 void move(int disks, int from, int to)
12 {
```

```
13    if (disks > 0)
14    {
15       int other = 6 - from - to;
16       move(disks - 1, from, other);
17       cout << "Move disk from peg " << from << " to " << to << endl;
18       move(disks - 1, other, to);
19    }
20 }
21
22 int main()
23 {
24    move(5, 1, 3);
25    return 0;
26 }
```

Program Run

```
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
```

These instructions may suffice for the priests, but unfortunately it is not easy for us to see what is going on. Let's improve the program so that it actually carries out the instructions and shows the contents of the towers after each move.

We represent each tower as a vector of integers, corresponding to the disk sizes. We use a class `Puzzle` to hold the three towers.

```
class Puzzle
{
```

```
public:
   Puzzle(int disks);
   void move_single_disk(int from, int to);
   void move(int ndisks, int from, int to);
   void print() const;
private:
   vector<int> towers[3];
};
```

The constructor populates the leftmost tower with the given number of disks, with the largest disk at the bottom:

```
Puzzle::Puzzle(int ndisks)
{
   for (int d = ndisks; d >= 1; d--)
   {
      towers[0].push_back(d);
   }
}
```

The `move_single_disk` member function moves a single disk from one tower to another:

```
void Puzzle::move_single_disk(int from, int to)
{
   int last = towers[from].size() - 1;
   int disk = towers[from][last];
   towers[from].pop_back();
   towers[to].push_back(disk);
}
```

As with the preceding solution, the `move` function recursively moves a smaller pile of disks, then a single disk, and then again the smaller pile:

```
void Puzzle::move(int disks, int from, int to)
{
   if (disks > 0)
   {
      int other = 3 - from - to;
      move(disks - 1, from, other);
      move_single_disk(from, to);
      print();
      move(disks - 1, other, to);
   }
}
```

We also print the towers. The program output is:

```
5 4 3 2 1 | |
5 4 3 2 | | 1
5 4 3 | 2 | 1
5 4 3 | 2 1 |
5 4 | 2 1 | 3
5 4 1 | 2 | 3
5 4 1 | | 3 2
5 4 | | 3 2 1
5 | 4 | 3 2 1
5 | 4 1 | 3 2
5 2 | 4 1 | 3
5 2 1 | 4 | 3
5 2 1 | 4 3 |
5 2 | 4 3 | 1
```

```
5 | 4 3 2 | 1
5 | 4 3 2 1 |
| 4 3 2 1 | 5
1 | 4 3 2 | 5
1 | 4 3 | 5 2
| 4 3 | 5 2 1
3 | 4 | 5 2 1
3 | 4 1 | 5 2
3 2 | 4 1 | 5
3 2 1 | 4 | 5
3 2 1 | | 5 4
3 2 | | 5 4 1
3 | 2 | 5 4 1
3 | 2 1 | 5 4
| 2 1 | 5 4 3
1 | 2 | 5 4 3
1 | | 5 4 3 2
| | 5 4 3 2 1
```

That's better. Now you can see how the disks move. You can check that all moves are legal—the disk size always decreases.

You can see that it takes $31 = 2^5 - 1$ moves to solve the puzzle for 5 disks. With 64 disks, it takes $2^{64} - 1 = 18446744073709551615$ moves. If the priests can move one disk per second, it takes about 585 billion years to finish the job. Because the earth is about 4.5 billion years old at the time this book is written, we don't have to worry too much whether the world will really come to an end when they are done.

worked_example_2/hanoi_moves.cpp

```
#include <iostream>
#include <vector>

using namespace std;

class Puzzle
{
public:
   Puzzle(int disks);
   void move_single_disk(int from, int to);
   void move(int ndisks, int from, int to);
   void print() const;
private:
   vector<int> towers[3];
};

Puzzle::Puzzle(int ndisks)
{
   for (int d = ndisks; d >= 1; d--)
   {
      towers[0].push_back(d);
   }
}

void Puzzle::move_single_disk(int from, int to)
{
   int last = towers[from].size() - 1;
   int disk = towers[from][last];
   towers[from].pop_back();
   towers[to].push_back(disk);
}
```

```
32
33 void Puzzle::move(int disks, int from, int to)
34 {
35    if (disks > 0)
36    {
37       int other = 3 - from - to;
38       move(disks - 1, from, other);
39       move_single_disk(from, to);
40       print();
41       move(disks - 1, other, to);
42    }
43 }
44
45 void Puzzle::print() const
46 {
47    for (int i = 0; i < 3; i++)
48    {
49       if (i > 0) cout << "| ";
50       for (int j = 0; j < towers[i].size(); j++)
51       {
52          cout << towers[i][j] << " ";
53       }
54    }
55    cout << endl;
56 }
57
58 int main()
59 {
60    Puzzle hanoi(5);
61    hanoi.print();
62    hanoi.move(5, 0, 2);
63    return 0;
64 }
```

CHAPTER 12

SORTING AND SEARCHING

© Volkan Ersoy/iStockphoto.

CHAPTER GOALS

To compare the selection sort and merge sort algorithms

To study the linear search and binary search algorithms

To appreciate that algorithms for the same task can differ widely in performance

To understand the big-Oh notation

To be able to estimate and compare the performance of algorithms

To write code to measure the running time of a program

CHAPTER CONTENTS

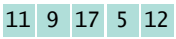
One of the most common tasks in data processing is sorting. For example, an array of employees often needs to be displayed in alphabetical order or sorted by salary. In this chapter, you will learn several sorting methods and techniques for comparing their performance. These techniques are useful not only for sorting algorithms, but also for analyzing other algorithms.

Once an array of elements is sorted, one can rapidly locate individual elements. You will study the *binary search* algorithm that carries out this fast lookup.

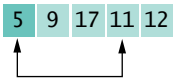
12.1 Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

A *sorting algorithm* rearranges the elements of an array so that they are stored in sorted order. In this section, we show you the first of several sorting algorithms, called **selection sort**. Consider the following array `a`:



An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in `a[3]`. You should move the 5 to the beginning of the array. Of course, there is already an element stored in `a[0]`, namely 11. Therefore you cannot simply move `a[3]` into `a[0]` without moving the 11 somewhere else. You don't yet know where the 11 should end up, but you know for certain that it should not be in `a[0]`. Simply get it out of the way by *swapping it* with `a[3]`:

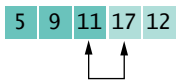


Now the first element is in the correct place. In the foregoing figure, the darker color indicates the portion of the array that is already sorted.

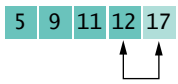
In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.

Next take the minimum of the remaining entries `a[1]...a[4]`. That minimum value, 9, is already in the correct place. You don't need to do anything in this case, simply extend the sorted area by one to the right:

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:



Now the unsorted region is only two elements long; keep to the same successful strategy. The minimum element is 12. Swap it with the first value, 17:



That leaves you with an unprocessed region of length 1, but of course a region of length 1 is always sorted. You are done.

If speed was not an issue for us, we could stop the discussion of sorting right here. However, the selection sort algorithm shows disappointing performance when run on large data sets, and it is worthwhile to study better sorting algorithms.

Here is the implementation of the selection sort algorithm:

sec01/selsort.cpp

```
1  #include <cstdlib>
2  #include <ctime>
3  #include <iostream>
4
5  using namespace std;
6
7  /**
8     Gets the position of the smallest element in an array range.
9     @param a the array
10    @param from the beginning of the range
11    @param to the end of the range
12    @return the position of the smallest element in
13    the range a[from]...a[to]
14 */
15 int min_position(int a[], int from, int to)
16 {
17    int min_pos = from;
18    for (int i = from + 1; i <= to; i++)
19    {
20       if (a[i] < a[min_pos]) { min_pos = i; }
21    }
22    return min_pos;
23 }
24
25 /**
26    Swaps two integers.
27    @param x the first integer to swap
28    @param y the second integer to swap
29 */
```

```
30 void swap(int& x, int& y)
31 {
32    int temp = x;
33    x = y;
34    y = temp;
35 }
36
37 /**
38    Sorts an array using the selection sort algorithm.
39    @param a the array to sort
40    @param size the number of elements in a
41 */
42 void selection_sort(int a[], int size)
43 {
44    int next; // The next position to be set to the minimum
45
46    for (next = 0; next < size - 1; next++)
47    {
48       // Find the position of the minimum starting at next
49       int min_pos = min_position(a, next, size - 1);
50       // Swap the next element and the minimum
51       swap(a[next], a[min_pos]);
52    }
53 }
54
55 /**
56    Prints all elements in an array.
57    @param a the array to print
58    @param size the number of elements in a
59 */
60 void print(int a[], int size)
61 {
62    for (int i = 0; i < size; i++)
63    {
64       cout << a[i] << " ";
65    }
66    cout << endl;
67 }
68
69 int main()
70 {
71    srand(time(0));
72    const int SIZE = 20;
73    int values[SIZE];
74    for (int i = 0; i < SIZE; i++)
75    {
76       values[i] = rand() % 100;
77    }
78    print(values, SIZE);
79    selection_sort(values, SIZE);
80    print(values, SIZE);
81    return 0;
82 }
```

Program Run

```
60 47 70 39 6 12 96 93 83 53 36 29 50 97 94 95 38 17 8 26
6 8 12 17 26 29 36 38 39 47 50 53 60 70 83 93 94 95 96 97
```

12.2 Profiling the Selection Sort Algorithm

To measure the performance of a program, one could simply run it and measure how long it takes by using a stopwatch. However, most of our programs run very quickly, and it is not easy to time them accurately in this way. Furthermore, when a program does take a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory (for which it should not be penalized) or for screen output (whose speed depends on the computer model, even for computers with identical CPUs). Instead we use the `time` function. The call

```
int now = time(0);
```

sets `now` to the number of seconds that have elapsed since January 1, 1970. We don't care about this value, but if we have two time measurements, then their difference yields the elapsed time:

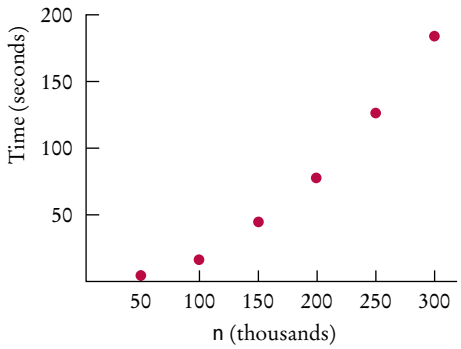
```
int before = time(0);
selection_sort(values, size);
int after = time(0);

cout << "Elapsed time = " << after - before
      << " seconds" << endl;
```

To measure the running time of a function, get the current time immediately before and after the function call.

By measuring the time just before and after the sorting, you don't count the time it takes to initialize the array or the time during which the program waits for the user to provide inputs. The table in Figure 1 shows the results of some sample runs.

These measurements were obtained on a Pentium processor with a clock speed of 2.40 GHz running Linux. On another computer, the actual numbers will differ, but the relationship between the numbers will be the same. Figure 1 shows a plot of the measurements.

As you can see, doubling the size of the data set more than doubles the time needed to sort it.



n	Seconds
50,000	4
100,000	17
150,000	46
200,000	79
250,000	125
300,000	183

Figure 1 Time Taken by Selection Sort

EXAMPLE CODE See sec02 of your companion code for the complete selection sort timing program.

12.3 Analyzing the Performance of the Selection Sort Algorithm

Let us count the number of operations that a program must carry out to sort an array using the selection sort algorithm. Actually, we don't know how many machine operations are generated for each C++ instruction or which of those instructions are more time-consuming than others, but we can make a simplification. Simply count how often an array element is *visited*. Each visit requires about the same amount of work by other operations, such as incrementing subscripts and comparing values.

Let n be the size of the array. First, you must find the smallest of n numbers. To achieve this, you must visit n elements. Then swap the elements, which takes two visits. (You may argue that there is a certain probability that you don't need to swap the values. That is true, and one can refine the computation to reflect that observation. As we will soon see, doing so would not affect the overall conclusion.) In the next step, you need to visit only $n - 1$ elements to find the minimum and then visit two of them to swap them. In the following step, $n - 2$ elements are visited to find the minimum. The last run visits two elements to find the minimum and requires two visits to swap the elements. Therefore, the total number of visits is

$$\begin{aligned} n + 2 + (n - 1) + 2 + \cdots + 2 + 2 &= \big(n + (n - 1) + \cdots + 2\big) + (n - 1) \cdot 2 \\ &= \big(2 + \cdots + (n - 1) + n\big) + (n - 1) \cdot 2 \\ &= \frac{n(n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

because

$$1 + 2 + \cdots + (n - 1) + n = \frac{n(n + 1)}{2}$$

After multiplying out and collecting terms of n, you find that the number of visits is

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

This is a quadratic equation in n. That explains why the graph of Figure 1 looks approximately like a parabola.

Now simplify the analysis further. When you plug in a large value for n (for example, 1,000 or 2,000), then $\frac{1}{2}n^2$ is 500,000 or 2,000,000. The lower term, $\frac{5}{2}n - 3$, doesn't contribute much at all; it is just 2,497 or 4,997, a drop in the bucket compared to the hundreds of thousands or even millions of comparisons specified by the $\frac{1}{2}n^2$ term. Just ignore these lower-level terms.

Next, ignore the constant factor $\frac{1}{2}$. You need not be interested in the actual count of visits for a single n. You need to compare the ratios of counts for different values of n^2. For example, you can say that sorting an array of 2,000 numbers requires four times as many visits as sorting an array of 1,000 numbers:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

The factor $\frac{1}{2}$ cancels out in comparisons of this kind. We will simply say, "The number of visits is of order n^2". That way, we can easily see that the number of comparisons increases fourfold when the size of the array doubles: $(2n)^2 = 4n^2$.

To indicate that the number of visits is of order n^2, computer scientists often use **big-Oh notation**: The number of visits is $O(n^2)$. This is a convenient shorthand.

To turn an exact expression such as

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

into big-Oh notation, simply locate the fastest-growing term, n^2, and ignore its constant coefficient, $\frac{1}{2}$ in this case, *no matter how large or small it may be.*

Computer scientists use big-Oh notation to describe how fast a function grows.

In general, the expression $f(n) = O(g(n))$ means that f grows no faster than g, or, more formally, that for all n larger than some threshold, the ratio $f(n)/g(n)$ is less than a constant value C. The function g is usually chosen to be very simple, such as n^2 in our example.

You observed before that the actual number of machine operations, and the actual amount of time that the computer spends on them, is approximately proportional to the number of element visits. Maybe there are about 10 machine operations (increments, comparisons, memory loads, and stores) for every element visit. The number of machine operations is then approximately $10 \times \frac{1}{2}n^2$. As before, we aren't interested in the coefficient, so we can say that the number of machine operations, and hence the time spent on the sorting, is of the order of n^2 or $O(n^2)$.

Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.

The sad fact remains that doubling the size of the array causes a fourfold increase in the time required for sorting it. When the size of an array increases by a factor of 100, the sorting time increases by a factor of 10,000. To sort an array of ten million entries takes 10,000 times as long as sorting 100,000 entries. If 100,000 entries can be sorted in about 17 seconds (as in our example), then sorting ten million entries requires about two days. You will see in the next section how one can dramatically improve the performance of the sorting process by choosing a more sophisticated algorithm.

Special Topic 12.1

Oh, Omega, and Theta

We have used the big-Oh notation somewhat casually in this chapter to describe the growth behavior of a function. Here is the formal definition of the big-Oh notation: Suppose we have a function $T(n)$. Usually, it represents the processing time of an algorithm for a given input of size n. But it could be any function. Also, suppose that we have another function $f(n)$. It is usually chosen to be a simple function, such as $f(n) = n^k$ or $f(n) = \log(n)$, but it too can be any function. We say

$$T(n) \text{ is } O(f(n))$$

if $T(n)$ grows at a rate that is bounded by $f(n)$. More formally, we require that for all n larger than some threshold, the ratio $T(n) / f(n) \leq C$ for some constant value C.

If $T(n)$ is a polynomial of degree k in n, then one can show that $T(n)$ is $O(n^k)$. Later in this chapter, we will encounter functions that are $O(\log(n))$ or $O(n \log(n))$. Some algorithms take much more time. For example, one way of sorting a sequence is to compute all of its permutations, until you find one that is in increasing order. Such an algorithm takes $O(n!)$ time, which is very bad indeed.

Table 1 shows common big-Oh expressions, sorted by increasing growth.

Table 1 Common Big-Oh Growth Rates

Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Strictly speaking, $T(n)$ is $O(f(n))$ means that T grows no faster than f. But it is permissible for T to grow much more slowly. Thus, it is technically correct to state that $T(n) = n^2 + 5n - 3$ is $O(n^3)$ or even $O(n^{10})$.

Computer scientists have invented additional notation to describe the growth behavior of functions more accurately. The assertion

$$T(n) \text{ is } \Omega(f(n))$$

means that T grows at least as fast as f, or, formally, that for all n larger than some threshold, the ratio $T(n) / f(n) \geq C$ for some constant value C. (The Ω symbol is the capital Greek letter omega.) For example, $T(n) = n^2 + 5n - 3$ is $\Omega(n^2)$ or even $\Omega(n)$.

The assertion

$$T(n) \text{ is } \Theta(f(n))$$

means that T and f grow at the same rate—that is, $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$. (The Θ symbol is the capital Greek letter theta.)

The Θ notation gives the most precise description of growth behavior. For example, $T(n) = n^2 + 5n - 3$ is $\Theta(n^2)$ but not $\Theta(n)$ or $\Theta(n^3)$.

The notations are very important for the precise analysis of algorithms. However, in casual conversation it is common to stick with big-Oh, while still giving an estimate as good as one can make.

Special Topic 12.2

Insertion Sort

Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence

```
a[0] a[1] ... a[k]
```

of an array is already sorted. (When the algorithm starts, we set `k` to `0`.) We enlarge the initial sequence by inserting the next array element, `a[k + 1]`, at the proper location. When we reach the end of the array, the sorting process is complete.

For example, suppose we start with the array

11	9	16	5	7

Of course, the initial sequence of length 1 is already sorted. We now add `a[1]`, which has the value 9. The element needs to be inserted before the element 11. The result is

9	11	16	5	7

Next, we add `a[2]`, which has the value 16. This element does not have to be moved.

9	11	16	5	7

We repeat the process, inserting `a[3]` or 5 at the very beginning of the initial sequence.

5	9	11	16	7

Finally, `a[4]` or 7 is inserted in its correct position, and the sorting is completed.

The following function implements the insertion sort algorithm:

```
/**
   Sorts an array, using insertion sort.
   @param a the array to sort
*/
void insertion_sort(int a[], int size)
{
   for (int i = 1; i < size; i++)
   {
      int next = a[i];
      // Move all larger elements up
      int j = i;
      while (j > 0 && a[j - 1] > next)
      {
         a[j] = a[j - 1];
         j--;
      }
      // Insert the element
      a[j] = next;
   }
}
```

How efficient is this algorithm? Let n denote the size of the array. We carry out $n - 1$ iterations. In the kth iteration, we have a sequence of k elements that is already sorted, and we need to insert a new element into the sequence. For each insertion, we need to visit the elements of the initial sequence until we have found the location in which the new element can be inserted. Then we need to move up the remaining elements of the sequence. Thus, $k + 1$ array elements are visited. Therefore, the total number of visits is

$$2 + 3 + \cdots + n = \frac{n(n+1)}{2} - 1$$

Insertion sort is an $O(n^2)$ algorithm.

We conclude that insertion sort is an $O(n^2)$ algorithm, on the same order of efficiency as selection sort.

Insertion sort has a desirable property: Its performance is $O(n)$ if the array is already sorted—see Exercise R12.19. This is a useful property in practical applications, in which data sets are often partially sorted.

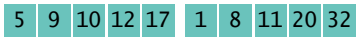
© Kirby Hamilton/iStockphoto.

Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.

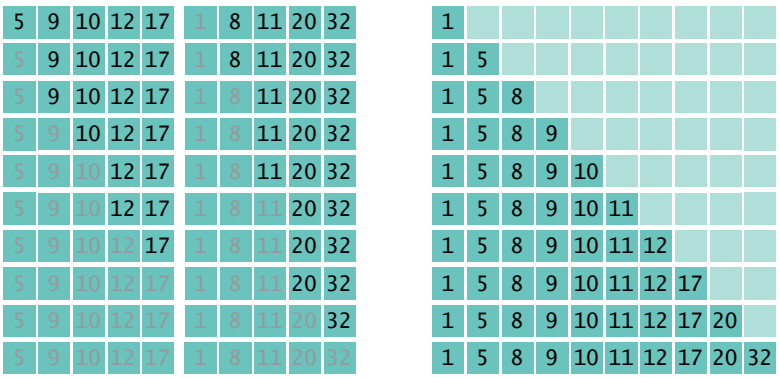
EXAMPLE CODE See special_topic_2 of your companion code for a program that illustrates sorting with insertion sort.

12.4 Merge Sort

In this section, you will learn about the **merge sort** algorithm, a much more efficient algorithm than selection sort. The basic idea behind merge sort is very simple. Suppose you have an array of 10 integers. Engage in a bit of wishful thinking and hope that the first half of the array is already perfectly sorted, and the second half is too, like this:



Now it is an easy matter to *merge* the two sorted arrays into a sorted array, simply by taking a new element from either the first or the second subarray and choosing the smaller of the elements each time:



In fact, you probably performed this merging before when you and a friend had to sort a pile of papers. You and the friend split the pile in half, each of you sorted your half, and then you merged the results together.

This is all well and good, but it doesn't seem to solve the problem for the computer. It still has to sort the first and second halves, because it can't very well ask a few buddies to pitch in. As it turns out, though, if the computer keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together, it carries out dramatically fewer steps than the selection sort requires.

© Rich Legg/iStockphoto.

In merge sort, one sorts each half, then merges the sorted halves.

The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, then merging the sorted halves.

Let us write a program that implements this idea. Because we will call the `merge_sort` function multiple times to sort portions of the array, we will supply the range of elements that we would like to have sorted:

```
void merge_sort(int a[], int from, int to)
{
   if (from == to) { return; }
   int mid = (from + to) / 2;

   // Sort the first and the second half
   merge_sort(a, from, mid);
   merge_sort(a, mid + 1, to);
   merge(a, from, mid, to);
}
```

The `merge` function is somewhat long but quite straightforward—see the following code listing for details.

sec04/mergesort.cpp

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

using namespace std;

/**
   Merges two adjacent ranges in an array.
   @param a the array with the elements to merge
   @param from the start of the first range
   @param mid the end of the first range
   @param to the end of the second range
*/
void merge(int a[], int from, int mid, int to)
{
   int n = to - from + 1; // Size of the range to be merged
   // Merge both halves into a temporary array b
   // We allocate the array dynamically because its size is only
   // known at run time—see Section 7.4
   int* b = new int[n];

   int i1 = from;
      // Next element to consider in the first half
   int i2 = mid + 1;
      // Next element to consider in the second half
   int j = 0; // Next open position in b

   // As long as neither i1 nor i2 is past the end, move the smaller
   // element into b

   while (i1 <= mid && i2 <= to)
   {
      if (a[i1] < a[i2])
      {
         b[j] = a[i1];
         i1++;
      }
      else
      {
         b[j] = a[i2];
         i2++;
      }
      j++;
   }

   // Note that only one of the two while loops below is executed

   // Copy any remaining entries of the first half
   while (i1 <= mid)
   {
      b[j] = a[i1];
      i1++;
      j++;
   }
   // Copy any remaining entries of the second half
   while (i2 <= to)
   {
```

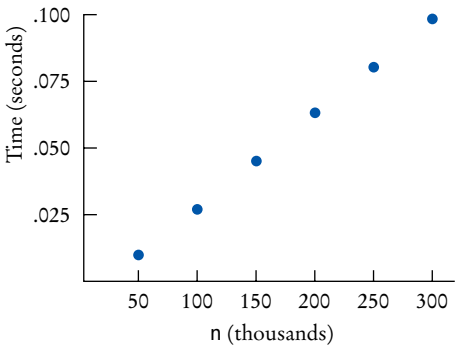
```
58          b[j] = a[i2];
59          i2++;
60          j++;
61       }
62
63       // Copy back from the temporary array
64       for (j = 0; j < n; j++)
65       {
66          a[from + j] = b[j];
67       }
68
69       // The temporary array is no longer needed
70       delete[] b;
71    }
72
73    /**
74       Sorts the elements in a range of an array.
75       @param a the array with the elements to sort
76       @param from start of the range to sort
77       @param to end of the range to sort
78    */
79    void merge_sort(int a[], int from, int to)
80    {
81       if (from == to) { return; }
82       int mid = (from + to) / 2;
83       // Sort the first half and the second half
84       merge_sort(a, from, mid);
85       merge_sort(a, mid + 1, to);
86       merge(a, from, mid, to);
87    }
88
89    /**
90       Prints all elements in an array.
91       @param a the array to print
92       @param size the number of elements in a
93    */
94    void print(int a[], int size)
95    {
96       for (int i = 0; i < size; i++)
97       {
98          cout << a[i] << " ";
99       }
100      cout << endl;
101   }
102
103   int main()
104   {
105      srand(time(0));
106      const int SIZE = 20;
107      int values[SIZE];
108      for (int i = 0; i < SIZE; i++)
109      {
110         values[i] = rand() % 100;
111      }
112      print(values, SIZE);
113      merge_sort(values, 0, SIZE - 1);
114      print(values, SIZE);
115      return 0;
116   }
```

12.5 Analyzing the Merge Sort Algorithm

The merge sort algorithm looks much more complicated than the selection sort algorithm, and it appears that it may well take much longer to carry out these repeated subdivisions. However, the timing results for merge sort look much better than those for selection sort (see the table in Figure 2). Sorting an array with 300,000 elements takes less than one second on our test machine, whereas the selection sort takes 183 seconds.

EXAMPLE CODE See sec05 of your companion code for the complete timing program for merge sort.

In order to get precise timing results, it is best to run the algorithm multiple times, and then divide the total time by the number of runs. Figure 2 shows typical results and a graph plotting the relationship. Note that the graph does not have a parabolic shape. Instead, it appears as if the running time grows approximately linearly with the size of the array.



n	Merge Sort (seconds)	Selection Sort (seconds)
50,000	0.010	4
100,000	0.028	17
150,000	0.045	46
200,000	0.064	79
250,000	0.080	125
300,000	0.097	183

Figure 2 Time Taken by Merge Sort

To understand why the merge sort algorithm is such a tremendous improvement, let us estimate the number of array element visits. First, we tackle the merge process that happens after the first and second halves have been sorted.

Each step in the merge process adds one more element to `b`. There are n elements in `b`. That element may come from the first or second half of `a`, and in most cases the elements from the two halves must be compared to see which one to take. Count that as 3 visits per element (one for `b` and one each for the two halves of `a`), or $3n$ visits total. Then you must copy back from `b` to `a`, yielding another $2n$ visits, for a total of $5n$.

If you let $T(n)$ denote the number of visits required to sort a range of n elements through the merge sort process, then you obtain

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

because sorting each half takes $T(n/2)$ visits. Actually, if n is not even, then you have one array of size $(n - 1)/2$ and one of size $(n + 1)/2$. Although it turns out that this detail does not affect the outcome of the computation, you can assume for now that n is a power of 2, say $n = 2^m$. This way, all arrays can be evenly divided into two parts.

Unfortunately, the formula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$

does not clearly tell you the relationship between n and $T(n)$. To understand the relationship, evaluate $T(n/2)$, using the same formula:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Therefore

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Do that again:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

hence

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

This generalizes from 2, 4, 8, to the kth power of 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Recall that we assume that $n = 2^m$; hence, for $k = m$,

$$\begin{aligned} T(n) &= 2^m T\left(\frac{n}{2^m}\right) + 5nm \\ &= nT(1) + 5nm \\ &= n + 5n\log_2(n) \end{aligned}$$

Because $n = 2^m$, you have $m = \log_2(n)$.

To establish the growth order, you drop the lower order term n and are left with $5n\log_2(n)$. Drop the constant factor 5. It is also customary to drop the base of the logarithm because all logarithms are related by a constant factor. For example,

$$\log_2(x) = \log_{10}(x) / \log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Hence we say that merge sort is an $O(n\log(n))$ algorithm.

Merge sort is an $O(n\log(n))$ algorithm. The $n\log(n)$ function grows much more slowly than n^2.

Is the $O(n\log(n))$ merge sort algorithm better than an $O(n^2)$ selection sort algorithm? You bet it is. Recall that it took $100^2 = 10{,}000$ times as long to sort ten million records as it took to sort 100,000 records with the $O(n^2)$ algorithm. With the $O(n\log(n))$ algorithm, the ratio is

$$\frac{10{,}000{,}000\log(10{,}000{,}000)}{100{,}000\log(100{,}000)} = 100\left(\frac{7}{5}\right) = 140$$

Suppose for the moment that merge sort takes the same time as selection sort to sort an array of 100,000 integers, that is, 17 seconds on the test machine. (Actually, as you

have seen, it is much faster than that.) Then it would take about 40 minutes to sort 10,000,000 integers. Contrast that with selection sort, which would take about two days for the same task. As you can see, even if it takes you several hours to learn about a better algorithm, that can be time well spent.

In this chapter you have barely begun to scratch the surface of this interesting topic. There are many sort algorithms, some with even better performance than the merge sort algorithm, and the analysis of these algorithms can be quite challenging. If you are a computer science major, you may revisit these important issues in later computer science classes.

Special Topic 12.3

The Quicksort Algorithm

Quicksort is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a range `a[from] ... a[to]` of the array `a`, first rearrange the elements in the range so that no element in the range `a[from] ... a[p]` is larger than any element in the range `a[p + 1] ... a[to]`. This step is called *partitioning* the range.

For example, suppose we start with a range

5	3	2	6	4	1	3	7

Here is a partitioning of the range. Note that the partitions aren't yet sorted.

3	3	2	1	4	6	5	7

You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm on the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

1	2	3	3	4	5	6	7

Quicksort is implemented recursively as follows:

```
void quicksort(int a[], int from, int to)
{
   if (from >= to) { return; }
   int p = partition(a, from, to);
   quicksort(a, from, p);
   quicksort(a, p + 1, to);
}
```

Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range, `a[from]`, as the pivot.

Now form two regions `a[from] ... a[i]`, consisting of values at most as large as the pivot and `a[j] ... a[to]`, consisting of values at least as large as the pivot. The region `a[i + 1] ... a[j - 1]` consists of values that haven't been analyzed yet. (See Figure 3.) At the beginning, both the left and right areas are empty; that is, `i = from - 1` and `j = to + 1`.

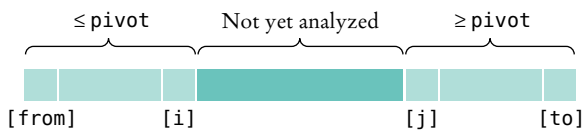


Figure 3 Partitioning a Range

Then keep incrementing `i` while `a[i] < pivot` and keep decrementing `j` while `a[j] > pivot`. Figure 4 shows `i` and `j` when that process stops.

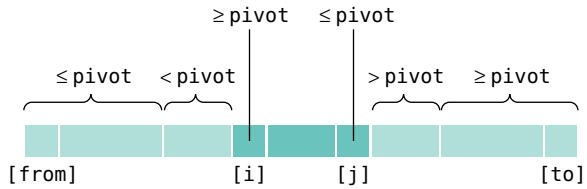


Figure 4 Extending the Partitions

Now swap the values in positions `i` and `j`, increasing both areas once more. Keep going while `i < j`. Here is the code for the `partition` function:

```
int partition(int a[], int from, int to)
{
   int pivot = a[from];
   int i = from - 1;
   int j = to + 1;
   while (i < j)
   {
      i++; while (a[i] < pivot) { i++; }
      j--; while (a[j] > pivot) { j--; }
      if (i < j) { swap(a[i], a[j]); }
   }
   return j;
}
```

On average, the quicksort algorithm is an $O(n \log(n))$ algorithm. Because it is simpler, it runs faster than merge sort in most cases. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* run-time behavior is $O(n^2)$. Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such "tuned" quicksort algorithms are commonly used because their performance is generally excellent. For example, the C library contains a function `qsort` that implements the quicksort algorithm.

EXAMPLE CODE See special_topic_3 of your companion code for the complete quicksort program.

12.6 Searching

Searching for an element in an array is an extremely common task. As with sorting, the right choice of algorithms can make a big difference.

12.6.1 Linear Search

Suppose you need to find the telephone number of your friend. If you have a telephone book, you can look up your friend's name quickly, because the telephone book is sorted alphabetically. However, now suppose you have a telephone number and you must know to whom it belongs (without actually calling the number). You could look through the telephone book, one number at a time, until you find the number. This would obviously be a tremendous amount of work.

This thought experiment shows the difference between a search through an unsorted data set and a search through a sorted data set.

A linear search examines all values in an array until it finds a match or reaches the end.

If you want to find a number in an array of values in arbitrary order, you must look through all elements until you have found a match or until you reach the end. This is called a **linear** or **sequential search**.

How long does a linear search take? If you assume that the value is present in the array `a`, then the average search visits $n/2$ elements. If it is not present, then all n elements must be inspected to verify the absence. Either way, a linear search is an $O(n)$ algorithm.

A linear search locates a value in an array in $O(n)$ steps.

Here is a function that performs a linear search through an array of integers `a` for a given value. The function then returns the index of the match, or –1 if the value does not occur in `a`.

sec06/lsearch.cpp

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

using namespace std;

/**
   Finds an element in an array.
   @param a the array with the elements to search
   @param size the number of elements in a
   @param value the value to search for
   @return the index of the first match, or -1 if not found
*/
int linear_search(int a[], int size, int value)
{
   for (int i = 0; i < size; i++)
   {
      if (a[i] == value)
      {
         return i;
      }
   }
   return -1;
}

int main()
{
   srand(time(0));
   const int SIZE = 20;
   int values[SIZE];
   for (int i = 0; i < SIZE; i++)
   {
      values[i] = rand() % 100;
      cout << values[i] << " ";
   }
   cout << endl;

   cout << "Enter number to search for: ";
   int target;
   cin >> target;
   int pos = linear_search(values, SIZE, target);
   cout << "Found in position " << pos << endl;
```

```
43     return 0;
44  }
```

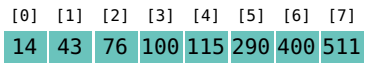
Program Run

```
46 99 45 57 64 95 81 69 11 97 6 85 61 88 29 65 83 88 45 88
Enter number to search for: 12
Found in position -1
```

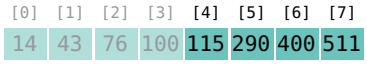
12.6.2 Binary Search

Now consider searching for an item in a sequence that has been previously sorted. Of course, you could still do a linear search, but it turns out you can do much better than that.

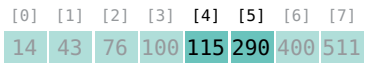
Consider the following sorted array a. The data set is:



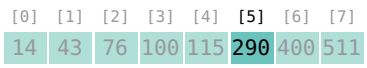
and you want to see whether the value 123 is in the data set. The last point in the first half of the data set, a[3], is 100. It is smaller than the value you are looking for; hence, you should look in the second half of the data set for a match, that is, in the array



Now the last value of the first half of this array is 290; hence, the value must be located in the array



The last value of the first half of this very short array is 115, which is smaller than the value that you are searching, so you must look in the second half:



It is trivial to see that you don't have a match, because 123 ≠ 290. If you wanted to insert 123 into the array, you would need to insert it just before a[5].

A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

This search process is called a **binary search**, because the size of the search is cut in half in each step. That cutting in half works only because you know that the array of values is sorted.

The following function implements a binary search in a sorted array of integers. It returns the position of the match if the search succeeds, or –1 if the value is not found in the array. Here we show a recursive version of the binary search algorithm.

sec06/bsearch.cpp

```
1  #include <cstdlib>
2  #include <ctime>
3  #include <iostream>
4
5  using namespace std;
6
7  /**
```

```
8     Finds an element in a sorted array.
9     @param a the sorted array with the elements to search
10    @param from the start of the range to search
11    @param to the end of the range to search
12    @param value the value to search for
13    @return the index of the first match, or -1 if not found
14 */
15 int binary_search(int a[], int from, int to, int value)
16 {
17    if (from > to)
18    {
19       return -1;
20    }
21
22    int mid = (from + to) / 2;
23    if (a[mid] == value)
24    {
25       return mid;
26    }
27    else if (a[mid] < value)
28    {
29       return binary_search(a, mid + 1, to, value);
30    }
31    else
32    {
33       return binary_search(a, from, mid - 1, value);
34    }
35 }
36
37 int main()
38 {
39    srand(time(0));
40    const int SIZE = 20;
41    int values[SIZE];
42    values[0] = 0;
43    for (int i = 1; i < SIZE; i++)
44    {
45       values[i] = values[i - 1] + rand() % 10;
46       cout << values[i] << " ";
47    }
48    cout << endl;
49
50    cout << "Enter number to search for: ";
51    int target;
52    cin >> target;
53    int pos = binary_search(values, 0, SIZE - 1, target);
54    cout << "Found in position " << pos << endl;
55    return 0;
56 }
```

Now determine the number of element visits required to carry out a search. Use the same technique as in the analysis of merge sort. Because you look at the middle element, which counts as one comparison, and then search either the left or the right array, you have

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Using the same equation,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

By plugging this result into the original equation, you get

$$T(n) = T\left(\frac{n}{4}\right) + 2$$

This generalizes to the kth power of 2:

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

As in the analysis of merge sort, you make the simplifying assumption that n is a power of 2, $n = 2^m$, where $m = \log_2(n)$. Then you obtain

$$T(n) = T(1) + \log_2(n)$$

Therefore, binary search is an $O(\log(n))$ algorithm.

This result makes intuitive sense. Suppose that n is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because $\log_2(100) \approx 6.64386$, and indeed the next larger power of 2 is $2^7 = 128$.

A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you only search the array once, then it is more efficient to pay for an $O(n)$ linear search than for an $O(n \log(n))$ sort and $O(\log(n))$ binary search. But if one makes a number of searches in the same array, then sorting it is definitely worthwhile.

Programming Tip 12.1

Library Functions for Sorting and Binary Search

If you need to sort or search values in your own programs, there is no need to implement your own algorithms. You can simply use functions in the C++ library. This note gives you a brief overview of the library functions for sorting and binary search.

You sort an array by calling the `sort` function with a pointer to the beginning and the end of the array:

```
sort(a, a + size);
```

Here *size* is the size of the array. For example,

```
int a[5] = { 60, 47, 70, 39, 6 };
sort(a, a + 5); // Now a contains 6, 39, 47, 60, 70
```

For a vector, the call looks slightly different:

```
sort(v.begin(), v.end());
```

The expressions `v.begin()` and `v.end()` are **iterators** that denote the beginning and ending positions of the vector. (As you will see in Chapter 14, an iterator denotes a position in a **container**.)

If you have a sorted array or vector, you can use the library's `binary_search` function to test whether it contains a given value. For example, the call

```
binary_search(a, a + size, value)
```

returns `true` if the array `a` contains `value`. (Unlike our binary search function from Section 12.6.2, the library function does not return the position where the value was found.)

To search a vector, you call

```
binary_search(v.begin(), v.end(), value)
```

To use the `sort` or `binary_search` functions, you must include the `<algorithm>` header.

Special Topic 12.4

Defining an Ordering for Sorting Objects

When you use the `sort` library function, you must ensure that it is able to compare elements. Suppose that you want to sort an array of `Employee` objects. The compiler will complain that it does not know how to compare two employees.

There are several ways to overcome this problem. The simplest is to define the `<` operator for `Employee` objects:

```
bool operator<(const Employee& a, const Employee& b)
{
   return a.get_salary() < b.get_salary();
}
```

The curious name `operator<` indicates that this function defines a comparison operator. For more information about defining your own operators, see Chapter 13.

This `<` operator compares employees by salary. If you call `sort` to sort an array of employees, they will be sorted by increasing salary.

EXAMPLE CODE See special_topic_4 of your companion code for an example program using the comparison operator for `Employee` objects.

12.7 Problem Solving: Estimating the Running Time of an Algorithm

In this chapter, you have learned how to estimate the running time of sorting algorithms. As you have seen, being able to differentiate between $O(n \log(n))$ and $O(n^2)$ running times has great practical implications. Being able to estimate the running times of other algorithms is an important skill. In this section, we will practice estimating the running time of vector algorithms.

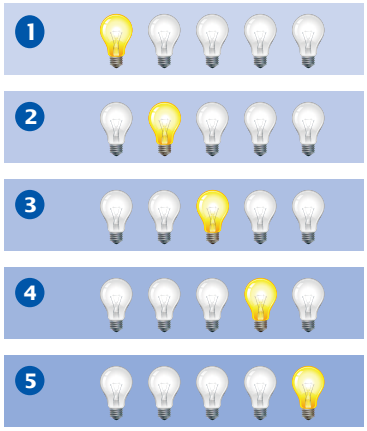
12.7.1 Linear Time

Let us start with a simple example, an algorithm that counts how many elements have a particular value:

```
int count = 0;
for (int i = 0; i < a.size(); i++)
{
   if (a[i] == value) { count++; }
}
```

What is the running time in terms of n, the length of the vector?

Start with looking at the pattern of vector element visits. Here, we visit each element once. It helps to visualize this pattern. Imagine the vector as a sequence of light bulbs. As the *i*th element gets visited, imagine the *i*th bulb lighting up.



(lightbulbs) © Kraska/iStockphoto.

Now look at the work per visit. Does each visit involve a fixed number of actions, independent of n? In this case, it does. There are just a few actions—read the element, compare it, maybe increment a counter.

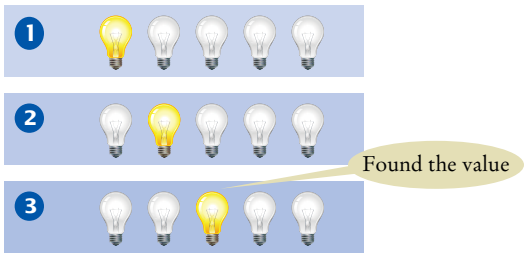
A loop with n iterations has $O(n)$ running time if each step consists of a fixed number of actions.

Therefore, the running time is n times a constant, or $O(n)$.

What if we don't always run to the end of the vector? For example, suppose we want to check whether the value occurs in the vector, without counting it:

```
boolean found = false;
for (int i = 0; !found && i < a.size(); i++)
{
   if (a[i] == value) { found = true; }
}
```

Then the loop can stop in the middle:



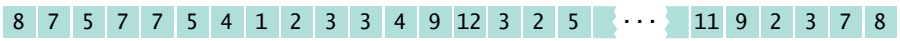
Is this still $O(n)$? It is, because in some cases the match may be at the very end of the vector. Also, if there is no match, one must traverse the entire vector.

12.7.2 Quadratic Time

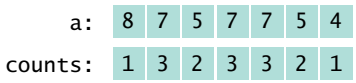
Now let's turn to a more interesting case. What if we do a lot of work with each visit? Here is an example: We want to find the most frequent element in a vector.

Suppose the vector is

It's obvious by looking at the values that 7 is the most frequent one. But now imagine a vector with a few thousand values.



We can count how often the value 8 occurs, then move on to count how often 7 occurs, and so on. For example, in the first vector, 8 occurs once, and 7 occurs three times. Where do we put the counts? Let's put them into a second vector of the same length.



Then we take the maximum of the counts. It is 3. We look up where the 3 occurs in the counts, and find the corresponding value. Thus, the most common value is 7.

Let us first estimate how long it takes to compute the counts.

```
for (int i = 0; i < a.size(); i++)
{
   counts[i] = Count how often a[i] occurs in a
}
```

A loop with n iterations has $O(n^2)$ running time if each step takes $O(n)$ time.

We still visit each vector element once, but now the work per visit is much larger. As you have seen in the previous section, each counting action is $O(n)$. When we do $O(n)$ work in each step, the total running time is $O(n^2)$.

This algorithm has three phases:

1. Compute all counts.
2. Compute the maximum.
3. Find the maximum in the counts.

We have just seen that the first phase is $O(n^2)$. Computing the maximum is $O(n)$—look at the algorithm in Section 6.2.4 and note that each step involves a fixed amount of work. Finally, we just saw that finding a value is $O(n)$.

The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.

How can we estimate the total running time from the estimates of each phase? Of course, the total time is the sum of the individual times, but for big-Oh estimates, we take the *maximum* of the estimates. To see why, imagine that we had actual equations for each of the times:

$$T_1(n) = an^2 + bn + c$$

$$T_2(n) = dn + e$$

$$T_3(n) = fn + g$$

Then the sum is

$$T(n) = T_1(n) + T_2(n) + T_3(n) = an^2 + (b + d + f)n + c + e + g$$

But only the largest term matters, so $T(n)$ is $O(n^2)$.

Thus, we have found that our algorithm for finding the most frequent element is $O(n^2)$.

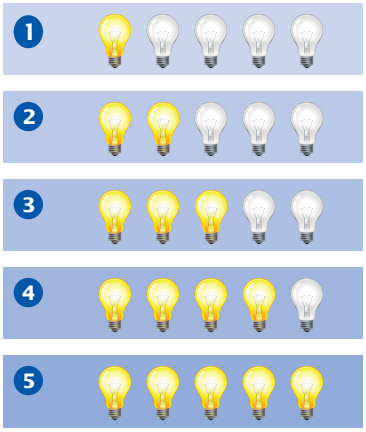
12.7.3 The Triangle Pattern

Let us see if we can speed up the algorithm from the preceding section. It seems wasteful to count elements again if we have already counted them.

Can we save time by eliminating repeated counting of the same element? That is, before counting `a[i]`, should we first check that it didn't occur in `a[0] ... a[i - 1]`?

Let us estimate the cost of these additional checks. In the *i*th step, the amount of work is proportional to *i*. That's not quite the same as in the preceding section, where you saw that a loop with *n* iterations, each of which takes $O(n)$ time, is $O(n^2)$. Now each step just takes $O(i)$ time.

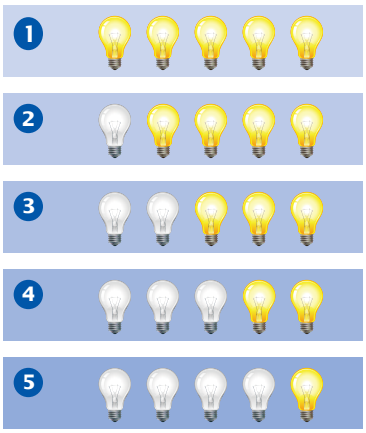
To get an intuitive feel for this situation, look at the light bulbs again. In the second iteration, we visit `a[0]` again. In the third iteration, we visit `a[0]` and `a[1]` again, and so on. The light bulb pattern is



A loop with *n* iterations has $O(n^2)$ running time if the *i*th step takes $O(i)$ time.

If there are *n* light bulbs, about half of the square above, or $n^2/2$ of them, light up. That's unfortunately still $O(n^2)$.

Here is another idea for time saving. When we count `a[i]`, there is no need to do the counting in `a[0] ... a[i - 1]`. If `a[i]` never occurred before, we get an accurate count by just looking at `a[i] ... a[n - 1]`. And if it did, we already have an accurate count. Does that help us? Not really—it's the triangle pattern again, but this time in the other direction.



That doesn't mean that these improvements aren't worthwhile. If an $O(n^2)$ algorithm is the best one can do for a particular problem, you still want to make it as fast as possible. However, we will not pursue this plan further because it turns out that we can do much better.

12.7.4 Logarithmic Time

An algorithm that cuts the size of work in half in each step runs in $O(\log(n))$ time.

Logarithmic time estimates arise from algorithms that cut work in half in each step. You have seen this in the algorithms for binary search and merge sort.

In particular, when you use sorting or binary search in a phase of an algorithm, you will encounter logarithmic time in the big-Oh estimates.

Consider this idea for improving our algorithm for finding the most frequent element. Suppose we first *sort* the vector:

That cost us $O(n \log(n))$ time. If we can complete the algorithm in $O(n)$ time, we will have found a better algorithm than the $O(n^2)$ algorithm of the preceding sections.

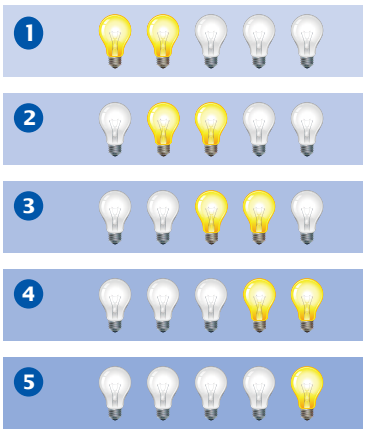
To see why this is possible, imagine traversing the sorted vector. As long as you find a value that was equal to its predecessor, you increment a counter. When you find a different value, save the counter and start counting anew:

Or in code,

```
int count = 0;
for (int i = 0; i < a.size(); i++)
{
   count++;
   if (i == a.size() - 1 || a[i] != a[i + 1])
   {
      counts[i] = count;
      count = 0;
   }
}
```

That's a constant amount of work per iteration, even though it visits two elements. $2n$ is still $O(n)$. Thus, we can compute the counts in $O(n)$ time from a sorted vector. The entire algorithm is now $O(n \log(n))$.

Note that we don't actually need to keep all counts, only the highest one that we encountered so far (see Exercise E12.8). That is a worthwhile improvement, but it does not change the big-Oh estimate of the running time.



EXAMPLE CODE See sec07 of your companion code for a program that compares the speed of algorithms for finding the most frequent element.

WORKED EXAMPLE 12.1

Enhancing the Insertion Sort Algorithm

Learn how to implement an improvement of the insertion sort algorithm shown in Special Topic 12.2. The enhanced algorithm is called *Shell sort* after its inventor, Donald Shell. See your E-Text or visit `wiley.com/go/bclo3`.

Computing & Society 12.1 The First Programmer

Before pocket calculators and personal computers existed, navigators and engineers used mechanical adding machines, slide rules, and tables of logarithms and trigonometric functions to speed up computations. Unfortunately, the tables—for which values had to be computed by hand—were notoriously inaccurate. The mathematician Charles Babbage (1791–1871) had the insight that if a machine could be constructed that produced printed tables automatically, both calculation and typesetting errors could be avoided. Babbage set out to develop a machine for this purpose, which he called a *Difference Engine* because it used successive differences to compute polynomials. For example, consider the function $f(x) = x^3$. Write down the values for $f(1)$, $f(2)$, $f(3)$, and so on. Then take the *differences* between successive values:

```
1
    7
8
    19
27
    37
64
    61
125
    91
216
```

Repeat the process, taking the difference of successive values in the second column, and then repeat once again:

```
1
    7
8       12
    19      6
27      18
    37      6
64      24
    61      6
125     30
    91
216
```

Now the differences are all the same. You can retrieve the function values by a pattern of additions—you need to know the values at the fringe of the pattern and the constant difference. You can try it out yourself: Write the highlighted numbers on a sheet of paper, and fill in the others by adding the numbers that are in the north and northwest positions.

This function was very attractive, because mechanical addition machines had been known for some time. They consisted of cog wheels, with 10 cogs per wheel, to represent digits, and mechanisms to handle the carry from one digit to the next. Mechanical multiplication machines, on the other hand, were fragile and unreliable. Babbage built a successful prototype of the Difference Engine and, with his own money and government grants, proceeded to build the table-printing machine. However, because of funding problems and the difficulty of building the machine to the required precision, it was never completed.

While working on the Difference Engine, Babbage conceived of a much grander vision that he called the *Analytical Engine*. The Difference Engine was designed to carry out a limited set of computations—it was no smarter than a pocket calculator is today. But Babbage realized that such a machine could be made *programmable* by storing programs as well as data. The internal storage of the Analytical Engine was to consist of 1,000 registers of 50 decimal digits each. Programs and constants were to be stored on punched cards—a technique that was, at that time, commonly used on looms for weaving patterned fabrics.

Ada Augusta, Countess of Lovelace (1815–1852), the only child of Lord Byron, was a friend and sponsor of Charles Babbage. Ada Lovelace was one of the first people to realize the potential of such a machine, not just for computing mathematical tables but for processing data that were not numbers. She is considered by many to be the world's first programmer.

Topham/The Image Works.

Replica of Babbage's Difference Engine

CHAPTER SUMMARY

Describe the selection sort algorithm.

- The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

Measure the running time of a function.

- To measure the running time of a function, get the current time immediately before and after the function call.

Use the big-Oh notation to describe the running time of an algorithm.

- Computer scientists use big-Oh notation to describe how fast a function grows.
- Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.
- Insertion sort is an $O(n^2)$ algorithm.

Describe the merge sort algorithm.

- The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, then merging the sorted halves.

Contrast the running times of the merge sort and selection sort algorithms.

- Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2.

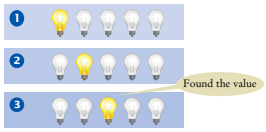
Describe the linear search and binary search algorithms and their running times.

- A linear search examines all values in an array until it finds a match or reaches the end.
- A linear search locates a value in an array in $O(n)$ steps.
- A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.
- A binary search locates a value in an array in $O(\log(n))$ steps.

Practice developing big-Oh estimates of algorithms.

- A loop with n iterations has $O(n)$ running time if each step consists of a fixed number of actions.
- A loop with n iterations has $O(n^2)$ running time if each step takes $O(n)$ time.

- The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.
- A loop with n iterations has $O(n^2)$ running time if the ith step takes $O(i)$ time.
- An algorithm that cuts the size of work in half in each step runs in $O(\log(n))$ time.



REVIEW EXERCISES

■■ **R12.1** *Checking against off-by-one errors.* When writing the selection sort algorithm of Section 12.1, a programmer must make the usual choices of `<` against `<=`, `size` against `size - 1`, and `next` against `next + 1`. This is fertile ground for off-by-one errors. Make code walkthroughs of the algorithm with arrays of length 0, 1, 2, and 3 and check carefully that all index values are correct.

■ **R12.2** What is the difference between searching and sorting?

■■ **R12.3** For the following expressions, what is the order of the growth of each?

a. $n^2 + 2n + 1$

b. $n^{10} + 9n^9 + 20n^8 + 145n^7$

c. $(n + 1)^4$

d. $(n^2 + n)^2$

e. $n + 0.001n^3$

f. n^3 $1000n^2 + 10^9$

g. $n + \log(n)$

h. $n^2 + n \log(n)$

i. $2^n + n^2$

j. $\dfrac{n^3 + 2n}{n^2 + 0.75}$

■ **R12.4** We determined that the actual number of visits in the selection sort algorithm is

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We then characterized this function as having $O(n^2)$ growth. Compute the actual ratios

$$T(2{,}000)/T(1{,}000)$$

$$T(5{,}000)/T(1{,}000)$$

$$T(10{,}000)/T(1{,}000)$$

and compare them with

$$f(2{,}000)/f(1{,}000)$$

$$f(5{,}000)/f(1{,}000)$$

$$f(10{,}000)/f(1{,}000)$$

where $f(n) = n^2$.

■ **R12.5** Suppose algorithm A takes five seconds to handle a data set of 1,000 records. If the algorithm A is an $O(n)$ algorithm, how long will it take to handle a data set of 2,000 records? Of 10,000 records?

■■ **R12.6** Suppose an algorithm takes five seconds to handle a data set of 1,000 records. Fill in the following table, which shows the approximate growth of the execution times depending on the complexity of the algorithm.

For example, because $3000^2/1000^2 = 9$, the $O(n^2)$ algorithm would take 9 times as long, or 45 seconds, to handle a data set of 3,000 records.

	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n \log(n))$	$O(2^n)$
1,000	5	5	5	5	5
2,000					
3,000		45			
10,000					

■■ **R12.7** Sort the following growth rates from slowest growth to fastest growth.

$O(n)$	$O(\log(n))$	$O(2^n)$	$O(n\sqrt{n})$
$O(n^3)$	$O(n^2 \log(n))$	$O(\sqrt{n})$	$O(n^{\log(n)})$
$O(n^n)$	$O(n \log(n))$		

■ **R12.8** What is the growth rate of the standard algorithm to find the minimum value of an array? Of finding both the minimum and the maximum?

■ **R12.9** What is the big-Oh time estimate of the following function in terms of n, the length of `a`? Use the "light bulb pattern" method of Section 12.7 to visualize your result.

```
void swap(vector<int> a)
{
   int i = 0;
   int j = a.size() - 1;
   while (i < j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
      i++;
      j--;
   }
}
```

■ **R12.10** A *run* is a sequence of adjacent repeated values (see Exercise R6.25). Describe an $O(n)$ algorithm to find the length of the longest run in an array.

■■ **R12.11** Consider the task of finding the most frequent element in an array of length `n`. Here are three approaches:

a. Sort the array, then find the longest run.

b. Allocate an array of counters of the same size as the original array. For each element, traverse the array and count how many other elements are equal to it, updating its counter. Then find the maximum count.

c. Keep variables for the most frequent element that you have seen so far and its frequency. For each index `i`, check whether `a[i]` occurs in `a[0] ... a[i - 1]`. If not, count how often it occurs in `a[i + 1] ... a[n - 1]`. If `a[i]` is more frequent than the most frequent element so far, update the variables.

Describe the big-Oh efficiency of each approach.

■■ **R12.12** Your task is to remove all duplicates from an array. For example, if the array has the values

```
4 7 11 4 9 5 11 7 3 5
```

then the array should be changed to

```
4 7 11 9 5 3
```

Here is a simple algorithm. Look at `a[i]`. Count how many times it occurs in `a`. If the count is larger than 1, remove it. What is the order of complexity of this algorithm?

■■■ **R12.13** Modify the merge sort algorithm to remove duplicates in the merging step to obtain an algorithm that removes duplicates from an array. Note that the resulting array does not have the same ordering as the original one. What is the efficiency of this algorithm?

■■ **R12.14** Consider the following algorithm to remove all duplicates from an array: Sort the array. For each element in the array, look at its next neighbor to decide whether it is present more than once. If so, remove it. Is this a faster algorithm than the one in Exercise R12.12?

■■■ **R12.15** Develop an $O(n \log(n))$ algorithm for removing duplicates from an array if the resulting array must have the same ordering as the original array. When a value occurs multiple times, all but its first occurrence should be removed.

■ **R12.16** Make a walkthrough of selection sort with the following data sets:

a. `4 7 11 4 9 5 11 7 3 5`

b. `–7 6 8 7 5 9 0 11 10 5 8`

■ **R12.17** Make a walkthrough of merge sort with the following data sets:

a. `5 11 7 3 5 4 7 11 4 9`

b. `9 0 11 10 5 8 –7 6 8 7 5`

■ **R12.18** Make a walkthrough of the following:

a. Linear search for 7 in `–7 1 3 3 4 7 11 13`

b. Binary search for 8 in `–7 2 2 3 4 7 8 11 13`

c. Binary search for 8 in `–7 1 2 3 5 7 10 13`

■■■ **R12.19** Why does insertion sort perform significantly better than selection sort if an array is already sorted?

■■■ **R12.20** Consider the following speedup of the insertion sort algorithm of Special Topic 12.2. For each element, use the enhanced binary search algorithm described in Exercise E12.9 that yields the insertion position for missing elements. Does this speedup have a significant impact on the efficiency of the algorithm?

■■ **R12.21** Consider the following algorithm known as *bubble sort*:

```
While the array is not sorted
   For each adjacent pair of elements
      If the pair is not sorted
         Swap its elements.
```

What is the big-Oh efficiency of this algorithm?

■■ **R12.22** The *radix sort* algorithm sorts an array of n integers with d digits, using ten auxiliary arrays. First place each value v into the auxiliary array whose index corresponds to the last digit of v. Then move all values back into the original array, preserving their order. Repeat the process, now using the next-to-last (tens) digit, then the hundreds digit, and so on. What is the big-Oh time of this algorithm in terms of n and d? When is this algorithm preferable to merge sort?

■■ **R12.23** A *stable sort* does not change the order of elements with the same value. This is a desirable feature in many applications. Consider a sequence of e-mail messages. If you sort by date and then by sender, you'd like the second sort to preserve the relative order of the first, so that you can see all messages from the same sender in date order. Is selection sort stable? Insertion sort? Why or why not?

■■ **R12.24** Give an $O(n)$ algorithm to sort an array of n bytes (numbers between –128 and 127). *Hint:* Use an array of counters.

■■ **R12.25** You are given a sequence of arrays of words, representing the pages of a book. Your task is to build an index (a sorted array of words), each element of which has an array of sorted numbers representing the pages on which the word appears. Describe an algorithm for building the index and give its big-Oh running time in terms of the total number of words.

■■ **R12.26** Given two arrays of n integers each, describe an $O(n \log(n))$ algorithm for determining whether they have an element in common.

■■■ **R12.27** Given an array of n integers and a value v, describe an $O(n \log(n))$ algorithm to find whether there are two values x and y in the array with sum v.

■■ **R12.28** Given two arrays of n integers each, describe an $O(n \log(n))$ algorithm for finding all elements that they have in common.

■■ **R12.29** Suppose we modify the quicksort algorithm from Special Topic 12.3, selecting the middle element instead of the first one as pivot. What is the running time on an array that is already sorted?

■■ **R12.30** Suppose we modify the quicksort algorithm from Special Topic 12.3, selecting the middle element instead of the first one as pivot. Find a sequence of values for which this algorithm has an $O(n^2)$ running time.

PRACTICE EXERCISES

■ **E12.1** Modify the selection sort algorithm to sort an array of strings by increasing length.

■ **E12.2** Modify the selection sort algorithm to sort a vector of integers.

■■ **E12.3** Write a program that automatically generates the table of sample runs for the selection sort algorithm. The program should ask for the smallest and largest value of n and the number of measurements, then make all sample runs and display the table.

■ **E12.4** Modify the merge sort algorithm to sort a vector of employees by salary.

■■■ **E12.5** Write a telephone lookup program. Read a data set of 1,000 names and telephone numbers from a file that contains the numbers in random order. Handle lookups by name and also reverse lookups by phone number. Use a binary search for both lookups.

■■ **E12.6** Implement a program that measures the performance of the insertion sort algorithm described in Special Topic 12.2.

■ **E12.7** Implement the bubble sort algorithm described in Exercise R12.21.

■■ **E12.8** Implement the algorithm described in Section 12.7.4, but remember only the value with the highest frequency so far:

```
int most_frequent = 0;
int highest_frequency = -1;
for (int i = 0; i < a.size(); i++)
   Count how often a[i] occurs in a[i + 1] . . . a[a.size() - 1]
   If it occurs more often than highest_frequency
      highest_frequency = that count
      most_frequent = a[i]
```

■■ **E12.9** Consider the binary search function in Section 12.6.2. If no match is found, the function returns –1. Modify the function so that it returns a `bool` value indicating whether a match was found. Add a reference parameter `pos`, which is set to the location of the match if the search was successful. If a match was not found, set `pos` to the index of the next larger value instead, or to the array size if the target value is larger than all the elements of the array.

■■■ **E12.10** Use the modification of the binary search function from Exercise E12.9 to sort an array. Make a second array of the same size as the array to be sorted. For each element in the first array, call binary search on the second array to find out where the new element should be inserted. Then move all elements above the insertion point up by one slot and insert the new element. Thus, the second array is always kept sorted. Implement this algorithm and measure its performance.

■■ **E12.11** Implement the `binary_search` function of Section 12.6.2 without recursion. *Hint:* While `from < to`, update either `from` or `to`, depending on which range should be searched.

■■ **E12.12** Implement the `merge_sort` function without recursion, where the size of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.

■■ **E12.13** Write a program that sorts a vector of `Employee` objects by the employee names and prints the results. Use the `sort` function from the C++ library.

■■■ **E12.14** Write a program that keeps an appointment book. Make a class `Appointment` that stores a description of the appointment, the appointment day, the starting time, and the ending time. Your program should keep the appointments in a sorted vector. Users can add appointments and print out all appointments for a given day. When a new appointment is added, use binary search to find where it should be inserted in the vector. Do not add it if it conflicts with another appointment.

PROGRAMMING PROJECTS

■■ **P12.1** The median m of a sequence of n elements is the element that would fall in the middle if the sequence was sorted. That is, $e \leq m$ for half the elements, and $m \leq e$ for the others. Clearly, one can obtain the median by sorting the sequence, but one can do quite a bit better with the following algorithm that finds the kth element of a

sequence between a (inclusive) and b (exclusive). (For the median, use $k = n/2$, $a = 0$, and $b = n$.)

```
select(k, a, b)
Pick a pivot p in the subsequence between a and b.
Partition the subsequence elements into three subsequences: the elements <p, =p, >p
Let n1, n2, n3 be the sizes of each of these subsequences.
if k < n1
   return select(k, 0, n1).
else if (k > n1 + n2)
   return select(k, n1 + n2, n).
else
   return p.
```

Implement this algorithm and measure how much faster it is for computing the median of a random large sequence, when compared to sorting the sequence and taking the middle element.

■■ **P12.2** Implement the following modification of the quicksort algorithm, due to Bentley and McIlroy. Instead of using the first element as the pivot, use an approximation of the median.

If $n \leq 7$, use the middle element. If $n \leq 40$, use the median of the first, middle, and last element. Otherwise compute the "pseudomedian" of the nine elements `a[i * (n - 1) / 8]`, where `i` ranges from 0 to 8. The pseudomedian of nine values is med(med(v_0, v_1, v_2), med(v_3, v_4, v_5), med(v_6, v_7, v_8)).

Compare the running time of this modification with that of the original algorithm on sequences that are nearly sorted or reverse sorted, and on sequences with many identical elements. What do you observe?

■■■ **P12.3** Bentley and McIlroy suggest the following modification to the quicksort algorithm when dealing with data sets that contain many repeated elements.

Instead of partitioning as

(where ≤ denotes the elements that are ≤ the pivot), it is better to partition as

However, that is tedious to achieve directly. They recommend to partition as

and then swap the two = regions into the middle. Implement this modification and check whether it improves performance on data sets with many repeated elements.

■ **P12.4** Implement the radix sort algorithm described in Exercise R12.22 to sort arrays of numbers between 0 and 999.

■ **P12.5** Implement the radix sort algorithm described in Exercise R12.22 to sort arrays of numbers between 0 and 999. However, use a single auxiliary array, not ten.

■■ **P12.6** Implement the radix sort algorithm described in Exercise R12.22 to sort arbitrary `int` values (positive or negative).

■■■ **P12.7** Implement the `merge_sort` function without recursion, where the size of the array is an arbitrary number. *Hint:* Keep merging adjacent areas whose size is a power of 2, and pay special attention to the last area whose size is less.

■■ **P12.8** It is common for people to name directories as `dir1`, `dir2`, and so on. When there are ten or more directories, the operating system displays them in dictionary order, as `dir1`, `dir10`, `dir11`, `dir12`, `dir2`, `dir3`, and so on. That is irritating, and it is easy to fix. Provide a comparison operator that compares strings that end in digit sequences in a way that makes sense to a human. First compare the part before the digits as strings, and then compare the numeric values of the digits.

■■■ **P12.9** Sometimes, directory or file names have numbers in the middle, and there may be more than one number, for example, `sec3_14.txt` or `sec10_1.txt`. Provide a comparison operator that can compare such strings in a way that makes sense to humans. Break each string into strings not containing digits and digit groups. Then compare two strings by comparing the first non-digit groups as strings, the first digit groups as integers, and so on.

WORKED EXAMPLE 12.1

Enhancing the Insertion Sort Algorithm

Problem Statement Implement an improvement of the insertion sort algorithm (shown in Special Topic 12.2) called *Shell sort* after its inventor, Donald Shell.

Shell sort is an enhancement of insertion sort that takes advantage of the fact that insertion sort is an $O(n)$ algorithm if the array is already sorted. Shell sort brings parts of the array into sorted order, then runs an insertion sort over the entire array, so that the final sort doesn't do much work.

A key step in Shell sort is to arrange the sequence into rows and columns, and then to sort each column separately. For example, if the array is

65	46	14	52	38	2	96	39	14	33	13	4	24	99	89	77	73	87	36	81

and we arrange it into four columns, we get

65	46	14	52
38	2	96	39
14	33	13	4
24	99	89	77
73	87	36	81

Now we sort each column:

14	2	13	5
24	33	14	39
38	46	36	52
65	87	89	77
73	99	96	81

Put together as a single array, we get

14	2	13	5	24	33	14	39	38	46	36	52	65	87	89	77	73	99	96	81

Note that the array isn't completely sorted, but many of the small numbers are now in front, and many of the large numbers are in the back.

We will repeat the process until the array is sorted. Each time, we use a different number of columns. Shell had originally used powers of two for the column counts. For example, on an array with 20 elements, he proposed using 16, 8, 4, 2, and finally one column. With one column, we have a plain insertion sort, so we know the array will be sorted. What is surprising is that the preceding sorts greatly speed up the process.

However, better sequences have been discovered. We will use the sequence of column counts

$$
\begin{aligned}
c_1 &= 1 \\
c_2 &= 4 \\
c_3 &= 13 \\
c_4 &= 40 \\
&\cdots \\
c_{i+1} &= 3c_i + 1
\end{aligned}
$$

That is, for an array with 20 elements, we first do a 13-sort, then a 4-sort, and then a 1-sort. This sequence is almost as good as the best known ones, and it is easy to compute.

We will not actually rearrange the array, but compute the locations of the elements of each column.

For example, if the number of columns c is 4, the four columns are located in the array as follows:

65				38				14				24				73			
	46				2				33				99				87		
		14				96				13				89				36	
			52				39				4				77				81

Note that successive column elements have distance c from another. The *k*th column is made up of the elements `a[k]`, `a[k + c]`, `a[k + 2 * c]`, and so on.

Now let's adapt the insertion sort algorithm to sort such a column. The original algorithm was

```
for (int i = 1; i < size; i++)
{
   int next = a[i];
   // Move all larger elements up
   int j = i;
   while (j > 0 && a[j - 1] > next)
   {
      a[j] = a[j - 1];
      j--;
   }
   // Insert the element
   a[j] = next;
}
```

The outer loop visits the elements `a[1]`, `a[2]`, and so on. In the *k*th column, the corresponding sequence is `a[k + c]`, `a[k + 2 * c]`, and so on. That is, the outer loop becomes

```
for (int i = k + c; i < size; i = i + c)
```

In the inner loop, we originally visited `a[j]`, `a[j - 1]`, and so on. We need to change that to `a[j]`, `a[j - c]`, and so on. The inner loop becomes

```
while (j >= c && a[j - c] > next)
{
   a[j] = a[j - c];
   j = j - c;
}
```

Putting everything together, we get the following function:

```
/**
   Sorts a column, using insertion sort.
   @param a the array to sort
   @param size the size of a
   @param k the index of the first element in the column
   @param c the gap between elements in the column
*/
void insertion_sort(int[] a, int size, int k, int c)
{
   for (int i = k + c; i < size; i = i + c)
   {
      int next = a[i];
      // Move all larger elements up
      int j = i;
      while (j >= c && a[j - c] > next)
      {
```

```
            a[j] = a[j - c];
            j = j - c;
         }
         // Insert the element
         a[j] = next;
      }
   }
```

Now we are ready to implement the Shell sort algorithm. First, we need to find out how many elements we need from the sequence of column counts. We generate the sequence values until they exceed the size of the array to be sorted.

```
vector<int> columns;
int c = 1;
while (c < size)
{
   columns.push_back(c);
   c = 3 * c + 1;
}
```

For each column count, we sort all columns:

```
for (int s = columns.size() - 1; s >= 0; s--)
{
   c = columns[s];
   for (int k = 0; k < c; k++)
   {
      insertion_sort(a, size, k, c);
   }
}
```

How good is the performance? Let's compare it with the library `sort` function (see Programming Tip 12.1) and insertion sort. The results show that Shell sort is a dramatic improvement over insertion sort:

```
Enter array size: 300000
Elapsed time with Shell sort: 0.093 seconds
Elapsed time with the library sort: 0.089 seconds
Elapsed time with insertion sort: 127 seconds
```

However, the library sort (which uses an $O(n \log(n))$ sorting algorithm) outperforms Shell sort. For this reason, Shell sort is not used in practice, but it is still an interesting algorithm that is surprisingly effective.

You may also find it interesting to experiment with Shell's original column sizes. In the sort function, simply replace

```
c = 3 * c + 1;
```

with

```
c = 2 * c;
```

You will find that the algorithm is about three times slower than the improved sequence. That is still much faster than plain insertion sort.

You will find the following program to demonstrate Shell sort and compare it to insertion sort and library sort in the `worked_example_1` folder of the book's companion code.

worked_example_1/sortdemo.cpp

```
1  #include <cstdlib>
2  #include <ctime>
3  #include <iostream>
4  #include <algorithm>
5
6  #include "shellsort.h"
```

```
 7
 8  using namespace std;
 9
10  /**
11     Fills an array with random integers between 0 and 99.
12     @param a the array to print
13     @param size the number of elements in a
14  */
15  void random_fill(int a[], int size)
16  {
17     for (int i = 0; i < size; i++)
18     {
19        a[i] = rand() % 100;
20     }
21  }
22
23  /**
24     Prints all elements in an array.
25     @param a the array to print
26     @param size the number of elements in a
27  */
28  void print(int a[], int size)
29  {
30     for (int i = 0; i < size; i++)
31     {
32        cout << a[i] << " ";
33     }
34     cout << endl;
35  }
36
37  const int MAX_SIZE = 50000000;
38  int values[MAX_SIZE];
39
40  int main()
41  {
42     srand(time(0));
43     int size = 20;
44
45     cout << "Demo with " << size << " elements:" << endl;
46     random_fill(values, size);
47     print(values, size);
48     shell_sort(values, size);
49     print(values, size);
50     cout << "Enter array size: ";
51     cin >> size;
52     const int ITERATIONS = 1000;
53
54     int before = time(0);
55     srand(before);
56     for (int i = 1; i <= ITERATIONS; i++)
57     {
58        random_fill(values, size);
59        shell_sort(values, size);
60     }
61     int after = time(0);
62     cout << "Elapsed time with Shell sort: "
63        << (after - before) * 1.0 / ITERATIONS << " seconds" << endl;
64
```

```
   srand(before); // Make sure we get the same arrays again
   before = time(0);
   for (int i = 1; i <= ITERATIONS; i++)
   {
      random_fill(values, size);
      sort(values, values + size);
   }
   after = time(0);
   cout << "Elapsed time with the library sort: "
      << (after - before) * 1.0 / ITERATIONS << " seconds" << endl;

   random_fill(values, size);
   before = time(0);
   insertion_sort(values, size, 0, 1);
   after = time(0);
   cout << "Elapsed time with insertion sort: " << after - before
      << " seconds" << endl;

   return 0;
}
```

worked_example_1/shellsort.cpp

```
#include <vector>

using namespace std;

/**
   Sorts a column, using insertion sort.
   @param a the array to sort
   @param size the size of a
   @param k the index of the first element in the column
   @param c the gap between elements in the column
*/
void insertion_sort(int a[], int size, int k, int c)
{
   for (int i = k + c; i < size; i = i + c)
   {
      int next = a[i];
      // Move all larger elements up
      int j = i;
      while (j >= c && a[j - c] > next)
      {
         a[j] = a[j - c];
         j = j - c;
      }
      // Insert the element
      a[j] = next;
   }
}

/**
   Sorts an array, using Shell sort.
   @param a the array to sort
   @param size the size of a
*/
void shell_sort(int a[], int size)
{
   // Generate the sequence values
   vector<int> columns;
```

```
38       int c = 1;
39       while (c < size)
40       {
41          columns.push_back(c);
42          c = 3 * c + 1;
43       }
44
45       // For each column count, sort all columns
46       for (int s = columns.size() - 1; s >= 0; s--)
47       {
48          c = columns[s];
49          for (int k = 0; k < c; k++)
50          {
51             insertion_sort(a, size, k, c);
52          }
53       }
54    }
```

worked_example_1/shellsort.h

```
1  void insertion_sort(int a[], int size, int k, int c);
2  void shell_sort(int a[], int size);
```

CHAPTER 13

ADVANCED C++

© Sky_Blue/iStock/Getty Images.

CHAPTER GOALS

To be able to implement overloaded operators in your own classes

To learn how to implement classes that manage their memory allocation and deallocation

To be able to use and define template classes and functions

CHAPTER CONTENTS

This chapter introduces several advanced C++ topics that are particularly useful for programmers who provide libraries that are used by other programmers. You will learn how to define the meaning of operators (such as + or <) that are applied to objects, how to write classes that automatically manage free store memory, and how to build class templates that can be instantiated with type parameters.

Using these techniques, you will understand how standard C++ classes and templates such as `string` and `vector<T>` actually work. In the following chapters, you will be able to implement your own data structures in a professional way.

13.1 Operator Overloading

When you concatenate or print strings, you use the + and << operators. These operators were originally intended for working with numbers, but the standard library has given them additional meaning when used with strings. Giving a new meaning to an operator is called *operator overloading*.

© shotsstudio/Getty Images.

Like a painting that can have a different meaning for each viewer, a C++ operator can have different meanings.

13.1.1 Operator Functions

You can define a new meaning for a C++ operator by defining a function whose name is `operator` followed by the operator symbol.

We use operators because they make expressions easier to read than functions. For example, the expression

```
a * x + b
```

could be written without operators as

```
sum(product(a, x), b)
```

However, most programmers would prefer the form with operators.

Naturally, that ease of reading greatly depends on the conventional meaning of the operators. In C++, you can define the behavior of operators in any way you choose. If you choose unwisely, an operator may well be more confusing to programmers than a well-chosen function name.

It is best to overload operators in contexts where they closely follow traditional usage. A good example is using + for concatenating strings. The concatenation of strings is related to the sum of numbers. Using - or == for concatenation would have been possible but confusing.

For that reason, we will not add overloaded operators to the `CashRegister` or `Question` classes that we used as examples for classes in Chapters 9 and 10. Instead, we will start out with this simple `Time` class that represents a point in time during the day, such as 9:30. For simplicity, we use "military" or 24 hour time, with hours between 0 and 23.

The `Time` *class represents a point in time with hours and minutes.*

sec01/time.h

```
1  #ifndef TIME_H
2  #define TIME_H
3
4  class Time
5  {
6  public:
7     Time();
8     Time(int h, int m);
9     int get_hours() const;
10    int get_minutes() const;
11 private:
12    int hours;
13    int minutes;
14 };
15
16 #endif
```

sec01/time.cpp

```
1  #include "time.h"
2
3  Time::Time()
4  {
5     hours = 0;
6     minutes = 0;
7  }
8
9  Time::Time(int h, int m)
10 {
```

```
11    hours = h;
12    minutes = m;
13 }
14
15 int Time::get_hours() const
16 {
17    return hours;
18 }
19
20 int Time::get_minutes() const
21 {
22    return minutes;
23 }
```

Given two `Time` objects, we may want to compute the difference between them:

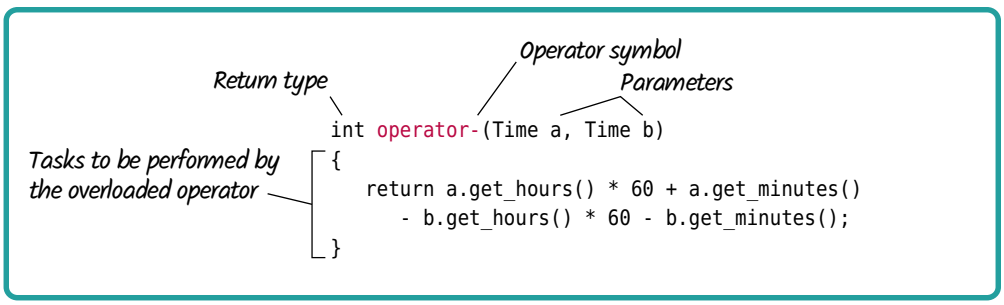
```
Time morning(9, 30);
Time lunch(12, 0);
int minutes_to_lunch = lunch - morning;
```

To achieve that, we define a function whose name is `operator` followed by the operator symbol, as shown in Syntax 13.1.

```
int operator-(Time a, Time b)
{
   return a.get_hours() * 60 + a.get_minutes()
      - b.get_hours() * 60 - b.get_minutes();
}
```

Note that the `operator-` function is not a member function. It is a nonmember function with two parameters. Alternatively, you can implement the operator function as a member function with one explicit parameter; see Section 13.1.4.

Syntax 13.1 Overloaded Operator Definition



The precedence and associativity of an overloaded operator is the same as that of the original operator.

In this way, you can redefine any of the existing C++ operators. However, you cannot define new operators that don't exist in C++, and you cannot change the precedence or associativity of the operators. That is, `x + y * z` means `x + (y * z)` or `operator+(x, operator*(y, z))` no matter how you define these operators.

Can you define the `+` operator to add two times? Of course you can, simply by defining an `operator+(Time a, Time b)`. But that doesn't mean you should. A `Time` object represents a point in time, not a duration. For example, 6:00 means "a particular time early in the morning", which is quite different from "6 hours". It does not make any sense to add two points in time, such as 6:00 and 9:30.

However, it does make sense to add a number of minutes to a `Time` object, resulting in a new `Time` object. Here is an overloaded `+` operator for that task.

```
Time operator+(Time a, int minutes)
{
   int result_minutes = a.get_hours() * 60 + a.get_minutes() + minutes;
   return Time((result_minutes / 60) % 24, result_minutes % 60);
}
```

For example,

```
Time too_early(6, 0);
Time wake_up = too_early + 45;
```

13.1.2 Overloading Comparison Operators

A commonly overloaded operator is the `==` operator, used to compare two values. Two `Time` values are equal if the number of minutes between them is zero. Therefore, you can define

```
bool operator==(Time a, Time b)
{
   return a - b == 0;
}
```

For completeness, it is a good idea to also define a `!=` operator:

```
bool operator!=(Time a, Time b)
{
   return a - b != 0;
}
```

If you want to sort an array of `Time` objects with the standard library `sort` function, as described in Programming Tip 12.1 and Special Topic 12.4, you need to define the `<` operator:

```
bool operator<(Time a, Time b)
{
   return a - b < 0;
}
```

13.1.3 Input and Output

You may want to print a `Time` object with the familiar `<<` notation. The first parameter of the operator`<<` function is the output stream, which is passed by reference because the output operations modify the stream. The second parameter is the `Time` object that is sent to the stream.

```
ostream& operator<<(ostream& out, Time a)
{
   out << a.get_hours() << ":"
      << setw(2) << setfill('0')
      << a.get_minutes();
   return out;
}
```

The `<<` operator returns the `out` stream. This is what enables chaining of the `<<` operator. For example,

```
cout << noon << "\n";
```

really means

```
(cout << noon) << "\n";
```

that is

```
operator<<(cout, noon) << "\n"
```

The call to `operator<<(cout, noon)` prints the time `noon` and then returns `cout`. Then `cout << "\n"` prints a newline. Special Topic 13.3 explains why the function return type is a reference `ostream&`.

You can also define an `operator>>` to read a `Time` object from an input stream.

```
istream& operator>>(istream& in, Time& a)
{
   int hours;
   char separator;
   int minutes;
   in >> hours;
   in.get(separator); // Read : character
   in >> minutes;
   a = Time(hours, minutes);
   return in;
}
```

Note that the `>>` operator returns the input stream, similar to the `<<` operator. However, unlike the `<<` operator, the `>>` operator must have a parameter of type `Time&`. The parameter is modified when it is filled with the input.

Basic operator overloading is relatively easy and many people find it fun. However, as already mentioned, it is best not to go overboard. Using inappropriate operators can make programs more difficult to read.

EXAMPLE CODE See `sec01` of your companion code for a sample program that defines several overloaded operators for the `Time` class.

13.1.4 Operator Members

Operators can be defined either as member or nonmember functions.

When defining an operator function whose first parameter type is a class, you can choose to turn it into a member function of that class. That is an advantage if you want to have access to the private implementation of the class. For example, instead of a function

```
Time operator+(Time a, int min)
```

you can define a member function

```
Time Time::operator+(int min) const
{
   int result_minutes = hours * 60 + minutes + min;
   return Time((result_minutes / 60) % 24, result_minutes % 60);
}
```

Note that this member function is defined as `const` because the implicit argument (that is, the first argument of the operator) is not modified.

When you define an operator as a member function, it is still invoked in the same way as an operator expression. The compiler translates the operator expression into a member function call. For example, the expression

```
t + 60
```

is now the call

```
t.operator+(60)
```

Special Topic 13.1

Overloading Increment and Decrement Operators

There is a problem with overloading the ++ and -- operators. There are actually two forms of these operators: a **prefix** form

```
++x;
```

and a **postfix** form

```
x++;
```

For numbers, both of these two forms have the same effect: they increment x. However, they return different values. The value of ++x is the value of x after it has been incremented. The value of x++ is the value of x *before* the increment. You notice the difference only if you combine the increment expression with another expression. For example,

```
int x = 4;
cout << x++ << endl; // Prints 4, the value before the increment
cout << x << endl;   // Prints 5
cout << ++x << endl; // Prints 6, the value after the increment
```

Some programmers use the value of the increment operator in complex expressions such as this one:

```
int i = -1;
while (s[++i] != '\0') . . .
```

We recommend against this style—it is confusing and a common source of programming errors. Use ++ only to increment a variable and never use the return value. Then it doesn't make any difference whether you use x++ or ++x.

Nevertheless, there are two separate ++ operators—the prefix form and the postfix form—and the compiler must distinguish between them when they are overloaded. To overload the prefix form for `Time` objects, to increment the object by one minute, you define

```
void operator++(Time& a)
```

To overload the postfix form, you define

```
void operator++(Time& a, int dummy)
```

The `int dummy` parameter is not used inside the function; it merely serves to differentiate the two `operator++` functions.

If we wanted to define both operators so that their values can be used, we would do it as follows:

```
Time operator++(Time& a)
{
   a = a + 1;
   return a;
}
```

and

```
Time operator++(Time& a, int dummy)
{
   Time before_increment = a;
   a = a + 1;
   return before_increment;
}
```

Alternatively, here are the operators implemented as member functions:

```
Time Time::operator++()
{
   *this = *this + 1;
   return *this;
}

Time Time::operator++(int dummy)
{
   Time before_increment = *this;
   *this = *this + 1;
   return before_increment;
}
```

Special Topic 13.2

Implicit Type Conversions

C++ will perform a wide variety of *implicit* conversions; that is, conversions from one type to another in situations where no explicit request appears. You are familiar with this concept from the example of mixed-type arithmetic. In an expression such as

```
6 * 3.1415926
```

the left operand is an integer, while the right operand is a floating-point number. The integer value is implicitly converted to a floating-point equivalent, and the multiplication acts on these two values.

You can define implicit type conversions for your own classes. For example, Worked Example 13.1 defines a `Fraction` class:

```
class Fraction
{
public:
   Fraction(int n, int d); // The fraction n/d
   Fraction(int n); // The fraction n/1
   Fraction operator+(Fraction other) const;
   . . .
};
```

When dealing with user-defined classes, there are two categories of type conversions to consider. The first is the conversion of a type *to* the new class, while the second is a conversion from the class to another type.

Conversions to the class are handled using constructors. For example, our `Fraction` class defines a constructor that takes a single integer argument. It is used when an `int` needs to be converted to a `Fraction`:

```
Fraction a(3, 4);
a = a + 2; // Actually a = a.operator+(Fraction(2))
```

A conversion from the class to another type is accomplished by writing a **conversion operator**. A conversion operator uses a type as the operator name, has no arguments, and does not specify a result type (because the result type is implicit in the name). Here is a conversion operator for converting a `Fraction` into a `double`:

```
Fraction::operator double() const
{
   // Convert numerator to double, then do division
   return numerator * 1.0 / denominator;
}
```

This conversion is used whenever a `Fraction` needs to be converted to a `double`:

```
double root = sqrt(a); // Actually sqrt(a.operator double())
```

An interesting and somewhat unorthodox use of conversion operators is found in the class `istream`. As you have seen in Chapter 8, a series of values can be read using a loop, as follows:

```
while (cin >> x) { . . . }
```

The value returned by the `>>` operator is type `istream`. This is not a legal type for a condition, but the `istream` class defines an `operator bool` that returns `true` while input is successful, and `false` at the end of input or when input fails. The conversion is applied, and the loop executes as long as the stream is not exhausted.

Sometimes, you do *not* want to use a constructor as a type converter. For example, the `vector` class has a constructor with an integer parameter, denoting the initial size. You would not want to implicitly convert an integer such as `10` into a `vector<int>(10)`. Therefore, that constructor is declared as `explicit`, making it ineligible for implicit type conversions.

Special Topic 13.3

Returning References

In Section 13.1.3, you saw how to overload `<<` and `>>` operators, and you learned that you should return the stream so that the operators can be chained:

```
istream& operator>>(istream& in, Time& a)
{
   // Read in hours and minutes
   a = Time(hours, minutes);
   return in;
}
```

Note that the return type is a *reference* to `ostream`, and not an `ostream` object. As you will see in Section 13.2, if you change the return type to `ostream`, without a reference, the function returns a *copy* of `in`. That would not work. You cannot copy streams, but even if you could, then the next operation in a chain

```
cin >> t >> u;
```

would affect the copy, not the original stream.

Instead, we want to return the exact same stream that we received. This is achieved by returning a reference. To understand how this works, recall what happens when an object is passed by reference (such as the `Time& a` parameter in the operator>> function).

The function doesn't receive a copy of the argument, but instead a pointer to it. You don't notice this because the compiler automatically applies the `*` operator whenever you use the parameter variable. For example,

```
a = Time(hours, minutes);
```

actually means

```
*(pointer to argument) = Time(hours, minutes);
```

The same happens when you return a reference. A pointer to the value is returned. For example, in the call

```
return in;
```

a pointer to `in` is returned. Because `in` was a reference parameter, it was already a pointer, and the same pointer is returned.

When returning a reference, you have to be careful that the pointer is valid after the function exits. In our example, that is the case because the pointer was passed into the function. But

it would not be valid to return a reference to a local variable that is removed when the function exits.

Another reason for returning a reference is to provide write access to the internals of an object. For example, the `string` class provides an `operator[]` that you can use to modify the contents of a string:

```
string greeting = "hello";
greeting[0] = 'H';
```

The expression `greeting[0]` cannot be a `char` value because it appears to the left of an `=` operator. Instead, it is a reference. Here is an implementation, assuming that the `string` class stores the characters in a `char[]` array named `buffer`:

```
char& string::operator[](int index)
{
   return buffer[index];
}
```

The `operator[]` function actually returns the *address* of `buffer[index]`. The statement `greeting[0] = 'H';` means:

```
*(returned pointer) = 'H';
```

There is a drawback to this `operator[]` function. It does not work on `const` objects. For that reason, the `string` class has a second version of the operator:

```
char string::operator[](int index) const
{
   return buffer[index];
}
```

WORKED EXAMPLE 13.1

A Fraction Class

Learn how to implement a new data type that represents a fraction (a ratio of two integer values). See your E-Text or visit `wiley.com/go/bclo3`.

13.2 Automatic Memory Management

In Chapter 7, you were introduced to the concepts of memory allocation using the `new` operator, and the need to release allocated memory using the `delete` operator. In C++, management of memory on the free store is explicitly under the direction of the programmer. However, when you write classes that need to allocate memory, the users of your classes cannot be expected to explicitly deallocate the memory. In the following sections, you will learn how to to implement classes that manage memory correctly.

13.2.1 Constructors That Allocate Memory

You have been using `string` objects since the beginning of this book. What you may not have realized is that the contents of a string are not actually stored within the `string` object itself. Instead, an object of type `string` holds a pointer to an array of

character values. Because the length of the string is not known at compile-time, that array must be allocated on the free store, as shown in Figure 1.

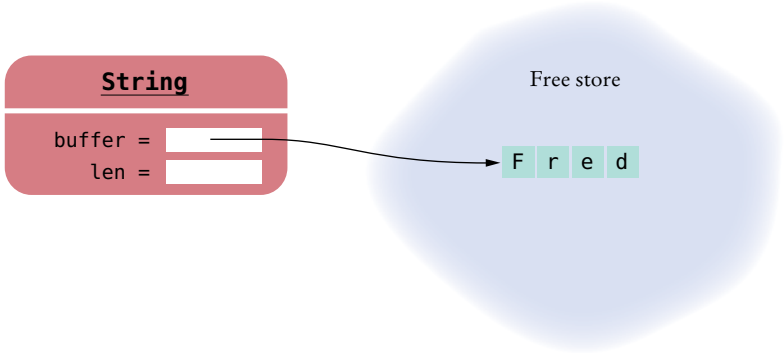


Figure 1 Characters of a `String` Object are Allocated on the Free Store

To examine the intricacies of memory management, we will define a simple `String` class that is similar to the standard library `string` class.

```
class String
{
public:
   String(); // Default constructor
   String(const char s[]); // Construction from character array
   . . .
private:
   char* buffer;
   int len;
};
```

When constructing a string from a zero-terminated character array, we need to copy the characters so that the string does not change if the original array is later modified. As already mentioned, that copy must be allocated on the free store:

```
String::String(const char s[])
{
   len = strlen(s);
   if (len > 0)
   {
      buffer = new char[len];
      for (int i = 0; i < len; i++)
      {
         buffer[i] = s[i];
      }
   }
   else
   {
      buffer = nullptr;
   }
}
```

Note that in our implementation we do not store a zero terminator.

The **default constructor** sets the `buffer` pointer to a null pointer:

```
String::String()
{
```

```
   len = 0;
   buffer = nullptr;
}
```

13.2.2 Destructors

Destructors are defined to take care of resource management when an object is no longer used.

A destructor is called when an object variable goes out of scope, or when an object on the free store is deleted.

When a `String` object is no longer in scope, the memory occupied by the object is reused. However, that poses a problem. Only the actual memory of the object, consisting of the `buffer` and `len` variables, is reclaimed. However, the array to which `buffer` points is not deleted. When the `String` object is no longer available, it is no longer possible to locate the pointer to the array, and a memory leak occurs (see Figure 2).

The C++ language has a special mechanism to overcome this potential problem. You can define a **destructor**, a function that is always called when an object variable is about to go out of scope, or when the memory that the object occupies is deleted.

```
void fun()
{
   String str;
   . . .
} // str.~String() automatically invoked here
```

The destructor for the `String` class should delete the array through the `buffer` pointer.

Here is the code for the destructor (see Syntax 13.2). Note that the name of the destructor is the name of the class, prefixed by the ~ symbol.

```
String::~String()
{
   delete[] buffer;
}
```

What should happen in the case that `buffer` is a null pointer? Calling `delete` on `nullptr` is safe—it simply does nothing. Therefore, you don't need a special case for that situation.

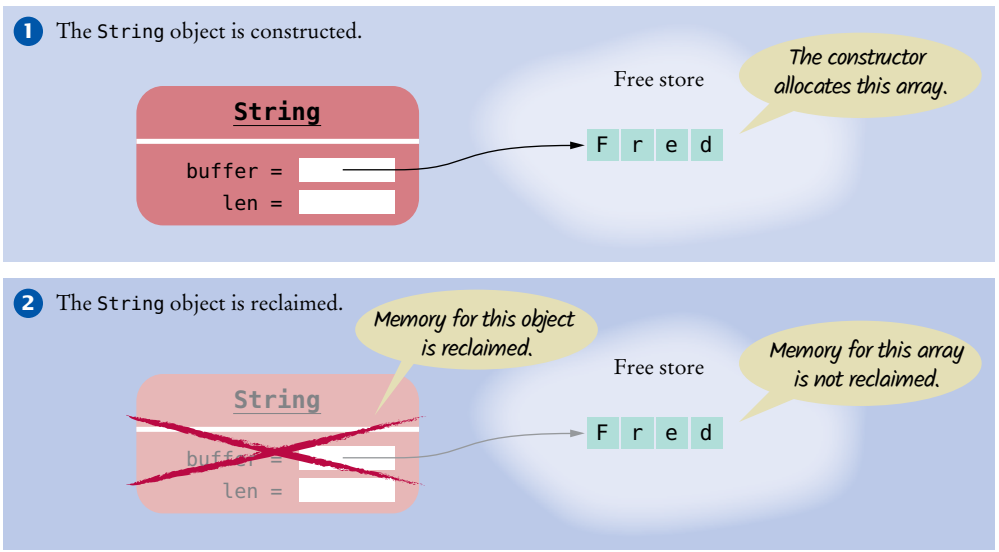
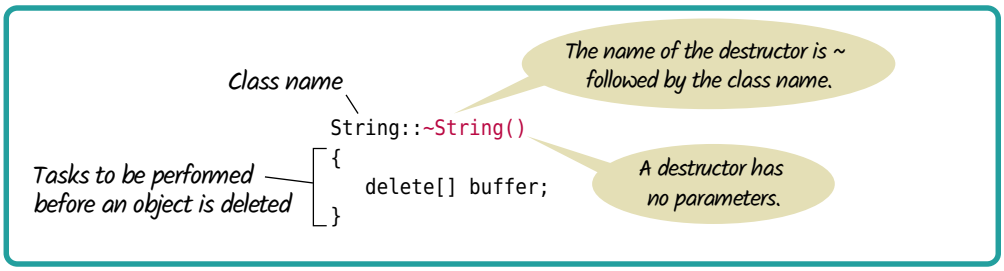


Figure 2 Reclaiming a `String` Object Without a Destructor

Syntax 13.2 Destructor Definition



You should think of a destructor as the opposite of a constructor. A constructor turns raw memory into an object. A destructor turns an object back into raw memory, recycling any remaining resources.

A class can have many overloaded constructors, but it can have at most one destructor, with no explicit parameters. After all, programmers don't explicitly invoke a destructor.

You should always supply a destructor when some amount of cleanup is required when an object goes out of scope. Very commonly, that cleanup involves recycling dynamic memory. But in some situations, a destructor might close a file or relinquish some other resource.

The standard library containers (`vector`, `list`, and so on) supply destructors that automatically recycle the dynamic memory that these classes use.

Destructors can ensure that free store memory is automatically recycled.

13.2.3 Overloading the Assignment Operator

Introducing a destructor solves an important problem. Objects that are no longer used do not cause memory leaks. However, the solution is not perfect. Consider the following situation:

```
String secretary("Fred");
String treasurer("Ann");
```

Part ❶ of Figure 3 shows the memory layout.

Now suppose that we assign one `String` object to another:

```
secretary = treasurer;
```

Assigning an object to another assigns each of the data members. That means:

```
secretary.buffer = treasurer.buffer;
secretary.len = treasurer.len;
```

This process is called *memberwise assignment*. Part ❷ of Figure 3 shows the situation after the assignment. You will immediately see a problem. The old character array of the `secretary` object is now orphaned—another **memory leak** has occurred.

Furthermore, there is a more subtle problem. Eventually, the `secretary` and `treasurer` objects will go out of scope. Their destructors will be invoked. Objects are destroyed in the opposite order of construction. That is, `treasurer` gets destroyed first. Part ❸ of Figure 3 shows what happens in the destructor. The array holding `"Ann"` is deleted.

If memberwise assignment is insufficient for a class, provide an overloaded `operator=`.

Note that the `buffer` pointer in the `secretary` object points to the deleted array. When the `secretary` object is destroyed, the character array is deleted twice (see Part ❹ of Figure 3). That is a fatal error that will compromise the internal structure of the free store.

The remedy is to overload the = operator to make the assignment safe. In the overloaded operator, we delete the array that is no longer used, and make a copy of the character array that is now used by both objects.

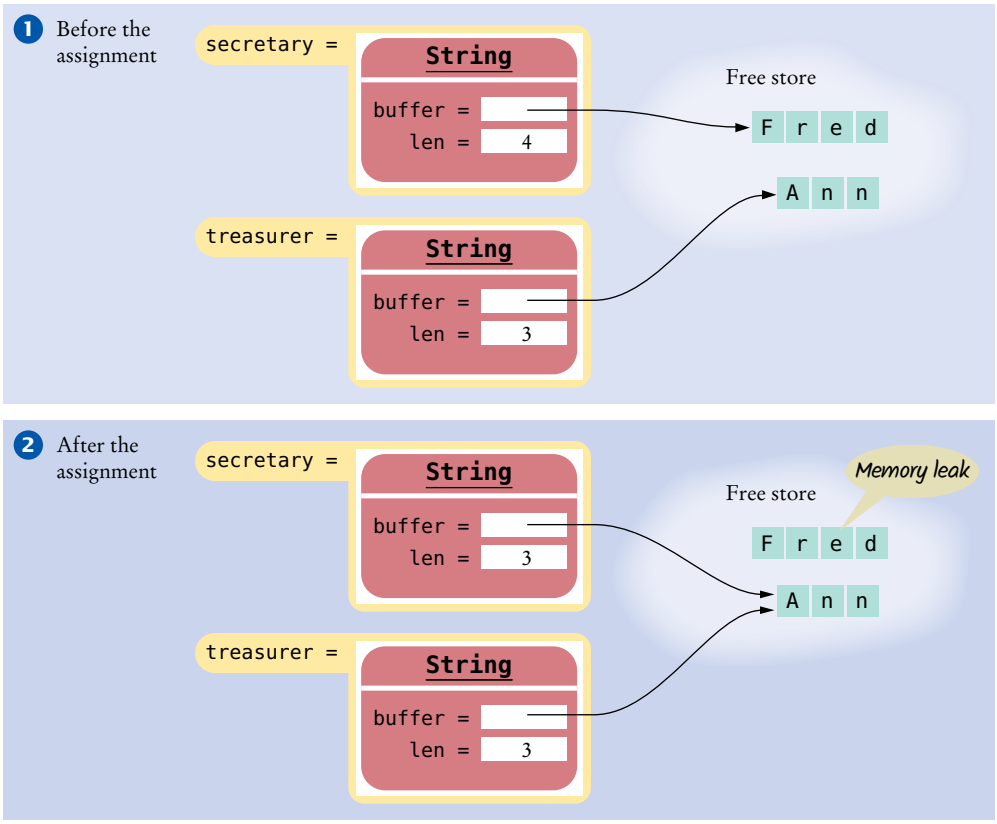


Figure 3 Assignment Without an Overloaded Assignment Operator

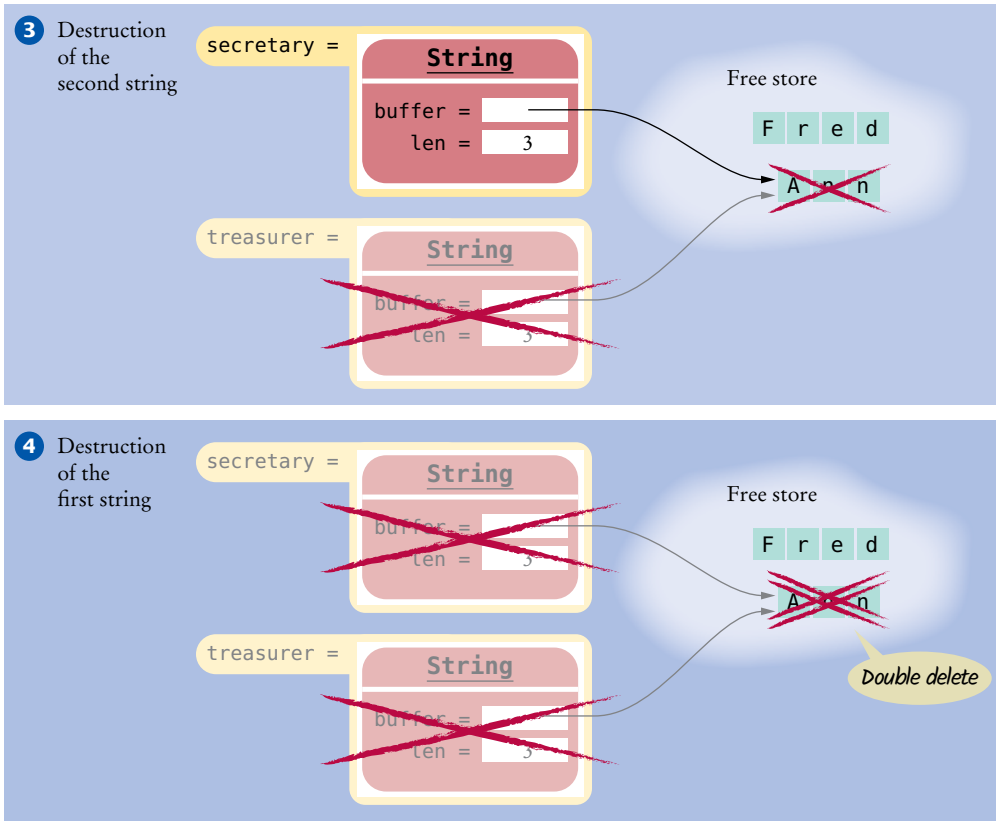


Figure 3 (continued) Assignment Without an Overloaded Assignment Operator

```
String& String::operator=(const String& other)
{
   . . .
   delete[] buffer;
   len = other.len;
   if (len > 0)
   {
      buffer = new char[len];
      for (int i = 0; i < len; i++)
      {
         buffer[i] = other.buffer[i];
      }
   }
   else
   {
      buffer = nullptr;
   }
   . . .
}
```

The basic concept behind the assignment operator is straightforward. The assignment first deletes the old character array, and then copies the array of the assigned object. It is also possible to reuse the old character array if it can hold the new string; see Exercise P13.15.

In the remainder of this section, we discuss several technical points that you have to know when overloading the = operator.

Unlike most other overloaded operators, `operator=` *must* be a member function. It is a syntax error to define a nonmember `operator=`.

The `operator=` function must take care not to carry out a destructive "self-assignment". It can happen that a programmer assigns an object to itself, for example in a context such as

```
a[0] = a[i];
```

where `a` is an array of objects and `i` is allowed to have the value `0`.

However, the `operator=` function immediately deletes the character array. If both strings are the same object, then the array is deleted and no longer available for copying. The test

```
if (this != &other)
```

tests whether the implicit and explicit argument have different addresses and are therefore different objects. Thus, an `operator=` function should include the test against self-assignment:

```
String& String::operator=(const String& other)
{
   if (this != &other)
   {
      . . .
   }
   . . .
}
```

Sometimes, programmers chain the = operator like this:

```
z = y = x;
```

To make this chaining work, an `operator=` function should always return `*this`, a reference to the left-hand side of the assignment:

```
String& String::operator=(const String& other)
{
   . . .
   return *this;
}
```

Because the = operator is right associative, the expression

```
z = y = x;
```

is equivalent to

```
z = (y = x);
```

The parenthesized expression is executed first. It becomes a function call

```
y.operator=(x);
```

That function call returns a reference to `y`. The second assignment is therefore the call

```
z.operator=(y);
```

The overall effect is that the contents of `x` is first copied to `y` and then to `z`.

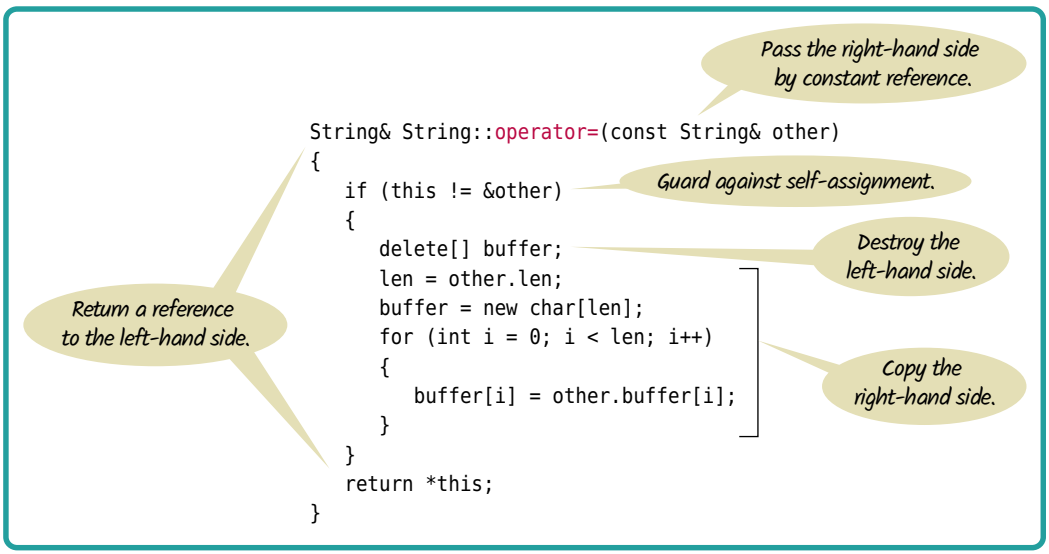
Putting everything together yields the following code for the assignment operator:

```
String& String::operator=(const String& other)
{
   if (this != &other)
   {
```

```
      delete[] buffer;
      len = other.len;
      if (len > 0)
      {
         buffer = new char[len];
         for (int i = 0; i < len; i++)
         {
            buffer[i] = other.buffer[i];
         }
      }
      else
      {
         buffer = nullptr;
      }
   }
   return *this;
}
```

Whenever you have a class with a destructor, you need to consider whether member-wise assignment is compatible with the destructor. If it is not (for example because two copies of a pointer would be deleted), you need to supply an assignment operator that makes a complete copy.

Syntax 13.3 Overloaded Assignment Operator



13.2.4 Copy Constructors

A copy constructor initializes an object as a copy of another object.

Consider the following definition of two `String` variables:

```
String treasurer("Ann");
String secretary = treasurer;
```

Even though the second definition looks like an assignment, `operator=` is not invoked. The purpose of `operator=` is to assign an object to an *existing* object. However, the `secretary` object has not yet been constructed. That is, the pointer `secretary.buffer` is a

random value. If you review the code for the `operator=` function, you will note that the first part deletes the old buffer. It would be fatal if `operator=` were to be executed with an uninitialized object because it would then delete an uninitialized pointer, causing a program crash or free store corruption.

Instead, the compiler invokes another memory management function, the **copy constructor**. The copy constructor defines how to construct an object of a class as a copy of another object of the same class.

If the programmer doesn't provide a copy constructor, the compiler provides one that copies all data members.

If you don't define a copy constructor, then the compiler provides a version that simply constructs the data members of the new object as copies of the corresponding data members of the existing object. For the `String` class, the default copy constructor would look like this:

```
String::String(const String& other)
{
   buffer = other.buffer;
   len = other.len;
}
```

However, this version of the copy constructor is inappropriate. It can lead to the same kind of errors as the default version of the assignment operator. You must define the copy constructor to make a copy of the buffer.

Here is a valid copy constructor for the `String` class:

```
String::String(const String& other)
{
   len = other.len;
   if (len > 0)
   {
      buffer = new char[len];
      for (int i = 0; i < len; i++)
      {
         buffer[i] = other.buffer[i];
      }
   }
   else
   {
      buffer = nullptr;
   }
}
```

© Alex Gumerov/iStockphoto.

Whenever an object is constructed as a copy of an existing object, the copy constructor is invoked.

Copy constructors are used for copying function arguments and return values.

A copy constructor is also invoked when an object is passed to a function as a parameter, such as this one:

```
void repeat(String str, int count)
{
  for (int i = 0; i < count; i++)
  {
     str.print(cout);
  }
  cout << endl;
}
```

Now consider a function call:

```
String name = "Fred";
repeat(name, 2);
```

When the function is called, the parameter variable `str` is allocated, and it needs to be constructed. The copy constructor constructs it as a copy of the `name` argument (see Figure 4).

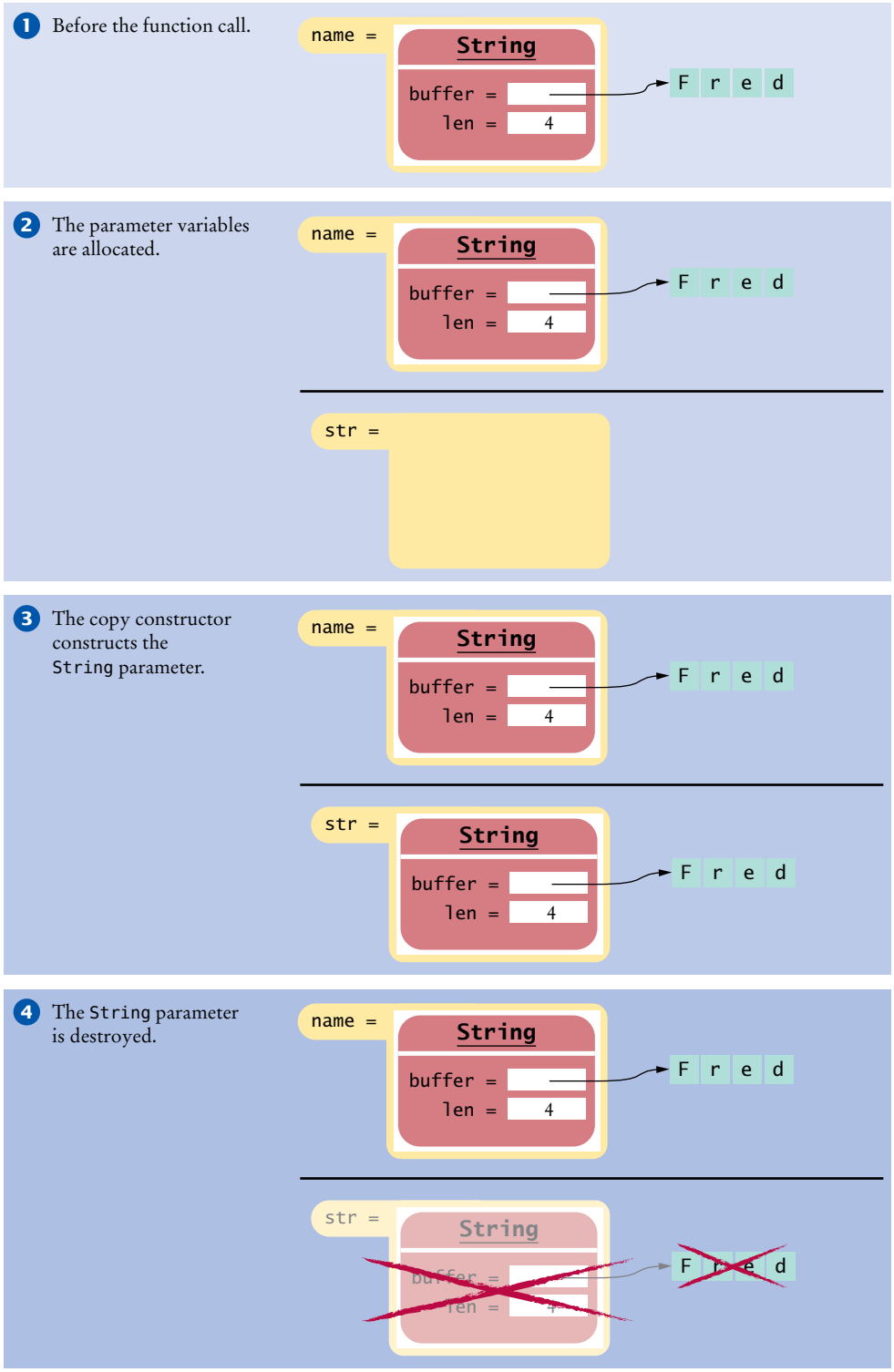
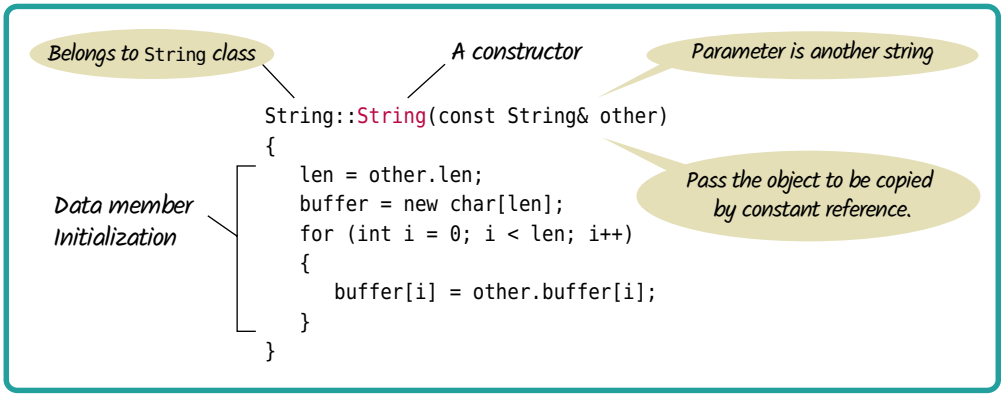


Figure 4 Initializing a Parameter with a Copy Constructor

When the `repeat` function exits, the parameter variable `str` goes out of scope, and the destructor is invoked. (This process of copying and destruction is expensive, and it would be better to use a reference parameter instead—see Programming Tip 13.1.)

Syntax 13.4 Copy Constructor



Constructors, destructors, and the assignment operator must all be defined to work together to manage internally allocated dynamic memory.

The assignment operator, copy constructor, and destructor are collectively called the "**big three**". You must implement them for any class that manages free store memory.

You only need to worry about the "big three" if your class manages free store memory. If you use the library classes, such as vector or list, there is nothing to worry about. These classes already implement the "big three" for you.

EXAMPLE CODE See sec02 of your companion code to see "the big three" implemented for the `String` class.

Table 1 summarizes the "Big 3" memory management functions and the optional move constructor and move assignment functions discussed in Special Topic 13.6. Collectively these functions are called the "Big 5".

Table 1 Memory Management Functions

Function	Example	Description

Copy Constructor	`String::String(const String& other)`
	`{`
	`  len = other.len;`
	`  buffer = new char[len];`
	`  for (int i = 0; i < len; i++)`
	`  {`
	`  buffer[i] = other.buffer[i];`
	`  }`
	`}`
		Initializes this object as a copy of `other`.

Destructor	`String::~String()`
	`{`
	`  delete[] buffer;`
	`}`
		Releases the resources (such as free store memory) used by this object.

Table 1 Memory Management Functions

Function	Example	Description

Assignment Operator	`String& String::operator=(const String& other)`
	`{`
	`  if (this != &other)`
	`  {`
	`  delete[] buffer;`
	`  len = other.len;`
	`  buffer = new char[len];`
	`  for (int i = 0; i < len; i++)`
	`  {`
	`  buffer[i] = other.buffer[i];`
	`  }`
	`  }`
	`  return *this;`
	`}`
		Releases the resources of this object, then makes this object a copy of the `other` object.

Move Constructor	`String::String(String&& other)`
	`{`
	`  len = other.len;`
	`  buffer = other.buffer;`
	`  other.buffer = nullptr;`
	`}`
		Moves the resources from the `other` object into this object, then prepares the `other` object for destruction. This constructor is optional and used for efficiently constructing an object from a temporary object that is about to be destroyed.

Move Assignment	`String& String::operator=(String&& other)`
	`{`
	`  delete[] buffer;`
	`  len = other.len;`
	`  buffer = other.buffer;`
	`  other.buffer = nullptr;`
	`  return *this;`
	`}`
		Releases the resources of this object, makes this object a copy of `other`, and prepares the `other` object for deletion. This operator is optional and used for efficiently assigning a temporary object that is about to be destroyed.

Programming Tip 13.1

Use Reference Parameters To Avoid Copies

As you have seen in Section 13.2.4, copying a function argument into a parameter variable invokes a copy constructor. For example, when you call

```
repeat(name, 2);
```

the character array `name.buffer` is copied when the function is called, and the copy is deleted when the function exits and the parameter variable is destroyed.

It is easy to avoid that cost, by using a constant reference:

```
void repeat(const String& str, int count)
{
   . . .
}
```

Now the `repeat` function receives a pointer to the argument, and not a copy. (References are just pointers in disguise, without the need of having to use the `&` and `*` operators. See Special

Topic 13.3.) Because the reference is declared as `const`, the function cannot modify the argument. Using a constant reference is more efficient than copying the argument, and just as safe. Unfortunately, it is more verbose.

Up to now, we were more concerned with simplicity than with efficiency, and we frequently passed objects into functions without using references. From now on, we will prefer constant references.

When the return value of a function is an object, it needs to be copied to the caller, again using a copy constructor. See Special Topic 13.6 to see how to reduce the cost of that copy.

Common Error 13.1

Defining a Destructor Without the Other Two Functions of the "Big Three"

It is often intuitively obvious to programmers when a class needs a destructor. If a class contains a pointer to free store memory, or an open file, or some other resource that requires cleanup, then the destructor gets to do the cleanup.

Such a user-defined destructor is incompatible with the automatically generated assignment and copy operations. Consider the example of a `Quiz` class that consists of questions, using the `Question` class and its derived classes of Chapter 10.

```
class Quiz
{
public:
   void add_question(Question* q);
   . . .
private:
   vector<Question*> questions;
};
```

Here, we use pointers to questions because of polymorphism. Different question objects have different content and capabilities. A user will add questions using code like this:

```
Question* q = new ChoiceQuestion(. . .);
quiz.add(q);
```

How will the questions get deleted? It seems reasonable to provide a destructor:

```
Quiz::~Quiz()
{
   for (int i = 0; i < questions.size(); i++)
   {
      delete questions[i];
   }
}
```

But what if a `Quiz` object is copied? Then all questions are deleted twice when the original and the copy go out of scope.

You have several choices:

1. Don't define a destructor. Instead, define a member function that the class user must explicitly invoke to free up the memory. Then it is up to the class user to track any copies.
2. Disallow copying by "deleting" the automatically generated assignment operator and copy constructor—see Special Topic 13.5.
3. Implement the assignment operator and copy constructor to make complete copies of all parts of the object, so that the original and the copy can be destroyed separately (see Exercise P13.17).

Special Topic 13.4

Virtual Destructors

Whenever a class that is intended to serve as a base class for inheritance declares a destructor, the destructor should be declared as virtual. To illustrate, consider the following class declaration:

```
class Question
{
public:
   . . .
   virtual ~Question();
private:
   string text;
   string answer;
};

class ChoiceQuestion : public Question
{
   . . .
private:
   vector<string> choices;
}

Question::~Question() {}
```

At first glance, it doesn't look as if `Question` and `ChoiceQuestion` need destructors. But every class has one, provided by the C++ compiler. That destructor invokes the destructors on all data members. And that is a good thing because `string` and `vector` members need to be properly destroyed.

When deleting pointers that can point to a base class or derived class, it is important that the correct destructor is invoked. This situation arises in the `Quiz` class of Common Error 13.1. Its destructor calls `delete` on the pointer `questions[i]`, which might point to a `Question` or `ChoiceQuestion` object. The `delete` operation, in turn, invokes the destructor of that object. That is, it must invoke either the `Question` or `ChoiceQuestion` destructor, depending on the actual type of the object to which `questions[i]` points.

If the destructor in `Question` is declared `virtual`, as shown, then the expression `delete questions[i]` causes the appropriate destructor to be executed. If the `virtual` designation is omitted, and a `ChoiceQuestion` is deleted, only the base class data members will be destroyed.

Whenever a class has at least one virtual function, it should also have a virtual destructor.

Special Topic 13.5

Suppressing Automatic Generation of Memory Management Functions

Sometimes, you want a class with a destructor that cleans up resources, but you don't want anyone to make copies. An example is the `ofstream` class. The destructor closes the file, which is a good thing because programmers often forget to close a file explicitly, which can cause data loss. (File operations are usually batched for increased performance, and the last batch may not be written to disk unless a file is closed.)

But we don't want to allow copying an `ofstream` object. What would that mean? Two objects that write to the same file? We certainly don't want the file to be closed when the first object goes out of scope.

If you try copying an `ofstream` object, the compiler will refuse. That is fine—you don't want to copy streams. When you pass an `ofstream` object to a function, you always use a reference because writing to the stream changes it.

If you want to forbid copying in your own classes, declare the copy constructor and `operator=` with the `delete` keyword, like this:

```
class Quiz
{
public:
   Quiz(const Quiz& other) = delete;
   Quiz& operator=(const Quiz& other) = delete;
   . . .
};
```

Then these memory management functions are not generated, and any attempt to copy an object will be flagged as an error.

Special Topic 13.6

Move Operations

Consider a function that returns a `String` object:

```
String String::substr(int start, int length) const
{
   String result;
   if (length > 0)
   {
      result.len = length;
      result.buffer = new char[length];
      for (int i = 0; i < length; i++)
      {
         result.buffer[i] = buffer[start + i];
      }
   }
   return result;
}
```

Let's look closely at a call to this function.

```
String name("Fred");
String initial = name.substr(0, 1);
```

An object `result` of type `String` is constructed in the function, and it must be destroyed when the function exits. Therefore, the string needs to be copied to `initial` before the destruction.

It is wasteful to make a copy of an object and then immediately destroy the original. It would be better if `result` was *moved* to the `initial` object.

For added efficiency, you can define a *move constructor* that specifies how such a movement should be carried out:

```
String::String(String&& other)
{
   len = other.len;
   buffer = other.buffer;
   other.buffer = nullptr;
}
```

The `String&&` type denotes a reference to a temporary object that will be destroyed after the call. In the move constructor, you can grab its data and set the `other` object to a state in which the destructor works correctly.

This is what our move constructor does. Consider what happens to `result` when it is returned from the `substr` member function and moved into the `initial` variable. The `initial` variable is constructed from `result` with the move constructor. The `len` and `buffer` members

are copied from the `result` object into the `initial` object. Then `result.buffer` is set to `nullptr` so that `result` can be safely destroyed.

There is also a "move" version of `operator=`, if the right hand side is a temporary object. Consider this scenario:

```
String treasurer("Fred");
treasurer = "Ann";
```

This actually means that a temporary `String` object is constructed from the character array `"Ann"`, which is then assigned to `treasurer` and subsequently destroyed. As before, you want to transfer the buffer from the `other` object, and set the `other` object to a state in which it can be destroyed.

```
String& String::operator=(String&& other)
{
   delete[] buffer;
   len = other.len;
   buffer = other.buffer;
   other.buffer = nullptr;
   return *this;
}
```

Another strategy for implementing move assignment is to swap the contents of the two objects (see Exercise P13.12). For both move operations, note that the reference to `other` is *not* declared as `const` because we modify the object for destruction.

Defining move operations is optional. If you don't provide them, then the regular copy construction or assignment will take place, making a copy instead of transferring the data.

Special Topic 13.7

Shared Pointers

As you have seen, it can be challenging to keep track of all pointers that point to an object, and to delete the object when the last pointer has gone away. The `shared_ptr` class automates this tracking. For example, here we create and use a shared pointer to a `Question` object:

```
shared_ptr<Question> q1(new Question);
q1->set_text("Who was the inventor of C++?");
q1->set_answer("Bjarne Stroustrup");
q1->display();
```

As you can see, the `->` operator of the `shared_ptr` class is overloaded to access the members of the underlying object.

Moreover, the `shared_ptr` class provides memory management operations that track the number of pointers to the object. When the last `shared_ptr` is destroyed, the object is deleted.

Consider again the `Quiz` class of Common Error 13.1. Because a `Quiz` managed a vector of `Question*` pointers, the class designer was obliged to worry about memory management, which is a distraction from the task of implementing quizzes. Shared pointers offer an effective solution. Change the implementation to:

```
class Quiz
{
   . . .
private:
   vector<shared_ptr<Question>> questions;
};
```

Now the automatically generated memory management operations do the right thing. When a `Quiz` object is copied, the vector is copied, and all shared pointers are copied and tracked. The

programmer implementing the `Quiz` class does not need to worry about memory management at all.

The tracking of `shared_ptr` objects has one drawback. It fails when there is a cycle of shared pointers. Exercise P13.18 explores this limitation.

Should you use shared pointers in your programs? If you use pointers for sharing objects or for polymorphism (that is, to describe objects that can belong to base or derived classes), you should seriously consider using shared pointers. They work exactly like regular pointers and exact a very modest management overhead.

Simply use `shared_pointer<`*Type*`>` instead of *Type*`*`. You use the `*` and `->` operators in the usual way.

Most importantly, you never have to call `delete`. In the uncommon case that you have cycles of shared pointers, you can break them by setting shared pointers to `nullptr` when the objects are no longer needed.

We will stick with regular pointers in the following chapters that focus on implementing data structures. In that context, it makes sense to manage pointers as efficiently as possible, without the overhead of shared pointers.

WORKED EXAMPLE 13.2

Tracing Memory Management of Strings

Learn how to add tracing messages to the memory management functions in the `String` class, then use them to analyze the calls to the assignment operator, copy constructor, and destructor. See your E-Text or visit `wiley.com/go/bclo3`.

13.3 Templates

A template produces functions or classes that depend on types.

The `vector` class differs from classes such as `string` or `ostream`. To use a `vector`, you have to specify the element type such as `vector<int>` or `vector<string>`. In C++, `vector` isn't a class—it is a *template* for a class. When you specify the component type, the compiler makes a class with the given type.

Just like a sewing template that provides instructions for making a dress, a C++ template specifies how to make a class or function.

In the following sections, you will learn how to write templates for making classes and functions.

13.3.1 Function Templates

A function template produces functions when the type variables are replaced with actual types.

Suppose you want to write a function that prints an array of integers, separated by commas. It's not hard to do:

```
void print(ostream& out, int data[], int size)
{
   for (int i = 0; i < size; i++)
   {
      if (i > 0) { out << ", "; }
      out << data[i];
   }
   out << endl;
}
```

In your next project, you need to do the same for an array of `double` values. Of course, you can copy and edit:

```
void print(ostream& out, double data[], int size)
{
   for (int i = 0; i < size; i++)
   {
      if (i > 0) { out << ", "; }
      out << data[i];
   }
   out << endl;
}
```

This works, but it is not entirely satisfactory. There ought to be a mechanism that automatically produces this function for *any* array type.

That is what a *template* provides. A template lets you write a function or a class definition with one or more **type parameters**. Here is a template for making a `print` function for arrays of any type:

```
template<typename T>
void print(ostream& out, T data[], int size)
{
   for (int i = 0; i < size; i++)
   {
      if (i > 0) { out << ", "; }
      out << data[i];
   }
   out << endl;
}
```

Now you can produce functions `print<int>`, `print<double>`, and so on.

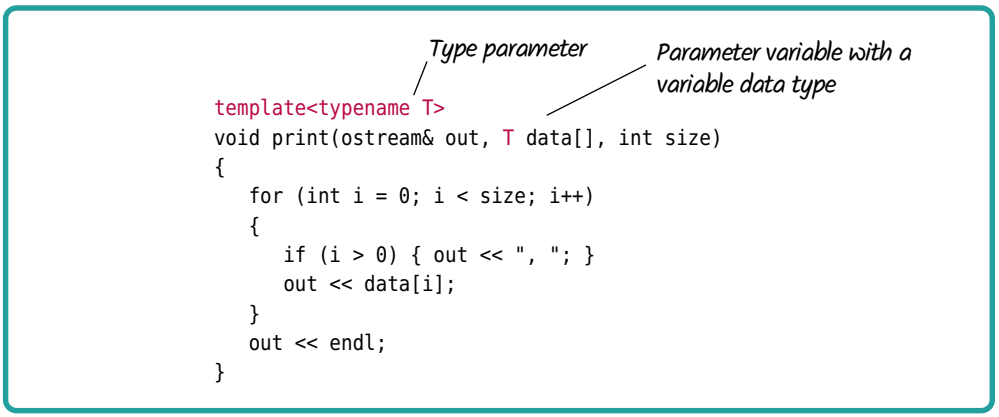
```
int a[] = { 1, 7, 2, 9 };
print<int>(cout, a, 4);
```

Template parameters are inferred from the arguments in a function invocation.

When you call a function that was defined as a template, the compiler will even infer the correct template type from the argument types:

```
double b[] = { 3.14159, 1.41421 };
print(cout, b, 2); // Will use print<double>
string c[] = { "Fred", "Sally", "Alice" };
print(cout, c, 3); // Will use print<string>
```

Syntax 13.5 Function Template



EXAMPLE CODE See sec03_01 of your companion code for a program that demonstrates the `print` function template.

13.3.2 Class Templates

A class template produces classes that depend on type parameters.

We now turn to templates that produce classes. Suppose, for example, you want to write a function that traverses an array and simultaneously keeps track of the minimum and the maximum.

```
int min = data[0];
int max = data[0];
for (int i = 1; i < size; i++)
{
   if (data[i] < min) { min = data[i]; }
   if (data[i] > max) { max = data[i]; }
}
```

But how can the function return both values? We can use a `Pair` class:

```
class Pair
{
public:
   Pair(int a, int b);
   int get_first() const;
   int get_second() const;
private:
   int first;
   int second;
};
```

Then our function can be completed like this:

```
Pair minmax(int data[], int size)
{
   . . .
   return Pair(min, max);
}
```

However, the `Pair` class is not very flexible. Suppose you want to gather a pair of `double` or `string` values. It is far more useful to define `Pair` as a template for producing classes `Pair<int>`, `Pair<double>`, `Pair<string>`, and so on.

The syntax is very similar to that of function templates. You specify the type parameter in the `template` header, and use it whenever the type is required:

```
template<typename T>
class Pair
{
public:
   Pair(T a, T b);
   T get_first() const;
   T get_second() const;
private:
   T first;
   T second;
};
```

Now you specify a function template for each member function:

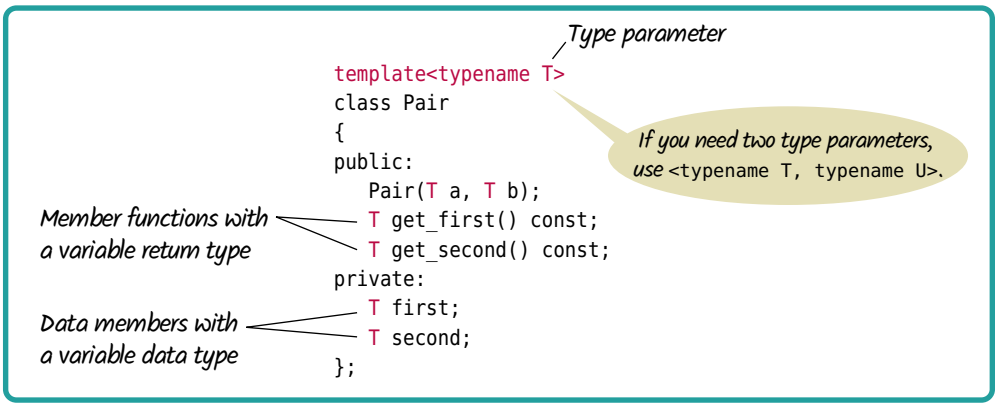
```
template<typename T>
T Pair<T>::get_first() const
{
   return first;
}

template<typename T>
T Pair<T>::get_second() const
{
   return second;
}
```

Each function name is prefixed by the "`Pair<T>::`" qualifier. And, of course, the **type variable** `T` is used whenever the element type is required.

When declaring a constructor, you use `Pair<T>::` for the class name, but the constructor name is `Pair` without another `T`.

```
template<typename T>
Pair<T>::Pair(T a, T b)
{
   first = a;
   second = b;
}
```

Syntax 13.6 Class Template



We used this `Pair` template as an example to show you how straightforward it is to define a class template. The standard library has a `pair` class that is similar, but it has two type parameters, one each for the first and second element. For example, a `pair<string, int>` holds a string and an integer. You will see many more examples of class templates in the coming chapters when you learn about container classes.

When you define a function template for use by other programmers, you need to put the entire template definition into a header file. This is different from non-template functions, where you put the function declaration into the header file and the definition (that is, the code) into the source file. From the perspective of the C++ compiler, a function template doesn't actually contain code. It contains instructions on how to produce the code for the function, and those instructions need to be visible in all source files that make use of the template.

EXAMPLE CODE See sec03_02 of your companion code for a program that demonstrates the `Pair` template class.

Special Topic 13.8

Non-Type Template Parameters

The examples up to this point have all used types as template parameters, specified using the `typename` reserved word. Although type names are the most common form of template parameter, you can also use compile-time constants.

For example, you can define a class for a two-dimensional array of values. Such an array is often called a matrix, and we use that term in this example:

```
template<typename T, int ROWS, int COLUMNS>
class Matrix
{
   . . .
private:
   T data[ROWS][COLUMNS];
};
```

To create an instance of this matrix the programmer would specify both the element type and the size:

```
Matrix<double, 3, 4> a; // A 3 × 4 matrix of double values
```

By using template parameters for the bounds, they become part of the type. This means that compatibility between variables of different sizes will be checked at compile time:

```
Matrix<int, 3, 4> a;
Matrix<double, 3, 4> b;
Matrix<int, 5, 7> c;
Matrix<int, 3, 4> d;
b = a; // Error, element types don't match.
c = a; // Error, sizes don't match, so types differ.
d = a; // OK. Element types and sizes match.
```

CHAPTER SUMMARY

Implement overloaded operators.

- You can define a new meaning for a C++ operator by defining a function whose name is `operator` followed by the operator symbol.
- The precedence and associativity of an overloaded operator is the same as that of the original operator.
- Operators can be defined either as member or nonmember functions.

Implement classes that manage their memory allocation and deallocation.

- Destructors are defined to take care of resource management when an object is no longer used.
- A destructor is called when an object variable goes out of scope, or when an object on the free store is deleted.
- If memberwise assignment is insufficient for a class, provide an overloaded `operator=`.
- A copy constructor initializes an object as a copy of another object.
- If the programmer doesn't provide a copy constructor, the compiler provides one that copies all data members.
- Copy constructors are used for copying function arguments and return values.
- Constructors, destructors, and the assignment operator must all be defined to work together to manage internally allocated dynamic memory.

Use and define template classes and functions.

- A template produces functions or classes that depend on types.
- A function template produces functions when the type variables are replaced with actual types.
- Template parameters are inferred from the arguments in a function invocation.
- A class template produces classes that depend on type parameters.

REVIEW EXERCISES

R13.1 When would you choose an overloaded operator for a particular operation, and when would you choose a function?

R13.2 Could you overload the unary negation operator so that `-T` would decrement the `Time` value `T` by 60 seconds? Would this be a good idea or not?

R13.3 Which operators does the `vector` class overload?

R13.4 Suppose you define a `Fraction` class and want to define an operator to raise a fraction to an integer power. Can you define a `**` operator so that, for example, `f ** 2` is `f` squared?

R13.5 Suppose you define a `Fraction` class and want to define an operator to raise a fraction to an integer power. What is the drawback of defining `operator^` so that, for example, `f ^ 2` is `f` squared? (*Hint:* What is `f ^ 2 + 1`?)

R13.6 Suppose you define a `Fraction` class with an `operator*` that multiplies two fractions, a constructor `Fraction(int)` and a type conversion `Fraction::operator double` (see Special Topic 13.2). What is the meaning of `3 * f`? Will the `3` turn into a `Fraction` or `f` into a `double`? Try it out.

R13.7 Suppose you want to overload the `*` operator so that `n * s` or `s * n` yields `n` repetitions of the `string` variable `s`. For example, if `s` is `"hi"`, then `3 * s` is `"hihihi"`. What functions do you need to define? Can any of them be member functions?

R13.8 What error is being committed in the assignment operator for the following class?

```
class String
{
public:
   String(const char s[]);
   String& operator=(const String& other);
private:
   char* buffer;
   int len;
};

String::String(const char s[])
{
   len = strlen(s);
   buffer = new char[len];
   for (int i = 0; i < len; i++)
   {
      buffer[i] = s[i];
   }
}

String& String::operator=(const String& other)
{
   if (this != &other)
   {
      len = other.len;
      buffer = new char[len];
      for (int i = 0; i < len; i++)
      {
         buffer[i] = other.buffer[i];
```

```
      }
   }
   return *this;
}
```

■■ **R13.9** What is the error in the assignment operator for this class?

```
class String
{
public:
   String(const char s[]);
   String& operator=(const String& other);
private:
   char* buffer;
   int len;
};

String::String(const char s[])
{
   len = strlen(s);
   buffer = new char[len];
   for (int i = 0; i < len; i++)
   {
      buffer[i] = s[i];
   }
}

String& String::operator=(const String& other)
{
   if (this != &other)
   {
      delete[] buffer;
      len = other.len;
      char* buffer = new char[len];
      for (int i = 0; i < len; i++)
      {
         buffer[i] = other[i];
      }
   }
   return *this;
}
```

■■ **R13.10** What output will be printed by the following program?

```
class Base
{
public:
   Base();
   ~Base();
};

void Base::Base()
{
   cout << "Constructing Base" << endl;
}

void Base::~Base()
{
   cout << "Destroying Base" << endl;
}
```

```
class Derived : public Base
{
public:
   Derived();
   ~Derived();
};

void Derived::Derived()
{
   cout << "Constructing Derived" << endl;
}

void Derived::~Derived()
{
   cout << "Destroying Derived" << endl;
}

int main()
{
   Base b;
   Derived d;
   return 0;
}
```

■■■ **R13.11** What output will be printed by the following program?

```
class Base
{
public:
   Base();
   Base(int v);
   virtual void display();
private:
   int value;
};

Base::Base()
{
   value = 7;
}

Base::Base(int v)
{
   value = v;
}

void Base::display()
{
   cout << "In base class, value is " << value << endl;
}

class Derived : public Base
{
public:
   Derived(int v);
   virtual void display();
private:
   int new_value;
};
```

```
Derived::Derived(int v)
{
   new_value = v;
}

void Derived::display()
{
   Base::display();
   cout << "In derived class, value is " << new_value << endl;
}

int main()
{
   Derived d(5);
   d.display();
}
```

■■ **R13.12** With the definitions of `Base` and `Derived` from Exercise R13.11, what is the output of this program?

```
int main()
{
   Base* pb = new Base;
   Base* pd = new Derived;
   delete pb;
   delete pd;
   return 0;
}
```

What can you do to make sure that the correct destructor is called?

■ **R13.13** What is the difference between destruction and deletion of an object?

■■ **R13.14** What problems could programmers encounter if they defined a destructor for a class but no assignment operator? Illustrate your description with an example class.

■■ **R13.15** What problems could programmers encounter if they defined a destructor for a class but no copy constructor? Illustrate your description with an example class.

■ **R13.16** Which objects are destroyed when the following function exits? Which values are deleted?

```
void f(const Fraction& a)
{
   Fraction b = a;
   Fraction* c = new Fraction(3, 4);
   Fraction* d = &a;
   Fraction* e = new Fraction(7,8);
   Fraction* f = c;
   delete f;
}
```

■ **R13.17** You can find the code for the `vector` template in the header file `<vector>`. Locate and copy the "big three" memory management functions in the class definition.

■ **R13.18** Why do you have to specify template parameters when you instantiate a class template, but not when you instantiate a function template?

■ **R13.19** How is a template class different from a template function?

■ **R13.20** How is a non-type template parameter different from a typename template parameter? When might you use such a value?

PRACTICE EXERCISES

■■ **E13.1** Overload the << operator for the `cashregister.cpp` class of Chapter 9 so that you can call

```
reg << price;
```

to add an item. Is this a wise choice of operator overloading? Why or why not?

■ **E13.2** Overload the << operator so that one can print a `vector<int>` to an `ostream`.

■■ **E13.3** Repeat Exercise E13.2, but use a template so that any `vector<T>` can be printed.

■■ **E13.4** Implement a class `Point` with an x- and y-coordinate, each of type `double`. Overload the `*` operator so that a point can be scaled by a `double` value. For example, `0.5 * Point(2, 3)` is the same as `Point(1, 1.5)`. Overload the `==` and `!=` operators to compare two points, and the << and >> operators to print and read points.

■ **E13.5** Implement a < operator for the `String` class of Section 13.2.

■■ **E13.6** Following Special Topic 13.3, add an `operator[]` to the `String` class of Section 13.2 so that `s[i]` yields the ith character of the string `s`, and `s[i] = c` updates it. Define a second version that reads a character, but does not update it, so that you can access `s[i]` in a function with a parameter `const String& s`.

■■ **E13.7** Implement a class `Duration` that, unlike the `Time` class, represents a time interval; for example, a duration of 1 hour and 30 minutes. Unlike `Time` objects, it makes sense to add two durations. Implement a suitable operator. Does it make sense to form the difference of two durations? If so, implement that as well.

■■ **E13.8** Add all relational operators to the `Duration` class of Exercise E13.7.

■■ **E13.9** Add input and output operators to the `Duration` class of Exercise E13.7 and Exercise E13.8.

■■ **E13.10** Change the `Time` class of Section 13.1 so that the difference of two `Time` objects is a `Duration`. Also implement adding a duration to a time.

■■ **E13.11** Implement the assignment operator and copy constructor for the `Quiz` class described in Common Error 13.1.

■■ **E13.12** Improve on the `substr` member function in Worked Example 13.2 to return a string consisting of as many characters from the original string as possible when the interval between `start` and `start + length` contains invalid positions.

■ **E13.13** Consider this class:

```
class Trace
{
public:
   Trace(string n);
   ~Trace();
private:
   string name;
};

Trace::Trace(string n) : name(n)
{
   cout << "Entering " << name << endl;
}
```

```
Trace::~Trace()
{
   cout << "Exiting " << name << endl;
}
```

Predict what the output will be in the following program, then test your prediction. Explain at what point in execution each message is generated.

```
void f(Trace t)
{
   cout << "Entering f" << endl;
}

int main()
{
   Trace tracer("main");
   f(tracer);
}
```

■■ **E13.14** Extend the class `Trace` from Exercise E13.13 with a copy constructor and an assignment operator, printing a short message in each. Use this class to demonstrate

a. the difference between initialization

```
Trace t("abc");
Trace u = t;
```

b. and assignment.

```
Trace t("abc");
Trace u("xyz");
u = t;
```

c. the fact that all constructed objects are automatically destroyed.

d. the fact that the copy constructor is invoked if an object is passed by value to a function.

e. the fact that the copy constructor is not invoked when a parameter is passed by reference.

f. the fact that the copy constructor is used to copy a return value to the caller.

■ **E13.15** Our `String` class always deletes the old character buffer and reallocates a new character buffer during assignment or in the copy constructor. This need not be done if the new value is smaller than the current value, and hence would fit in the existing buffer. Rewrite the `String` class so that each instance will maintain an integer data member indicating the size of the buffer, and reallocate a new buffer only when necessary.

■■ **E13.16** Write a class `OptionalEmployee` that stores an optional element of type `Employee*`. Here is a typical usage

```
class Department
{
private:
   . . .
   OptionalEmployee secretary;
};
```

Supply a member function

```
bool exists() const
```

that tests whether the optional element exists, a member function

```
void set(Employee* t)
```

to set it, and

```
Employee* get() const
```

to get it. As an internal representation, use a Boolean data member and a data member of type `Employee*`. Provide the necessary memory management functions to ensure that the data member is recovered when the enclosing object (for example, the instance of `Department`) is deleted.

■■ **E13.17** In Exercise E13.16, implement an `Optional` class template so that the `secretary` can be defined as `Optional<Employee>`.

■ **E13.18** Add a `swap` member function to the `Pair` class of Section 13.3.2 that swaps the first and second elements.

■ **E13.19** Add a `sort` member function to the `Pair` class of Section 13.3.2 that sorts the two elements.

■ **E13.20** Turn the `minmax` function in Section 13.3.2 into a template.

■ **E13.21** Rewrite the selection sort algorithm described in Section 12.1 as a template function, using the less than operator to compare two elements.

■ **E13.22** Rewrite the merge sort algorithm described in Section 12.4 as a template function.

■ **E13.23** Rewrite the binary search algorithm described in Section 12.6.2 as a template function.

PROGRAMMING PROJECTS

■■ **P13.1** Define a class `Money` with two integer data members, `dollars` and `cents`. Overload arithmetic operators, comparison operators, and input and output operators for your class. Should you overload the `*` and `/` operators? What argument types should they accept? Overload the `%` operator so that if `n` is a floating-point value, `n % m` yields `n` percent of the money amount `m`.

■■ **P13.2** Define a class `Complex` for complex numbers. Provide implementations for addition, subtraction, multiplication, and the stream input and output operators. Implement the compound assignment operators, such as `+=`, for each of the supported binary arithmetic operations.

■■■ **P13.3** Define a class `BigInteger` that stores arbitrarily large integers by keeping their digits in a `vector<int>`. Supply a constructor `BigInteger(string)` that reads a sequence of digits from a string. Overload the `+`, `-`, and `*` operators to add, subtract, and multiply the digit sequences. Overload the `<<` operator to send the big integer to a stream. For example,

```
BigInteger a("123456789");
BigInteger b("987654321");
cout << a * b;
```

prints `121932631112635269`.

■■■ **P13.4** Implement a class for polynomials. Store the coefficients in a vector of floating-point values. Then provide operators for addition, subtraction, multiplication, and output.

■■ **P13.5** Define a class `Set` that stores a finite set of integers. (In a set, the order of elements does not matter, and every element can occur at most once.) Supply `add` and `remove` member functions to add and remove set elements. Overload the `|` and `&` operators to compute the union and intersection of the set, and the `<<` operator to send the set contents to a stream.

■■ **P13.6** Continue Exercise P13.5 and overload the `~` operator to compute the complement of a set. That is, `~a` is the set of all integers that are not present in the set `a`. *Hint:* Add a `bool` data member to the `Set` class to keep track of whether a set is finite or has a finite complement.

■■■ **P13.7** Implement an "associative array" that uses strings instead of integer indexes for keys and stores values of type `double`. Overload the index operator (`operator[]`) to provide access as in the following example:

```
AssociativeArray prices;
prices["Toaster Oven"] = 19.95;
prices["Car Vacuum"] = 24.95;
cout << prices["Toaster oven"] << endl;
```

Hint: `operator[]` needs to return a `double&`. See Special Topic 13.3.

■■ **P13.8** Turn the `AssociativeArray` class of Exercise P13.7 into a template `AssociativeArray<K, V>`. Use a `vector<pair<K, V>>` to hold the key/value pairs. You don't have to overload the `[]` operator. Instead, provide member functions `put` and `get`:

```
AssociativeArray<string, double> prices;
prices.put("Toaster Oven", 19.95);
prices.put("Car Vacuum", 24.95);
cout << prices.get("Toaster Oven") << endl;
```

■■■ **P13.9** Improve the `Fraction` class of Worked Example 13.1 to handle values that represent infinity. When the denominator is zero, the value is $+\infty$ if the numerator is positive and $-\infty$ if the numerator is negative. Represent a "Not a Number" or NaN value as an object with numerator and denominator equal to zero. You will not have to change the implementation for multiplication and division, but adding $+\infty$ and $-\infty$, or subtracting two infinities of the same sign is NaN. You also need to adjust the comparison operators. Any comparison involving NaN is automatically false. Finally, adjust the input and output operators.

■■ **P13.10** Implement the operators in Worked Example 13.1 in the most efficient way possible. Always use reference parameters (see Programming Tip 13.1), and directly manipulate data members instead of calling other operators.

■■■ **P13.11** Implement the `move` operators described in Special Topic 13.6 for the `String` class of Worked Example 13.2.

■■ **P13.12** Implement the move assignment operator of Special Topic 13.6 by swapping the buffers of the left- and right-hand sides. Add trace messages showing the memory addresses of the buffers, as they are allocated and deleted, and the functions in which these events occur. Write a program that demonstrates the move assignment.

■■ **P13.13** Provide a `usubstr` member function for the `String` class of Worked Example 13.2 so that `str.usubstr(i, n)` produces a substring starting at the `i`th Unicode character stored in `str`, and containing `n` Unicode characters. Assume that the string is encoded as UTF-8 (see Special Topic 8.2).

■■■ **P13.14** Implement an optimized representation for short strings with the `String` class of Worked Example 13.2. Add a four-character array data member that you use instead of the buffer on the free store if the length is at most four.

■■ **P13.15** Change the implementation of the `String` class in Section 13.2.3 so that the assignment operator reuses the `buffer` array if its size is sufficient to hold the assigned string. You will need to add a `size` data member to track the size of the array which might be longer than the string length. Add trace messages showing the memory addresses and sizes of the buffers as they are allocated and deleted, and the functions in which these events occur. Write a program that demonstrates the assignment operator with strings of different sizes.

■■ **P13.16** Reimplement the quiz program of Section 10.4 using shared pointers (see Special Topic 13.7).

■■■ **P13.17** Common Error 13.1 describes the need for a copy constructor and assignment operator for a `Quiz` class that stores a vector of `Question*` pointers. You will implement those operations in this exercise. As pointed out in Special Topic 13.4, you must add a virtual destructor to the `Quiz` class. Moreover, you need a virtual `copy` function that makes a copy of each question type:

```
Question* Question::copy() const { return new Question(*this); }
Question* ChoiceQuestion::copy() const { return new ChoiceQuestion(*this); }
```

Using these functions, define the copy contructor, assignment operator, and destructor for the `Quiz` class. Add trace messages to the copy functions and the destructors of the `Question` and `ChoiceQuestion` classes and write a program that demonstrates the correct functioning of the memory managemnent functions.

■■ **P13.18** Implement a class `Person` with a data member `shared_ptr<Person> best_friend` (see Special Topic 13.7). Construct three `Person` objects on the free store so that Peter's best friend is Paul, Paul's best friend is Mary, and Mary's best friend is Peter. Add a trace message to the destructor and demonstrate that the objects are not destroyed when the program exits unless you first break the cycle by setting one of the `best_friend` members to `nullptr`.

■■ **P13.19** Define a class `Set` that stores integers in a dynamically allocated array of integers.

```
class Set
{
public:
   void add(int n);
   bool contains(int n) const;
   int get_size() const;
   . . .
private:
   int* elements;
   int size;
};
```

In a set, the order of elements does not matter, and every element can occur at most once.

Supply the `add`, `contains`, and `get_size` member functions and the "big three" memory management functions.

■■ **P13.20** Make the `Set` class from Exercise P13.19 into a template class.

■■ **P13.21** Create a template definition for a fixed-size array class. The declaration

```
Array<int, 10> data;
```

should create an array of 10 integer values. Overload the index operator `[]` to provide access to the elements. (See Special Topic 13.3.)

■■ **P13.22** Define an array class that allows the user to set a lower bound for index values that is different from zero. That is, a declaration such as

```
LBArray<int, 1955, 1975> data;
```

should create an array with 21 integer values, indexed using the integer values 1955 to 1975.

■■ **Engineering P13.23** Implement a class `Vector` that models a mathematical vector in three-dimensional space. Support adding two vectors with the + operator, multiplying a vector with a scalar, and multiplying two vectors (the dot product).

■■■ **Engineering P13.24** Using the `Vector` class from Exercise P13.23, provide operators

```
double& Vector::operator[](int index)
double Vector::operator[](int index) const
```

that can be used to access the x, y, and z-components of a vector. (The second version of the operator is needed to access components of a `const Vector&` parameter.)

■■ **Engineering P13.25** Implement the cross product for the `Vector` class of Exercise P13.23. Pick a suitable operator, or, if none exists, provide a function instead.

■■ **Engineering P13.26** Implement a class `Matrix` that models a 3×3 matrix. Support adding and multiplying matrices, multiplying a matrix with a scalar, and multiplying a matrix with a `Vector` (see Exercise P13.23).

■■■ **Engineering P13.27** Using the `Matrix` class from Exercise P13.26, overload the bracket operator so that one can access a matrix element as `m[i][j]`. *Hint:* Add a class `MatrixRow`, have `Matrix::operator[]` return a `MatrixRow`, and `MatrixRow::operator[]` return a `double&`.

■■■ **Engineering P13.28** Complete the implementation of the `Matrix` template of Special Topic 13.8.

WORKED EXAMPLE 13.1

A Fraction Class

Problem Statement Imagine you wish to implement a new data type that represents a fraction (a ratio of two integer values). Fractional numbers can be declared with both a numerator and denominator, or simply a numerator:

```
Fraction x; // Represents 0/1
Fraction y(7); // Represents 7/1
Fraction z(3, 4); // Represents 3/4
```

You want fractions to act just like other numbers. That means you should be able to add and subtract fractional values, compare them to other fractions, perform input and output of fractions, and so on:

```
if (y < z) { x = z - y; }
else { x = y + z; }
cout << "Value is " << x << "\n";
```

We define our `Fraction` class as follows:

```
class Fraction
{
public:
   Fraction(int n, int d);
   Fraction(int n);
   Fraction();
   Fraction operator+(Fraction other) const;
   Fraction operator*(Fraction other) const;
   bool operator<(Fraction other) const;
   . . .
private:
   int numerator;
   int denominator;
};
```

We implement the `operator` functions as member functions that need to access the internal representation. Let us start with addition and multiplication.

Recall that

$$a/b + c/d = (a \cdot d + b \cdot c)/(b \cdot d)$$

and

$$a/b \cdot c/d = (a \cdot c)/(b \cdot d)$$

For example, $1/2 + 1/3 = (1 \cdot 3 + 2 \cdot 1)/(2 \cdot 3) = 5/6$. However, sometimes, you need to reduce the result to lowest common terms. For example, $1/2 + 3/2 = (1 \cdot 2 + 2 \cdot 3)/(2 \cdot 2) = 8/4$.

The best place for that reduction is in the constructor:

```
Fraction::Fraction(int n, int d)
{
   int g = gcd(n, d); // The greatest common divisor
   if (g == 0)
   {
      numerator = 0;
      denominator = 1;
   }
   else if (d > 0)
   {
```

```
      numerator = n / g;
      denominator = d / g;
   }
   else
   {
      numerator = -n / g;
      denominator = -d / g;
   }
}
```

The greatest common divisor is computed with Euclid's algorithm:

```
int gcd(int a, int b)
{
   int m = abs(a);
   int n = abs(b);
   while (n != 0)
   {
      int r = m % n;
      m = n;
      n = r;
   }
   return m;
}
```

Now we can implement the addition of fractions:

```
Fraction Fraction::operator+(Fraction other) const
{
   return Fraction(numerator * other.denominator
      + denominator * other.numerator,
      denominator * other.denominator);
}
```

You may wonder what happens if the denominator is zero. It is possible to interpret such objects as "infinity"—see Exercise P13.9.

The multiplication is similar:

```
Fraction Fraction::operator*(Fraction other) const
{
   return Fraction(numerator * other.numerator,
      denominator * other.denominator);
}
```

This operator multiplies two `Fraction` objects, but it can also be used to multiply a `Fraction` with an integer such as:

```
Fraction y = x * 3; // x.operator*(Fraction(3))
```

As described in Special Topic 13.2, the value 3 is automatically converted to a `Fraction`, using the constructor with a single parameter.

However, if the integer is the left operand, the conversion is not applied:

```
Fraction y = 3 * x; // 3 is not converted to a Fraction
```

We need to define an `operator*` that can process an `int` and a `Fraction`. It can't be a member function of `Fraction`, so we define it as a regular function:

```
Fraction operator*(int n, Fraction f)
{
   return f * n;
}
```

There are two subtraction operators: a **unary** version that denotes the negative of a fraction as `-f`, and a **binary** version that subtracts two fractions as `f - g`. The unary version is implemented as an `operator-` with one argument:

```
Fraction operator-(Fraction f)
{
   return f * -1;
}
```

Here is the binary version:

```
Fraction operator-(Fraction f, Fraction g)
{
   return f + g * -1;
}
```

In these cases, we chose to express operators in terms of existing ones. This makes the code easy to understand. It is also possible to implement each operator directly in terms of the numerators and the denominators of the operands—see Exercise P13.10.

For division, we would like to use the same strategy as for subtraction. However, there is no suitable operator for computing the multiplicative inverse, so we implement a member function and use it in the definition of the division operator.

```
Fraction Fraction::inverse() const
{
   return Fraction(denominator, numerator);
}

Fraction operator/(Fraction f, Fraction g)
{
   return f * g.inverse();
}
```

Next, we turn to relational operators. Because we ensure in the constructor that denominators are positive, we know that $a/b < c/d$ when $a \cdot d < b \cdot c$:

```
bool Fraction::operator<(Fraction other) const
{
   return numerator * other.denominator < denominator * other.numerator;
}
```

Now we can define all other relational operators in terms of the < operator:

```
bool operator>(Fraction f, Fraction g)
{
   return g < f;
}

bool operator>=(Fraction f, Fraction g)
{
   return !(f < g);
}

bool operator<=(Fraction f, Fraction g)
{
   return !(g < f);
}

bool operator!=(Fraction f, Fraction g)
{
   return f < g || g < f;
}

bool operator==(Fraction f, Fraction g)
{
   return !(f < g || g < f);
}
```

Finally, here are the implementations of the output and input operators. Because we have no member functions to access the numerator and denominator of a fraction, and we need access to the internals of the class, we call helper functions, `print` and `read`.

```
ostream& operator<<(ostream& out, Fraction f)
{
   f.print(out);
   return out;
}

void Fraction::print(ostream& out) const
{
   out << numerator << "/" << denominator;
}

istream& operator>>(istream& in, Fraction& f)
{
   f.read(in);
   return in;
}

void Fraction::read(istream& in)
{
   char separator; // For reading / character
   in >> numerator >> separator >> denominator;
}
```

Here is the complete program. Note how the non-member operator functions are declared in the header file `fraction.h` and defined in the source file `fraction.cpp`.

worked_example_1/fraction.h

```
#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>

using namespace std;

class Fraction
{
public:
   /**
      Constructs a fraction n / d.
      @param n the numerator
      @param d the denominator
   */
   Fraction(int n, int d);
   /**
      Constructs a fraction n / 1.
      @param n the numerator
   */
   Fraction(int n);
   /**
      Constructs the fraction 0 / 1.
   */
   Fraction();

   /**
      Comoutes the inverse of this fraction.
      @return the inverse (denominator / numerator)
```

```
30     */
31     Fraction inverse() const;
32     /**
33        Prints this fraction to a stream.
34        @param out the output stream
35     */
36     void print(ostream& out) const;
37     /**
38        Reads a fraction from a stream and stores the input
39        in this fraction.
40        @param in the input stream
41     */
42     void read(istream& in);
43
44     Fraction operator+(Fraction other) const;
45     Fraction operator*(Fraction other) const;
46     bool operator<(Fraction other) const;
47  private:
48     int numerator;
49     int denominator;
50  };
51
52  Fraction operator-(Fraction f);
53  Fraction operator-(Fraction f, Fraction g);
54  Fraction operator*(int n, Fraction f);
55  Fraction operator/(Fraction f, Fraction g);
56  bool operator>(Fraction f, Fraction g);
57  bool operator<=(Fraction f, Fraction g);
58  bool operator>=(Fraction f, Fraction g);
59  bool operator==(Fraction f, Fraction g);
60  bool operator!=(Fraction f, Fraction g);
61  ostream& operator<<(ostream& out, Fraction f);
62  istream& operator>>(istream& in, Fraction& a);
63
64  #endif
```

worked_example_1/fraction.cpp

```
1   #include <cstdlib>
2   #include "fraction.h"
3
4   using namespace std;
5
6   int gcd(int a, int b)
7   {
8      int m = abs(a);
9      int n = abs(b);
10     while (n != 0)
11     {
12        int r = m % n;
13        m = n;
14        n = r;
15     }
16     return m;
17  }
18
19  Fraction::Fraction()
20  {
21     numerator = 0;
22     denominator = 1;
```

```
23 }
24
25 Fraction::Fraction(int n)
26 {
27    numerator = n;
28    denominator = 1;
29 }
30
31 Fraction::Fraction(int n, int d)
32 {
33    int g = gcd(n, d); // The greatest common divisor
34    if (g == 0)
35    {
36       numerator = 0;
37       denominator = 1;
38    }
39    else if (d > 0)
40    {
41       numerator = n / g;
42       denominator = d / g;
43    }
44    else
45    {
46       numerator = -n / g;
47       denominator = -d / g;
48    }
49 }
50
51 Fraction Fraction::operator+(Fraction other) const
52 {
53    return Fraction(numerator * other.denominator
54       + denominator * other.numerator,
55       denominator * other.denominator);
56 }
57
58 Fraction Fraction::operator*(Fraction other) const
59 {
60    return Fraction(numerator * other.numerator,
61       denominator * other.denominator);
62 }
63
64 Fraction operator*(int n, Fraction f)
65 {
66    return f * n;
67 }
68
69 Fraction operator-(Fraction f)
70 {
71    return f * -1;
72 }
73
74 Fraction operator-(Fraction f, Fraction g)
75 {
76    return f + g * -1;
77 }
78
79 Fraction Fraction::inverse() const
80 {
81    return Fraction(denominator, numerator);
82 }
```

```
 83
 84 Fraction operator/(Fraction f, Fraction g)
 85 {
 86    return f * g.inverse();
 87 }
 88
 89 bool Fraction::operator<(Fraction other) const
 90 {
 91    return numerator * other.denominator < denominator * other.numerator;
 92 }
 93
 94 bool operator>(Fraction f, Fraction g)
 95 {
 96    return g < f;
 97 }
 98
 99 bool operator>=(Fraction f, Fraction g)
100 {
101    return !(f < g);
102 }
103
104 bool operator<=(Fraction f, Fraction g)
105 {
106    return !(g < f);
107 }
108
109 bool operator!=(Fraction f, Fraction g)
110 {
111    return f < g || g < f;
112 }
113
114 bool operator==(Fraction f, Fraction g)
115 {
116    return !(f < g || g < f);
117 }
118
119 ostream& operator<<(ostream& out, Fraction f)
120 {
121    f.print(out);
122    return out;
123 }
124
125 void Fraction::print(ostream& out) const
126 {
127    out << numerator << "/" << denominator;
128 }
129
130 istream& operator>>(istream& in, Fraction& f)
131 {
132    f.read(in);
133    return in;
134 }
135
136 void Fraction::read(istream& in)
137 {
138    char separator; // For reading / character
139    in >> numerator >> separator >> denominator;
140 }
```

worked_example_1/fractiondemo.cpp

```
#include <iostream>
#include "fraction.h"

using namespace std;

int main()
{
   Fraction f(1, 2);
   Fraction g(1, 3);
   cout << "f + g: " << f + g << endl;
   cout << "f * g: " << f * g << endl;
   cout << "f / g: " << f / g << endl;
   cout << "Enter a fraction: ";
   Fraction h;
   cin >> h;
   if (f == h)
   {
      cout << "Your input equals 1/2" << endl;
   }
   else
   {
      cout << "f - h: " << f - h << endl;
   }
   return 0;
}
```

WORKED EXAMPLE 13.2

Tracing Memory Management of Strings

Problem Statement In this Worked Example, we will complete the implementation of the `String` class and add tracing messages to the memory management functions. We will then analyze the calls to assignment operators, copy constructors, and destructors.

In the preceding sections, you saw how to implement the memory management functions of a `String` class that manages a buffer of `char` values. In the buffer, we do not use the `'\0'` terminator. Also, when the string is empty, we don't allocate any buffer at all but set the length to zero and the buffer to `nullptr`.

First, let us implement the string member functions that we always use: `length`, `substr`, and concatenation with the `+` operator.

The `length` member function is trivial:

```
int String::length() const
{
   return len;
}
```

Here is an implementation of the `substr` member function. For simplicity, we don't do error checking. (See Exercise E13.12 for an improvement.)

```
String String::substr(int start, int length) const
{
   String result;
   if (length > 0)
   {
      result.len = length;
      result.buffer = new char[length];
      for (int i = 0; i < length; i++)
      {
         result.buffer[i] = buffer[start + i];
      }
   }
   return result;
}
```

Here we take advantage of the fact that a member function of the `String` class has access to the private data of any `String` object. In particular, the `substr` function manipulates the `result` object, attaching a buffer that holds the substring characters.

The `operator+` member function uses the same approach.

```
String String::operator+(const String& other) const
{
   String result;
   result.len = len + other.len;
   result.buffer = new char[result.len];
   for (int i = 0; i < len; i++)
   {
      result.buffer[i] = buffer[i];
   }
   for (int i = 0; i < other.len; i++)
   {
      result.buffer[len + i] = other.buffer[i];
   }
   return result;
}
```

Note that if either of the strings is empty, its buffer is `nullptr`, but the loop for copying its characters is never entered (because `len` is 0).

The `operator==` member function compares two strings.

```
bool String::operator==(const String& other) const
{
   if (len != other.len) { return false; }
   for (int i = 0; i < len; i++)
   {
      if (buffer[i] != other.buffer[i]) { return false; }
   }
   return true;
}
```

For printing a string, we need to access the characters in the buffer. Unfortunately, we cannot make `operator<<` into a `String` member function because the left parameter is an `ostream`, not a `String` (see Section 13.1.4). Instead, we implement it as a regular function that calls a `print` member function:

```
ostream& operator<<(ostream& out, const String& str)
{
   str.print(out);
   return out;
}

void String::print(ostream& out) const
{
   for (int i = 0; i < len; i++)
   {
      out << buffer[i];
   }
}
```

Note that the string is printed a character at a time because it is not zero terminated.

These are all the member functions that we need for our demonstration program. Let us return to the memory management functions. Each of them prints out a diagnostic message:

```
String::String()
{
   cout << "Constructing empty string" << endl;
   . . .
}

String::String(const char s[])
{
   cout << "Constructing \"" << s << "\"" << endl;
   . . .
}

String::String(const String& other)
{
   cout << "Copying \"" << other << "\"" << endl;
   . . .
}

String& String::operator=(const String& other)
{
   cout << "Assigning \"" << other << "\" to \""
      << *this << "\"" << endl;
   . . .
}
```

```
String::~String()
{
   cout << "Destroying \"" << *this << "\"" << endl;
   . . .
}
```

In our test program, we demonstrate the difference between copy construction and assignment. Then we show what happens when an object is passed to a function. We use the following function that counts the number of spaces in a string. This function uses the `length`, `substr`, and `operator==` member functions:

```
int spaces(String str)
{
   int count = 0;
   String space(" ");
   for (int i = 0; i < str.length(); i++)
   {
      if (str.substr(i, 1) == space) { count++; }
   }
   return count;
}
```

worked_example_2/stringdemo.cpp

```
#include <iostream>
#include "string.h"

using namespace std;

int spaces(String str)
{
   int count = 0;
   String space(" ");
   for (int i = 0; i < str.length(); i++)
   {
      if (str.substr(i, 1) == space) { count++; }
   }
   return count;
}

int main()
{
   String name("Mary Ann");
   String name2 = name; // Copy constructor
   String name3("Fred");
   name3 = name; // operator=

   int result = spaces(name);
   cout << "Spaces: " << result << endl;

   return 0;
} // str and space are destroyed here
```

Here is the output of running the program:

```
Constructing "Mary Ann"
Copying "Mary Ann"
Constructing "Fred"
Assigning "Mary Ann" to "Fred"
Copying "Mary Ann"
Constructing " "
```

```
Constructing empty string
Destroying "M"
Constructing empty string
Destroying "a"
Constructing empty string
Destroying "r"
Constructing empty string
Destroying "y"
Constructing empty string
Destroying " "
Constructing empty string
Destroying "A"
Constructing empty string
Destroying "n"
Constructing empty string
Destroying "n"
Destroying " "
Destroying "Mary Ann"
Spaces: 1
Destroying "Mary Ann"
Destroying "Mary Ann"
Destroying "Mary Ann"
```

That's a lot of activity for such a short program. Let's see what happens.

The first part of the program constructs three strings. The first and third string use the `String(const char s[])` constructor, and the second a copy constructor.

```
String name("Mary Ann");
String name2 = name; // Copy constructor
String name3("Fred");
```

That is shown in the trace messages.

```
Constructing "Mary Ann"
Copying "Mary Ann"
Constructing "Fred"
```

In the next line, we assign to an existing string:

```
name3 = name; // operator=
```

Because `name3` already exists, `operator=` is called. The trace output shows this:

```
Assigning "Mary Ann" to "Fred"
```

Now look at the end of the output. The `main` function has three objects of type `String`. All three of them have now been set to `"Mary Ann"`. In the last three lines of output, you can see how they are all destroyed:

```
Destroying "Mary Ann"
Destroying "Mary Ann"
Destroying "Mary Ann"
```

The remainder of the output shows what happens when calling the `spaces` function.

When the function is called, the `str` parameter is constructed with a copy constructor:

```
Copying "Mary Ann"
```

Then the `space` variable is initialized.

```
Constructing " "
```

Now, a loop visits each substring of length 1:

```
for (int i = 0; i < str.length(); i++)
{
   if (str.substr(i, 1) == space) { count++; }
}
```

The trace messages show the construction and destruction of many short strings:

```
Constructing empty string
Destroying "M"
Constructing empty string
Destroying "a"
Constructing empty string
Destroying "r"
. . .
```

The `substr` member function produces these strings using the default constructor and attaches a buffer with the substring. Of course, all these objects need to be destroyed.

Finally, when the `spaces` function returns, it destroys its local variables, namely the `space` string and the `str` parameter variable:

```
Destroying " "
Destroying "Mary Ann"
```

The eagle-eyed reader may have noticed something surprising about the substrings of length 1. These substrings are produced in the `substr` function and then copied into the `spaces` function. However, there is no trace of that copy in the program output.

The compiler carried out an optimization in which the return value was passed as an additional reference parameter. You can turn off this optimization (with g++, by using the `-fno-elide-constructors` flag), and then you see the additional copies.

There are cases when the compiler cannot carry out this optimization. A more generally useful strategy is to use *move operations*—see Special Topic 13.6.

As you can see, there is a signifant amount of copying that happens behind the scenes. If we are concerned only about correctness, that is not a concern. But programmers who want their code to be as efficient as possible need to understand when these copies happen, and what can be done to minimize the cost.

Here is the code for the `String` class with the trace messages used in this example.

worked_example_2/string.h

```
#ifndef STRING_H
#define STRING_H

#include <iostream>

using namespace std;

class String
{
public:
   String();
   String(const char s[]);
   String(const String& other);
   String& operator=(const String& other);
   ~String();

   String operator+(const String& other) const;
   bool operator==(const String& other) const;

   /**
      Yields the length of this string.
      @return the length
   */
   int length() const;
   /**
      Yields a substring of this string.
      @param start the position of the first character to copy
```

```
28          @param length the number of characters to copy
29          @return the substring
30       */
31       String substr(int start, int length) const;
32       /**
33          Prints this fraction to a stream.
34          @param out the output stream
35       */
36       void print(ostream& out) const;
37    private:
38       char* buffer;
39       int len;
40    };
41
42    ostream& operator<<(ostream& out, const String& str);
43    bool operator!=(const String& s, const String& t);
44
45    #endif
```

worked_example_2/string.cpp

```
 1  #include "string.h"
 2  #include <cstring>
 3
 4  String::String()
 5  {
 6     cout << "Constructing empty string" << endl;
 7     len = 0;
 8     buffer = nullptr;
 9  }
10
11  String::String(const char s[])
12  {
13     cout << "Constructing \"" << s << "\"" << endl;
14     len = strlen(s);
15     if (len > 0)
16     {
17        buffer = new char[len];
18        for (int i = 0; i < len; i++)
19        {
20           buffer[i] = s[i];
21        }
22     }
23     else
24     {
25        buffer = nullptr;
26     }
27  }
28
29  String::String(const String& other)
30  {
31     cout << "Copying \"" << other << "\"" << endl;
32     len = other.len;
33     if (len > 0)
34     {
35        buffer = new char[len];
36        for (int i = 0; i < len; i++)
37        {
38           buffer[i] = other.buffer[i];
39        }
```

```
   }
   else
   {
      buffer = nullptr;
   }
}

String& String::operator=(const String& other)
{
   cout << "Assigning \"" << other << "\" to \""
      << *this << "\"" << endl;
   if (this != &other)
   {
      delete[] buffer;
      len = other.len;
      if (len > 0)
      {
         buffer = new char[len];
         for (int i = 0; i < len; i++)
         {
            buffer[i] = other.buffer[i];
         }
      }
      else
      {
         buffer = nullptr;
      }
   }
   return *this;
}

String::~String()
{
   cout << "Destroying \"" << *this << "\"" << endl;
   delete[] buffer;
}

int String::length() const
{
   return len;
}

String String::substr(int start, int length) const
{
   String result;
   if (length > 0)
   {
      result.len = length;
      result.buffer = new char[length];
      for (int i = 0; i < length; i++)
      {
         result.buffer[i] = buffer[start + i];
      }
   }
   return result;
}

String String::operator+(const String& other) const
{
```

```
   String result;
   result.len = len + other.len;
   result.buffer = new char[result.len];
   for (int i = 0; i < len; i++)
   {
      result.buffer[i] = buffer[i];
   }
   for (int i = 0; i < other.len; i++)
   {
      result.buffer[len + i] = other.buffer[i];
   }
   return result;
}

bool String::operator==(const String& other) const
{
   if (len != other.len) { return false; }
   for (int i = 0; i < len; i++)
   {
      if (buffer[i] != other.buffer[i]) { return false; }
   }
   return true;
}

void String::print(ostream& out) const
{
   for (int i = 0; i < len; i++)
   {
      out << buffer[i];
   }
}

ostream& operator<<(ostream& out, const String& str)
{
   str.print(out);
   return out;
}

bool operator!=(const String& s, const String& t)
{
   return !(s == t);
}
```

CHAPTER 14

LINKED LISTS, STACKS, AND QUEUES

© andrea laurita/iStockphoto.

CHAPTER GOALS

To become familiar with the list, stack, and queue data types

To understand the implementation of linked lists

To understand the efficiency of vector and list operations

To study applications of stacks and queues

CHAPTER CONTENTS

In this chapter, we introduce a new data structure, the *linked list*. A linked list is made up of nodes, each of which is connected to the neighboring nodes. You will learn how to use lists and the related stack and queue types. You will study the implementation of linked lists and analyze when linked lists are more efficient than arrays or vectors.

14.1 Using Linked Lists

A **linked list** is a data structure for collecting a sequence of objects, such that addition and removal of elements in the middle of the sequence is efficient.

To understand the need for such a data structure, imagine a program that maintains a vector of employee records, sorted by the last name of the employees. When a new employee is hired, an object needs to be inserted into the vector. Unless the company happens to hire employees in dictionary order, it is likely that a new employee object needs to be inserted into the middle of the vector. In that case, many other objects must be moved toward the end. Conversely, if an employee leaves the company, the hole in the sequence needs to be closed by moving all objects that came after it. Moving a large number of objects can involve a substantial amount of computer time. We would like to structure the data in a way that minimizes this cost.

A linked list consists of a number of nodes, each of which has a pointer to the neighboring nodes.

Rather than storing the data in a single block of memory, a linked list uses a different strategy. Each value is stored in its own memory block, together with the locations of the neighboring blocks in the sequence (see Figure 1).

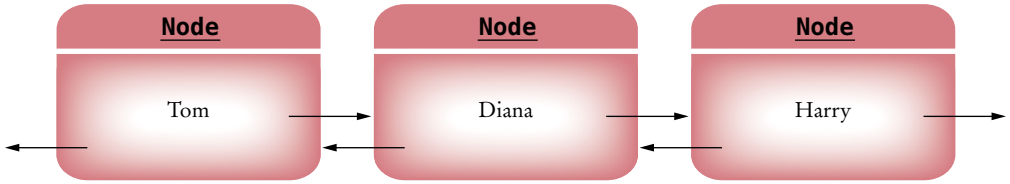


Figure 1
A Linked List

Adding and removing elements in the middle of a linked list is efficient.

It is now an easy matter to add another value to the sequence (see Figure 2), or to remove a value from the sequence (Figure 3), without moving the others.

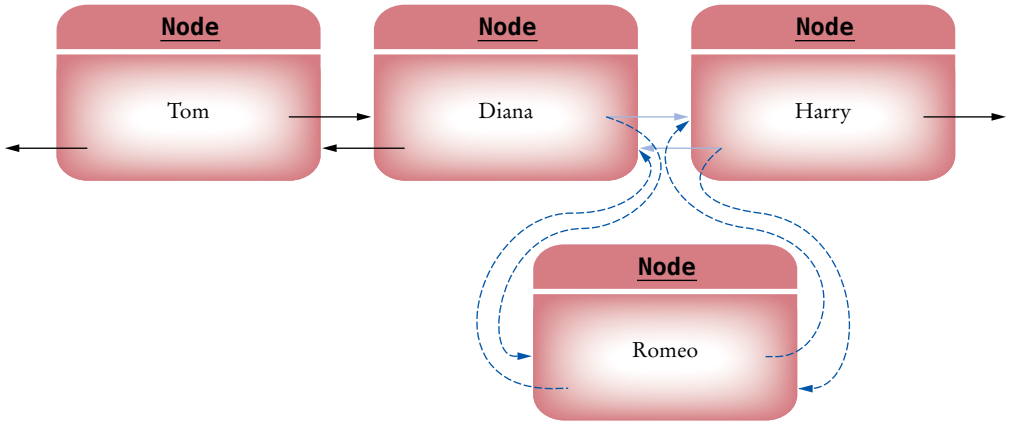


Figure 2
Adding a Node to a Linked List

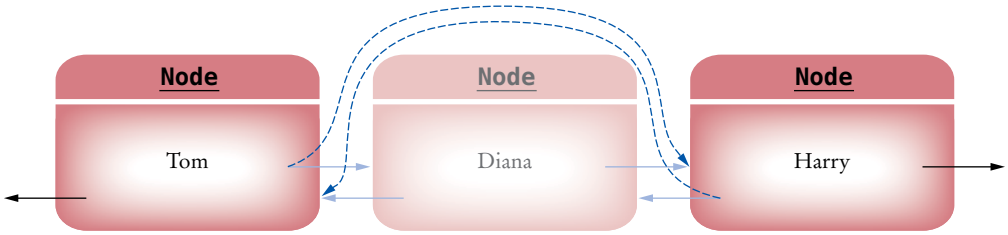


Figure 3 Removing a Node from a Linked List

What's the catch? Linked lists allow speedy insertion and removal, but element access can be slow. For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term **random access** is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence. For example, a binary search requires random access, whereas a linear search requires only sequential access.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

The standard C++ library has an implementation of the linked list container structure. In this section, you will learn how to use the standard linked list structure. Later you will look "under the hood" and find out how to implement linked lists. (The linked list of the standard C++ library has links going in both directions. Such a list is often called a **doubly-linked list**. A **singly-linked list** lacks the links to the predecessor elements.)

Just like `vector`, the standard `list` is a *template:* You can declare lists for different types. For example, to make a list of strings, define an object of type `list<string>`. Then you can use the `push_back` function to add strings to the end of the list. The following code segment defines a list of strings, `names`, and adds three strings to it:

```
list<string> names;
names.push_back("Tom");
names.push_back("Diana");
names.push_back("Harry");
```

This code looks exactly like the code that you would use to build a vector of strings. There is, however, one major difference. Suppose you want to access the last element in the list. You cannot directly refer to `names[2]`. Because the values are not stored in one contiguous block in memory, there is no immediate way to access the third element. Instead, you must visit each element in turn, starting at the beginning of the list and then proceeding to the next element.

You can inspect and edit a linked list with an iterator. An iterator points to a node in a linked list.

To visit an element, you use a *list iterator*. An **iterator** marks a *position* in the list. To get an iterator that marks the beginning position in the list, you define an iterator variable, then call the `begin` function of the `list` class to get the beginning position:

```
list<string>::iterator pos = names.begin();
```

If you iterate over a constant list (such as a `const list<string>&` parameter), the `begin` and `end` member functions yield a `list<string>::const_iterator`. This is the same as an iterator, except that you cannot modify the list using it.

Some programmers find type names such as `list<string>::iterator` and `list<string>::const_iterator` too cumbersome and prefer to use the `auto` reserved word (see Special Topic 2.3):

```
auto pos = names.begin();
```

To move the iterator to the next position, use the `++` operator:

```
pos++;
```

You can also move the iterator backward with the `--` operator:

```
pos--;
```

You use the `*` operator to find the value that is stored in the position marked by the iterator:

```
string value = *pos;
```

You have to be careful to distinguish between the iterator `pos`, which represents a position in the list, and the value `*pos`, which represents the value that is stored in the list. For example, if you change `*pos`, then you update the contents in the list:

```
*pos = "Romeo"; // The list value at the position is changed
```

If you change `pos`, then you merely change the current position.

```
pos = names.begin(); // The position is again at the beginning of the list
```

To insert another string before the iterator position, use the `insert` function:

```
names.insert(pos, "Romeo");
```

The `insert` function inserts the new element *before* the iterator position, rather than after it. This convention makes it easy to insert a new element before the first value of the list:

```
pos = names.begin();
names.insert(pos, "Romeo");
```

That raises the question of how you insert a value after the end of the list. Each list has an *end position* that does not correspond to any value in the list but that points past the list's end. The `end` function returns that position:

```
pos = names.end(); // Points past the end of the list
names.insert(pos, "Juliet"); // Insert past the end of the list
```

It is an error to compute

```
string value = *names.end(); // Error
```

The `end` position does not point to any value, so you cannot look up the value at that position. This error is equivalent to the error of accessing `v[10]` in a vector with ten elements.

The `end` position has another useful purpose: it is the stopping point for traversing the list. The following code iterates over all elements of the list and prints them out:

```
pos = names.begin();
while (pos != names.end())
{
   cout << *pos << endl;
   pos++;
}
```

The traversal can be described more concisely with a `for` loop:

```
for (pos = names.begin(); pos != names.end(); pos++)
{
```

```
   cout << *pos << endl;
}
```

Of course, this looks very similar to the typical `for` loop for traversing an array:

```
for (i = 0; i < size; i++)
{
   cout << a[i] << endl;
}
```

There is a convenient shortcut for this loop, called the range based `for` loop, which was discussed in Special Topic 6.5:

```
for (const string& element : names)
{
   cout << element << " ";
}
```

To remove an element from a list, you move an iterator to the position that you want to remove, then call the `erase` function. The following code erases the second element of the list:

```
pos = names.begin();
pos++;
pos = names.erase(pos);
```

The iterator that is passed to `erase` now points to a storage location that has been deleted, and can no longer be used. Therefore, the `erase` function returns an iterator that points to the element after the one that has been erased. It is a good idea to immediately replace the original iterator with the one that the `erase` function returns.

At the end of the preceding code sample, `pos` points to the element that was previously the third element and is now the second element.

In addition to the `push_back` member function, there is a `push_front` member function that adds an element at the front. The `pop_front` and `pop_back` member functions remove the first and last elements.

Table 1 Working with Linked Lists

`list<string> names;`	An empty list.
`names.push_back("Harry");`	Adds an element to the end of the list.
`names.push_front("Sally");`	Adds an element to the front of the list.
`int n = names.size();`	The current size of the list.
`list<string>::iterator iter = names.begin();`	Gets an iterator pointing to the first element of the list.
`iter++;`	Moves the iterator to the next element.
`string name = *iter;`	Yields the element to which the iterator points.
`if (pos != list.end())`	Tests whether the iterator still points to an element.
`names.insert(iter, "Fred");`	Inserts an element before the iterator.
`iter = names.erase(iter);`	Removes the element to which the iterator points. `iter` now points to the element following the removed element.
`names.pop_front(); names.pop_back();`	Removes the first and last elements of the list.

Here is a short example program that adds elements to a list, inserts and erases list elements, and finally traverses the resulting list:

sec01/listdemo.cpp

```
#include <string>
#include <list>
#include <iostream>

using namespace std;

int main()
{
   list<string> names;

   names.push_back("Tom");
   names.push_back("Diana");
   names.push_back("Harry");
   names.push_back("Juliet");

   // Add a value in fourth place

   list<string>::iterator pos = names.begin();
   // Or auto pos = names.begin();
   pos++;
   pos++;
   pos++;

   names.insert(pos, "Romeo");

   // Remove the value in second place

   pos = names.begin();
   pos++;

   names.erase(pos);

   // Print all values

   for (pos = names.begin(); pos != names.end(); pos++)
   {
      cout << *pos << " ";
   }
   cout << endl;

   // Print all values with range-based loop

   for (const string& element : names)
   {
      cout << element << " ";
   }
   cout << endl;

   return 0;
}
```

Program Run

```
Tom Harry Romeo Juliet
Tom Harry Romeo Juliet
```

14.2 Implementing Linked Lists

The previous section showed you how to put linked lists to use. However, because the implementation of the `list` class is hidden from you, you had to take it on faith that the list values are really stored in separate memory blocks. We will now walk through an implementation of the list, node, and iterator classes.

For simplicity, we will implement linked lists of strings. To implement the linked list class in C++ that can hold values of arbitrary types, you need to use templates. To implement iterators that behave exactly like the ones in the C++ library, you also need to know about operator overloading. Finally, you need to add memory management. Worked Example 14.1 shows how to add these features.

14.2.1 The Classes for Lists, Nodes, and Iterators

When implementing a linked list, we need to define list, node, and iterator classes.

The `list` class of the standard library defines many useful member functions. For simplicity, we will only study the implementation of the most useful ones: `push_back`, `insert`, `erase`, and the iterator operations. We call our class `List`, with an uppercase `L`, to differentiate it from the standard `list` class template.

We will implement a doubly-linked list, as the standard C++ library does. At first glance, a singly-linked list might seem simpler. However, insertion and removal in the middle of the list is easier to implement with a doubly-linked list (see Exercise P14.2).

A linked list stores each value in a separate object, called a *node*. A node object holds a value, together with pointers to the previous and next nodes:

```
class Node
{
public:
   Node(string element);
private:
   string data;
   Node* previous;
   Node* next;
friend class List;
friend class Iterator;
};
```

A list node contains pointers to the next and previous nodes.

Note the `friend` declarations. They indicate that the member functions of the `List` and `Iterator` classes are allowed to inspect and modify the data members of the `Node` class, which we will write presently.

A class should not grant friendship to another class lightly, because it breaks the privacy protection. In this case, it makes sense, though, because the list and iterator functions do all the necessary work and the node class is just an artifact of the implementation that is invisible to the users of the list class. Note that no code other than the member functions of the list and iterator classes can access the data members of the node class, so the data integrity is still guaranteed.

A list object holds the locations of the first and last nodes in the list:

```
class List
{
public:
   List();
   void push_back(string element);
```

```
   void insert(Iterator iter, string element);
   Iterator erase(Iterator iter);
   Iterator begin();
   Iterator end();
private:
   Node* first;
   Node* last;
friend class Iterator;
};
```

A list object contains pointers to the first and last nodes.

If the list is empty, then the `first` and `last` pointers are `nullptr`. Note that a list object stores no data; it just knows where to find the node objects that store the list contents.

An iterator contains a pointer to the current node, and to the list that contains it.

Finally, an *iterator* denotes a position in the list. It holds a pointer to the node that denotes its current position, and a pointer to the list that created it. Our iterator class uses member functions `get`, `next`, `previous`, and `equals` instead of operators `*`, `++`, `--`, and `==`. For example, we will call `pos.next()` instead of `pos++`.

```
class Iterator
{
public:
   Iterator();
   string get() const;
   void next();
   void previous();
   bool equals(Iterator other) const;
private:
   Node* position;
   List* container;
friend class List;
};
```

If the iterator points past the end of the list, then the `position` pointer is `nullptr`. In that case, the `previous` member function uses the `container` pointer to move the iterator back from the past-the-end position to the last element of the list. (This is only one possible choice for implementing the past-the-end position. Another choice would be to store an actual dummy node at the end of the list. Some implementations of the standard `list` class do just that.)

14.2.2 Implementing Iterators

Iterators are created by the `begin` and `end` member functions of the `List` class. The `begin` function creates an iterator whose `position` pointer points to the first node in the list. The `end` function creates an iterator whose `position` pointer is `nullptr`.

```
Iterator List::begin()
{
   Iterator iter;
   iter.position = first;
   iter.container = this;
   return iter;
}

Iterator List::end()
{
```

```
   Iterator iter;
   iter.position = nullptr;
   iter.container = this;
   return iter;
}
```

The `next` function (which is the equivalent of the `++` operator) advances the iterator to the next position. This is a very typical operation in a linked list; let us study it in detail. The `position` pointer points to the current node in the list. That node has a data member `next`. Because `position` is a node pointer, you use the `->` operator to access the data member `next`:

```
position->next
```

That `next` data member is itself a pointer, pointing to the next node in the linked list (see Figure 4). To make `position` point to that next node, write

```
position = position->next;
```

Thus, the `next` function is simply:

```
void Iterator::next()
{
   position = position->next;
}
```

Note that you can evaluate `position->next` only if `position` is not `nullptr`, because it is an error to dereference a `nullptr` pointer. That is, it is illegal to advance the iterator once it is in the past-the-end position. Our implementation does not check for errors such as this one; neither does the implementation of the standard C++ library. (In C++, it is the responsibility of the programmer using a data structure to use it correctly.

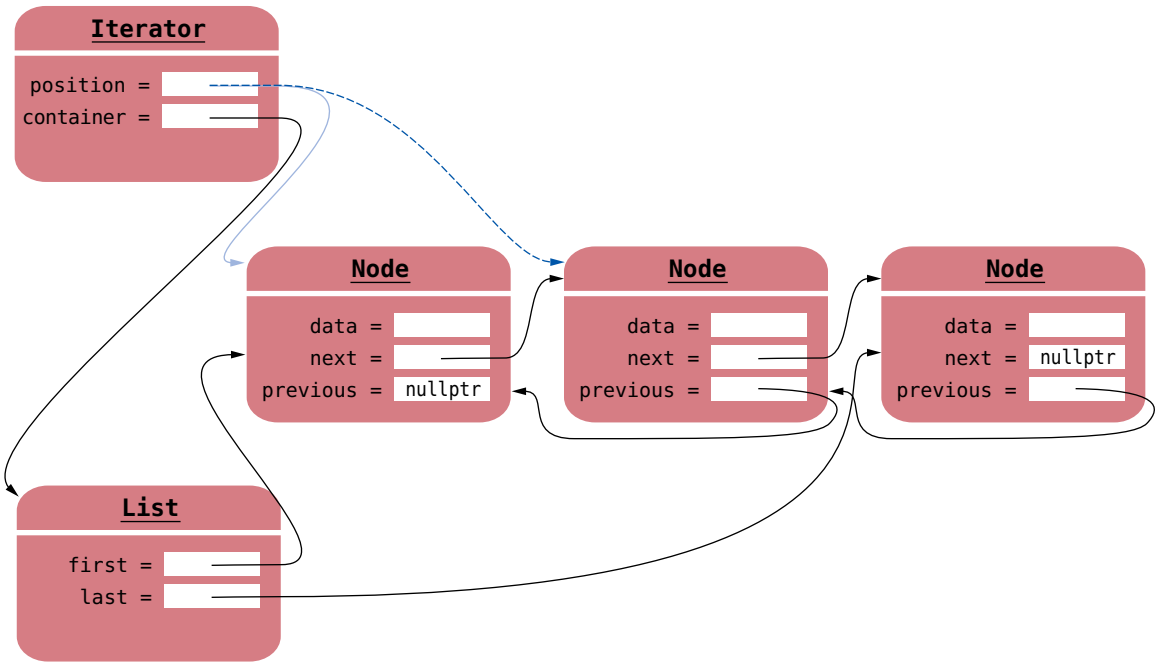


Figure 4 Advancing an Iterator

The previous function (which is the equivalent of the -- operator) is a bit more complex. In the ordinary case, you move the position backward with the instruction

```
position = position->previous;
```

However, if the iterator is currently past the end, then you must make it point to the last element in the list.

```
void Iterator::previous()
{
   if (position == nullptr)
   {
      position = container->last;
   }
   else
   {
      position = position->previous;
   }
}
```

The get function (which is the equivalent of the * operator) simply returns the data value of the node to which position points—that is, position->data. It is illegal to call get if the iterator points past the end of the list:

```
string Iterator::get() const
{
   return position->data;
}
```

Finally, the equals function (which is the equivalent of the == operator) compares two position pointers:

```
bool Iterator::equals(Iterator other) const
{
  return position == other.position;
}
```

14.2.3 Implementing Insertion and Removal

In the last section you saw how to implement the iterators that traverse an existing list. Now you will see how to build up lists by adding and removing elements, one step at a time.

First, we will implement the push_back function. It appends an element to the end of the list (see Figure 5). Make a new node:

```
Node* new_node = new Node(element);
```

List nodes are allocated on the free store, using the new operator.

This new node must be integrated into the list after the node to which the last pointer points. That is, the data member next of the last node (which is currently nullptr) must be updated to new_node. Also, the data member previous of the new node must point to what used to be the last node:

```
new_node->previous = last; ❶
last->next = new_node; ❷
```

Finally, you must update the last pointer to reflect that the new node is now the last node in the list:

```
last = new_node; ❸
```

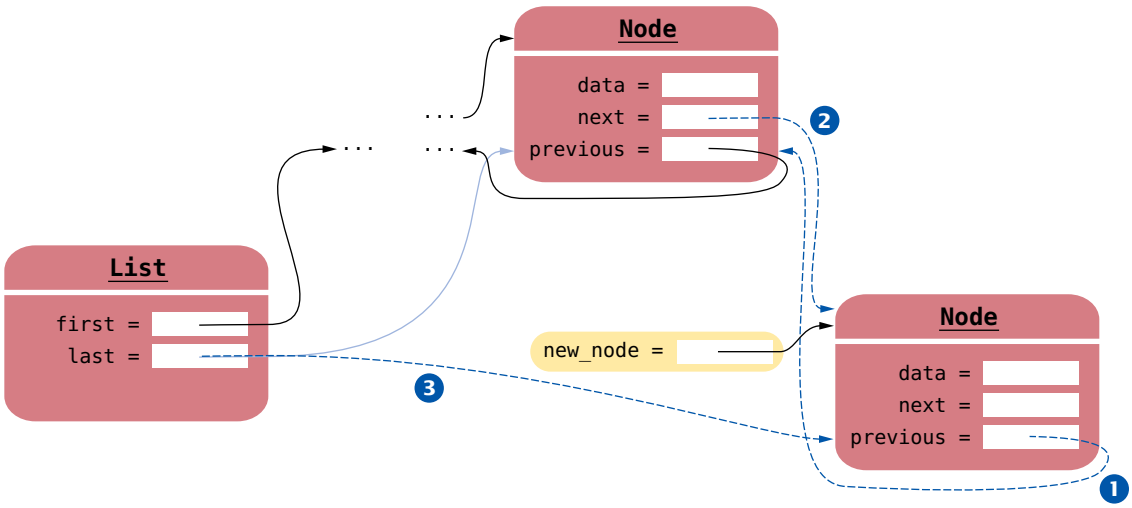


Figure 5 Appending a Node to the End of a Linked List

However, there is a special case when `last` is `nullptr`, which can happen only when the list is empty. After the call to `push_back`, the list has a single node—namely, `new_node`. In that case, both `first` and `last` must be set to `new_node`:

```
void List::push_back(string element)
{
   Node* new_node = new Node(element);
   if (last == nullptr) // List is empty
   {
      first = new_node;
      last = new_node;
   }
   else
   {
      new_node->previous = last;
      last->next = new_node;
      last = new_node;
   }
}
```

Inserting an element in the middle of a linked list is a little more difficult, because the node pointers in the *two* nodes surrounding the new node need to be updated. The function declaration is

```
void List::insert(Iterator iter, string element)
```

That is, a new node containing `element` is inserted before `iter.position` (see Figure 6).

Give names to the surrounding nodes. Let `before` be the node before the insertion location, and let `after` be the node after that. That is,

```
Node* after = iter.position;
Node* before = after->previous;
```

What happens if `after` is `nullptr`? After all, it is illegal to apply `->` to `nullptr`. In this situation, you are inserting past the end of the list. Simply call `push_back` to handle that case separately. Otherwise, you need to insert `new_node` between `before` and `after`:

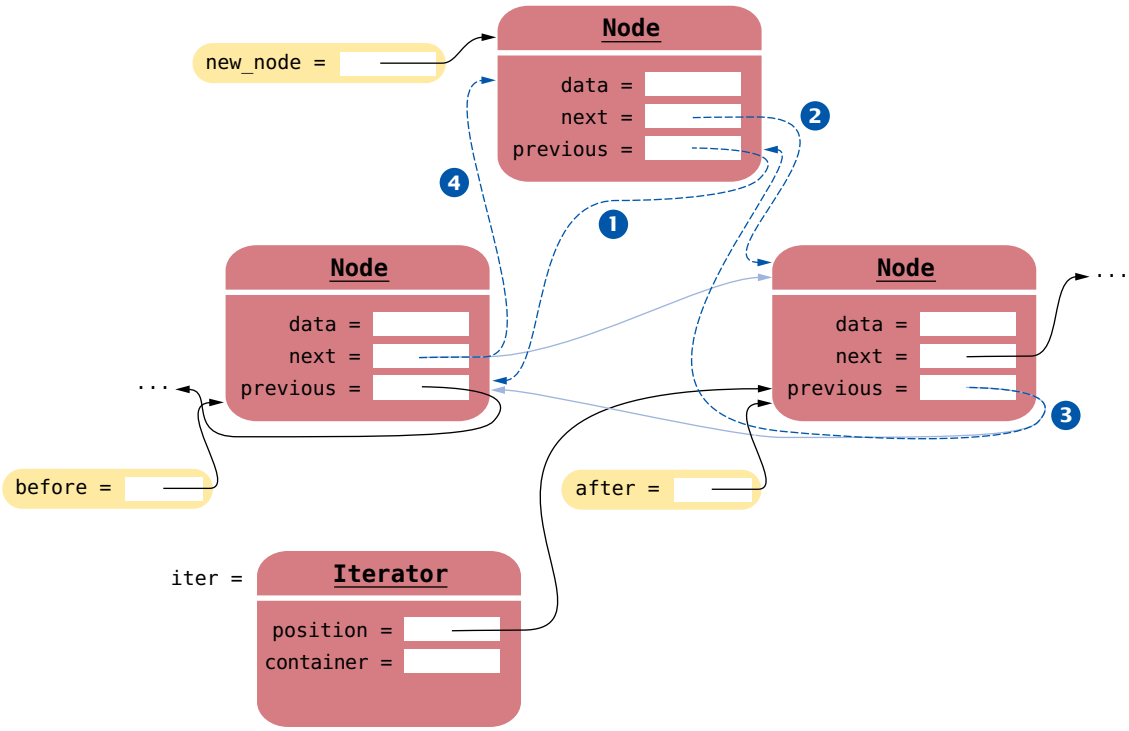


Figure 6 Inserting a Node into a Linked List

```
new_node->previous = before; 1
new_node->next = after; 2
```

You must also update the nodes from `before` and `after` to point to the new node:

```
after->previous = new_node; 3
before->next = new_node;   // If before != nullptr 4
```

However, you must be careful. You know that `after` is not `nullptr`, but it is possible that `before` is `nullptr`. In that case, you are inserting at the beginning of the list and need to adjust `first`:

```
if (before = nullptr) // Insert at beginning
{
   first = new_node;
}
else
{
   before->next = new_node;
}
```

Here is the complete code for the `insert` function:

```
void List::insert(Iterator iter, string element)
{
   if (iter.position == nullptr)
   {
      push_back(element);
      return;
   }
```

```
      Node* after = iter.position;
      Node* before = after->previous;
      Node* new_node = new Node(element);
      new_node->previous = before;
      new_node->next = after;
      after->previous = new_node;
      if (before == nullptr) // Insert at beginning
      {
         first = new_node;
      }
      else
      {
         before->next = new_node;
      }
   }
```

Finally, look at the implementation of the `erase` function:

```
Iterator List::erase(Iterator iter)
```

You want to remove the node to which `iter.position` points. As before, give names to the node to be removed, the node before it, and the node after it:

```
Node* remove = iter.position;
Node* before = remove->previous;
Node* after = remove->next;
```

You need to update the `next` and `previous` pointers of the `before` and `after` nodes to bypass the node that is to be removed (see Figure 7).

```
before->next = after;  // If before != nullptr 1
after->previous = before;  // If after != nullptr 2
```

However, as before, you need to cope with the possibility that `before`, `after`, or both are `nullptr`. If `before` is `nullptr`, you are erasing the first element in the list. It has no predecessor to update, but you must change the `first` pointer of the list. Conversely,

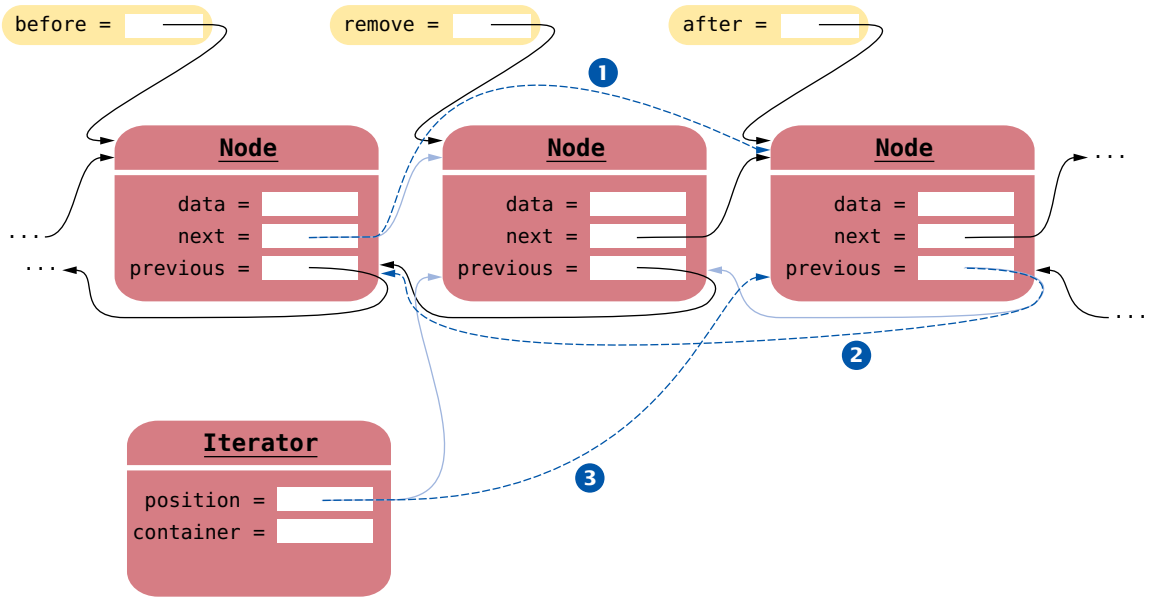


Figure 7 Removing a Node from a Linked List

if `after` is `nullptr`, you are erasing the last element of the list and must update the `last` pointer of the list:

```
if (remove == first)
{
   first = after;
}
else
{
   before->next = after;
}
if (remove == last)
{
   last = before;
}
else
{
   after->previous = before;
}
```

You must adjust the iterator position so it no longer points to the removed element.

```
iter.position = after; ❸
```

When a list node is erased, it is recycled to the free store with the `delete` operator.

Finally, you must remember to recycle the removed node:

```
delete remove;
```

Here is the complete `erase` function. Note that the function returns an iterator to the element following the erased one:

```
Iterator List::erase(Iterator iter)
{
   Node* remove = iter.position;
   Node* before = remove->previous;
   Node* after = remove->next;
   if (remove == first)
   {
      first = after;
   }
   else
   {
      before->next = after;
   }
   if (remove == last)
   {
      last = before;
   }
   else
   {
      after->previous = before;
   }
   delete remove;
   Iterator r;
   r.position = after;
   r.container = this;
   return r;
}
```

Implementing these linked list operations is somewhat complex. It is also error-prone. If you make a mistake and misroute some of the pointers, you can get subtle errors. For example, if you make a mistake with a `previous` pointer, you may never

Implementing operations that modify a linked list is challenging—you need to make sure that you update all node pointers correctly.

notice it until you traverse the list backwards. If a node has been deleted, then that same storage area may later be reallocated for a different purpose, and if you have kept a pointer to it, following that invalid node pointer will lead to disaster. You must exercise special care when implementing any operations that manipulate the node pointers directly.

Here is a program that puts our linked list to use and demonstrates the `insert` and `erase` operations:

sec02/list.h

```
#ifndef LIST_H
#define LIST_H

#include <string>

using namespace std;

class List;
class Iterator;

class Node
{
public:
   /**
      Constructs a node with a given data value.
      @param element the data to store in this node
   */
   Node(string element);
private:
   string data;
   Node* previous;
   Node* next;
friend class List;
friend class Iterator;
};

class List
{
public:
   /**
      Constructs an empty list.
   */
   List();
   /**
      Appends an element to the list.
      @param element the value to append
   */
   void push_back(string element);
   /**
      Inserts an element into the list.
      @param iter the position before which to insert
      @param element the value to insert
   */
   void insert(Iterator iter, string element);
   /**
      Removes an element from the list.
      @param iter the position to remove
```

```
48          @return an iterator pointing to the element after the
49          erased element
50       */
51       Iterator erase(Iterator iter);
52       /**
53          Gets the beginning position of the list.
54          @return an iterator pointing to the beginning of the list
55       */
56       Iterator begin();
57       /**
58          Gets the past-the-end position of the list.
59          @return an iterator pointing past the end of the list
60       */
61       Iterator end();
62    private:
63       Node* first;
64       Node* last;
65    friend class Iterator;
66    };
67
68    class Iterator
69    {
70    public:
71       /**
72          Constructs an iterator that does not point into any list.
73       */
74       Iterator();
75       /**
76          Looks up the value at a position.
77          @return the value of the node to which the iterator points
78       */
79       string get() const;
80       /**
81          Advances the iterator to the next node.
82       */
83       void next();
84       /**
85          Moves the iterator to the previous node.
86       */
87       void previous();
88       /**
89          Compares two iterators
90          @param other the iterator to compare with this iterator
91          @return true if this iterator and other are equal
92       */
93       bool equals(Iterator other) const;
94    private:
95       Node* position;
96       List* container;
97    friend class List;
98    };
99
100   #endif
```

sec02/list.cpp

```
1  #include <string>
2  #include "list.h"
3
```

```
4  using namespace std;
5
6  Node::Node(string element)
7  {
8     data = element;
9     previous = nullptr;
10    next = nullptr;
11 }
12
13 List::List()
14 {
15    first = nullptr;
16    last = nullptr;
17 }
18
19 void List::push_back(string element)
20 {
21    Node* new_node = new Node(element);
22    if (last == nullptr) // List is empty
23    {
24       first = new_node;
25       last = new_node;
26    }
27    else
28    {
29       new_node->previous = last;
30       last->next = new_node;
31       last = new_node;
32    }
33 }
34
35 void List::insert(Iterator iter, string element)
36 {
37    if (iter.position == nullptr)
38    {
39       push_back(element);
40       return;
41    }
42
43    Node* after = iter.position;
44    Node* before = after->previous;
45    Node* new_node = new Node(element);
46    new_node->previous = before;
47    new_node->next = after;
48    after->previous = new_node;
49    if (before == nullptr) // Insert at beginning
50    {
51       first = new_node;
52    }
53    else
54    {
55       before->next = new_node;
56    }
57 }
58
59 Iterator List::erase(Iterator iter)
60 {
61    Node* remove = iter.position;
62    Node* before = remove->previous;
63    Node* after = remove->next;
```

```
64       if (remove == first)
65       {
66          first = after;
67       }
68       else
69       {
70          before->next = after;
71       }
72       if (remove == last)
73       {
74          last = before;
75       }
76       else
77       {
78          after->previous = before;
79       }
80       delete remove;
81       Iterator r;
82       r.position = after;
83       r.container = this;
84       return r;
85    }
86
87    Iterator List::begin()
88    {
89       Iterator iter;
90       iter.position = first;
91       iter.container = this;
92       return iter;
93    }
94
95    Iterator List::end()
96    {
97       Iterator iter;
98       iter.position = nullptr;
99       iter.container = this;
100      return iter;
101   }
102
103   Iterator::Iterator()
104   {
105      position = nullptr;
106      container = nullptr;
107   }
108
109   string Iterator::get() const
110   {
111      return position->data;
112   }
113
114   void Iterator::next()
115   {
116      position = position->next;
117   }
118
119   void Iterator::previous()
120   {
121      if (position == nullptr)
122      {
123         position = container->last;
```

```
   }
   else
   {
      position = position->previous;
   }
}

bool Iterator::equals(Iterator other) const
{
   return position == other.position;
}
```

sec02/listdemo.cpp

```
#include <string>
#include <iostream>
#include "list.h"

using namespace std;

int main()
{
   List names;

   names.push_back("Tom");
   names.push_back("Diana");
   names.push_back("Harry");
   names.push_back("Juliet");

   // Add a value in fourth place

   Iterator pos = names.begin();
   pos.next();
   pos.next();
   pos.next();

   names.insert(pos, "Romeo");

   // Remove the value in second place

   pos = names.begin();
   pos.next();

   names.erase(pos);

   // Print all values

   for (pos = names.begin(); !pos.equals(names.end()); pos.next())
   {
      cout << pos.get() << endl;
   }

   return 0;
}
```

Program Run

```
Tom Harry Romeo Juliet
```

WORKED EXAMPLE 14.1

Implementing a Linked List Template

Learn how to create a linked list template class. See your E-Text or visit `wiley.com/go/bclo3`.

14.3 The Efficiency of List, Array, and Vector Operations

In this section, we will formally analyze how efficient the fundamental operations on linked lists, arrays, and vectors are. We will consider these operations:

- Getting the kth element
- Adding and removing an element at a given position (an iterator or index)
- Adding and removing an element at the end

Locating the kth element is an $O(k)$ operation for linked lists.

To get the kth element of a linked list, you start at the beginning of the list and advance the iterator k times. Suppose it takes an amount of time T to advance the iterator once. This quantity is independent of the iterator position—advancing an iterator does some checking and then it follows the `next` pointer. Therefore, advancing the iterator to the kth element consumes kT time. Therefore, locating the kth element is an $O(k)$ operation.

Locating an element is an $O(1)$ operation for arrays and vectors.

To get the kth element of an array, we use an expression such as `a[k]`. This is executed in a constant amount of time that is independent of k. We say that accessing an array element takes $O(1)$ time.

To analyze the situation for vectors, we need to peek under the hood and see how the `vector` class is implemented.

© Kris Hanke/iStockphoto.

To get to the kth node of a linked list, one must skip over the preceding nodes.

A vector maintains a pointer to an array of elements termed the **buffer**. An integer data member, called the **capacity**, is the maximum number of elements that can be stored in the current buffer. The buffer is usually larger than is necessary to hold the current elements in the container. The *size* is the number of elements actually being held by the container. Because vectors use zero-based indexing, the size can also be interpreted as the first free location in the array. Figure 8 shows vector internals. The size is stored in the data member `len` because the class has a member function named `size`.

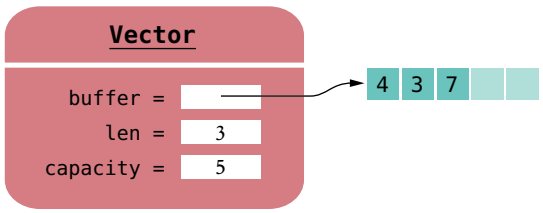


Figure 8 Internal Data Members Maintained by Vector

The kth element is accessed through the expression `buffer[k]`, which is done in constant or $O(1)$ time.

Here is a simplified implementation of the standard `vector<int>`, which we call `Vector`. We provide a member function `at` instead of the `[]` operator to access the elements:

```
class Vector
{
public:
   . . .
   int& at(int k);
private:
   int* buffer;
   int len;
   int capacity;
};

int& Vector::at(int k)
{
   return buffer[k];
}
```

Next, consider the task of adding an element in the middle of a list, array, or vector. For a linked list, we assume that we already have an iterator to the insertion location. It might have taken some time to get there, but we are now concerned with the cost of insertion after the position has been established.

Adding an element in a linked list is an $O(1)$ operation.

As you saw in Figure 6, you add an element by modifying the `previous` and `next` pointers of the new node and the surrounding nodes. This operation takes a constant number of steps, independent of the position. The same holds for removing an element. We conclude that list insertion and removal are $O(1)$ operations.

Adding an element in the middle of an array or vector of size n is an $O(n)$ operation.

For arrays and vectors, the situation is less rosy. To insert an element at position k, the elements with higher index values need to move (see Figure 9). How many elements are affected? For simplicity, we will assume that insertions happen at random locations. On average, each insertion moves $n/2$ elements, where n is the size of the array or vector.

The same argument holds for removing an element. On average, $n/2$ elements need to be moved. Therefore, we say that array and vector insertion and removal are $O(n)$ operations.

There is one situation where adding an element to an array or vector isn't so costly: when the insertion happens at the end. The `push_back` member function carries out that operation.

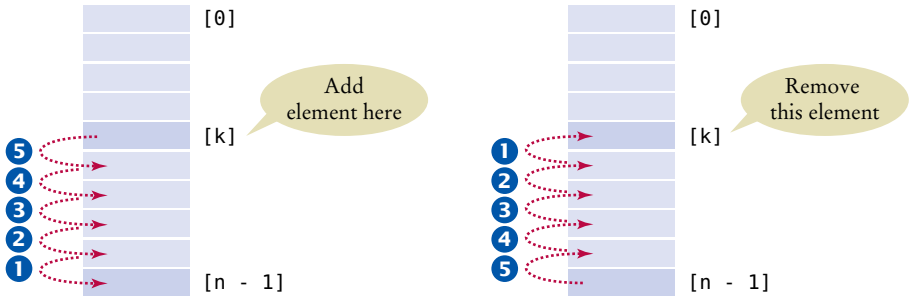


Figure 9 Inserting and Removing Vector Elements

Adding an element to the end of an array is an $O(1)$ operation.

If the size of the vector is less than the capacity, the new element is simply moved into place and the size is incremented, as shown in Figure 10. This is an $O(1)$ operation.

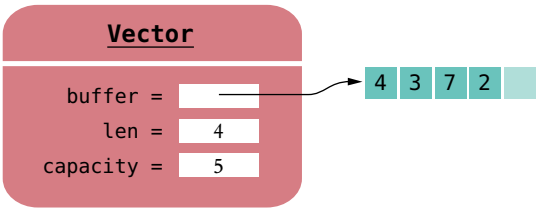


Figure 10 Vector After push_back

If, however, the size is equal to the capacity, it means that no more space is available. With an array, there is nothing to be done—the element cannot be inserted. Vectors, on the other hand, can grow. In order to make new space, a new and larger buffer is allocated. This new buffer is typically twice the size of the current buffer (see Figure 11). The existing elements are then copied into the new buffer, the old buffer is deleted, and insertion takes place as before. Reallocation is an $O(n)$ operation because all elements need to be copied to the new buffer.

© Craig Dingle/iStockphoto.

When a vector is completely full, we must move the contents to a larger array.

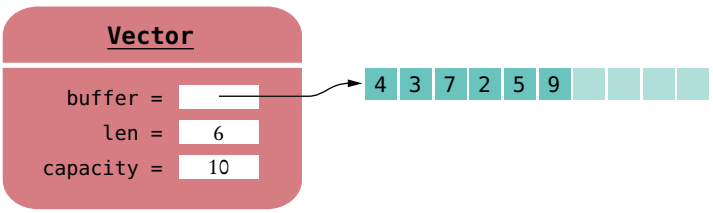


Figure 11 Vector After a Buffer Reallocation

Here is the implementation of the member function that grows the buffer if necessary:

```
void Vector::grow_if_necessary()
{
   if (len == capacity)
   {
      capacity = 2 * capacity;
      int* larger_buffer = new int[capacity];
      for (int i = 0; i < len; i++)
      {
         larger_buffer[i] = buffer[i];
      }
      delete[] buffer; // The old buffer is no longer needed
      buffer = larger_buffer;
   }
}
```

If we carefully analyze the total cost of a sequence of `push_back` operations, it turns out that these reallocations are not as expensive as they first appear. The key observation is that reallocation does not happen very often. Suppose we start with a vector of capacity 5 and double the size with each reallocation. We must reallocate when the buffer reaches sizes 5, 10, 20, 40, 80, 160, 320, 640, 1280, and so on.

Let us assume that one insertion without reallocation takes time T_1 and that reallocation of k elements takes time kT_2. What is the cost of 1,280 `push_back` operations? Of course, we pay $1280 \cdot T_1$ for the insertions. The reallocation cost is

$$\begin{aligned} 5T_2 + 10T_2 + 20T_2 + 40T_2 + \cdots + 1280T_2 &= (1 + 2 + 4 + \cdots + 256) \cdot 5 \cdot T_2 \\ &= 511 \cdot 5 \cdot T_2 \\ &< 512 \cdot 5 \cdot T_2 \\ &= 1280 \cdot 2 \cdot T_2 \end{aligned}$$

Therefore, the total cost is a bit less than

$$1280 \cdot (T_1 + 2T_2)$$

In general, the total cost of n `push_back` operations is less than $n \cdot (T_1 + 2T_2)$. Because the second factor is a constant, we conclude that n `push_back` operations take $O(n)$ time.

We know that it isn't quite true that an individual `push_back` operation takes $O(1)$ time. After all, occasionally a `push_back` is unlucky and must reallocate the buffer. But if the cost of that reallocation is distributed over the preceding `push_back` operations, then the surcharge for each of them is still a constant amount.

An element can be added to the end of a vector in amortized $O(1)$ time.

We say that `push_back` takes *amortized* $O(1)$ time, which is written as $O(1)+$. (Accountants say that a cost is amortized when it is distributed over multiple periods.)

In our implementation, we do not shrink the array when elements are removed. However, it turns out that you can (occasionally) shrink the array and still have $O(1)+$ performance for removing the last element (see Exercise R14.24).

Finally, we note that the `push_back` operation for a linked list takes $O(1)$ time, provided that the linked list implementation maintains a pointer to the last element of the list. Table 2 summarizes the execution times that we discussed in this section.

Table 2 Execution Times for Container Operations

Operation	Array/Vector	Linked List
Add/remove element at end	$O(1)+$	$O(1)$
Add/remove element in the middle (at a given index/iterator position)	$O(n)$	$O(1)$
Get kth element	$O(1)$	$O(k)$

EXAMPLE CODE See sec03 of your companion code for a program that demonstrates the `Vector` implementation.

14.4 Stacks and Queues

In this section, you will consider two common data types that allow insertion and removal of items at the ends only, not in the middle.

A stack is a container of items with "last in, first out" retrieval.

A **stack** lets you insert and remove elements at one end only, traditionally called the *top* of the stack. To visualize a stack, think of a stack of pancakes.

New items can be added (or pushed) to the top of the stack. Items are removed (or popped) from the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, also called *last in*, *first out* or *LIFO* order. For example, if you push strings `"Tom"`, `"Diana"`, and `"Harry"` onto a stack, and then pop them one by one, then you will first see `"Harry"`, then `"Diana"`, and finally `"Tom"`.

© John Madden/iStockphoto.

The last pancake that has been added to this stack will be the first one that is consumed.

To obtain a stack in the standard C++ library, you use the `stack` template:

```
stack<string> s;
s.push("Tom");
s.push("Diana");
s.push("Harry");
while (s.size() > 0)
{
   cout << s.top() << endl;
   s.pop();
}
```

The `pop` member function removes the top of the stack without returning a value. If you want to obtain the value before popping it, first call `top`, then `pop`.

Table 3 Working with Stacks

`stack<int> s;`	Makes an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `{ 1, 2, 3 }`, with the top of the stack at the end.

`int n = s.top();`	Sets `n` to 3, the most recently pushed element.
`s.pop();`	Removes the top; `s` is now `{ 1, 2 }`.
`n = s.size();`	Sets `n` to the current size of the stack.

A queue is a container of items with "first in, first out" retrieval.

A **queue** lets you add items to one end of the queue (the *back* or *tail*) and remove them from the other end of the queue (the *front* or *head*). To visualize a queue, simply think of people lining up. People join the back of the queue and wait until they have reached the front of the queue. Queues store items in a *first in*, *first out* or *FIFO* fashion. Items are removed in the same order in which they have been added.

The standard `queue` template implements a queue in C++. As with stacks, the addition and removal operations are called `push` and `pop`. The `front` member function yields

the first element of the queue (that is, the next one to be removed). The `back` member function yields the element that was most recently added. You cannot access any other elements of the queue. Here is an example of using a queue:

```
queue<string> q;
q.push("Tom");
q.push("Diana");
q.push("Harry");
while (q.size() > 0)
{
   cout << q.front() << endl;
   q.pop();
}
```

Jack Hollingsworth/Photodisc/Getty Images.

A Queue

Table 4 Working with Queues

`queue<int> q;`	Makes an empty queue.

`q.push(1);`
`q.push(2);`
`q.push(3);`
	Adds to the back of the queue; `q` is `{ 1 }`, then `{ 1, 2 }`, then `{ 1, 2, 3 }`. (The back of the queue is at the right.)

`int n = q.front();`	Sets `n` to 1, the first pushed element.
`q.pop();`	Removes the front element; `q` is now `{ 2, 3 }`.
`n = s.size();`	Sets `n` to the current size of the queue.

In the standard C++ library, the `push` and `pop` functions of the `stack` and `queue` classes have $O(1)$ efficiency.

Figure 12 contrasts the behaviors of the stack and queue data types.

There are many uses of stacks and queues in computer science. For example, consider an algorithm that attempts to find a path through a maze. When the algorithm encounters an intersection, it pushes the location on the stack, and then it explores the first branch. If that branch is a dead end, it returns to the location at the top of the stack and explores the next untried branch. If all branches are dead ends, it pops the location off the stack, revealing a previously encountered intersection. Another

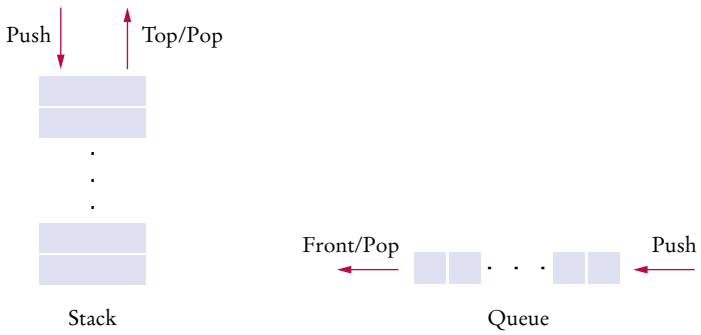


Figure 12 Stack and Queue Behavior

The Undo key pops commands off a stack so that the last command is the first to be undone.

important example is the run-time stack that a processor keeps to organize the variables of nested functions. Whenever a new function is called, its parameters and local variables are pushed onto a stack. When the function exits, they are popped off again. This stack makes recursive function calls possible.

As an example for the use of a queue, consider a printer that receives requests to print documents from multiple applications. If each of the applications sends printing data to the printer at the same time, then the printouts will be garbled. Instead, each application places all data to be sent to the printer into a file and inserts that file into the *print queue*. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the first in, first out rule, which is a fair arrangement for users of the shared printer.

The following sample program demonstrates the first-in, first-out order of a queue and the last-in, first-out order of a stack.

sec04/fifolifo.cpp

```
#include <iostream>
#include <string>
#include <queue>
#include <stack>

using namespace std;

int main()
{
   cout << "FIFO order:" << endl;

   queue<string> q;
   q.push("Tom");
   q.push("Diana");
   q.push("Harry");

   stack<string> s;
   while (q.size() > 0)
   {
      string name = q.front();
      q.pop();
      cout << name << endl;
      s.push(name);
   }

   cout << "LIFO order:" << endl;

   while (s.size() > 0)
   {
      cout << s.top() << endl;
      s.pop();
   }

   return 0;
}
```

Program Run

```
FIFO order:
Tom
Diana
Harry
```

```
LIFO order:
Harry
Diana
Tom
```

14.5 Implementing Stacks and Queues

The stack and queue data types are very simple. Elements are added and retrieved, either in *last-in, first-out* order or in *first-in, first-out* order. Stacks and queues are examples of **abstract data types**. We only specify how the operations must behave, not how they are implemented. In the following sections, we will study several implementations of stacks and queues and determine how efficient they are.

14.5.1 Stacks as Linked Lists

A stack can be implemented as a linked list, adding and removing elements at the front.

Let us first implement a stack as a sequence of nodes. New elements are added (or "pushed") to an end of the sequence, and they are removed (or "popped") from the same end.

Which end? It is up to us to choose, and we will make the least expensive choice: to add and remove elements at the front (see Figure 13).

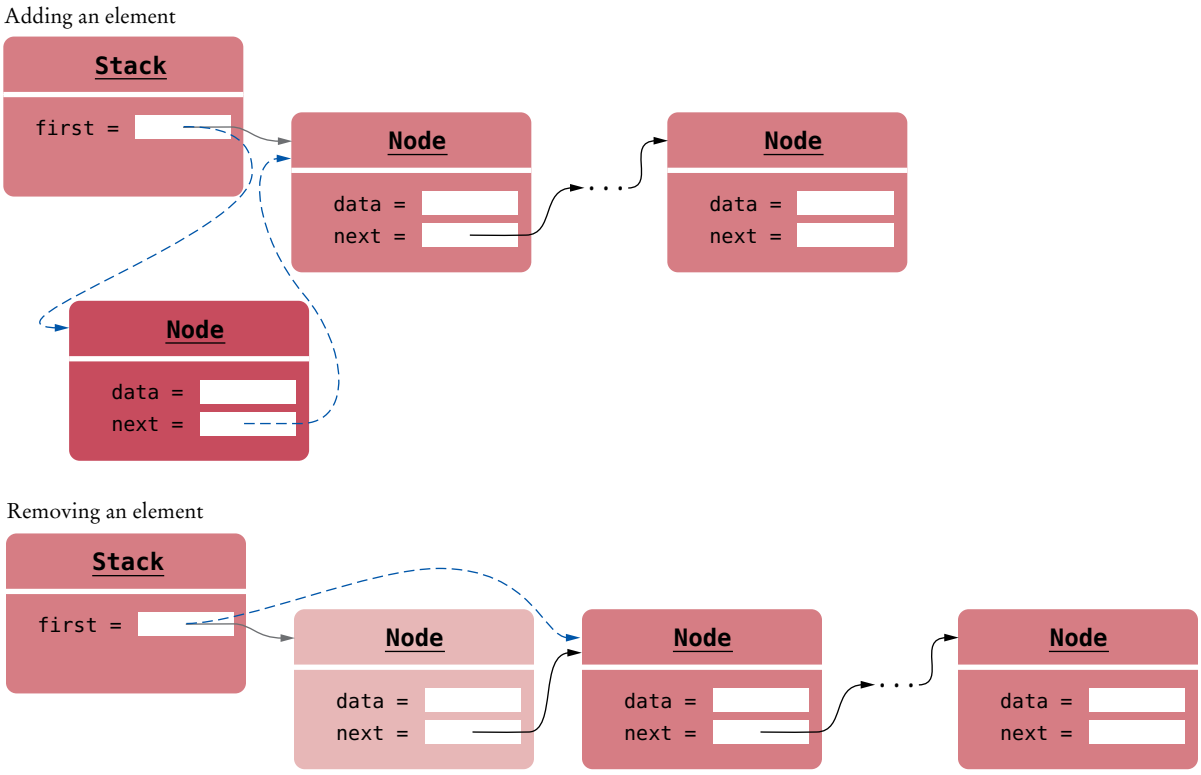


Figure 13 Push and Pop for a Stack Implemented as a Linked List

The push and pop operations run in constant time or $O(1)$ because each operation involves a fixed set of actions, independent of the size of the stack.

Here is the complete implementation. In Exercises P14.18 and P14.22, you are asked to implement this class as a template, and to provide a copy constructor, assignment operator, and destructor.

sec05_01/stack.h

```
#ifndef STACK_H
#define STACK_H

#include <string>

using namespace std;

/**
   An implementation of a stack as a sequence of nodes.
*/
class Stack
{
public:
   /**
      Constructs an empty stack.
   */
   Stack();

   /**
      Adds an element to the top of the stack.
      @param element the element to add
   */
   void push(string element);

   /**
      Yields the element on the top of the stack.
      @return the top element
   */
   string top() const;

   /**
      Removes the element from the top of the stack.
   */
   void pop();

   /**
      Yields the number of elements in this stack.
      @return the size
   */
   int size() const;

private:
   class Node
   {
   public:
      string data;
      Node* next;
   };

   Node* first;
   int len;
};
```

```
53
54 #endif
```

sec05_01/stack.cpp

```
1  #include "stack.h"
2
3  Stack::Stack()
4  {
5     first = nullptr;
6     len = 0;
7  }
8
9  void Stack::push(string element)
10 {
11    Node* new_node = new Node;
12    new_node->data = element;
13    new_node->next = first;
14    first = new_node;
15    len++;
16 }
17
18 string Stack::top() const
19 {
20    return first->data;
21 }
22
23 void Stack::pop()
24 {
25    Node* to_delete = first;
26    first = first->next;
27    delete to_delete;
28    len--;
29 }
30
31 int Stack::size() const
32 {
33    return len;
34 }
```

sec05_01/stackdemo.cpp

```
1  #include <iostream>
2  #include "stack.h"
3
4  using namespace std;
5
6  int main()
7  {
8     Stack s;
9     s.push("Tom");
10    s.push("Diana");
11    s.push("Harry");
12    while (s.size() > 0)
13    {
14       cout << s.top() << endl;
15       s.pop();
16    }
17
18    return 0;
19 }
```

Program Run

```
Harry
Diana
Tom
```

14.5.2 Stacks as Arrays

When implementing a stack as an array, add and remove elements at the back.

In the preceding section, you saw how a stack was implemented as a sequence of nodes. In this section, we will instead store the values in an array, thus saving the storage of the node references.

Again, it is up to us at which end of the array we place new elements. This time, it is better to add and remove elements at the back of the array (see Figure 14).

Of course, an array may eventually fill up as more elements are pushed on the stack. As with the `Vector` implementation of Section 14.3, the array must grow when it gets full.

The `push` operation runs in amortized constant time, for the same reason that appending elements to a vector is an $O(1)$+ operation. The same is true for the `pop` operation.

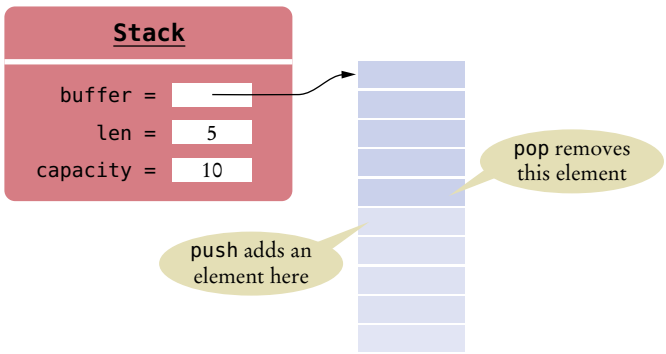


Figure 14 A Stack Implemented as an Array

EXAMPLE CODE See sec05_02 of your companion code for a stack implemented as an array.

14.5.3 Queues as Linked Lists

A queue can be implemented as a linked list, adding elements at the back and removing them at the front.

We now turn to the implementation of a queue. When implementing a queue as a sequence of nodes, we add nodes at one end and remove them at the other. In a singly-linked node sequence, it is not possible to remove the last node in $O(1)$ time, even if one has a pointer to the last node. It is also necessary to change the `next` pointer of the preceding node to `nullptr`. Locating the preceding node would be an $O(n)$ operation. Therefore, it is best to remove elements at the front and add them at the back (see Figure 15).

The `push` and `pop` operations of a queue are $O(1)$ operations. Each of them involves a fixed number of steps that is independent of the size of the queue. Note that we need a reference to the last node so that we can efficiently add elements.

EXAMPLE CODE See sec05_03 of your companion code for a queue implemented as a sequence of nodes.

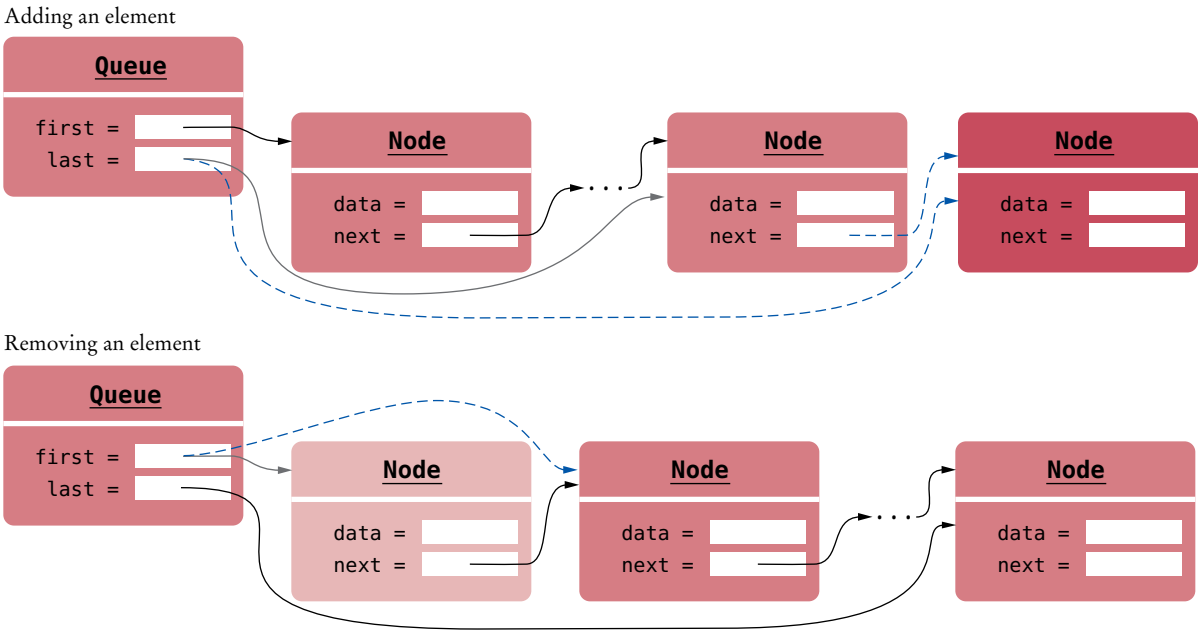


Figure 15 A Queue Implemented as a Linked List

14.5.4 Queues as Circular Arrays

In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.

When storing queue elements in an array, we have a problem: elements get added at one end of the array and removed at the other. But adding or removing the first element of an array is an $O(n)$ operation, so it seems that we cannot avoid this expensive operation, no matter which end we choose for adding elements and which for removing them.

However, we can solve this problem with a trick. We add elements at the end, but when we remove them, we don't actually move the remaining elements. Instead, we increment the index at which the head of the queue is located (see Figure 16).

© ihsanyildizli/iStockphoto.

In a circular array, we wrap around to the beginning after the last element.

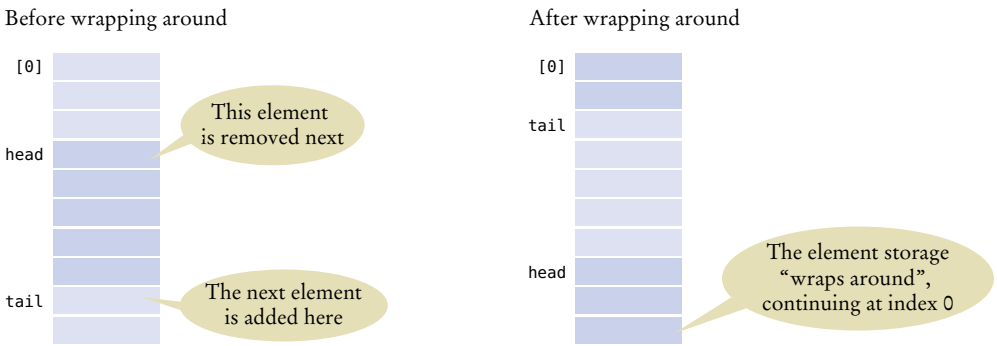


Figure 16 Queue Elements in a Circular Array

After adding sufficiently many elements, the last element of the array will be filled. However, if there were also a few calls to `pop`, then there is additional room in the front of the array. Then we "wrap around" and start storing elements again at index `0`—see part 2 of Figure 16. For that reason, the array is called "circular".

Eventually, of course, the tail reaches the head, and a larger array must be allocated.

As you can see from the source code that follows, adding or removing an element requires a bounded set of operations, independent of the queue size, except for array reallocation. However, as discussed in Section 14.3, reallocation happens rarely enough that the total cost is still amortized constant time, $O(1)+$.

Table 5 Efficiency of Stack and Queue Operations

	Stack as Linked List	Stack as Array	Queue as Linked List	Queue as Circular Array
Add an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$
Remove an element.	$O(1)$	$O(1)+$	$O(1)$	$O(1)+$

EXAMPLE CODE See sec05_04 of your companion code for a queue implemented as a circular array.

14.6 Stack and Queue Applications

Stacks and queues are, despite their simplicity, very versatile data structures. In the following sections, you will see some of their most useful applications.

14.6.1 Balancing Parentheses

A stack can be used to check whether parentheses in an expression are balanced.

In Common Error 2.4, you saw a simple trick for detecting unbalanced parentheses in an expression such as

```
-(b * b - (4 * a * c ) ) / (2 * a)
 1       2          1 0   1     0
```

Increment a counter when you see a `(` and decrement it when you see a `)`. The counter should never be negative, and it should be zero at the end of the expression.

That works for expressions in C++, but in mathematical notation, one can have more than one kind of parentheses, such as

$$-\{ [b \cdot b - (4 \cdot a \cdot c)] / (2 \cdot a) \}$$

To see whether such an expression is correctly formed, place the parentheses on a stack:

When you see an opening parenthesis, push it on the stack.
When you see a closing parenthesis, pop the stack.

```
If the opening and closing parentheses don't match
   The parentheses are unbalanced. Exit.
If at the end the stack is empty
   The parentheses are balanced.
Else
   The parentheses are not balanced.
```

Here is a walkthrough of the sample expression:

Stack	Unread expression	Comments
Empty	-{ [b * b - (4 * a * c)] / (2 * a) }	
{	[b * b - (4 * a * c)] / (2 * a) }	
{ [	b * b - (4 * a * c)] / (2 * a) }	
{ [(	4 * a * c)] / (2 * a) }	
{ [	] / (2 * a) }	(matches)
{	/ (2 * a) }	[matches]
{ (	2 * a) }	
{	}	(matches)
Empty	No more input	{ matches }
		The parentheses are balanced

EXAMPLE CODE See sec06_01 of your companion code for a program that uses a stack to check for balanced parentheses.

14.6.2 Evaluating Reverse Polish Expressions

Use a stack to evaluate expressions in reverse Polish notation.

Consider how you write arithmetic expressions, such as (3 + 4) × 5. The parentheses are needed so that 3 and 4 are added before multiplying the result by 5.

However, you can eliminate the parentheses if you write the operators *after* the numbers, like this: 3 4 + 5 × (see Special Topic 14.1). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 × to 35. It gets trickier for complex expressions. For example, 3 4 5 + × means to compute 4 5 + (that is, 9), and then evaluate 3 9 ×. If we evaluate this expression left-to-right, we need to leave the 3 somewhere while we work on 4 5 +. Where? We put it on a stack. The algorithm for evaluating reverse Polish expressions is simple:

```
If you read a number
   Push it on the stack.
Else if you read an operator
   Pop two values off the stack.
   Combine the values with the operator.
   Push the result back onto the stack.
Else if there is no more input
   Pop and display the result.
```

Here is a walkthrough of evaluating the expression 3 4 5 + ×:

Stack	Unread expression	Comments
Empty	3 4 5 + ×	
3	4 5 + ×	Numbers are pushed on the stack
3 4	5 + ×	
3 4 5	+ ×	
3 9	×	Pop 4 and 5, push 4 5 +
27	No more input	Pop 3 and 9, push 3 9 ×
Empty		Pop and display the result, 27

The following program simulates a reverse Polish calculator:

sec06_02/calculator.cpp

```
#include <iostream>
#include <stack>
#include <string>

using namespace std;

/*
   This calculator uses the reverse Polish notation.
*/
int main()
{
   stack<int> results;
   cout << "Enter numbers and operators, separated by spaces, " << endl
      << "P to print, Q to quit." << endl;
   bool done = false;
   while (!done)
   {
      string input;
      cin >> input;

      if (input == "+" || input == "-" || input == "*" || input == "/")
      {
         // If the command is an operator,
         // pop the arguments and push the result

         if (results.size() < 2)
         {
            cout << "Insufficient arguments" << endl;
            return 1;
         }

         // Note that the second argument is on the top of the stack

         int arg2 = results.top();
         results.pop();
         int arg1 = results.top();
         results.pop();

         if (input == "+")
         {
```

```
41             results.push(arg1 + arg2);
42          }
43          else if (input == "-")
44          {
45             results.push(arg1 - arg2);
46          }
47          else if (input == "*")
48          {
49             results.push(arg1 * arg2);
50          }
51          else
52          {
53             results.push(arg1 / arg2);
54          }
55       }
56       else if (input == "Q" || input == "q"
57          || input == "P" || input == "p")
58       {
59          if (results.size() > 0)
60          {
61             cout << results.top() << endl;
62          }
63          if (input == "Q" || input == "q")
64          {
65             done = true;
66          }
67       }
68       else
69       {
70          // Not an operator--push the input value
71
72          results.push(stoi(input));
73       }
74    }
75    return 0;
76 }
```

14.6.3 Evaluating Algebraic Expressions

Using two stacks, you can evaluate expressions in standard algebraic notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

Use two stacks to evaluate algebraic expressions.

First, consider a simple example, the expression 3 + 4. We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4	Comments
1	3		+ 4	
2	3	+	4	
3	4 3	+	No more input	Evaluate the top.
4	7			The result is 7.

This operation is fundamental to the algorithm. We call it "evaluating the top".

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, * and / have a higher (and equal) precedence.

Consider the expression 3 × 4 + 5. Here are the first processing steps:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × 4 + 5	Comments
1	3		× 4 + 5	
2	3	×	4 + 5	
3	4 3	×	+ 5	Evaluate × before +.

Because × has a higher precedence than +, we are ready to evaluate the top:

	Number stack	Operator stack		Comments
4	12	+	5	
5	5 12	+	No more input	Evaluate the top.
6	17			That is the result.

With the expression, 3 + 4 × 5, we add × to the operator stack because we must first read the next number; then we can evaluate × and then the +:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 + 4 × 5	Comments
1	3		+ 4 × 5	
2	3	+	4 + 5	

3	4 3	+	× 5	Don't evaluate + yet.
4	4 3	× +	5	

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

	Number stack	Operator stack		Comments
5	5 4 3	× +	No more input	Evaluate the top.
6	20 3	+		Evaluate top again.
7	23			That is the result.

To see how parentheses are handled, consider the expression 3 × (4 + 5). A (is pushed on the operator stack. The + is pushed as well. When we encounter the), we know that we are ready to evaluate the top until the matching (reappears:

	Number stack Empty	Operator stack Empty	Unprocessed input 3 × (4 + 5)	Comments
1	3		× (4 + 5)	
2	3	×	(4 + 5)	
3	3	(×	4 + 5)	Don't evaluate × yet.
4	4 3	(×	+ 5)	
5	4 3	+ (×	5)	
6	5 4 3	+ (×	)	Evaluate the top.
7	9 3	(×	No more input	Pop (.
8	9 3	×		Evaluate top again.
9	27			That is the result.

Here is the algorithm:

```
If you read a number
   Push it on the number stack.
Else if you read a (
   Push it on the operator stack.
Else if you read an operator op
   While precedence(top of operator stack) ≥ precedence(op)
      Evaluate the top.
   Push op on the operator stack.
Else if you read a )
   While the top of the stack is not a (
      Evaluate the top.
   Pop the (.
Else if there is no more input
   While the operator stack is not empty
      Evaluate the top.
```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper member function that evaluates the topmost operator with the topmost numbers:

```
Evaluate the top:
Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.
```

EXAMPLE CODE See sec06_03 of your companion code to get the complete code for the expression calculator.

14.6.4 Backtracking

Use a stack to remember choices you haven't yet made so that you can backtrack to them.

Suppose you are inside a maze. You need to find the exit. What should you do when you come to an intersection? You can continue exploring one of the paths, but you will want to remember the other ones. If your chosen path didn't work, you can go back to one of the other choices and try again.

© Skip ODonnell/iStockphoto.

A stack can be used to track positions in a maze.

Of course, as you go along one path, you may reach further intersections, and you need to remember your choice again. Simply use a stack to remember the paths that still need to be tried. The process of returning to a choice point and trying another choice is called *backtracking*. By using a stack, you return to your more recent choices before you explore the earlier ones.

Figure 17 shows an example. We start at a point in the maze, at position (3, 4). There are four possible paths. We push them all on a stack ❶. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4). We now push two choices on the stack, going west or east ❷. Both of them lead to dead ends ❸ ❹.

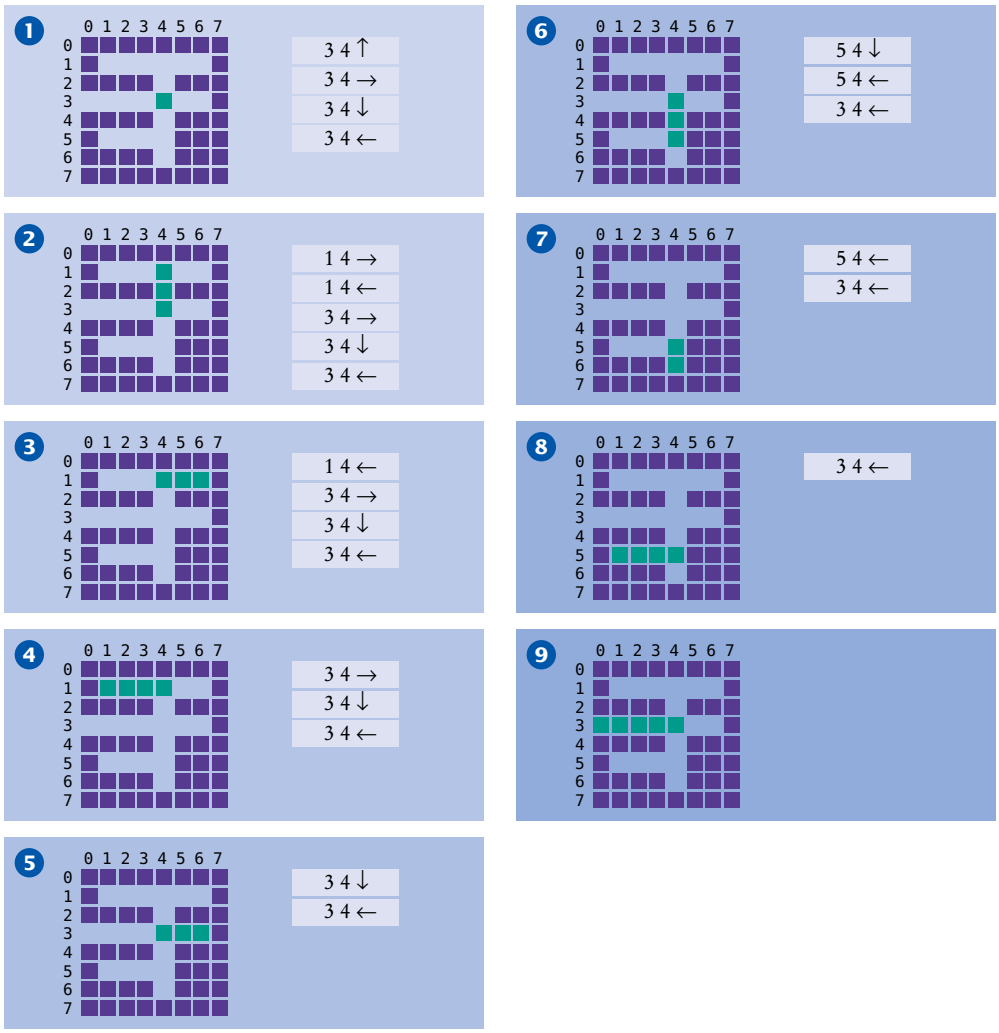


Figure 17 Backtracking Through a Maze

Now we pop off the path from (3, 4) going east. That too is a dead end ❺. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack ❻. They both lead to dead ends ❼ ❽.

Finally, the path from (3, 4) going west leads to an exit ❾.

Using a stack, we have found a path out of the maze. Here is the pseudocode for our maze-finding algorithm:

```
Push all paths from the point on which you are standing on a stack.
While the stack is not empty
   Pop a path from the stack.
   Follow the path until you reach an exit, intersection, or dead end.
   If you found an exit
      Congratulations!
   Else if you found an intersection
      Push all paths meeting at the intersection, except the current one, onto the stack.
```

This algorithm will find an exit from the maze, provided that the maze has no *cycles*. If it is possible that you can make a circle and return to a previously visited intersection along a different sequence of paths, then you need to work harder—see Exercise E14.21.

How you implement this algorithm depends on the description of the maze. In the example code, we use a two-dimensional array of characters, with spaces for corridors and asterisks for walls, like this:

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

In the example code, a `Path` object is constructed with a starting position and a direction (North, East, South, or West). The `Maze` class has a member function `extend` that extends a path until it reaches an intersection or exit, or until it is blocked by a wall, and a member function `paths_from` that computes all paths from an intersection point.

Note that you can use a queue instead of a stack in this algorithm. Then you explore the earlier alternatives before the later ones. This can work just as well for finding an answer, but it isn't very intuitive in the context of exploring a maze—you would have to imagine being teleported back to the initial intersections rather than just walking back to the last one.

EXAMPLE CODE See sec06_04 of your companion code for a complete program demonstrating backtracking.

Special Topic 14.1

Reverse Polish Notation

In the 1920s, the Polish mathematician Jan Łukasiewicz realized that it is possible to dispense with parentheses in arithmetic expressions, provided that you write the operators *before* their arguments, for example, `+ 3 4` instead of `3 + 4`. Thirty years later, Australian computer scientist Charles Hamblin noted that an even better scheme would be to have the operators *follow* the operands. This was termed **reverse Polish notation** or RPN.

Standard Notation	Reverse Polish Notation
3 + 4	3 4 +
3 + 4 × 5	3 4 5 × +
3 × (4 + 5)	3 4 5 + ×
(3 + 4) × (5 + 6)	3 4 + 5 6 + ×
3 + 4 + 5	3 4 + 5 +

Reverse Polish notation might look strange to you, but that is just an accident of history. Had earlier mathematicians realized its advantages, today's schoolchildren might be using it and not worrying about precedence rules and parentheses.

In 1972, Hewlett-Packard introduced the HP 35 calculator that used reverse Polish notation. The calculator had no keys labeled with parentheses or an equals symbol. There is just a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as "the calculators that have no equal".

Over time, calculator vendors have adapted to the standard algebraic notation rather than forcing its users to learn a new notation. However, those users who have made the effort to learn reverse Polish notation tend to be fanatic proponents, and to this day, some Hewlett-Packard calculator models still support it.

Courtesy of Nigel Tout.

The Calculator with No Equal

CHAPTER SUMMARY

Describe the linked list data structure and the use of list iterators.

- A linked list consists of a number of nodes, each of which has a pointer to the neighboring nodes.
- Adding and removing elements in the middle of a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You can inspect and edit a linked list with an iterator. An iterator points to a node in a linked list.

Explain how linked lists are implemented.

- When implementing a linked list, we need to define list, node, and iterator classes.
- A list node contains pointers to the next and previous nodes.

- A list object contains pointers to the first and last nodes.
- An iterator contains a pointer to the current node, and to the list that contains it.
- List nodes are allocated on the free store, using the `new` operator.
- When a list node is erased, it is recycled to the free store with the `delete` operator.
- Implementing operations that modify a linked list is challenging—you need to make sure that you update all node pointers correctly.

Know the efficiencies of the fundamental operations on lists, arrays, and vectors.

- Locating the kth element is an $O(k)$ operation for linked lists.
- Locating an element is an $O(1)$ operation for arrays and vectors.
- Adding an element in a linked list is an $O(1)$ operation.
- Adding an element in the middle of an array or vector of size n is an $O(n)$ operation.
- Adding an element to the end of an array is an $O(1)$ operation.
- An element can be added to the end of a vector in amortized $O(1)$ time.

Describe the stack and queue data structures.

- A stack is a container of items with "last in, first out" retrieval.
- A queue is a container of items with "first in, first out" retrieval.

Compare different implementations of stacks and queues.

- A stack can be implemented as a linked list, adding and removing elements at the front.
- When implementing a stack as an array, add and remove elements at the back.
- A queue can be implemented as a linked list, adding elements at the back and removing them at the front.
- In a circular array implementation of a queue, element locations wrap from the end of the array to the beginning.

Solve programming problems using stacks and queues.

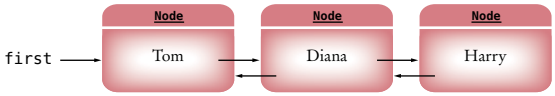
- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven't yet made so that you can backtrack to them.

REVIEW EXERCISES

■ **R14.1** If a list has n elements, how many legal positions are there for inserting a new element? For erasing an element?

■■ **R14.2** What happens if you keep advancing an iterator past the end of the list? Before the beginning of the list? What happens if you look up the value at an iterator that is past the end? If you erase the past-the-end position? All these are illegal operations, of course. What does the list implementation of your compiler do in these cases?

■■ **R14.3** The following code edits a linked list consisting of three nodes.



Draw a diagram showing how they are linked together after the following code is executed.

```
Node* p1 = first->next;
Node* p2 = first;
while (p2->next != nullptr) { p2 = p2->next; }
first->next = p2;
p2->next = p1;
p1->next = nullptr;
p2->previous = first;
p1->previous = p2;
last = p1;
```

■■ **R14.4** Explain what the following code prints.

```
list<string> names;
list<string>::iterator p = names.begin();
names.insert(p, "Tom");
p = names.begin();
names.insert(p, "Diana");
p++;
names.insert(p, "Harry");
for (p = names.begin(); p != names.end(); p++)
{
   cout << *p << endl;
}
```

■■ **R14.5** The `insert` procedure of Section 14.2.3 inserts a new element before the iterator position. To understand the updating of the nodes, draw before/after node diagrams for the following four scenarios.

a. The list is completely empty.

b. The list is not empty, and the iterator is at the beginning of the list.

c. The list is not empty, and the iterator is at the end of the list.

d. The list is not empty, and the iterator is in the middle of the list.

■ **R14.6** What advantages do lists have over vectors? What disadvantages do they have?

■ **R14.7** Suppose you need to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred

lookups against the collection every day. Would you use a vector or a linked list to store the information?

■ **R14.8** Suppose you need to keep a collection of appointments. Would you use a linked list or a vector of `Appointment` objects?

■ **R14.9** Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?

■ **R14.10** Consider the efficiency of locating the kth element in a linked list of length n. If $k > n/2$, it is more efficient to start at the end of the list and move the iterator to the previous element. Why doesn't this increase in efficiency improve the big-Oh estimate of random access in a linked list?

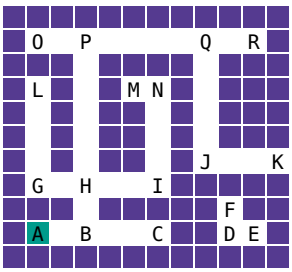
■ **R14.11** Explain why inserting an element into the middle of a list is faster than inserting an element into the middle of a vector.

■ **R14.12** Explain why the `push_back` operation with a vector is usually constant time, but occasionally much slower.

■■ **R14.13** Suppose a vector implementation were to add 10 elements at each reallocation instead of doubling the capacity. Show that the `push_back` operation no longer has amortized constant time.

■■ **R14.14** What is the big-Oh efficiency of selection sort when it is applied to a linked list?

■ **R14.15** Consider the algorithm for traversing a maze from Section 14.6.4. Assume that we start at position A and push in the order West, South, East, and North. In which order will the lettered locations of the sample maze be visited?



■ **R14.16** Repeat Exercise R14.15, using a queue instead of a stack.

■ **R14.17** What is the big-Oh efficiency of replacing all negative values in a linked list of integers with zeroes? Of removing all negative values?

■ **R14.18** What is the big-Oh efficiency of replacing all negative values in a vector of integers with zeroes? Of removing all negative values?

■ **R14.19** Show that the introduction of the `get_size` member function in Exercise E14.11 does not affect the big-Oh efficiency of the other list operations.

■■ **R14.20** Given the `get_size` member function of Exercise E14.11 and the `get` member function of Exercise E14.12, what is the big-Oh efficiency of this loop?

```
for (int i = 0; i < my_list.get_size(); i++) { cout << my_list.get(i) << endl; }
```

■■ **R14.21** Given the `get_size` member function of Exercise E14.11 and the `get` member function of Exercise E14.13, what is the big-Oh efficiency of this loop?

```
for (int i = 0; i < my_list.get_size(); i++) { cout << my_list.get(i) << endl; }
```

■ **R14.22** A linked list implementor, hoping to improve the speed of accessing elements, provides an array of `Node` pointers, pointing to every tenth node. Then the operation `get(n)` looks up the pointer at position `n - n / 10` and follows `n % 10` links.

a. With this implementation, what is the efficiency of the `get` operation?

b. What is the disadvantage of this implementation?

■ **R14.23** Consider a vector implementation with a `pop_back` member function that shrinks the internal array to half of its size when it is at most half full. Give a sequence of `push_back` and `pop_back` calls that does not have amortized $O(1)$ efficiency.

■■■ **R14.24** Suppose the `Vector` implementation of Section 14.3 had a `pop_back` member function that shrinks the internal array by 50 percent when it is less than 25 percent full. Show that any sequence of `push_back` and `pop_back` calls has amortized $O(1)$ efficiency.

■ **R14.25** Given a queue with $O(1)$ operations to add and remove elements at the head and tail of a sequence, what is the big-Oh efficiency of moving the element at the head of the queue to the tail? Of moving the element at the tail of the queue to the head? (The order of the other queue elements should be unchanged.)

■ **R14.26** A deque (double-ended queue) is a data structure with operations to add and remove elements at the head and tail of a sequence. What is the $O(1)$ efficiency of these operations if the deque is implemented as

a. a singly-linked list?

b. a doubly-linked list?

c. a circular array?

■■ **R14.27** In our circular array implementation of a queue, can you compute the value of `len` from the values of the `head` and `tail` data members? Why or why not?

■ **R14.28** Draw the contents of a circular array implementation of a queue `q`, with an initial array size of 10, after each of the following loops:

a. `for (int i = 1; i <= 5; i++) { q.push(i); }`

b. `for (int i = 1; i <= 3; i++) { q.pop(); }`

c. `for (int i = 1; i <= 10; i++) { q.push(i); }`

d. `for (int i = 1; i <= 8; i++) { q.pop(); }`

■■ **R14.29** Suppose you are stranded on a desert island on which stacks are plentiful, but you need a queue. How can you implement a queue using two stacks? What is the big-Oh running time of the queue operations?

■■ **R14.30** Suppose you are stranded on a desert island on which queues are plentiful, but you need a stack. How can you implement a stack using two queues? What is the big-Oh running time of the stack operations?

■ **R14.31** Why does the `Iterator` class in Section 14.2 need a pointer to the list?

■■ **R14.32** Describe an alternative implementation of the `Iterator` class in Section 14.2 that does not need a pointer to the list.

■■ **R14.33** When you call the `erase` member function to remove a list element, the iterator becomes invalid. What happens if you use it to insert an element? Explain the problem and draw a diagram.

■■ **R14.34** Continue Exercise R14.33 by providing a code example demonstrating the problem.

■ **R14.35** Describe an implementation of the `Iterator` class in Section 14.2 that does not suffer from the problem of Exercise R14.33.

■■ **R14.36** What happens when you call the `erase` or `insert` member function of the `List` class in Section 14.2 with an `Iterator` that doesn't point into the given list? Explain the problem and draw a diagram.

■■ **R14.37** Continue Exercise R14.36 by providing a code example demonstrating the problem.

■ **R14.38** Describe an implementation of the `erase` and `insert` member functions in Section 14.2 that fixes the problem of Exercise R14.36.

PRACTICE EXERCISES

■ **E14.1** Write a function that prints all values in a linked list, starting from the end of the list.

■ **E14.2** Write a function

```
void downsize(list<string>& names)
```

that removes every second value from a linked list.

■ **E14.3** Write a function `maximum` that computes the largest element in a `list<int>`.

■■■ **E14.4** Write a function `sort` that sorts the elements of a linked list (without copying them into a vector).

■■ **E14.5** Write a function `merge` that merges two lists into one, alternating elements from each list until the end of one of the lists has been reached, then appending the remaining elements of the other list. For example, merging the lists containing `A B C` and `D E F G H` should yield the list `A D B E C F G H`.

■ **E14.6** Provide a linked list of integers by modifying the `Node`, `List`, and `Iterator` classes of Section 14.2 to hold integers instead of strings.

■■ **E14.7** Write a member function `List::reverse()` that reverses the nodes in a list.

■ **E14.8** Write a member function `List::push_front()` that adds a value to the beginning of a list.

■ **E14.9** Write a member function `List::swap(List& other)` that swaps the elements of this list and `other`. Your member function should work in $O(1)$ time.

■ **E14.10** Write a member function `List::get_size()` that computes the number of elements in the list, by counting the elements until the end of the list is reached.

■ **E14.11** Add a `size` data member to the `List` class of Section 14.2. Modify the `insert` and `erase` functions to update the data member `size` so that it always contains the correct size. Change the `get_size()` function of Exercise E14.10 to take advantage of this data member.

■ **E14.12** Add a member function `string List::get(int n)` to the linked list class in Section 14.2.

■■■ **E14.13** Improve the efficiency of the `get` member function of Exercise E14.12 by storing (or "caching") the last known (*node*, *index*) pair. If `n` is larger than the last known index, start from the corresponding node instead of the front of the list. Be sure to discard the last known pair when it is no longer accurate. (This can happen when another member function edits the list).

■■ **E14.14** Use a stack to reverse the words of a sentence. Keep reading words until you have a word that ends in a period, adding them onto a stack. When you have a word with a period, pop the words off and print them. Stop when there are no more words in the input. For example, you should turn the input

```
Mary had a little lamb. Its fleece was white as snow.
```

into

```
Lamb little a had mary. Snow as white was fleece its.
```

Pay attention to capitalization and the placement of the period.

■ **E14.15** Your task is to break a number into its individual digits, for example, to turn 1729 into 1, 7, 2, and 9. It is easy to get the last digit of a number *n* as `n % 10`. But that gets the numbers in reverse order. Solve this problem with a stack. Your program should ask the user for an integer, then print its digits separated by spaces.

■■ **E14.16** A homeowner rents out parking spaces in a driveway during special events. The driveway is a "last-in, first-out" stack. Of course, when a car owner retrieves a vehicle that wasn't the last one in, the cars blocking it must temporarily move to the street so that the requested vehicle can leave. Write a program that models this behavior, using one stack for the driveway and one stack for the street. Use integers as license plate numbers. Positive numbers add a car, negative numbers remove a car, zero stops the simulation. Print out the stack after each operation is complete.

■ **E14.17** It is common to make the `pop` operation return the value that has been removed from a stack or queue. Modify all four implementations in Section 14.5 in this way.

■ **E14.18** Add a `%` (remainder) operator to the expression calculator of Section 14.6.3.

■■ **E14.19** Add a ^ (power) operator to the expression calculator of Section 14.6.3. For example, 2 ^ 3 evaluates to 8. As in mathematics, your power operator should be evaluated from the right. That is, 2 ^ 3 ^ 2 is 2 ^ (3 ^ 2), not (2 ^ 3) ^ 2. (That's more useful because you could get the latter as 2 ^ (3 × 2).)

■ **E14.20** Write a program that checks whether a sequence of HTML tags is properly nested. For each opening tag, such as `<p>`, there must be a closing tag `</p>`. A tag such as `<p>` may have other tags inside, for example

```
<p> <ul> <li> </li> </ul> <a> </a> </p>
```

The inner tags must be closed before the outer ones. Your program should process a file containing tags. For simplicity, assume that the tags are separated by spaces, and that there is no text inside the tags.

■ **E14.21** Modify the maze solver program of Section 14.6.4 to handle mazes with cycles. Keep a set of visited intersections. When you have previously seen an intersection, treat it as a dead end and do not add paths to the stack.

■ **E14.22** Add a member function `first_to_last` to the implementation of a queue in Section 14.5.3. The member function moves the element at the head of the queue to the tail of the queue. The element that was second in line will now be at the head.

■ **E14.23** Add a member function `last_to_first` to the implementation of a queue in Section 14.5.3. The member function moves the element at the tail of the queue to the head.

■ **E14.24** Add a member function `first_to_last`, as described in Exercise E14.22, to the circular array implementation of a queue in Section 14.5.4.

■ **E14.25** Add a member function `last_to_first`, as described in Exercise E14.23, to the circular array implementation of a queue in Section 14.5.4.

■ **E14.26** Write a function that removes all strings of a given length from a linked list and returns them in another linked list.

■ **E14.27** Implement the following operator function that prints the contents of a list of strings, separated by commas and enclosed in braces.

```
ostream& operator<<(ostream& out, const list<string>& lst)
```

■■ **E14.28** Implement an `operator>>` function that can read a `list<string>` that was produced by the output operator of Exercise E14.27.

■■ **E14.29** Implement the operators of Exercise E14.27 and Exercise E14.28 so that they can be applied to a `list<T>`.

■■ **E14.30** Change the increment and decrement operators of the iterator in Worked Example 14.1 so that they return the appropriate value (that is, the iterator before the move or a reference to the iterator after the move).

■■ **E14.31** The `*` operator of the iterator in Worked Example 14.1 is not as useful as the one in the standard list class. You cannot use it to modify a list, by calling

```
*iter = . . .;
```

Rectify this by providing two iterators: a `ConstIterator` whose `*` operator returns a `const T&`, and an `Iterator` whose `*` operator returns a `T&`. You now need two `begin` member functions:

```
List<T>::Iterator List<T>::begin()
List<T>::ConstIterator List<T>::begin() const
```

When calling `begin()` on a `const List<T>` (or more likely, a `const List<T>&`), the second member function is called. Provide a program that demonstrates that fact. Also provide two versions for the `end` member function.

PROGRAMMING PROJECTS

■■ **P14.1** Turn the linked list implementation into a *circular list:* Have the `previous` pointer of the first node point to the last node, and the `next` pointer of the last node point to the first node. Then remove the `last` pointer in the `List` class because the value can now be obtained as `first->previous`. Reimplement the member functions so that they have the same effect as before.

■■ **P14.2** Turn the linked list implementation into a **singly-linked list***:* Drop the `previous` pointer of the nodes and the `previous` member function of the iterator. Reimplement the other member functions so that they have the same effect as before. *Hint:* In

order to remove an element in constant time, iterators should store the predecessor of the current node.

■■ **P14.3** Modify the linked list implementation to use a *dummy node* for the past-the-end position whose `data` member is unused. A past-the-end iterator should point to the dummy node. Remove the `container` pointer in the iterator class. Reimplement the member functions so that they have the same effect as before.

■■ **P14.4** Improve the linked list implementation in Section 14.2 by printing an error message whenever any of the member functions of the `List` or `Iterator` classes is invoked with an invalid argument, or when the list or iterator is in an invalid state for the given operation. For example, you should detect a call to `get` or `next` when an iterator is at the end of the list, or a call to `erase` with an iterator that points to a different list.

■■ **P14.5** Write a class `Polynomial` that stores a polynomial such as

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a linked list of terms. A term contains the coefficient and the power of x. For example, you would store $p(x)$ as

$$(5,10),(9,7),(-1,1),(-10,0)$$

Supply member functions to add, multiply, and print polynomials. Supply a constructor that makes a polynomial from a single term. For example, the polynomial `p` can be constructed as

```
Polynomial p(Term(-10, 0));
p.add(Polynomial(Term(-1, 1)));
p.add(Polynomial(Term(9, 7)));
p.add(Polynomial(Term(5, 10)));
```

Then compute $p(x) \times p(x)$.

```
Polynomial q = p.multiply(p);
q.print();
```

■■ **P14.6** Using a queue of vectors, implement a non-recursive variant of the merge sort algorithm as follows. Start by inserting vectors of length 1 for each element into the queue. Keep removing pairs of vectors from the queue, merging them into a single vector, and adding the result back into the queue. Stop when the queue has size 1.

■■■ **P14.7** In a paint program, a "flood fill" fills all empty pixels of a drawing with a given color, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a 10×10 array of integers that are initially 0.

Prompt for the starting row and column.
Push the (row, column) pair on a stack.

You will need to provide a simple `Pair` class.

Repeat the following operations until the stack is empty:

Pop off the (row, column) pair from the top of the stack.
If it has not yet been filled, fill th corresponding array location with numbers 1, 2, 3, and so on (to show the order in which the square is filled.)
Push the coordinates of any unfilled neighbors in the north, east, south, or west direction on the stack.

When you are done, print the entire array.

■■■ **P14.8** Repeat Exercise P14.7, but use a queue instead.

■■■ **P14.9** Modify the expression calculator of Section 14.6.3 to convert an expression into reverse Polish notation. *Hint:* Instead of evaluating the top and pushing the result, append the instructions to a string.

■■ **P14.10** Use a stack to enumerate all permutations of a string. Suppose you want to find all permutations of the string `meat`.

```
Push the string +meat on the stack.
While the stack is not empty
   Pop off the top of the stack.
   If that string ends in a + (such as tame+)
      Remove the + and add the string to the list of permutations.
   Else
      Remove each letter in turn from the right of the +.
      Insert it just before the +.
      Push the resulting string on the stack.
```

For example, after popping `e+mta`, you push `em+ta`, `et+ma`, and `ea+mt`.

■■ **P14.11** Repeat Exercise P14.10, but use a queue instead.

■■ **Business P14.12** Suppose you buy 100 shares of a stock at $12 per share, then another 100 at $10 per share, and then sell 150 shares at $15. You have to pay taxes on the gain, but exactly what is the gain? In the United States, the FIFO rule holds: You first sell all shares of the first batch for a profit of $300, then 50 of the shares from the second batch, for a profit of $250, yielding a total profit of $550. Write a program that can make these calculations for arbitrary purchases and sales of shares in a single company. The user enters commands `buy` *quantity price*, `sell` *quantity price* (which causes the gain to be displayed), and `quit`. *Hint:* Keep a queue of objects of a class `Block` that contains the quantity and price of a block of shares.

■■ **P14.13** The linked list implementation of Section 14.2 used pointers to the first and last node to provide $O(1)$ insertion and removal at both ends of the list. Change the implementation to a *circular linked list*, where the the `next` pointer of the last node points to the first node and the `previous` pointer of the first node points to the last node. Then drop the `last` pointer and use `first->previous` instead to access the last element.

■■ **P14.14** A deque (double-ended queue) is a data structure with operations `push_front`, `pop_front`, `push_back`, `pop_back`, and `size`. Implement a deque as a circular array, so that these operations have amortized constant time.

■■ **P14.15** Turn the `Vector` class of Section 14.3 into a class template.

■■ **P14.16** Add memory management functions to the the `Vector` class of Section 14.3.

■■■ **P14.17** The `insert` and `erase` member functions of the `Vector` class of Section 14.3 use integer parameters to specify positions. The vector class in the standard library uses an iterator instead. Define an `Iterator` class for `Vector` and change these member functions to use it. Also provide `begin`, `end`, `get`, and `equals` member functions as in Section 14.2.

■■ **P14.18** Turn the `Stack` class of Section 14.5.1 into a class template.

■■ **P14.19** Turn the `Stack` class of Section 14.5.2 into a class template.

■■ **P14.20** Turn the `Queue` class of Section 14.5.3 into a class template.

- **P14.21** Turn the `Queue` class of Section 14.5.4 into a class template.
- **P14.22** Add memory management functions to the the `Stack` class of Section 14.5.1.
- **P14.23** Add memory management functions to the the `Stack` class of Section 14.5.2.
- **P14.24** Add memory management functions to the the `Queue` class of Section 14.5.3.
- **P14.25** Add memory management functions to the the `Queue` class of Section 14.5.4.

WORKED EXAMPLE 14.1

Implementing a Linked List Template

In this Worked Example, we turn the linked list class of Section 14.2 into a template class that can store elements of any type. We also add memory management operations so that lists can be safely copied and destroyed. Finally, we use nested classes and overloaded operators so that our iterators work exactly like the list iterators in the standard library.

Operators

The standard list iterators use operators: `*iter` fetches the element to which the iterator points, `++` and `--` move the iterator forward and backward, and you can use `==` and `!=` to compare iterators.

It is straightforward to provide these operators. Just change the names of the member functions `get`, `next`, `previous`, and `equals` to operators:

```
class Iterator
{
public:
   string operator*() const;
   void operator++(int dummy);
   void operator--(int dummy);
   void operator==(Iterator other) const;
};
```

Make the same change in the member function definitions. Note that the `++` and `--` operator functions receive a dummy `int` argument so that we can use their postfix form:

```
for (Iterator iter = names.begin(); iter != names.end(); iter++)
```

If we want to enable use of the prefix form `++iter` (which we have never used in this book), we also need to supply a member function

```
void Iterator::operator++();
```

without a parameter. In fact, it is a good idea to do that, in order to enable use of the range-based `for` loop. The compiler translates the loop

```
for (string element : names)
{
   . . .
}
```

into the equivalent of the loop

```
for (auto iter = names.begin(); iter != names.end(); ++iter)
{
   auto element = *iter;
   . . .
}
```

The range-based `for` loop uses the prefix version because its implementation is usually more efficient when the operators return a value. (The prefix version returns the value *after* the increment. The postfix version returns the value *before* the increment and is therefore at a disadvantage: it must make a copy of the old value. The prefix version can just return a reference to itself—see Exercise E14.30.)

Memory Management

When implementing the `List` class in Section 14.2, we were not concerned with memory management. However, when providing a class for use by the programming public, we cannot ignore this issue. When a `List` object goes out of scope, all of its nodes have been allocated on

the free store, and someone needs to deallocate them. Programmers expect that this happens in the destructor. As you have seen in Section 13.2, you also need to provide a copy constructor and `operator=` once you have a destructor.

In most cases, the `operator=` destroys the left-hand side and then copies the right-hand side into the left hand side. However, we cannot call destructors explicitly, and we cannot invoke constructors on existing objects. Instead, let us put the destruction and copy construction activities into two helper functions `free` and `copy`. Then we call them in the "big 3" memory management functions as follows:

```
List::List(const List& other)
{
   copy(other);
}

List::~List()
{
   free();
}

List& List::operator=(const List& other)
{
   if (this != &other)
   {
      free();
      copy(other);
   }
}
```

Now we just have two functions to implement.

The `free` function deallocates all nodes. It is straightforward to traverse all nodes and to delete them. You just have to be careful about one issue. As soon as a node is passed to `delete`, you can no longer use its data members. (Some free store implementations overwrite part of the deleted memory with data for internal use.) Therefore, you need to save the pointer to the next node before deleting a node:

```
void List::free()
{
   Node* to_delete = first;
   while (to_delete != nullptr)
   {
      Node* next_to_delete = to_delete->next;
      delete to_delete;
      to_delete = next_to_delete;
   }
}
```

The `copy` member function traverses the list that is to be copied, makes a copy of each node, and links the nodes together. The member function must also update the `first` and `last` pointers.

```
void List::copy(const List& other)
{
   Node* just_copied = nullptr;
   Node* next_to_copy = other.first;
   first = nullptr;
   last = nullptr;
   while (next_to_copy != nullptr)
   {
      Node* copy = new Node(next_to_copy->data);
      copy->previous = just_copied;
      if (just_copied == nullptr)
      {
```

```
            first = copy;
         }
         else
         {
            just_copied->next = copy;
         }
         if (next_to_copy == other.last)
         {
            last = copy;
         }
         next_to_copy = next_to_copy->next;
         just_copied = copy;
      }
   }
```

We make both of these helper functions private because they should never be called by a class user. The `free` function assumes that the freed object is never used again, and the `copy` function assumes that it is called on an uninitialized object.

References

Since we are thinking about memory management, this is a good time to make a simple but useful improvement to the `List` class. In order to avoid copying `string` objects, we should use constant references for all object parameters. Simply change parameters such as

```
void push_back(string element);
Iterator erase(Iterator iter);
```

to

```
void push_back(const string& element)
Iterator erase(const Iterator& iter);
```

Remember to update the member function definitions as well.

What about return values? You cannot return a reference to a local variable. For example, it would be an error to change the return type of the `begin` member function from `Iterator` to `Iterator&`:

```
Iterator& List::begin()
{
   Iterator iter;
   iter.position = first;
   iter.container = this;
   return iter; // Error—cannot return a reference to a local variable
}
```

A reference is a pointer in disguise. Here, we would return a pointer to `iter`, an object that is being reclaimed as soon as the `begin` function exits.

However, there is one place where we can return a reference. The `Iterator::get` member function returns data in a node. The node is allocated on the free store, and it still exists after the `get` member function exits:

```
const string& Iterator::operator*() const
{
   return position->data;
}
```

Nested Classes

In Section 14.2, the `List` class used two helper classes, `Node` and `Iterator`. Other data structures also use nodes and iterators. To avoid conflicts, we could rename these particular classes into `ListNode` and `ListIterator`. However, there is a more elegant approach. We can *nest* the `Node` and `Iterator` classes inside the `List` class. We will make the `Node` class private because users of the list

don't need to know about nodes. However, the `Iterator` is public and known to the world as `List::Iterator`. The following arrangement achieves this effect:

```
class List
{
private:
   class Node
   {
   public:
      Node(const string& element);
      string data;
      Node* previous;
      Node* next;
   };
public:
   class Iterator
   {
   public:
      Iterator();
      const string& operator*() const;
      void operator++();
      void operator++(int dummy);
      void operator--();
      void operator--(int dummy);
      bool operator==(const Iterator& other) const;
      bool operator!=(const Iterator& other) const;
   private:
      Node* position;
      List* container;
   friend class List;
   };
   // List member functions
private:
   void copy(const List& other);
   void free();
   Node* first;
   Node* last;
};
```

We first define the `Node` class because the `Iterator` class needs to know about nodes. Note that you can have multiple private and public sections in a class definition. We make the `Node` class private at the beginning of the definition of the `List` class. Because it is private, we don't need to bother making its data private. Then the `Iterator` class is defined in a public section, together with the public member functions. Finally, as we have always done, we declare the data members in a private section at the end of the class definition.

When you define a member function of a nested class, you use the full class name, including the enclosing class. For example:

```
const string& List::Iterator::operator*() const
{
   return position->data;
}
```

Use the same notation when a member function of the enclosing class returns a nested class object:

```
List::Iterator List::begin()
{
   Iterator iter;
   iter.position = first;
   iter.container = this;
```

```
   return iter;
}
```

However, once you get past the name of the member function, you don't have to use that prefix again. Note for example the declaration of `iter` in the `begin` member function. You are now inside the scope of `List`, and `List::` is not necessary. Here is another example:

```
void List::insert(const Iterator& iter, const string& element)
{
   . . .
}
```

The `Iterator` inside the parameter list comes after the name of the member function (or, technically, after the `List::` preceding the name), and therefore is in the `List` scope. Finally, consider the `erase` member function of the `List` class:

```
List::Iterator List::erase(const Iterator& iter)
{
   . . .
}
```

The return type needs the `List::` prefix, but the parameter type does not.

Templates

Now that we have the helper classes inside the `List` class, we need to define only a single class template:

```
template<typename T>
class List
{
   . . .
};
```

Every time that we used a `string` as parameter type, return type, or data member type, we now use `T`:

```
template<typename T>
class List
{
private:
   class Node
   {
      . . .
      T data;
   };
public:
   class Iterator
   {
      . . .
      const T& operator*() const;
      . . .
   };
   void push_back(const T& element);
   . . .
};
```

Provide a template for each member function:

```
template <typename T>
void List<T>::push_back(const T& element)
{
   . . .
}
```

For member functions of the nested classes, add <T> to the List:: prefix:

```
template <typename T>
const T& List<T>::Iterator::operator*() const
{
   return position->data;
}
```

Remember, the List class is the template. Iterator is a normal class inside List<T>.

There is a nasty complication when a member function template has a nested class return type. Consider the List<T>::begin() function. Logically, we would define it like this:

```
template <typename T>
List<T>::Iterator List<T>::begin()
{
   . . .
}
```

After all, the function returns an Iterator, and we need to prefix it with List<T>:: because we haven't yet entered the List<T> scope.

However, at this point, the compiler holds out the possibility that List<T>::Iterator could be something other than a type, even though we defined it as a class. (Unfortunately, it is possible to define a special version of the template for a specific type, and then make Iterator mean something else in that special version.) To overcome that problem, the C++ standard requires that we confirm that List<T>::Iterator is actually a type, by adding typename to it:

```
template <typename T>
typename List<T>::Iterator List<T>::begin()
```

There is nothing charitable one can say about this syntax. C++ is a complex language that developed over years, and occasionally one encounters such a quirk.

Apart from this complication, turning the List class into a template is straightforward. As you can see, there is a long journey from the basic list implementation to a professional version, and there is still room for improvement (see Exercises E14.30 and E14.31).

You can now change the demo code to use types List<string> and List<string>::Iterator, and use operators with the iterator.

worked_example_1/list.h

```
#ifndef LIST_H
#define LIST_H

using namespace std;

template <typename T>
class List
{
private:
   class Node
   {
   public:
      /**
         Constructs a node with a given data value.
         @param element the data to store in this node
      */
      Node(const T& element);
      T data;
      Node* previous;
      Node* next;
   };
public:
   class Iterator
```

```
24    {
25    public:
26       /**
27          Constructs an iterator that does not point into any list.
28       */
29       Iterator();
30       /**
31          Looks up the value at a position.
32          @return the value of the node to which the iterator points
33       */
34       const T& operator*() const;
35       /**
36          Advances the iterator to the next node.
37       */
38       void operator++();
39       void operator++(int dummy);
40       /**
41          Moves the iterator to the previous node.
42       */
43       void operator--();
44       void operator--(int dummy);
45
46       bool operator==(const Iterator& other) const;
47       bool operator!=(const Iterator& other) const;
48    private:
49       Node* position;
50       List* container;
51    friend class List;
52    };
53
54    /**
55       Constructs an empty list.
56    */
57    List();
58
59    List(const List& other);
60    ~List();
61    List& operator=(const List& other);
62
63    /**
64       Appends an element to the list.
65       @param element the value to append
66    */
67    void push_back(const T& element);
68    /**
69       Inserts an element into the list.
70       @param iter the position before which to insert
71       @param element the value to append
72    */
73    void insert(const Iterator& iter, const T& element);
74    /**
75       Removes an element from the list.
76       @param iter the position to remove
77       @return an iterator pointing to the element after the
78       erased element
79    */
80    Iterator erase(const Iterator& iter);
81    /**
82       Gets the beginning position of the list.
83       @return an iterator pointing to the beginning of the list
```

```
84       */
85       Iterator begin();
86       /**
87          Gets the past-the-end position of the list.
88          @return an iterator pointing past the end of the list
89       */
90       Iterator end();
91    private:
92       void copy(const List& other);
93       void free();
94
95       Node* first;
96       Node* last;
97    };
98
99    template <typename T>
100   List<T>::Node::Node(const T& element)
101   {
102      data = element;
103      previous = nullptr;
104      next = nullptr;
105   }
106
107   template <typename T>
108   List<T>::List()
109   {
110      first = nullptr;
111      last = nullptr;
112   }
113
114   template <typename T>
115   void List<T>::push_back(const T& element)
116   {
117      Node* new_node = new Node(element);
118      if (last == nullptr) // List is empty
119      {
120         first = new_node;
121         last = new_node;
122      }
123      else
124      {
125         new_node->previous = last;
126         last->next = new_node;
127         last = new_node;
128      }
129   }
130
131   template <typename T>
132   void List<T>::insert(const Iterator& iter, const T& element)
133   {
134      if (iter.position == nullptr)
135      {
136         push_back(element);
137         return;
138      }
139
140      Node* after = iter.position;
141      Node* before = after->previous;
142      Node* new_node = new Node(element);
143      new_node->previous = before;
```

```
144    new_node->next = after;
145    after->previous = new_node;
146    if (before == nullptr) // Insert at beginning
147    {
148       first = new_node;
149    }
150    else
151    {
152       before->next = new_node;
153    }
154 }
155
156 template <typename T>
157 typename List<T>::Iterator List<T>::erase(const Iterator& iter)
158 {
159    Node* remove = iter.position;
160    Node* before = remove->previous;
161    Node* after = remove->next;
162    if (remove == first)
163    {
164       first = after;
165    }
166    else
167    {
168       before->next = after;
169    }
170    if (remove == last)
171    {
172       last = before;
173    }
174    else
175    {
176       after->previous = before;
177    }
178    delete remove;
179    Iterator r;
180    r.position = after;
181    r.container = this;
182    return r;
183 }
184
185 template <typename T>
186 typename List<T>::Iterator List<T>::begin()
187 {
188    Iterator iter;
189    iter.position = first;
190    iter.container = this;
191    return iter;
192 }
193
194 template <typename T>
195 typename List<T>::Iterator List<T>::end()
196 {
197    Iterator iter;
198    iter.position = nullptr;
199    iter.container = this;
200    return iter;
201 }
202
```

```
203 template <typename T>
204 List<T>::Iterator::Iterator()
205 {
206    position = nullptr;
207    container = nullptr;
208 }
209
210 template <typename T>
211 const T& List<T>::Iterator::operator*() const
212 {
213    return position->data;
214 }
215
216 template <typename T>
217 void List<T>::Iterator::operator++()
218 {
219    position = position->next;
220 }
221
222 template <typename T>
223 void List<T>::Iterator::operator++(int dummy)
224 {
225    operator++();
226 }
227
228 template <typename T>
229 void List<T>::Iterator::operator--()
230 {
231    if (position == nullptr)
232    {
233       position = container->last;
234    }
235    else
236    {
237       position = position->previous;
238    }
239 }
240
241 template <typename T>
242 void List<T>::Iterator::operator--(int dummy)
243 {
244    operator--();
245 }
246
247 template <typename T>
248 bool List<T>::Iterator::operator==(const Iterator& other) const
249 {
250    return position == other.position;
251 }
252
253 template <typename T>
254 bool List<T>::Iterator::operator!=(const Iterator& other) const
255 {
256    return position != other.position;
257 }
258
259 template <typename T>
260 void List<T>::free()
261 {
262    Node* to_delete = first;
```

```
263    while (to_delete != nullptr)
264    {
265       Node* next_to_delete = to_delete->next;
266       delete to_delete;
267       to_delete = next_to_delete;
268    }
269 }
270
271 template <typename T>
272 void List<T>::copy(const List& other)
273 {
274    Node* just_copied = nullptr;
275    Node* next_to_copy = other.first;
276    first = nullptr;
277    last = nullptr;
278    while (next_to_copy != nullptr)
279    {
280       Node* copy = new Node(next_to_copy->data);
281       copy->previous = just_copied;
282       if (just_copied == nullptr)
283       {
284          first = copy;
285       }
286       else
287       {
288          just_copied->next = copy;
289       }
290       if (next_to_copy == other.last)
291       {
292          last = copy;
293       }
294       next_to_copy = next_to_copy->next;
295       just_copied = copy;
296    }
297 }
298
299 template <typename T>
300 List<T>::List(const List& other)
301 {
302    copy(other);
303 }
304
305 template <typename T>
306 List<T>::~List()
307 {
308    free();
309 }
310
311 template <typename T>
312 List<T>& List<T>::operator=(const List& other)
313 {
314    if (this != &other)
315    {
316       free();
317       copy(other);
318    }
319 }
320
321 #endif
```

worked_example_1/listdemo.cpp

```
#include <string>
#include <iostream>
#include "list.h"

using namespace std;

int main()
{
   List<string> names;

   names.push_back("Tom");
   names.push_back("Diana");
   names.push_back("Harry");
   names.push_back("Juliet");

   // Add a value in fourth place

   List<string>::Iterator pos = names.begin();
   pos++;
   pos++;
   pos++;

   names.insert(pos, "Romeo");

   // Remove the value in second place

   pos = names.begin();
   pos++;

   names.erase(pos);

   // Print all values

   for (pos = names.begin(); pos != names.end(); pos++)
   {
      cout << *pos << " ";
   }
   cout << endl;

   // Make a copy

   List<string> names2 = names;

   // Print with range-based for loop

   for (auto element : names2)
   {
      cout << element << " ";
   }
   cout << endl;

   return 0;
}
```

Program Run

```
Tom Harry Romeo Juliet
Tom Harry Romeo Juliet
```

CHAPTER 15

SETS, MAPS, AND HASH TABLES

CHAPTER GOALS

To become familiar with the set and map data types

To be able to use the set and map classes of the C++ standard library

To implement a hash table and understand the efficiency of its operations

CHAPTER CONTENTS

Arrays and linked lists arrange their elements in a linear sequence. In this chapter, you will see how to use and implement *unordered* collections in which the order in which elements are inserted does not matter. Perhaps surprisingly, it is more efficient to find elements in an unordered collection than in a linear sequence.

We start out introducing the set and map classes of the standard C++ class library. In the last section, you will learn how to implement the hash table data structure that is the basis of unordered collections.

15.1 Sets

Arrays, vectors, and linked lists have one characteristic in common: These data structures keep the elements in the same order in which you inserted them. However, in many applications, you don't really care about the order of the elements in a collection. You can then make a very useful tradeoff: Instead of keeping elements in order, you can find them quickly.

A set is an unordered collection of distinct elements.

In mathematics and computer science, an unordered collection of distinct items is called a **set**. As a typical example, consider a print server: a computer that has access to multiple printers. The server may keep a collection of objects representing available printers. The order of the objects doesn't really matter.

The fundamental operations on a set are

- Adding an element.
- Removing an element.
- Finding an element.
- Traversing all elements.

© JohnnyGreig/iStockphoto.

In a set of printers, order doesn't matter.

Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.

A set rejects duplicates. If an object is already in the set, an attempt to add it again is ignored. That's useful in many programming situations. For example, if we keep a set of available printers, each printer should occur at most once in the set. Adding elements that are already in the set, as well as removing elements that are not in the set, are valid operations, but they do not change the set.

The order of elements in a set does not matter. This fact is the key to making sets more efficient than arrays or linked lists. A set implementation can arrange the set elements so that finding, adding, and removing elements is efficient. Two completely different implementation strategies are commonly used for sets, called *hash tables* and *binary search trees*.

The basic idea of a **hash table** is simple. Set elements are grouped into smaller collections of elements that share the same characteristic. You can imagine a hash table of books as having a group for each color, so that books of the same color are in the same group. To find whether a book is already present, you just need to check it against the books in the same color group. Actually, hash tables don't use colors, but integer values (called **hash codes**) that can be computed from the elements. In order to use a hash table, there must be a mechanism for computing those hash codes. In C++, hash codes can be computed for common types such as numbers and strings. For other types, you need to provide the computation—see Special Topic 15.2. One way to compute the hash code of a color is to add up the red, green, and blue values.

© Alfredo Ragazzoni/iStockphoto.

On this shelf, books of the same color are grouped together. Similarly, in a hash table, objects with the same hash code are placed in the same group.

When you iterate over the elements of a hash table, you visit elements in the order of their hash codes, which usually appears to be quite random. For that reason, the C++ set implementation that makes use of a hash table is called an *unordered set*. In Section 15.3, you will see exactly how hash tables are implemented.

A **binary search tree** uses a different strategy for arranging its elements. Elements are kept in sorted order. For example, a set of books might be arranged by height, or alphabetically by author and title. The elements are not stored in an array—that would make adding and removing elements too inefficient. Instead, they are stored in nodes, as in a linked list. However, the nodes are not arranged in a linear sequence but in a tree shape. We will examine such trees in detail in the next chapter. A set using this implementation strategy is called an *ordered set*.

© Volkan Ersoy/iStockphoto.

An ordered set keeps its elements in sorted order.

In order to use an ordered set, it must be possible to compare the elements and determine which one is "larger". You can use an ordered set

for numbers and strings. For other types, you can define an `operator<`, as described in Section 13.1.2.

Unordered sets are usually more efficient than ordered sets.

As a rule of thumb, you should choose an ordered set if you want to visit the set's elements in sorted order. Otherwise choose an unordered set—as long as the hash function is well chosen, it is a bit more efficient.

In C++, you use the `unordered_set` and `set` class templates to construct unordered and ordered sets. For example, an unordered set of strings is declared as follows:

```
unordered_set<string> names;
```

You use the `insert` and `erase` member functions to add and remove elements:

```
names.insert("Romeo");
names.insert("Juliet");
names.insert("Romeo"); // Has no effect: "Romeo" is already in the set
names.erase("Juliet");
names.erase("Juliet"); // Has no effect: "Juliet" is no longer in the set
```

To determine whether a value is in the set, use the `count` member function. It returns 1 if the value is in the set, 0 otherwise.

```
int c = names.count("Romeo"); // count returns 1
```

An iterator of an `unordered_set` visits elements in seemingly random order.

Finally, you can visit the elements of a set with an iterator. The iterator visits the elements in seemingly random order. If the set was ordered, elements would be visited in increasing order. For example, consider what happens when we continue our set example as follows:

```
names.insert("Tom");
names.insert("Diana");
names.insert("Harry");
unordered_set<string>::iterator pos;
for (pos = names.begin(); pos != names.end(); pos++)
{
   cout << *pos << " ";
}
```

An iterator of an ordered set visits elements in increasing order.

The code prints the set elements `Diana Harry Romeo Tom` in some order.

A good illustration of the use of sets is a program to check for misspelled words. Assume you have a file containing correctly spelled words (that is, a dictionary), and a second file you wish to check. The program reads a file containing a dictionary of correctly spelled words into a set, then reads words from the second file and tests whether each is in the set, printing the word if it is not found.

```
int main()
{
   unordered_set<string> dictionary = read_words("../../words.txt");
   unordered_set<string> book = read_words("../../alice.txt");

   // Iterate over all words in the book
   for (auto iter = book.begin(); iter != book.end(); iter++)
   {
      string word = *iter;

      // Print the word if it is not in the dictionary
      if (dictionary.count(word) == 0)
      {
         cout << word << endl;
      }
   }
}
```

EXAMPLE CODE See sec01 of your companion code for the complete spell check program.

Table 1 Working with Sets

`unordered_set<string> names;`	Constructs an unordered set of strings. Use `set<string>` if you need to traverse the elements in sorted order.
`names.insert("Romeo");`	Now `names.size()` is 1.
`names.insert("Juliet");`	Now `names.size()` is 2.
`names.insert("Romeo");`	Now `names.size()` is still 2. You can't add duplicates.
`if (names.count("Fred") > 0)`	The `count` function returns 0 if the element is not present in the set, or 1 if it is present.

`for (set<string>::iterator p = names.begin();`
`  p != names.end(); p++)`
`{`
`  cout << *p << endl;`
`}`
	This is how you iterate over all elements of a set. With an unordered set, you do not know in which order the elements will be visited.

`names.erase("Romeo");`	Now `names.size()` is 1.
`names.erase("Romeo");`	It is not an error to erase an element that is not present. The call has no effect.

15.2 Maps

A map keeps associations between keys and values.

A **map** is a data type that keeps associations between *keys* and *values*. Every key in the map has a unique value, but a value may be associated with several keys. Figure 1 gives a typical example: a map that associates names with colors. This map might describe the favorite colors of various people.

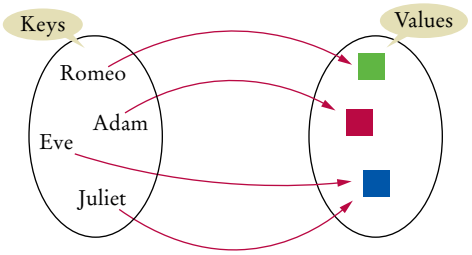


Figure 1 A Map

A map is implemented as a set of key/value pairs. For example, the map shown in Figure 1 is implemented as the set `{ (Romeo, ■), (Adam, ■), (Eve, ■), (Juliet, ■) }`. Therefore, the C++ standard library has two map implementations, `unordered_map` and `map`, in which the underlying sets are unordered or ordered.

With the `unordered_map` and `map` class templates in the standard library, you use the `[]` operator to associate keys and values. Here is an example:

```
unordered_map<string, double> scores;
scores["Tom"] = 90;
scores["Diana"] = 86;
scores["Harry"] = 100;
```

You can read a score back with the same notation:

```
cout << "Tom's score: " << scores["Tom"];
```

If the key is not present in the map, the [] operator automatically inserts the key and a default value. If you don't want that (for example, because you have a constant reference to a map), use the at member function instead:

```
int toms_score = scores.at("Tom");
```

To find out whether a key is present in the map, use the find member function. It yields an iterator that points to the entry with the given key, or past the end of the container if the key is not present.

A map iterator yields key/value pairs. If the map is ordered, they are visited in increasing key order.

The iterator of a map<K, V> with key type K and value type V yields elements of type pair<K, V>. The pair class is a simple class defined in the <utility> header that stores a pair of values. It has two public (!) data members, first and second. Therefore, you have to go through this process to see if a key is present:

```
unordered_map<string, double>::iterator pos = scores.find("Harry"); // Call find
if (pos == scores.end()) // Check if there was a match
{
   cout << "No match for Harry";
}
else
{
   cout << "Harry's score: " << (*pos).second;
      // pos points to a pair<string, double>
}
```

As with pointers, you can write pos->second instead of (*pos).second.

The following loop shows how you iterate over the contents of a map:

```
unordered_map <string, double>::iterator pos;
for (pos = scores.begin(); pos != scores.end(); pos++)
{
   cout << "The score of " << pos->first << " is " << pos->second << endl;
}
```

To remove a key/value pair from a map, call the erase member function:

```
scores.erase("Tom");
```

If the key is present, it is removed, together with its associated value. If you have an iterator that already points to the key/value pair that you want to remove, you can pass that to the erase member function:

```
scores.erase(pos);
```

A simple example to illustrate the use of maps is a telephone database. The database associates names with telephone numbers. One member function inserts elements into the database. There are member functions to look up the number associated with a given name. The member function print_all produces a listing of all entries. Because we use an unordered map, the program does not list the telephone records in alphabetical order. If an ordered traversal is important, use a map instead of an unordered_map.

sec02/tele.cpp

```
1  #include <iostream>
2  #include <unordered_map>
3  #include <utility>
4  #include <string>
5  #include <vector>
```

```
6
7  using namespace std;
8
9  /**
10    TelephoneDirectory maintains a map of name/number pairs.
11 */
12 class TelephoneDirectory
13 {
14 public:
15    /**
16       Add a new name/number pair to database.
17       @param name the new name
18       @param number the new number
19    */
20    void add_entry(string name, int number);
21
22    /**
23       Find the number associated with a name.
24       @param name the name being searched
25       @return the associated number, or zero
26       if not found in database
27    */
28    int find_entry(string name) const;
29
30    /**
31       Print all entries.
32    */
33    void print_all() const;
34 private:
35    unordered_map<string, int> database;
36 };
37
38 void TelephoneDirectory::add_entry(string name, int number)
39 {
40    database[name] = number;
41 }
42
43 int TelephoneDirectory::find_entry(string name) const
44 {
45    auto pos = database.find(name);
46    if (pos == database.end())
47    {
48       return 0; // Not found
49    }
50    else
51    {
52       return pos->second;
53    }
54 }
55
56 void TelephoneDirectory::print_all() const
57 {
58    for (auto pos = database.begin(); pos != database.end(); pos++)
59    {
60       cout << pos->first << ": " << pos->second << endl;
61    }
62 }
63
64 int main()
65 {
```

```
   TelephoneDirectory data;
   data.add_entry("Fred", 7235591);
   data.add_entry("Mary", 3841212);
   data.add_entry("Sarah", 3841212);
   cout << "Number for Fred: " << data.find_entry("Fred") << endl;
   cout << "All names and numbers:" << endl;
   data.print_all();
   return 0;
}
```

Program Run

```
Number for Fred: 7235591
All names and numbers:
Fred: 7235591
Mary: 3841212
Sarah: 3841212
```

Table 2 Working with Maps

`unordered_map<string, int> scores;`	Constructs an unordered map whose keys have type `string` and whose values are integers. Use a `map<string, int>` instead if you need to traverse the keys in sorted order.
`scores["Harry"] = 90;` `scores["Sally"] = 95;`	Adds keys and values to the map.
`scores["Sally"] = 100;`	Modifies the value of an existing key.
`int n = scores["Sally"];` `int n2 = scores["Diana"];`	Gets the value associated with a key, or a default (zero for integers) if the key is not present. `n` is 100, `n2` is 0. *Caution:* The map now contains the key `"Diana"` with value 0.
`int n = scores.at("Sally");` `int n2 = scores.at("Diana");`	Gets the value associated with a key, or a default (zero for integers) if the key is not present, *without* adding the key to the map. Use the `at` member function when you have a constant reference to a map.
`auto pos = scores.find("Harry");`	Gets an iterator pointing to the pair with key `"Harry"`, or past the end of the map. The `auto` reserved word lets you avoid the cumbersome iterator type `map<string, int>::iterator`.

`if (pos != scores.end())`
`{`
`  harrys_score = pos->second;`
`}`
	With a map iterator, `pos->first` is the key and `pos->second` is the value of the pair to which `pos` points.

`for (auto p = scores.begin(); p != scores.end(); p++)`
`{`
`  cout << p->first << " " << p->second << endl;`
`}`
	This is how you traverse all key/value pairs of a map.

`scores.erase("Harry");` `scores.erase(pos);`	To erase a key/value pair, pass the key or an iterator to the `erase` member function.

Programming Tip 15.1

Use the `auto` Type for Iterators

If you look at the source code for the `tele.cpp` program, you will notice that the member functions for finding and printing declare the map iterators with the `auto` reserved word instead of the actual iterator types.

This is a good idea because the actual types are very unwieldy. At first glance, it seems as if the `pos` iterator should be declared as

```
unordered_map<string, int>::iterator pos;
```

However, if you have a closer look at the `TelephoneDirectory` class in the `tele.cpp` program, you will notice that the `find_entry` and `print_all` member functions are declared as `const`. Therefore, the `database` member is `const` and `database.begin()` returns a `const_iterator`. That is, the correct declaration would be

```
unordered_map<string, int>::const_iterator pos;
```

That's a mouthful. The `auto` reserved word was invented to relieve programmers from having to write and decipher such unwieldy type names.

Special Topic 15.1

Multisets and Multimaps

A multiset (or bag) is similar to a set, but elements can occur multiple times.

A set cannot contain duplicates. A *multiset* (also called a bag) is an unordered collection that can contain multiple copies of an element. An example is a grocery bag that contains some grocery items more than once.

In the C++ library, the `unordered_multiset` and `multiset` class templates implement this data type. You use a multiset in the same way as a set. When you insert an element multiple times, the element count reflects the number of insertions. Each call to `erase` decrements the element count until it reaches 0.

```
unordered_multiset<string> names;
names.insert("Romeo");
names.insert("Juliet");
names.insert("Romeo"); // Now names.count("Romeo") is 2
names.erase("Juliet"); // Now names.count("Juliet") is 0
names.erase("Juliet"); // Has no effect: "Juliet" is no
                       // longer in the bag
```

© studiocaspar/iStockphoto.

A grocery bag that contains some items more than once can be modeled with a multiset.

A multimap can have multiple values associated with the same key.

A *multimap* can have multiple values associated with the same key. Instead of using the `[]` operator, you insert pairs.

Here is an example:

```
unordered_multimap<string, string> friends;
friends.insert(make_pair("Tom", "Diana"));  // Diana is a friend of Tom
friends.insert(make_pair("Tom", "Harry"));  // Harry is also a friend of Tom
```

The `make_pair` function (also defined in the `<utility>` header) makes a `pair` object from its arguments.

To enumerate all values associated with a key, you obtain a pair iterator that defines the range containing all pairs with the given key:

```
auto tom_range = friends.equal_range("Tom");
   // pair<unordered_multimap<string, string>::iterator>
```

Then you visit all pairs in that range:

```
cout << "Tom's friends: ";
for (auto pos = tom_range.first; pos != tom_range.second; pos++)
{
   cout << pos->second << " ";
}
```

To erase an entry, call the `erase` member function with an iterator pointing to it:

```
friends.erase(tom_range.first); // Erases Tom's first friend
```

EXAMPLE CODE See special_topic_1 of your companion code for a program that demonstrates multiset and multimap.

WORKED EXAMPLE 15.1

Word Frequency

Learn how to create a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file. See your E-Text or visit `wiley.com/go/bclo3`.

© Ermin Gutenberger/iStockphoto.

15.3 Implementing a Hash Table

In Section 15.1, you were introduced to the set data structure and its two implementations in the C++ standard library, hash tables and binary search trees. In these sections, you will see how hash tables are implemented and how efficient their operations are.

15.3.1 Hash Codes

The basic idea behind hashing is to place values into an array, at a location that can be determined from the value itself. Each value has a **hash code**, an integer value that is computed from a value in such a way that different values are likely to yield different hash codes.

Let us look at one way of computing a hash code for a string. We want to combine the `char` values of the string into an integer. We could simply add up all `char` values, but then permutations such as `"eat"` and `"tea"` would have the same hash code. A better way is to scramble the codes a bit:

```
int hash_code(const string& str)
{
   int h = 0;
   for (int i = 0; i < str.length(); i++)
   {
      h = 31 * h + str[i];
   }
   return h;
}
```

A hash function computes an integer value from an object.

For example, the hash code of `"eat"` is

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

The hash code of `"tea"` is quite different, namely

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

(Use the Unicode table from Appendix C to look up the character values: `'a'` is 97, `'e'` is 101, and `'t'` is 116.)

Table 3 shows some examples of strings and their hash codes with this particular hash function.

Table 3 Sample Strings and Their Hash Codes

String	Hash Code	String	Hash Code
`"Adam"`	2035631	`"Juliet"`	–2065036585
`"Eve"`	70068	`"Katherine"`	2079199209
`"Harry"`	69496448	`"Sue"`	83491
`"Jim"`	74478	`"Ugh"`	84982
`"Joe"`	74656	`"VII"`	84982

A good hash function minimizes *collisions*—identical hash codes for different values.

It is possible for two or more distinct objects to have the same hash code; this is called a **collision**. For example, the strings `"Ugh"` and `"VII"` happen to have the same hash code, but these collisions are very rare for strings (see Exercise P15.6).

15.3.2 Hash Tables

A hash table uses the hash code to determine where to store each element.

A hash code is used as an array index into a **hash table**, an array that stores the set elements. In the simplest implementation of a hash table, you could make a very long array and insert each element at the location of its hash code (see Figure 2).

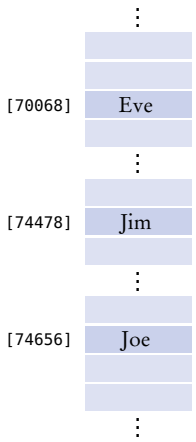


Figure 2 A Simplistic Implementation of a Hash Table

A good hash function produces different hash codes for each value so that they are scattered about in a hash table.

If there are no collisions, it is a very simple matter to find out whether a value is already present in the set or not. Compute its hash code and check whether the array position with that hash code is already occupied. This doesn't require a search through the entire array!

Of course, it is not feasible to allocate an array that is large enough to hold all possible integer index positions. Therefore, we must pick an array of some reasonable size and then "compress" the hash code to become a valid array index. Compression can be easily achieved by using the remainder operation:

```
int h = hash_code(x);
h = h % len;
if (h < 0) { h = -h; }
```

See Exercise E15.17 for an alternative compression technique.

A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.

After compressing the hash code, it becomes more likely that several elements will collide. There are several techniques for handling collisions. The most common one is called *separate chaining*. All colliding elements are collected in a linked list of elements with the same position value (see Figure 3). Such a list is called a "bucket". Special Topic 15.3 discusses *open addressing*, in which colliding elements are placed in empty locations of the hash table.

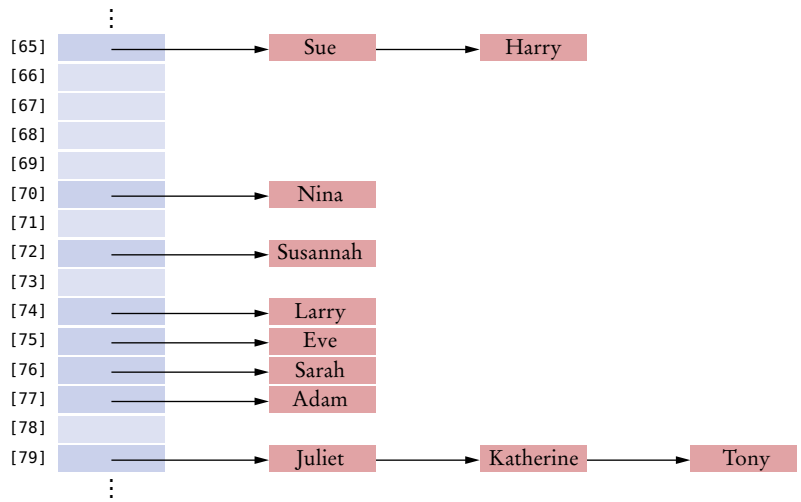
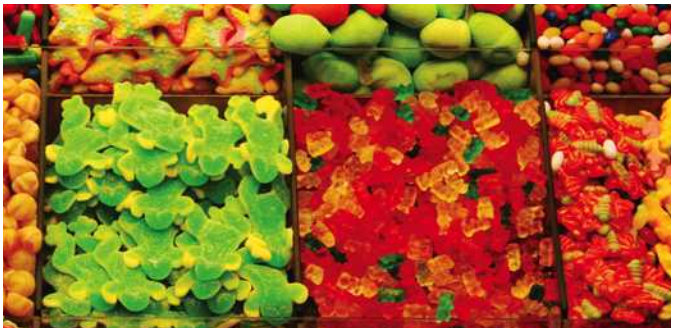


Figure 3 A Hash Table with Buckets to Store Elements with the Same Hash Code

In the following, we will use the first technique. Each entry of the hash table points to a sequence of nodes containing elements with the same (compressed) hash code.



Elements with the same hash code are placed in the same bucket.

15.3.3 Finding an Element

Let's assume that our hash table has been filled with a number of elements. Now we want to find out whether a given element is already present.

Here is the algorithm for finding an element `x` in a hash table:

1. Compute the hash code and compress it. This gives an index `h` into the hash table.
2. Iterate through the elements of the bucket at position `h`. For each element of the bucket, check whether it is equal to `x`.
3. If a match is found among the elements of that bucket, then `x` is in the set. Otherwise, it is not.

If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or $O(1)$ time.

How efficient is this operation? It depends on the hash code computation. In the best case, in which there are no collisions, all buckets either are empty or have a single element.

But in practice, some collisions will occur. We need to make some assumptions that are reasonable in practice.

First, we assume that the hash code does a good job scattering the elements into different buckets. In practice, the hash functions described in Special Topic 15.2 work well.

Next, we assume that the table is large enough. This is measured by the *load factor* $F = n/L$, where n is the number of elements and L the table length. For example, if the table is an array of length 1,000, and it has 700 elements, then the load factor is 0.7.

If the load factor gets too large, the elements should be moved into a larger table. A reasonable choice for the load factor is 0.75, which means that the table should be reallocated when it is more than 3/4 full.

Under these assumptions, each bucket can be expected to have, on average, F elements.

Finally, we assume that the hash code, its compression, and the equality test can be computed in bounded time, independent of the size of the set.

Now let us compute the cost of finding an element. Computing the array index takes constant time, due to our last assumption. Now we traverse a chain of buckets, which on average has a bounded length F. Finally, we test each bucket element for

equality, which we also assume to be $O(1)$. The entire operation takes constant or $O(1)$ time.

15.3.4 Adding and Removing Elements

Adding an element is an extension of the algorithm for finding an object. First compute the hash code to locate the bucket in which the element should be inserted:

1. Compute the compressed hash code `h`.
2. Iterate through the elements of the bucket at position `h`. For each element of the bucket, check whether it is equal to `x` (using the equality operator of the element type).
3. If a match is found among the elements of that bucket, then exit.
4. Otherwise, add a node containing `x` to the beginning of the node sequence.
5. If the load factor exceeds a fixed threshold, reallocate the table.

As described in the preceding section, the first three steps are $O(1)$. Inserting at the beginning of a node sequence is also $O(1)$. As with vectors, we can choose the new table to be twice the size of the old table, and amortize the cost of reallocation over the preceding insertions. That is, adding an element to a hash table is $O(1)+$.

Removing an element is equally simple. First compute the hash code to locate the bucket in which the element should be inserted. Try finding the element in that bucket. If it is present, remove it. Otherwise, do nothing. Again, this is a constant time operation. If we shrink a table that becomes too sparse, the cost is $O(1)+$.

15.3.5 Iterating over a Hash Table

An iterator for a linked list points to the current node in a list. A hash table has multiple node chains. When we are at the end of one chain, we need to move to the start of the next one. Therefore, the iterator also needs to store the bucket number (see Figure 4).

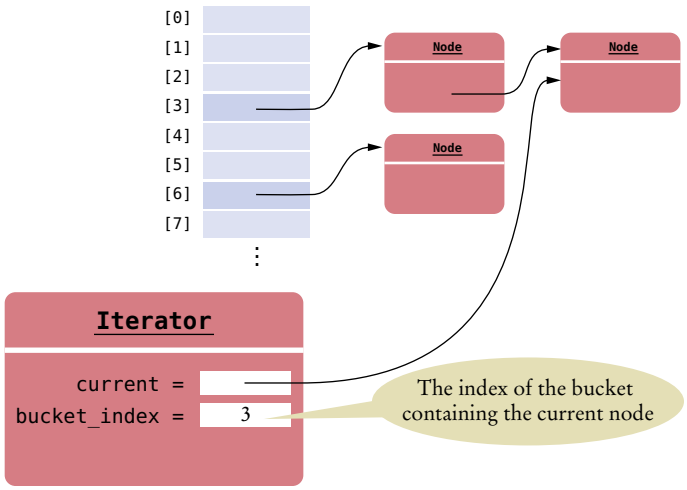


Figure 4
An Iterator to a Hash Table

When the iterator points into the middle of a node chain, then it is easy to advance it to the next element. However, when the iterator points to the last node in a chain, or if we have not yet found the first bucket, then we must skip past all empty buckets. When we find a non-empty bucket, we advance the iterator to its first node:

```
if (bucket_index >= 0 && current->next != nullptr)
{
   // Advance in the same bucket
   current = current->next;
}
else
{
   // Move to the next bucket
   do
   {
      bucket_index++;
   }
   while (bucket_index < container->buckets.size()
      && container->buckets[bucket_index] == nullptr);
   if (bucket_index < container->buckets.size())
   {
      // Start of next bucket
      current = container->buckets[bucket_index];
   }
   else
   {
      // No more buckets
      current = nullptr;
   }
}
```

As you can see, the cost of iterating over all elements of a hash table is proportional to the table length. Note that the table length could be in excess of $O(n)$ if the table is sparsely filled. This can be avoided if we shrink the table when the load factor gets too small. In that case, iterating over the entire table is $O(n)$, and each iteration step is $O(1)$.

Table 4 summarizes the efficiency of the operations on a hash table.

Table 4 Hash Table Efficiency

Operation	Hash Table
Find an element.	$O(1)$
Add/remove an element.	$O(1)+$
Iterate through all elements.	$O(n)$

Here is an implementation of a hash table. For simplicity, we do not reallocate the table when it grows or shrinks. Exercise E15.16 asks you to provide this enhancement.

sec03/hashtable.h

```
1  #ifndef HASHTABLE_H
2  #define HASHTABLE_H
3
4  #include <string>
5  #include <vector>
```

```
 6
 7  using namespace std;
 8
 9  /**
10     Computes the hash code for a string.
11     @param str a string
12     @return the hash code
13  */
14  int hash_code(const string& str);
15
16  class HashTable;
17  class Iterator;
18
19  class Node
20  {
21  private:
22     string data;
23     Node* next;
24
25  friend class HashTable;
26  friend class Iterator;
27  };
28
29  class Iterator
30  {
31  public:
32     /**
33        Looks up the value at a position.
34        @return the value of the node to which the iterator points
35     */
36     string get() const;
37     /**
38        Advances the iterator to the next node.
39     */
40     void next();
41     /**
42        Compares two iterators.
43        @param other the iterator to compare with this iterator
44        @return true if this iterator and other are equal
45     */
46     bool equals(const Iterator& other) const;
47  private:
48     const HashTable* container;
49     int bucket_index;
50     Node* current;
51
52  friend class HashTable;
53  };
54
55  /**
56     This class implements a hash table using separate chaining.
57  */
58  class HashTable
59  {
60  public:
61     /**
62        Constructs a hash table.
63        @param nbuckets the number of buckets
64     */
65     HashTable(int nbuckets);
```

```
66
67    /**
68       Tests for set membership.
69       @param x the potential element to test
70       @return 1 if x is an element of this set, 0 otherwise
71    */
72    int count(const string& x);
73
74    /**
75       Adds an element to this hash table if it is not already present.
76       @param x the element to add
77    */
78    void insert(const string& x);
79
80    /**
81       Removes an element from this hash table if it is present.
82       @param x the potential element to remove
83    */
84    void erase(const string& x);
85
86    /**
87       Returns an iterator to the beginning of this hash table.
88       @return a hash table iterator to the beginning
89    */
90    Iterator begin() const;
91
92    /**
93       Returns an iterator past the end of this hash table.
94       @return a hash table iterator past the end
95    */
96    Iterator end() const;
97
98    /**
99       Gets the number of elements in this set.
100      @return the number of elements
101   */
102   int size() const;
103
104 private:
105    vector<Node*> buckets;
106    int current_size;
107
108 friend class Iterator;
109 };
110
111 #endif
```

sec03/hashtable.cpp

```
1  #include "hashtable.h"
2
3  int hash_code(const string& str)
4  {
5     int h = 0;
6     for (int i = 0; i < str.length(); i++)
7     {
8        h = 31 * h + str[i];
9     }
10    return h;
11 }
```

```
12
13 HashTable::HashTable(int nbuckets)
14 {
15    for (int i = 0; i < nbuckets; i++)
16    {
17       buckets.push_back(nullptr);
18    }
19    current_size = 0;
20 }
21
22 int HashTable::count(const string& x)
23 {
24    int h = hash_code(x);
25    h = h % buckets.size();
26    if (h < 0) { h = -h; }
27
28    Node* current = buckets[h];
29    while (current != nullptr)
30    {
31       if (current->data == x) { return 1; }
32       current = current->next;
33    }
34    return 0;
35 }
36
37 void HashTable::insert(const string& x)
38 {
39    int h = hash_code(x);
40    h = h % buckets.size();
41    if (h < 0) { h = -h; }
42
43    Node* current = buckets[h];
44    while (current != nullptr)
45    {
46       if (current-> data == x) { return; }
47          // Already in the set
48       current = current->next;
49    }
50    Node* new_node = new Node;
51    new_node->data = x;
52    new_node->next = buckets[h];
53    buckets[h] = new_node;
54    current_size++;
55 }
56
57 void HashTable::erase(const string& x)
58 {
59    int h = hash_code(x);
60    h = h % buckets.size();
61    if (h < 0) { h = -h; }
62
63    Node* current = buckets[h];
64    Node* previous = nullptr;
65    while (current != nullptr)
66    {
67       if (current->data == x)
68       {
69          if (previous == nullptr)
70          {
71             buckets[h] = current->next;
```

```
72            }
73            else
74            {
75               previous->next = current->next;
76            }
77            delete current;
78            current_size--;
79            return;
80         }
81         previous = current;
82         current = current->next;
83      }
84   }
85
86   int HashTable::size() const
87   {
88      return current_size;
89   }
90
91   Iterator HashTable::begin() const
92   {
93      Iterator iter;
94      iter.current = nullptr;
95      iter.bucket_index = -1;
96      iter.container = this;
97      iter.next();
98      return iter;
99   }
100
101  Iterator HashTable::end() const
102  {
103     Iterator iter;
104     iter.current = nullptr;
105     iter.bucket_index = buckets.size();
106     iter.container = this;
107     return iter;
108  }
109
110  string Iterator::get() const
111  {
112     return current->data;
113  }
114
115  bool Iterator::equals(const Iterator& other) const
116  {
117     return current == other.current;
118  }
119
120  void Iterator::next()
121  {
122     if (bucket_index >= 0 && current->next != nullptr)
123     {
124        // Advance in the same bucket
125        current = current->next;
126     }
127     else
128     {
129        // Move to the next bucket
130        do
131        {
```

```
132             bucket_index++;
133          }
134          while (bucket_index < container->buckets.size()
135             && container->buckets[bucket_index] == nullptr);
136          if (bucket_index < container->buckets.size())
137          {
138             // Start of next bucket
139             current = container->buckets[bucket_index];
140          }
141          else
142          {
143             // No more buckets
144             current = nullptr;
145          }
146       }
147    }
```

EXAMPLE CODE See sec03 of your companion code to see the spell check program you saw in Section 15.1, modified to use this hash table implementation.

Special Topic 15.2

Implementing Hash Functions

The unordered sets and maps in the C++ library use the `hash` template class that is defined in the `<functional>` header. To obtain a **hash function** for a specific type, construct an object like this:

```
hash<string> string_hash_code;
```

This object acts like a function because the class overloads the **function call operator**.

When a class provides a member function with name `operator()`, that function is called when an object is followed by arguments enclosed in parentheses. For example, the `hash<string>` class provides a member function

```
int hash<string>::operator(const string&) const;
```

You can obtain hash codes by passing a value of the given type:

```
int h = string_hash_code("eat");
```

The `hash` template is defined for numeric types `int`, `double`, `char`, and so on, `bool`, `string`, pointers, and a small number of types that we have not used in this book. If you want to form a hash table for another type, then you need to define the `hash` template for that type.

For example, suppose we have a class `Task` that models a task that needs to be accomplished, with a description and a priority:

```
class Task
{
public:
   Task(string description, int priority);
   string get_description() const;
   int get_priority() const;
private:
   string description;
   int priority;
};
```

We want to form an `unordered_set<Task>`. By default, this declaration will not compile because the compiler does not know how to compute hash codes of `Task` objects.

You need to define a `hash<Task>` class with an overloaded function call operator. Moreover, this definition must be in the `std` namespace. This task requires a number of advanced C++ concepts that go well beyond the scope of this book. However, you can easily produce the required code if you are willing to take the syntax on faith:

```
namespace std
{
   template<>
   class hash<Task>
   {
   public:
      int operator()(const Task &t) const;
   };
};
```

If you declare the `Task` class in a `task.h` header file, the definition of `hash<Task>` goes in the same header file. Next, you need to implement the overloaded function call operator so that it computes the hash code of a `Task` object:

```
namespace std
{
   int hash<Task>::operator()(const Task &t) const
   {
      hash<string> string_hash_code;
      int h = string_hash_code(t.get_description())
         + t.get_priority();
      return h;
   }
}
```

This definition goes in the source file `task.cpp`.

Overload `hash<T>::operator()` by combining the hash codes for the data members.

As you can see, the hash code of a `Task` object is obtained by combining the hash codes of all data members. An integer member should simply be hashed to itself. For members of other types, use the `hash` template to obtain a hash function, and then invoke that function. In the preceding code example, we did that for the `description` data member. Its type is `string`, so we defined and invoked a hash function of type `hash<string>`.

When you supply your own `hash` class for a type, you must also provide a compatible `operator==` function. That operator is used to differentiate between two objects that happen to have the same hash code.

A class's hash function must be compatible with its equality operator.

The equality operator and hash function must be *compatible* with each other. Two values that are equal must yield the same hash code.

For the `Task` class, we can add a function that compares the data members that were used to compute the hash code:

```
bool operator==(const Task& a, const Task& b)
{
   return a.get_description() == b.get_description()
      && a.get_priority() == b.get_priority();
}
```

Place this definition into the `task.cpp` source file. The corresponding declaration

```
bool operator==(const Task& a, const Task& b);
```

belongs in the header file.

In this way, you can define a hash function for any type, and you can then collect values of that type in unordered sets and maps.

EXAMPLE CODE See special_topic_2 of your companion code for a program that demonstrates the hash function for the `Task` class.

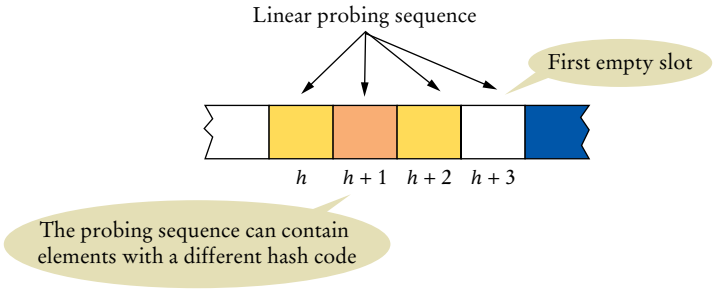
Special Topic 15.3
Open Addressing

In the preceding sections, you studied a hash table implementation that uses separate chaining for collision handling, placing all elements with the same hash code in a bucket. This implementation is fast and easy to understand, but it requires storage for the links to the nodes. If one places the elements directly into the hash table, then one doesn't need to store any links. This alternative technique is called *open addressing*. It can be beneficial if one must minimize the memory usage of a hash table.

Of course, open addressing makes collision handling more complicated. If you have two elements with (compressed) hash code h, and the first one is placed at index h, then the second must be placed in another location.

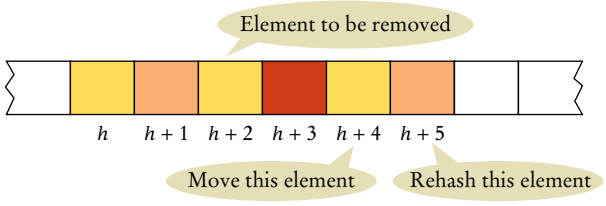
There are different techniques for placing colliding elements. The simplest is *linear probing*. If possible, place the colliding element at index $h + 1$. If that slot is occupied, try $h + 2$, $h + 3$, and so on, wrapping around to 0, 1, 2, and so on, if necessary. This sequence of index values is called the *probing sequence*. (You can see other probing sequences in Exercises P15.15 and P15.16.) If the probing sequence contains no empty slots, one must reallocate to a larger table.

How do we find an element in such a hash table? We compute the hash code and traverse the probing sequence until we either find a match or an empty slot. As long as the hash table is not too full, this is still an $O(1)$ operation, but it may require more comparisons than with separate chaining. With separate chaining, we only compare elements with the same hash code. With open addressing, there may be some elements with different hash codes that happen to lie on the probing sequence.



Adding an element is similar. Try finding the element first. If it is not present, add it in the first empty slot in the probing sequence.

Removing an element is trickier. You cannot simply empty the slot at which you find the element. Instead, you must traverse the probing sequence, look for the last element with the same hash code, and move that element into the slot of the removed element. Then rehash all elements that follow until you reach an empty slot (Exercise P15.14).



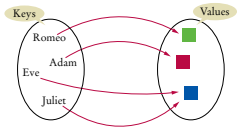
Alternatively, you can replace the removed element with a special "inactive" marker that, unlike an empty slot, does not indicate the end of a probing sequence. When adding another element, you can overwrite an inactive slot (Exercise P15.17).

CHAPTER SUMMARY

Describe the set data type and its implementation in the C++ library.

- A set is an unordered collection of distinct elements.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- Unordered sets are usually more efficient than ordered sets.
- An iterator of an `unordered_set` visits elements in seemingly random order.
- An iterator of an ordered `set` visits elements in increasing order.

Describe the map data type and its implementation in the C++ library.



- A map keeps associations between key and value objects.
- A map iterator yields key/value pairs. If the map is ordered, they are visited in increasing key order.
- A multiset (or bag) is similar to a set, but elements can occur multiple times.
- A multimap can have multiple values associated with the same key.

Understand the implementation of a hash table and the efficiencies of its operations.

- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.
- A hash table uses the hash code to determine where to store each element.
- A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.
- If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or $O(1)$ time.
- Overload `hash<T>::operator()` by combining the hash codes for the data members.
- A class's hash function must be compatible with its equality operator.

REVIEW EXERCISES

- **R15.1** A school web site keeps a collection of web sites that are blocked at student computers. Should the program that checks for blocked sites use a vector, linked list, or set for storing the site addresses?
- **R15.2** A library wants to track which books are checked out to which patrons. Should they use a map or a multimap from books to patrons?
- **R15.3** A library wants to track which patrons have checked out which books. Should they use a map or a multimap from patrons to books?
- **R15.4** How is a map similar to a vector? How is it different?
- **R15.5** An `unordered_set<T>` can be implemented as a hash table whose nodes store data of type `T`. How can you implement an `unordered_map<T>`?
- **R15.6** An `unordered_set<T>` can be implemented as a hash table whose nodes store data of type `T`. How can you implement an `unordered_multiset<T>`?
- **R15.7** What is the difference between a set and a map?
- **R15.8** Can a map have two keys with the same value? Two values with the same key?
- **R15.9** Verify the hash code of the string `"Juliet"` in Table 3.
- **R15.10** Verify that the strings `"VII"` and `"Ugh"` have the same hash code.
- **R15.11** Craig Coder doesn't like the fact that he has to implement a hash function for the objects that he wants to collect in a hash table. "Why not assign a unique ID to each object?" he asks. What is wrong with his idea?
- **R15.12** Craig Coder has another brilliant idea. If a hash function is supposed to produce scrambled values, why not just return a random number? What would happen if he did that with the hash table implementation of Section 15.3?
- **R15.13** Suppose you have an unordered set of strings. What happens if you modify one of the strings in the set, for example by changing the first character?
- **R15.14** What does the `Iterator::next` member function in Section 15.3 do when the iterator is past the end of the hash table?

PRACTICE EXERCISES

- **E15.1** Reimplement the `Polynomial` class of Exercise P14.5 by using an `unordered_map<int, double>` to store the coefficients.
- **E15.2** Write functions

```
unordered_set<int> set_union(unordered_set<int> a, unordered_set set<int> b)
unordered_set<int> intersection(unordered_set<int> a, unordered_set set<int> b)
```

  that compute the set union and intersection of the sets `a` and `b`. (Don't name the first function `union`—that is a reserved word in C++.)
- **E15.3** Implement the *sieve of Eratosthenes:* a method for computing prime numbers, known to the ancient Greeks. This algorithm computes all prime numbers up to n. Choose an n. First insert all numbers from 2 to n into a set. Then erase all multiples

of 2 (except 2); that is, 4, 6, 8, 10, 12, ... Erase all multiples of 3; that is, 6, 9, 12, 15, ... Go up to $\sqrt{n}$. Then print the set.

■■ **E15.4** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

```
Carl: B+
Joe: C
Sarah: A
```

■■ **E15.5** Read all words from a file and add them to a map whose keys are the first letters of the words and whose values are sets of words that start with that same letter. Then print out the word sets in alphabetical order.

■■ **E15.6** Read all words from a file and add them to a map whose keys are word lengths and whose values are comma-separated strings of words of the same length. Then print out those strings, in increasing order by the length of their entries.

■ **E15.7** Write a program that reads text from a file and breaks it up into individual words. Insert the words into a set. At the end of the input file, print all words, followed by the size of the resulting set. This program determines how many unique words a text file has.

■ **E15.8** Insert all words from a large file (such as the novel "War and Peace", which is available on the Internet) into an unordered set and an ordered set. Time the results. Which data structure is more efficient?

■■ **E15.9** Change the `tele.cpp` program in Section 15.2 to a directory that maps names to street addresses. Provide a member function

```
unordered_set<string> Directory::find_names(string address) const
```

that yields all names for a given address.

■■■ **E15.10** Repeat Exercise E15.9. Use an `unordered_multimap` data member that stores all names for a given street address.

■■ **E15.11** Given a `BankAccount` class with data members `string owner` and `double balance`, define a hash function and compatible `operator==` (see Special Topic 15.2).

■■ **E15.12** You keep a set of `Point` objects for a scientific experiment. (A `Point` has x- and y-coordinates.) Define a suitable `operator<` so that you can form a `set<Point>`.

■■ **E15.13** Define `hash<Point>` and a compatible equality operator for the `Point` class of Exercise E15.12

■■ **E15.14** A labeled point has x- and y-coordinates and a string label. Provide a class `LabeledPoint` and define `hash<LabeledPoint>` and a compatible equality operator.

■■ **E15.15** Provide an `operator<` for the `LabeledPoint` class of Exercise E15.14. Sort points first by their x-coordinates. If two points have the same x-coordinate, sort them by their y-coordinates. If two points have the same x- and y-coordinates, sort them by their label. Write a tester program that checks all cases by inserting points into an ordered set.

■■ **E15.16** Reallocate the buckets of the hash table implementation in Section 15.3 when the load factor is greater than 1.0 or less than 0.5, doubling or halving its size. Note that you need to recompute the hash values of all elements.

■ **E15.17** Implement the hash table in Section 15.3, using the "MAD (multiply-add-divide) method" for hash code compression. For that method, you choose a prime number p larger than the length L of the hash table and two values a and b between 1 and $p - 1$. Then reduce h to $|((a\,h + b)\,\%\,p)\,\%\,L|$.

■ **E15.18** Add functions to count collisions to the hash table in Section 15.3 and the one in Exercise E15.17. Insert all words from a dictionary (in `/usr/share/dict/words` or in `words.txt` in your companion code) into both hash table implementations. Does the MAD method reduce collisions? (Use a table size that equals the number of words in the file. Choose p to be the next prime greater than L, $a = 3$, and $b = 5$.)

PROGRAMMING PROJECTS

■■ **P15.1** Write a program that counts how often each word occurs in a text file. Use an `unordered_multiset<string>`.

■■ **P15.2** Repeat Exercise P15.1, but use an `unordered_map<string, int>`.

■■■ **P15.3** Write a program that reads a C++ source file and produces an index of all identifiers in the file. For each identifier, print all line numbers in which the identifier occurs.

■■ **P15.4** Read all words from a list of words and add them to a map whose keys are the phone keypad spellings of the word, and whose values are sets of words with the same code. For example, 26337 is mapped to the set `{ "Andes", "coder", "codes", . . . }`. Then keep prompting the user for numbers and print out all words in the dictionary that can be spelled with that number. In your solution, use a map that maps letters to digits.

© klenger/iStockphoto.

■■■ **P15.5** Reimplement Exercise P15.4 so that the keys of the map are objects of class `Student`. A student should have a first name, a last name, and a unique integer ID. For grade changes and removals, lookup should be by ID. The printout should be sorted by last name. If two students have the same last name, then use the first name as a tie breaker. If the first names are also identical, then use the integer ID. *Hint:* Use two maps.

■■ **P15.6** Try to find two words with the same hash code in a large file. Keep an `unordered_map<int, unordered_set<string>>`. When you read in a word, compute its hash code h and put the word in the set whose key is h. Then iterate through all keys and print the sets whose size is greater than one.

■■ **P15.7** Add a `previous` member function to the hash table iterator in Section 15.3.

■■ **P15.8** Make the `get` and `next` member functions of the hash table iterator in Section 15.3 safe by printing an error message if they are called inappropriately.

■■■ **P15.9** It is unsafe to have multiple iterators to the same hash table if the table is modified through one of them. Implement the following scheme for detecting *concurrent modifications*. Keep a `modcount` data member in the hash table that you increment with each operation that changes the contents. Each iterator also has a `modcount` that

is initialized with the `modcount` of the table. Before any iterator operation, check whether the `modcount` is still the same as that of the underlying table, and print an error message if it is not. Provide a program that demonstrates the error detection. Try the same program with an `unordered_set` to see what happens without the error detection.

■■■ **P15.10** Sometimes, it is advantageous to iterate over hash table elements in the order in which they have been inserted. Implement a `LinkedHashTable` and a corresponding iterator class in which each node has a link to the node that was inserted immediately afterward. Update these links in the `insert` and `erase` member functions.

■■ **P15.11** Turn the hash table class of Section 15.3 into a template. In order to compute the hash code of a value of type `T`, construct a hash function as an object of type `hash<T>`, and then invoke it on the value, as described in Special Topic 15.2.

■■■ **P15.12** Following Worked Example 14.1, turn the hash table template of Exercise P15.11 into an `unordered_set` template with a nested `Iterator` class that overloads operators in the same way as the standard set iterators.

■■■ **P15.13** Implement the copy constructor, assignment operator, and destructor for the hash table class in Section 15.3.

■■■ **P15.14** Implement a hash table with open addressing. When removing an element that is followed by other elements with the same hash code, replace it with the last such element and rehash the remaining elements of the probing sequence. Use an empty string to denote an empty slot. Set a separate flag if an empty string is inserted.

■■■ **P15.15** Modify Exercise P15.14 to use *quadratic probing*. The ith index in the probing sequence is computed as $(h + i^2) \% L$.

■■■ **P15.16** Modify Exercise P15.14 to use *double hashing*. The ith index in the probing sequence is computed as $(h + i\, h_2(k)) \% L$, where k is the original hash key before compression and h_2 is a function mapping integers to non-zero values. A common choice is $h_2(k) = 1 + k \% q$ for a prime q less than L.

■■■ **P15.17** Modify Exercise P15.14 so that you mark removed elements with an "inactive" element. You can't use an empty string—that is already used for empty elements. Instead, use the string `"INACTIVE"` and set a separate flag if that string is inserted.

WORKED EXAMPLE 15.1

Word Frequency

Problem Statement Write a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file.

For example, the following is the beginning of the output that results from processing the book *Alice in Wonderland*:

```
a                653
abide              1
able               1
about             97
above              4
absence            1
absurd             2
```

© Ermin Gutenberger/iStockphoto.

Step 1 Determine how you access the values.

In our case, the values are the word frequencies. We have a frequency value for every word. That is, we want to use a map that maps words to frequencies.

Step 2 Determine the element types of keys and values.

Each word is a `string` and each frequency is an `int`. Therefore, we need a map from string to integer.

Step 3 Determine whether element or key order matters.

We are supposed to print the words in sorted order, so we will use an ordered map, `map<string, int>`.

Step 4 For an ordered collection (that is, a list or vector), determine which operations must be efficient.

We skip this step because we use a map, not an ordered collection.

Step 5 For sets and maps, check whether you need to provide a hash function or comparison operator.

The key type for our map is `string`, and comparison of strings is already defined. Therefore, we need to do nothing further.

Step 6 Having chosen your collection, design your algorithm.

The algorithm for completing our task is fairly simple. Here is the pseudocode:

```
For each word in the input file
   Clean the word (change uppercase to lowercase), remove non-letters.
   If the word is already present in the frequencies map
      Increment the frequency.
   Else
      Set the frequency to 1.
```

Here is the program code:

worked_example_1/wordfreq.cpp

```
/*
   This program prints the frequencies of all words in "Alice in Wonderland".
*/

#include <iostream>
#include <iomanip>
#include <fstream>
#include <map>
#include <string>

using namespace std;

/**
   Changes letters to lowercase and removes characters that are not letters.
   @param s a string
   @return a string with all the letters from s, converted to lowercase
*/
string clean(const string& s)
{
   string r = "";
   for (int i = 0; i < s.length(); i++)
   {
      char c = s[i];
      if ('a' <= c && c <= 'z') // Append lowercase letters
      {
         r = r + c;
      }
      else if ('A' <= c && c <= 'Z') // If uppercase, turn to lowercase
      {
         c = c - 'A' + 'a';
         r = r + c;
      }
   }
   return r;
}

int main()
{
   map<string, int> frequencies;
   fstream in("../../alice.txt");
   string word;
   while (in >> word)
   {
      word = clean(word);

      // Get an iterator to the old frequency count

      auto pos = frequencies.find(word);

      // If there was none, put 1; otherwise, increment the count

      if (pos == frequencies.end())
      {
         frequencies[word] = 1;
```

```
55          }
56          else
57          {
58             pos->second++;
59          }
60       }
61
62       // Print all words and counts
63
64       for (auto p = frequencies.begin(); p != frequencies.end(); p++)
65       {
66          cout << setw(20) << left << p->first << setw(10) << right
67             << p->second << endl;
68       }
69       return 0;
70    }
```

CHAPTER 16

TREE STRUCTURES

© DNY59/iStockphoto.

CHAPTER GOALS

- To study trees and binary trees
- To understand how binary search trees can implement sets
- To learn how red-black trees provide performance guarantees for set operations
- To choose appropriate functions for tree traversal

CHAPTER CONTENTS

In this chapter, we study data structures that organize elements hierarchically, creating arrangements that resemble trees. These data structures offer better performance for adding, removing, and finding elements than the linear structures you have seen so far. You will learn about commonly used tree-shaped structures and study their implementation and performance.

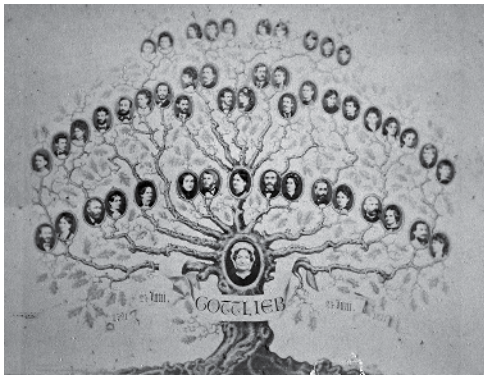
16.1 Basic Tree Concepts

A tree is composed of nodes, each of which can have child nodes.

The root is the node with no parent. A leaf is a node with no children.

In computer science, a **tree** is a hierarchical data structure composed of *nodes*. Each node has a sequence of *child nodes*, and one of the nodes is the *root node*.

Like a linked list, a tree is composed of nodes, but with a key difference. In a linked list, a node can have only one child node, so the data structure is a linear chain of nodes. In a tree, a node can have more than one child. The resulting shape resembles an actual tree with branches. However, in computer science, it is customary to draw trees upside-down, with the root on top (see Figure 1).



Austrian Archives/Imagno/GettyImages, Inc.

A family tree shows the descendants of a common ancestor.

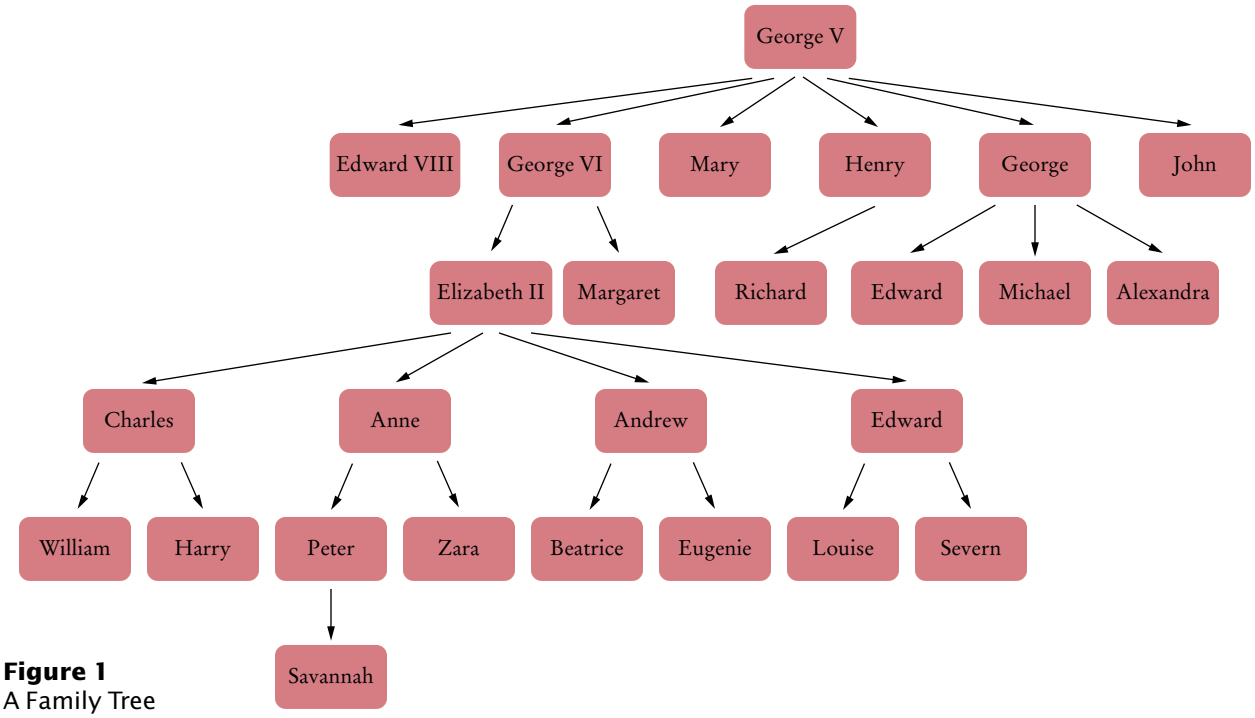
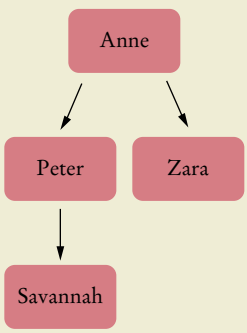


Figure 1
A Family Tree

Trees are commonly used to represent hierarchical relationships. When we talk about nodes in a tree, it is customary to use intuitive words such as roots and leaves, but also parents, children, and siblings—see Table 1 for commonly used terms.

Table 1 Tree Terminology

Term	Definition	Example (using Figure 1)
Node	The building block of a tree: A tree is composed of linked nodes.	This tree has 26 nodes: George V, Edward VIII, ..., Savannah.
Child	Each node has, by definition, a sequence of links to other nodes called its child nodes.	The children of Elizabeth II are Charles, Anne, Andrew, and Edward.
Leaf	A node with no child nodes.	This tree has 16 leaves, including William, Harry, and Savannah.
Interior node	A node that is not a leaf.	George V or George VI, but not Mary.
Parent	If the node c is a child of the node p, then p is a parent of c.	Elizabeth II is the parent of Charles.
Sibling	If the node p has children c and d, then these nodes are siblings.	Charles and Anne are siblings.
Root	The node with no parent. By definition, each tree has one root node.	George V.
Path	A sequence of nodes $c_1, c_2, ..., c_k$ where c_{i+1} is a child of c_i.	Elizabeth II, Anne, Peter, Savannah is a path of length 4.
Descendant	d is a descendant of c if there is a path from c to d.	Peter is a descendant of Elizabeth II but not of Henry.
Ancestor	c is an ancestor of d if d is a descendant of c.	Elizabeth II is an ancestor of Peter, but Henry is not.
Subtree	The subtree rooted at node n is the tree formed by taking n as the root node and including all its descendants.	The subtree with root Anne is 
Height	The number of nodes in the longest path from the root to a leaf. (Some authors define the height to be the number of edges in the longest path, which is one less than the height used in this book.)	The tree in Figure 1 has height 6. The longest path is George V, George VI, Elizabeth II, Anne, Peter, Savannah.

Trees have many applications in computer science; see, for example, Figure 2 and Figure 3.

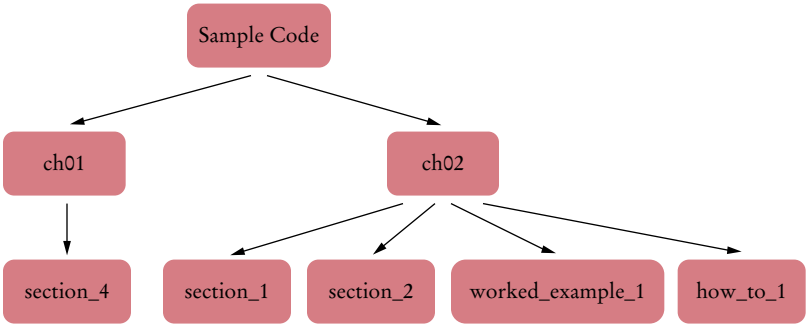


Figure 2 A Directory Tree

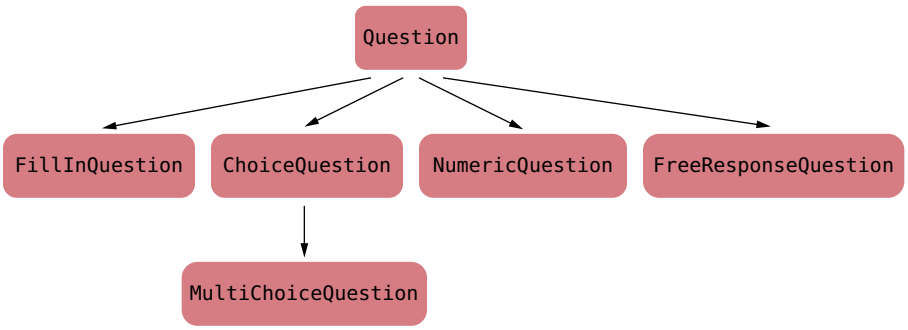


Figure 3 An Inheritance Tree

A tree class uses a node class to represent nodes and has a data member for the root node.

There are multiple ways of implementing a tree. Here we present an outline of a simple implementation that is further explored in Exercise P16.1. A node holds a data item and a vector of pointers to the child nodes. A tree holds a pointer to the root node.

```
class Node
{
private:
   int size() const;
   string data;
   vector<Node*> children;
friend class Tree;
};

class Tree
{
public:
   Tree();
   Tree(string root_data);
   void add_subtree(const Tree& subtree);
   int size() const;
   . . .
private:
   Node* root;
};
```

```
. . .

Tree::Tree(string root_data)
{
   root = new Node;
   root->data = root_data;
}

void Tree::add_subtree(const Tree& subtree)
{
   root->children.push_back(subtree.root);
}
```

Note that, as with linked lists, the `Node` class is considered an implementation detail. Users of the class only work with `Tree` objects.

When computing properties of trees, it is often convenient to use recursion. For example, consider the task of computing the tree *size*, that is, the number of nodes in the tree. Compute the sizes of its subtrees, add them up, and add one for the root.

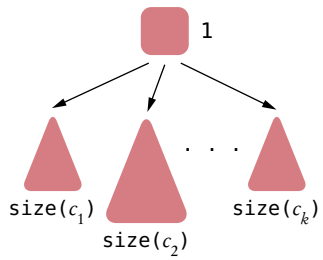
For example, in Table 1, the tree with root node Elizabeth II has four subtrees, with node counts 3, 4, 3, and 3, yielding a count of 1 + 3 + 4 + 3 + 3 = 14 for that tree.

Formally, if r is the root node of a tree, then

$$\texttt{size}(r) = 1 + \texttt{size}(c_1) + ... + \texttt{size}(c_k), \text{ where } c_1 ... c_k \text{ are the children of } r$$

© Yvette Harris/iStockphoto.

When computing tree properties, it is common to recursively visit smaller and smaller subtrees.



Many tree properties are computed with recursive functions.

To implement this size function, first provide a recursive helper:

```
int Node::size() const
{
   int sum = 0;
   for (Node* child : children) { sum = sum + child->size(); }
   return 1 + sum;
}
```

Then call this helper function from a function of the `Tree` class:

```
int Tree::size() const { return root->size(); }
```

It is useful to allow an *empty tree*; a tree whose root node is a null pointer. This is analogous to an empty list—a list with no elements. Because we can't invoke the helper function on a null pointer, we need to refine the `Tree` class's `size` function:

```
int Tree::size() const
{
   if (root == nullptr) { return 0; }
   else { return root->size(); }
}
```

EXAMPLE CODE See sec01 of your companion code for the `Tree` class and the recursive `size` function.

16.2 Binary Trees

A binary tree consists of nodes, each of which has at most two child nodes.

In the following sections, we discuss **binary trees**, trees in which each node has at most two children. As you will see throughout this chapter, binary trees have many very important applications.

In a binary tree, each node has a left and a right child node.

16.2.1 Binary Tree Examples

In this section, you will see several typical examples of binary trees. Figure 4 shows a *decision tree* for guessing an animal from one of several choices. Each non-leaf node contains a question. The left subtree corresponds to a "yes" answer, and the right subtree to a "no" answer.

This is a binary tree because every node has either two children (if it is a decision) or no children (if it is a conclusion). Exercises E16.4 and P16.7 show you how you can build decision trees that ask good questions for a particular data set.

A decision tree contains questions used to decide among a number of options.

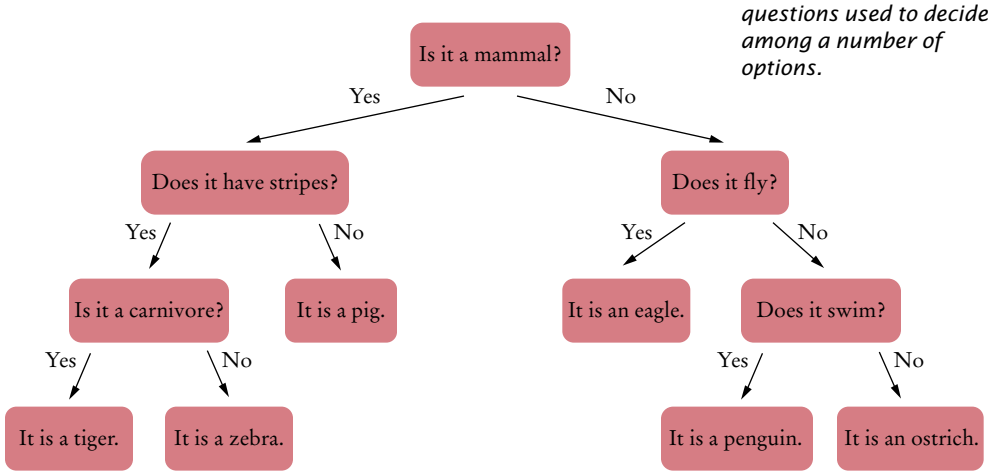


Figure 4 A Decision Tree for an Animal Guessing Game

In a Huffman tree, the left and right turns on the paths to the leaves describe binary encodings.

Another example of a binary tree is a *Huffman tree*. In a Huffman tree, the leaves contain symbols that we want to encode. To encode a particular symbol, walk along the path from the root to the leaf containing the symbol, and produce a zero for every left turn and a one for every right turn. For example, in the Huffman tree of Figure 5, an H is encoded as 0001 and an A as 10. Worked Example 16.1 shows how to build a Huffman tree that gives the shortest codes for the most frequent symbols.

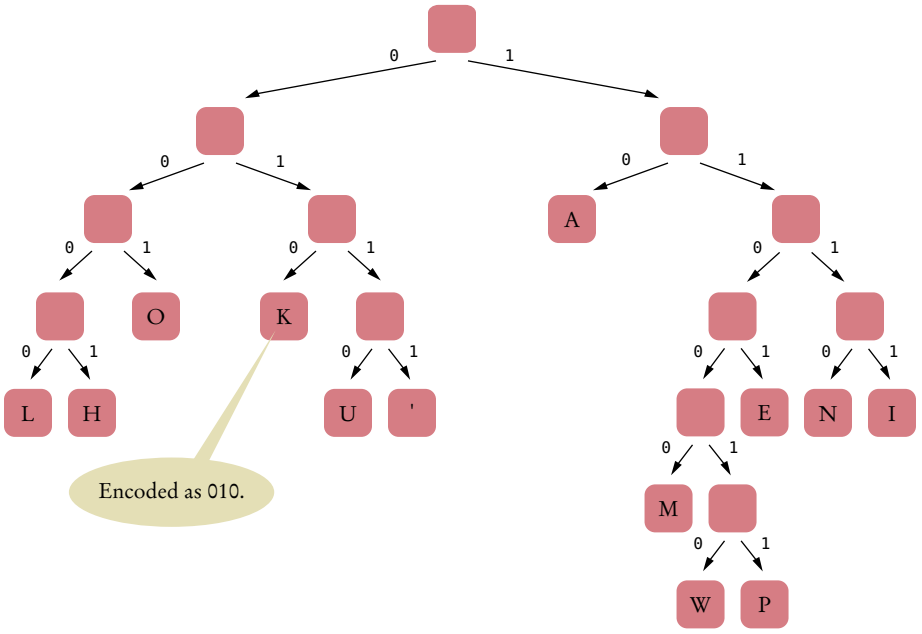


Figure 5 A Huffman Tree for Encoding the Thirteen Characters of Hawaiian Text

An expression tree shows the order of evaluation in an arithmetic expression.

Binary trees are also used to show the evaluation order in arithmetic expressions. For example, Figure 6 shows the trees for the expressions

```
(3 + 4) * 5
3 + 4 * 5
```

The leaves of the expression trees contain numbers, and the interior nodes contain the operators. Provided that each operator has two operands, the tree is binary.

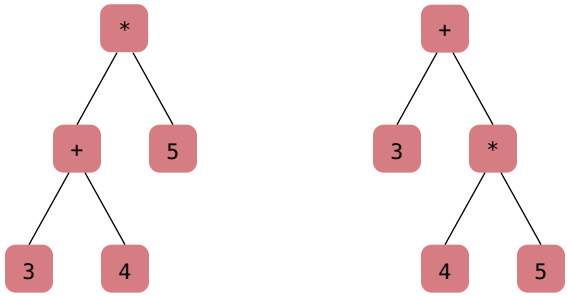


Figure 6 Expression Trees

16.2.2 Balanced Trees

In a balanced tree, all paths from the root to the leaves have approximately the same length.

When we use binary trees to store data, as we will in Section 16.3, we would like to have trees that are **balanced**. In a balanced tree, all paths from the root to one of the leaf nodes have approximately the same length. Figure 7 shows examples of a balanced and an unbalanced tree.

Recall that the height of a tree is the number of nodes in the longest path from the root to a leaf. The trees in Figure 7 have height 5. As you can see, for a given height, a balanced tree can hold more nodes than an unbalanced tree.

We care about the height of a tree because many tree operations proceed along a path from the root to a leaf, and their efficiency is better expressed by the height of the tree than the number of elements in the tree.

© Emrah Turudu/iStockphoto.

In a balanced binary tree, each subtree has approximately the same number of nodes.

A binary tree of height h can have up to $n = 2^h - 1$ nodes. For example, a completely filled binary tree of height 4 has $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ nodes (see Figure 8).

In other words, $h = \log_2(n + 1)$ for a completely filled binary tree. For a balanced tree, we still have $h \approx \log_2 n$. For example, the height of a balanced binary tree with 1,000 nodes is approximately 10 (because $1000 \approx 1024 = 2^{10}$). A balanced binary tree with 1,000,000 nodes has a height of approximately 20 (because $10^6 \approx 2^{20}$). As you will see in Section 16.3, you can find any element in such a tree in about 20 steps. That is a lot faster than traversing the 1,000,000 elements of a linked list.

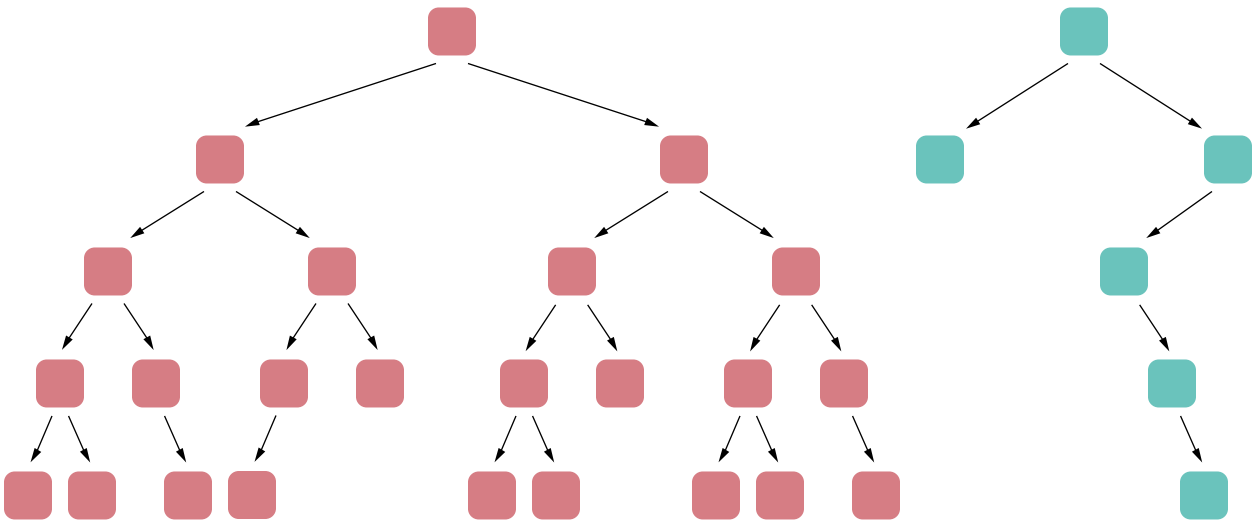


Figure 7 Balanced and Unbalanced Trees

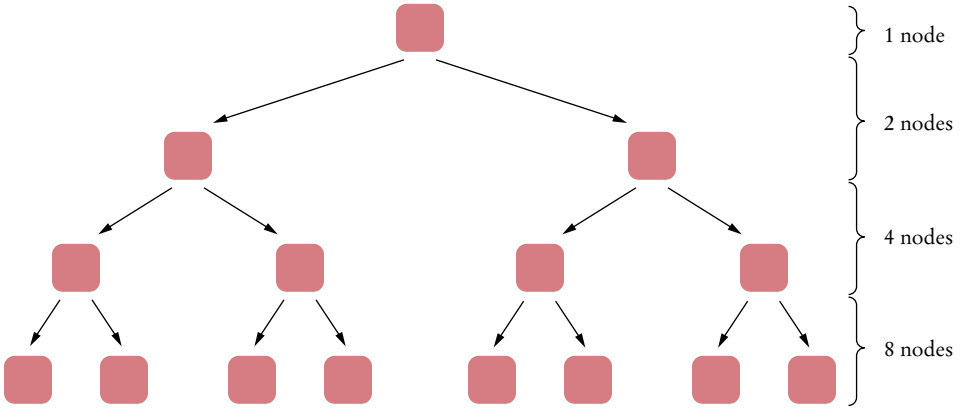


Figure 8 A Completely Filled Binary Tree of Height 4

16.2.3 A Binary Tree Implementation

Every node in a binary tree has pointers to two children, a left child and a right child. Either one may be a null pointer. A node in which both children are `nullptr` is a leaf.

```
class Node
{
private:
   string data;
   Node* left;
   Node* right;
friend class BinaryTree;
};
```

As with linked lists, the nodes are a private implementation detail. We provide a separate class with a public interface for working with tree objects:

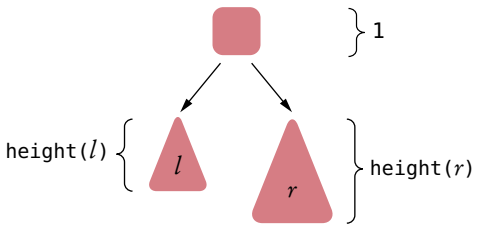
```
class BinaryTree
{
public:
   BinaryTree(); // An empty tree
   BinaryTree(string root_data); // A tree with one node
   BinaryTree(string root_data, BinaryTree left, BinaryTree right);
   . . .
private:
   . . .
   Node* root;
};

BinaryTree::BinaryTree(string root_data, BinaryTree left, BinaryTree right)
{
   root = new Node;
   root->data = root_data;
   root->left = left.root;
   root->right = right.root;
}
```

As with general trees, we often use recursion to define operations on binary trees. Consider computing the height of a tree; that is, the number of nodes in the longest path from the root to a leaf.

To get the height of the tree t, take the larger of the heights of the children and add one, to account for the root.

$$\texttt{height}(t) = 1 + \texttt{max}(\texttt{height}(l), \texttt{height}(r))$$

where l and r are the left and right subtrees.



When we implement this function, we could add a `height` function to the `Node` class. However, nodes can be null pointers and you can't call a function on a null pointer. It is easier to make the recursive helper function a function of the `Tree` class, like this:

```
int BinaryTree::height(const Node* n) const
{
   if (n == nullptr) { return 0; }
   else { return 1 + max(height(n->left), height(n->right)); }
}
```

To get the height of the tree, we provide this public function:

```
int BinaryTree::height() const { return height(root); }
```

Note that there are two `height` functions: a public function with no arguments, returning the height of the tree, and a private recursive helper function, returning the height of a subtree with a given node as its root.

EXAMPLE CODE See sec02 of your companion code for a program that uses a binary tree to implement the animal guessing game in Figure 4.

WORKED EXAMPLE 16.1

Building a Huffman Tree

Learn how to build a Huffman tree for compressing the color data of an image. See your E-Text or visit `wiley.com/go/bclo3`.

Charlotte and Emily Horstmann.

16.3 Binary Search Trees

A set implementation is allowed to rearrange its elements in any way it chooses so that it can find elements quickly. Suppose a set implementation *sorts* its entries. Then it can use **binary search** to locate elements quickly. Binary search takes $O(\log(n))$ steps, where n is the size of the set. For example, binary search in an array of 1,000 elements is able to locate an element in at most 10 steps by cutting the size of the search interval in half in each step.

If we use an array to store the elements of a set in sorted order, inserting or removing an element is an $O(n)$ operation. In the following sections, you will see how

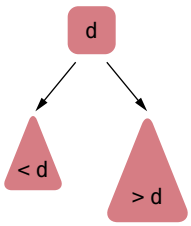
tree-shaped data structures can keep elements in sorted order with more efficient insertion and removal.

16.3.1 The Binary Search Property

All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.

A **binary search tree** is a binary tree in which all nodes fulfill the following property:

- The data values of *all* descendants to the left are less than the data value stored in the node, and *all* descendants to the right have greater data values.



The tree in Figure 9 is a binary search tree.

We can verify the binary search property for each node in Figure 9. Consider the node "Juliet". All descendants to the left have data before "Juliet". All descendants to the right have data after "Juliet". Move on to "Eve". There is a single descendant to the left, with data "Adam" before "Eve", and a single descendant to the right, with data "Harry" after "Eve". Check the remaining nodes in the same way.

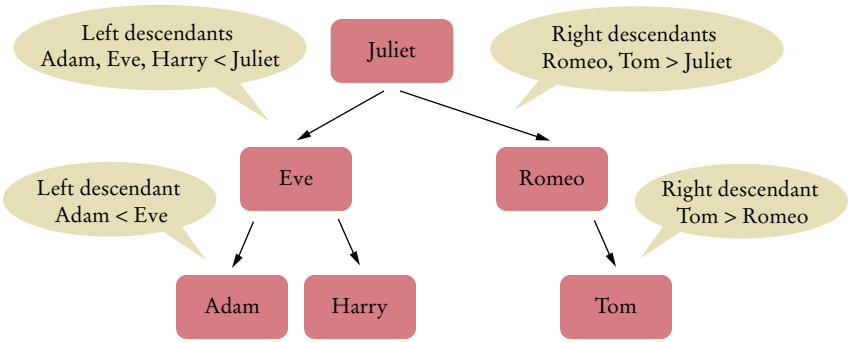


Figure 9
A Binary Search Tree

Figure 10 shows a binary tree that is not a binary search tree. Look carefully—the root node passes the test, but its two children do not.

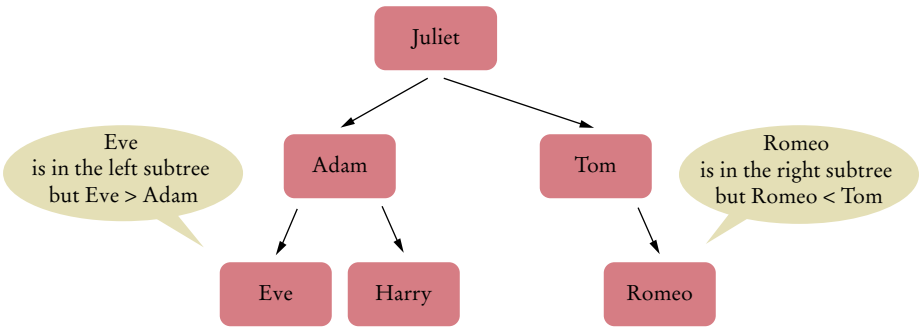


Figure 10
A Binary Tree That Is Not a Binary Search Tree

Here is the definition of the `BinarySearchTree` class that we will implement in the following sections:

```
class BinarySearchTree
{
public:
   BinarySearchTree();
   void insert(string element);
   int count(string element) const;
   void erase(string element);
   void print() const;
private:
   void print(Node* parent) const;
   void add_node(Node* parent, Node* new_node) const;
   Node* root;
};
```

16.3.2 Insertion

To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a `nullptr` node.

To insert data into the tree, use the following algorithm:

- If you encounter a non-null node pointer, look at its data value. If the data value of that node is larger than the value you want to insert, continue the process with the left child. If the node's data value is smaller than the one you want to insert, continue the process with the right child. If the node's data value is the same as the one you want to insert, you are done, because a set does not store duplicate values.
- If you encounter a null node pointer, replace it with the new node.

For example, consider the tree in Figure 11. It is the result of the following statements:

```
BinarySearchTree tree;
tree.insert("Juliet"); 1
tree.insert("Tom");    2
tree.insert("Diana"); 3
tree.insert("Harry");  4
```

We want to insert a new element `Romeo` into it:

```
tree.insert("Romeo"); 5
```

Start with the root node, `Juliet`. `Romeo` comes after `Juliet`, so you move to the right subtree. You encounter the node `Tom`. `Romeo` comes before `Tom`, so you move to the left

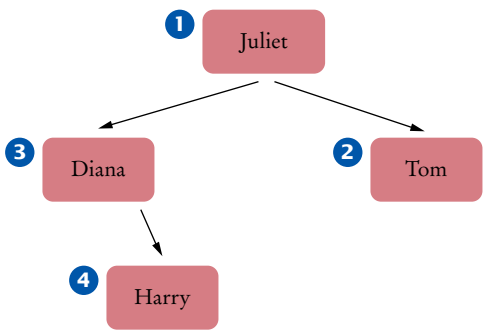


Figure 11 Binary Search Tree After Four Insertions

subtree. But there is no left subtree. Hence, you insert a new `Romeo` node as the left child of `Tom` (see Figure 12).

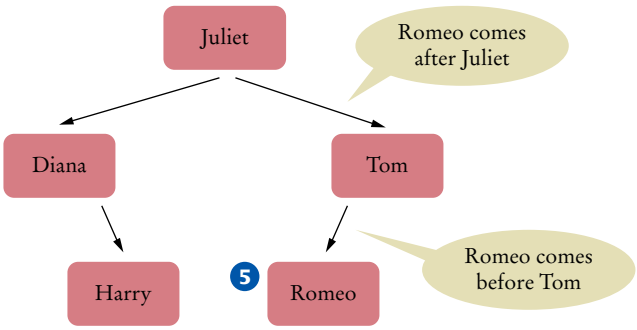


Figure 12 Binary Search Tree After Five Insertions

You should convince yourself that the resulting tree is still a binary search tree. When `Romeo` is inserted, it must end up as a right descendant of `Juliet`—that is what the binary search tree condition means for the root node `Juliet`. The root node doesn't care where in the right subtree the new node ends up. Moving along to `Tom`, the right child of `Juliet`, all it cares about is that the new node `Romeo` ends up somewhere on its left. There is nothing to its left, so `Romeo` becomes the new left child, and the resulting tree is again a binary search tree.

Here is the code for the `insert` function of the `BinarySearchTree` class:

```
void BinarySearchTree::insert(string element)
{
   Node* new_node = new Node;
   new_node->data = element;
   new_node->left = nullptr;
   new_node->right = nullptr;
   if (root == nullptr) { root = new_node; }
   else { add_node(root, new_node); }
}
```

If the tree is empty, simply set its root to the new node. Otherwise, you know that the new node must be inserted somewhere within the nodes, and you can call `add_node` on the root node to perform the insertion. The `add_node` function checks whether the new value is less than the value stored in the node passed to it. If so, the element is inserted in the left subtree; if not, it is inserted in the right subtree:

```
void BinarySearchTree::add_node(Node* parent, Node* new_node) const
{
   if (new_node->data < parent->data)
   {
      if (parent->left == nullptr) { parent->left = new_node; }
      else { add_node(parent->left, new_node); }
   }
   else if (new_node->data > parent->data)
   {
      if (parent->right == nullptr) { parent->right = new_node; }
      else { add_node(parent->right, new_node); }
   }
}
```

Let's trace the calls to `add_node` when inserting `Romeo` into the tree in Figure 11. The first call to `add_node` is

```
add_node(root, new_node)
```

Because `root` points to `Juliet`, you compare `Juliet` with `Romeo` and find that you must call

```
add_node(parent->right, new_node)
```

The node `parent->right` is `Tom`. Compare the data values again (`Tom` vs. `Romeo`) and find that you must now move to the left. Because `parent->right->left` is `nullptr`, set `parent->right->left` to `new_node`, and the insertion is complete (see Figure 12).

Unlike a linked list or an array, and like a hash table, a binary search tree has no *insert positions.* You cannot select the position where you would like to insert an element into a binary search tree. The data structure is *self-organizing;* that is, each element finds its own place.

16.3.3 Removal

We will now discuss the removal algorithm. Our task is to remove a node from the tree. Of course, we must first *find* the node to be removed. That is a simple matter, due to the characteristic property of a binary search tree. Compare the data value to be removed with the data value that is stored in the root node. If it is smaller, keep looking in the left subtree. Otherwise, keep looking in the right subtree.

Let us now assume that we have located the node that needs to be removed. First, let us consider the easiest case. If the node to be removed has no children at all, then the parent link is simply set to `nullptr` (Figure 13).

When the node to be removed has only one child, the situation is still simple (see Figure 14).

When removing a node with only one child from a binary search tree, the child replaces the node to be removed.

To remove the node, simply modify the parent link that points to the node so that it points to the child instead.

The case in which the node to be removed has two children is more challenging. Rather than removing the node, it is easier to replace its data value with the next larger value in the tree. That replacement preserves the binary search tree property. (Alternatively, you could use the largest element of the left subtree—see Exercise P16.4).

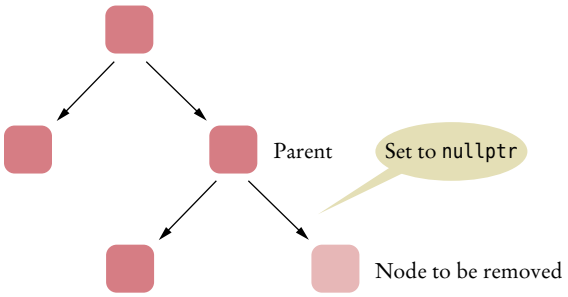


Figure 13 Removing a Node with No Children

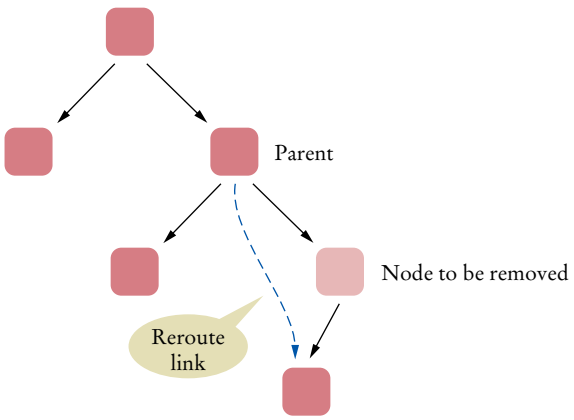


Figure 14 Removing a Node with One Child

When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.

To locate the next larger value, go to the right subtree and find its smallest data value. Keep following the left child links. Once you reach a node that has no left child, you have found the node containing the smallest data value of the subtree. Now remove that node—it is easily removed because it has at most one child to the right. Then store its data value in the original node that was slated for removal. Figure 15 shows the details.

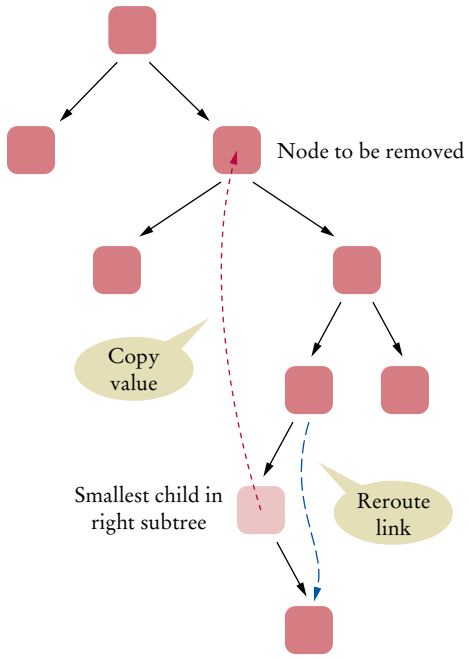


Figure 15 Removing a Node with Two Children

At the end of this section, you will find the source code for the `BinarySearchTree` class. It contains the `insert` and `erase` functions that we just described, a `count` function that tests whether a value is present in a binary search tree, and a `print` function that we will analyze in Section 16.4.

16.3.4 Efficiency of the Operations

In a balanced tree, all paths from the root to the leaves have about the same length.

Now that you have seen the implementation of this data structure, you may well wonder whether it is any good. Like nodes in a linked list, the nodes are allocated one at a time. No existing elements need to be moved when a new element is inserted or removed; that is an advantage. How fast insertion and removal are, however, depends on the shape of the tree. These operations are fast if the tree is **balanced**.

Because the operations of finding, adding, and removing an element process the nodes along a path from the root to a leaf, their execution time is proportional to the height of the tree, and not to the total number of nodes in the tree.

For a balanced tree, we have $h \approx O(\log(n))$. Therefore, inserting, finding, or removing an element is an $O(\log(n))$ operation. On the other hand, if the tree happens to be *unbalanced*, then binary tree operations can be slow—in the worst case, as slow as insertion into a linked list. Table 2 summarizes these observations.

Table 2 Efficiency of Binary Search Tree Operations

Operation	Balanced Binary Search Tree	Unbalanced Binary Search Tree
Find an element.	$O(\log(n))$	$O(n)$
Add an element.	$O(\log(n))$	$O(n)$
Remove an element.	$O(\log(n))$	$O(n)$

If a binary search tree is balanced, then adding, locating, or removing an element takes $O(\log(n))$ time.

If elements are added in fairly random order, the resulting tree is likely to be well balanced. However, if the incoming elements happen to be in sorted order already, then the resulting tree is completely unbalanced. Each new element is inserted at the end, and the entire tree must be traversed every time to find that end!

Binary search trees work well for random data, but if you suspect that the data in your application might be sorted or have long runs of sorted data, you should not use a binary search tree. There are more sophisticated tree structures whose functions keep trees balanced at all times. In these tree structures, one can guarantee that finding, adding, and removing elements takes $O(\log(n))$ time. The standard C++ library uses *red-black trees*, a special form of balanced binary trees, to implement ordered sets and maps. We discuss these data structures in Section 16.5.

sec03/binary_search_tree.h

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include <string>

using namespace std;

class Node
{
private:
   string data;
   Node* left;
   Node*right;
friend class BinarySearchTree;
};

/*
   This class implements a binary search tree whose
   nodes hold strings.
*/
class BinarySearchTree
{
public:
   /**
      Constructs an empty tree.
   */
   BinarySearchTree();

   /**
      Inserts a new node into the tree.
      @param element the element to insert
```

```
32       */
33       void insert(string element);
34
35       /**
36          Tries to find an element in the tree.
37          @param element the element to find
38          @return 1 if the element is contained in the tree
39       */
40       int count(string element) const;
41
42       /**
43          Tries to remove an element from the tree. Does nothing
44          if the element is not contained in the tree.
45          @param element the element to remove
46       */
47       void erase(string element);
48
49       /**
50          Prints the contents of the tree in sorted order.
51       */
52       void print() const;
53
54    private:
55       /**
56          Prints a node and all of its descendants in sorted order.
57          @param parent the root of the subtree to print
58       */
59       void print(Node* parent) const;
60
61       /**
62          Inserts a new node as a descendant of a given node.
63          @param parent the root node
64          @param new_node the node to insert
65       */
66       void add_node(Node* parent, Node* new_node) const;
67
68       Node* root;
69    };
70
71    #endif
```

sec03/binary_search_tree.cpp

```
1  #include <iostream>
2  #include "binary_search_tree.h"
3
4  using namespace std;
5
6  BinarySearchTree::BinarySearchTree()
7  {
8     root = nullptr;
9  }
10
11 void BinarySearchTree::insert(string element)
12 {
13    Node* new_node = new Node;
14    new_node->data = element;
15    new_node->left = nullptr;
16    new_node->right = nullptr;
17    if (root == nullptr) { root = new_node; }
```

```
18    else { add_node(root, new_node); }
19 }
20
21 int BinarySearchTree::count(string element) const
22 {
23    Node* current = root;
24    while (current != nullptr)
25    {
26       if (element < current->data)
27       {
28          current = current->left;
29       }
30       else if (element > current->data)
31       {
32          current = current->right;
33       }
34       else return 1;
35    }
36    return 0;
37 }
38
39 void BinarySearchTree::erase(string element)
40 {
41    // Find node to be removed
42
43    Node* to_be_removed = root;
44    Node* parent = nullptr;
45    bool found = false;
46    while (!found && to_be_removed != nullptr)
47    {
48       if (element == to_be_removed->data)
49       {
50          found = true;
51       }
52       else
53       {
54          parent = to_be_removed;
55          if (element < to_be_removed->data)
56          {
57             to_be_removed = to_be_removed->left;
58          }
59          else
60          {
61             to_be_removed = to_be_removed->right;
62          }
63       }
64    }
65
66    if (!found) { return; }
67
68    // to_be_removed contains element
69
70    // If one of the children is empty, use the other
71
72    if (to_be_removed->left == nullptr || to_be_removed->right == nullptr)
73    {
74       Node* new_child;
75       if (to_be_removed->left == nullptr)
76       {
77          new_child = to_be_removed->right;
```

```
78          }
79          else
80          {
81             new_child = to_be_removed->left;
82          }
83
84          if (parent == nullptr) // Found in root
85          {
86             root = new_child;
87          }
88          else if (parent->left == to_be_removed)
89          {
90             parent->left = new_child;
91          }
92          else
93          {
94             parent->right = new_child;
95          }
96          return;
97       }
98
99       // Neither subtree is empty
100
101      // Find smallest element of the right subtree
102
103      Node* smallest_parent = to_be_removed;
104      Node* smallest = to_be_removed->right;
105      while (smallest->left != nullptr)
106      {
107         smallest_parent = smallest;
108         smallest = smallest->left;
109      }
110
111      // smallest contains smallest child in right subtree
112
113      // Move contents, unlink child
114
115      to_be_removed->data = smallest->data;
116      if (smallest_parent == to_be_removed)
117      {
118         smallest_parent->right = smallest->right;
119      }
120      else
121      {
122         smallest_parent->left = smallest->right;
123      }
124   }
125
126   void BinarySearchTree::print() const
127   {
128      print(root);
129      cout << endl;
130   }
131
132   void BinarySearchTree::print(Node* parent) const
133   {
134      if (parent == nullptr) { return; }
135      print(parent->left);
136      cout << parent->data << " ";
137      print(parent->right);
```

```
138 }
139
140 void BinarySearchTree::add_node(Node* parent, Node* new_node) const
141 {
142    if (new_node->data < parent->data)
143    {
144       if (parent->left == nullptr) { parent->left = new_node; }
145       else { add_node(parent->left, new_node); }
146    }
147    else if (new_node->data > parent->data)
148    {
149       if (parent->right == nullptr) { parent->right = new_node; }
150       else { add_node(parent->right, new_node); }
151    }
152 }
```

sec03/treedemo.cpp

```
1  #include <iostream>
2  #include "binary_search_tree.h"
3
4  using namespace std;
5
6  /*
7     This program tests the binary search tree class.
8  */
9  int main()
10 {
11    BinarySearchTree t;
12    t.insert("D");
13    t.insert("B");
14    t.insert("A");
15    t.insert("C");
16    t.insert("F");
17    t.insert("E");
18    t.insert("I");
19    t.insert("G");
20    t.insert("H");
21    t.insert("J");
22    t.erase("A"); // Removing leaf
23    t.erase("B"); // Removing element with one child
24    t.erase("F"); // Removing element with two children
25    t.erase("D"); // Removing root
26    t.print();
27    cout << "Expected: C E G H I J" << endl;
28    cout << t.count("A") << " " << t.count("J") << endl;
29    cout << "Expected: 0 1" << endl;
30    return 0;
31 }
```

16.4 Tree Traversal

We often want to visit all elements in a tree. There are many different orderings in which one can visit, or *traverse*, the tree elements. The following sections introduce the most common ones.

16.4.1 Inorder Traversal

Suppose you inserted a number of data values into a binary search tree. What can you do with them? It turns out to be surprisingly simple to print all elements in sorted order. You *know* that all data in the left subtree of any node must come before the root node and before all data in the right subtree. That is, the following algorithm will print the elements in sorted order:

Print the left subtree.
Print the root data.
Print the right subtree.

To visit all elements in a tree, visit the root and recursively visit the subtrees.

Let's try this out with the tree in Figure 12 on page 531. The algorithm tells us to

1. Print the left subtree of `Juliet`; that is, `Diana` and descendants.
2. Print `Juliet`.
3. Print the right subtree of `Juliet`; that is, `Tom` and descendants.

How do you print the subtree starting at `Diana`?

1. Print the left subtree of `Diana`. There is nothing to print.
2. Print `Diana`.
3. Print the right subtree of `Diana`, that is, `Harry`.

That is, the left subtree of `Juliet` is printed as

```
Diana Harry
```

The right subtree of `Juliet` is the subtree starting at `Tom`. How is it printed? Again, using the same algorithm:

1. Print the left subtree of `Tom`, that is, `Romeo`.
2. Print `Tom`.
3. Print the right subtree of `Tom`. There is nothing to print.

Thus, the right subtree of `Juliet` is printed as

```
Romeo Tom
```

Now put it all together: the left subtree, `Juliet`, and the right subtree:

```
Diana Harry Juliet Romeo Tom
```

The tree is printed in sorted order.

It is very easy to implement this `print` function. We start with a recursive helper function:

```
void BinarySearchTree::print(Node* parent) const
{
   if (parent == nullptr) { return; }
   print(parent->left);
   cout << parent->data << " ";
   print(parent->right);
}
```

To print the entire tree, start this recursive printing process at the root:

```
void BinarySearchTree::print() const
{
   print(root);
```

```
      cout << endl;
   }
```

This visitation scheme is called *inorder traversal* (visit the left subtree, the root, the right subtree). There are two related traversal schemes, called *preorder traversal* and *postorder traversal*, which we discuss in the next section.

16.4.2 Preorder and Postorder Traversals

We distinguish between preorder, inorder, and postorder traversal.

In Section 16.4.1, we visited a binary tree in order: first the left subtree, then the root, then the right subtree. By modifying the visitation rules, we obtain other traversals.

In preorder traversal, we visit the root *before* visiting the subtrees, and in postorder traversal, we visit the root *after* the subtrees.

Preorder(n)
Visit n.
For each child c of n
* Preorder(c).*

Postorder(n)
For each child c of n
* Postorder(c).*
Visit n.

© Pawel Gaul/iStockphoto.

When visiting all nodes of a tree, one needs to choose a traversal order.

These two visitation schemes will not print a binary search tree in sorted order. However, they are important in other applications. Here is an example.

In Section 16.2, you saw trees for arithmetic expressions. Their leaves store numbers, and their interior nodes store operators. The expression trees describe the order in which the operators are applied. Let's apply postorder traversal to the expression trees in Figure 6.

The first tree yields

```
3 4 + 5 *
```

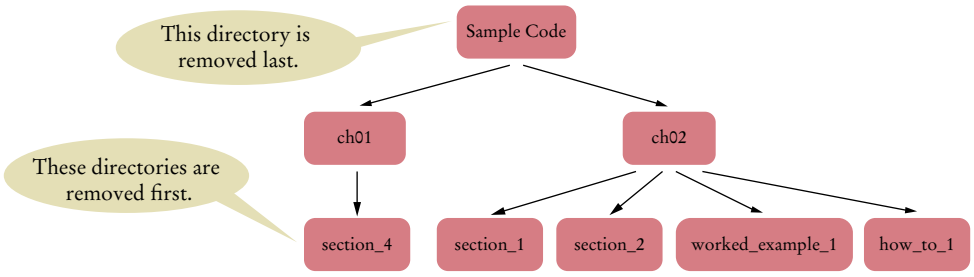
whereas the second tree yields

```
3 4 5 * +
```

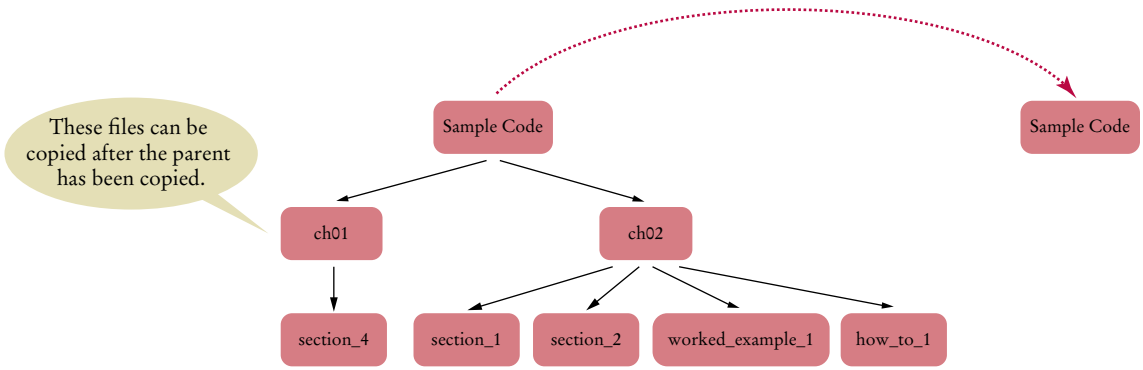
Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.

You can interpret the traversal result as an expression in "reverse Polish notation" (see Special Topic 14.1), or equivalently, instructions for a stack-based calculator (see Section 14.6.2).

Here is another example of the importance of traversal order. Consider the task of removing all directories from a directory tree such as the following, with the restriction that you can only remove a directory when it contains no other directories. In this case, you use a postorder traversal.



Conversely, if you want to copy the directory tree, you start copying the root, because you need a target directory into which to place the children. This calls for preorder traversal.



Note that pre- and postorder traversal can be defined for any trees, not just binary trees (see the sample code for this section). However, inorder traversal makes sense only for binary trees.

16.4.3 The Visitor Pattern

In the preceding sections, we simply printed each tree node that we visited. Often, we want to process the nodes in some other way. There is a "design pattern" (that is, a standard recipe to a commonly occurring problem), called the *visitor pattern*, that provides a solution. Define a class

```
class Visitor
{
public:
   virtual void visit(string data);
};
```

The `preorder` function receives an object of this class type, and calls its `visit` function:

```
void Tree::preorder(Visitor &v) const { preorder(root, v); }

void Tree::preorder(Node* n, Visitor& v) const
{
   if (n == nullptr) { return; }
   v.visit(n->data);
   for (Node* c : n->children) { preorder(c, v); }
}
```

Functions for postorder and, for a binary tree, inorder traversals can be implemented in the same way.

Let's say we want to count short names (with at most five letters). The following visitor will do the job.

```
class ShortNameCounter : public Visitor
{
public:
   void visit(string data);
   int get() const;
```

```
private:
   int counter = 0;
};

void ShortNameCounter::visit(string data)
{
   if (data.length() <= 5) { counter++; }
}

int ShortNameCounter::get() const
{
   return counter;
}

ShortNameCounter v2;
t1.preorder(v2);
cout << "Short names: " << v2.get() << endl;
```

Here, the visitor object accumulates the count. After the visit is complete, we can obtain the result.

16.4.4 Depth-First and Breadth-First Search

The traversals in the preceding sections are expressed using recursion. If you want to process the nodes of a tree, you supply a visitor, which is applied to all nodes. Sometimes, it is useful to use an iterative approach instead. Then you can stop processing nodes when a goal has been met.

To visit the nodes of a tree iteratively, we replace the recursive calls with a stack that keeps track of the children that need to be visited. Here is the algorithm:

Push the root node on a stack.
While the stack is not empty
  Pop the stack; let n be the popped node.
  Process n.
  Push the children of n on the stack, starting with the last one.

© David Jones/iStockphoto.

In a depth-first search, one moves as quickly as possible to the deepest nodes of the tree.

Depth-first search uses a stack to track the nodes that it still needs to visit.

This algorithm is called *depth-first search* because it goes deeply into the tree and then backtracks when it reaches the leaves (see Figure 16). Note that the tree can be an arbitrary tree—it need not be binary.

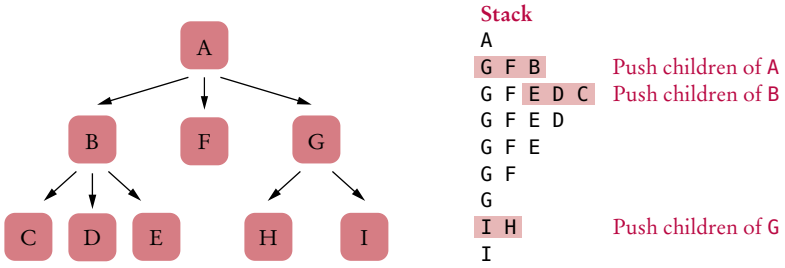


Figure 16 Depth-First Search

We push the children on the stack in right-to-left order so that the visit starts with the leftmost path. In this way, the nodes are visited in preorder. If the leftmost child had been pushed first, we would still have a depth-first search, just in a less intuitive order.

Breadth-first search first visits all nodes on the same level before visiting the children.

If we replace the stack with a queue, the visitation order changes. Instead of going deeply into the tree, we first visit all nodes at the same level before going on to the next level. This is called *breadth-first search* (Figure 17).

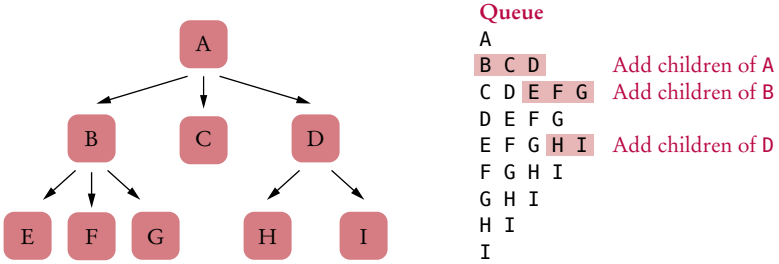


Figure 17 Breadth-First Search

Here is the implementation of the breadth first algorithm:

```
void Tree::breadth_first(Visitor& v) const
{
   if (root == nullptr) { return; }
   queue<Node*> q;
   q.push(root);
   while (q.size() > 0)
   {
      Node* n = q.front();
      q.pop();
      v.visit(n->data);
      for (Node* c : n->children) { q.push(c); }
   }
}
```

For depth-first search, replace the queue with a stack (Exercise E16.10). Exercise E16.12 shows how to stop the search before all elements have been visited.

16.4.5 Tree Iterators

The C++ standard library uses trees to implement sets, and iterators to process the elements, like this:

```
set<string> t = . . .
set<string>::iterator iter = t.begin();
string first = *iter;
iter++;
. . .
```

It is easy to implement such an iterator with depth-first or breadth-first search. Make the stack or queue into a data member of the iterator object. The `next` function executes one iteration of the loop that you saw in the last section.

```
class BreadthFirstIterator
{
```

```
public:
   BreadthFirstIterator(Node* root);
   . . .
   void next();
   . . .
private:
   queue<Node*> q;
};

BreadthFirstIterator::BreadthFirstIterator(Node* root)
{
   if (root != nullptr) { q.push(root); }
}

void BreadthFirstIterator::next()
{
   Node* n = q.front();
   q.pop();
   for (Node* c : n->children) { q.push(c); }
}
```

Note that there is no `visit` function. The user of the iterator receives the node data, processes it, and decides whether to call `next` again.

This iterator produces the nodes in breadth-first order. For a binary search tree, one would want the nodes in sorted order instead. Exercise P16.9 shows how to implement such an iterator.

EXAMPLE CODE See sec04 of your companion code for a program that demonstrates preorder, postorder, and breadth-first traversal in a tree.

16.5 Red-Black Trees

As you saw in Section 16.3, insertion and removal in a binary search tree are $O(\log(n))$ operations *provided that the tree is balanced*. In this section, you will learn about **red-black trees**, a special kind of binary search tree that rebalances itself after each insertion or removal. With red-black trees, we can guarantee efficiency of these operations. In fact, the standard C++ `set` and `map` classes also use red-black trees.

16.5.1 Basic Properties of Red-Black Trees

In a red-black tree, node coloring rules ensure that the tree is balanced.

A red-black tree is a binary search tree with the following additional properties:

- Every node is colored red or black.
- The root is black.
- A red node cannot have a red child (the "no double reds" rule).
- All paths from the root to a `nullptr` have the same number of black nodes (the "equal exit cost" rule).

Of course, the nodes aren't actually colored. Each node simply has a flag to indicate whether it is considered red or black. (The choice of these colors is traditional; one could have equally well used some other attributes. Perhaps, in an alternate universe, students learn about chocolate-vanilla trees.)

Figure 18 shows an example of a red-black tree.

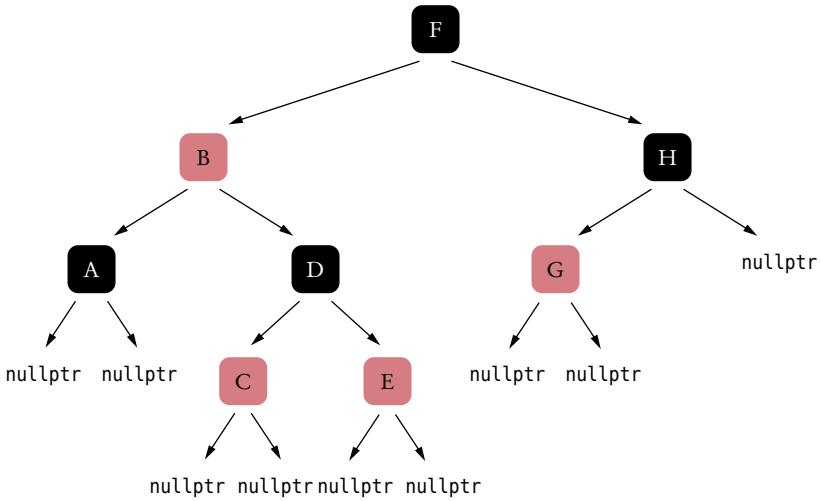


Figure 18 A Red-Black Tree

Instead of thinking of the colors, imagine each node to be a toll booth. As you travel from the root to a `nullptr` (an exit), you have to pay $1 at each black toll booth, but the red toll booths are free. The cost of traveling along a path is the sum of the charges; that is, the number of black nodes on the path. The "equal exit cost" rule says that the cost of any trip that starts at the root and ends at an exit is the same, no matter which exit you choose.

© Virginia N/iStockphoto.

Think of each node of a red-black tree as a toll booth. The total toll to each exit is the same.

Figures 19 and 20 show examples of trees that violate the "equal exit cost" and "no double reds" rules.

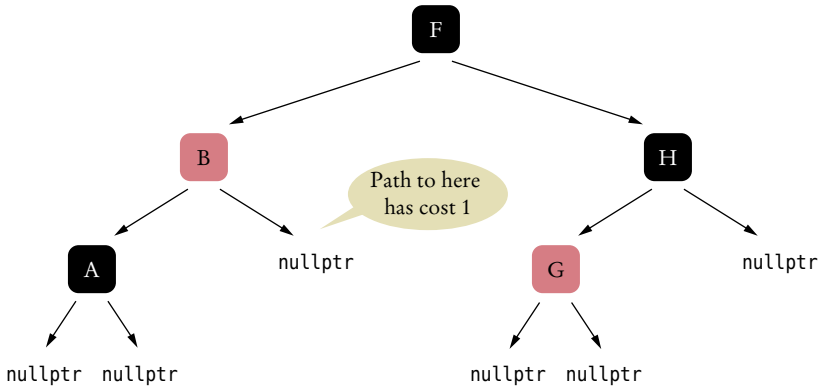


Figure 19 A Tree that Violates the "Equal Exit Cost" Rule

Note that the "equal exit cost" rule does not just apply to paths that end in a leaf, but to any path from the root to a node with one or two empty children. For example, in Figure 19, the path F–B violates the equal exit cost, yet B is not a leaf.

The "equal exit cost" rule eliminates highly unbalanced trees. You can't have null pointers high up in the tree. In other words, the nodes that aren't near the leaves need to have two children.

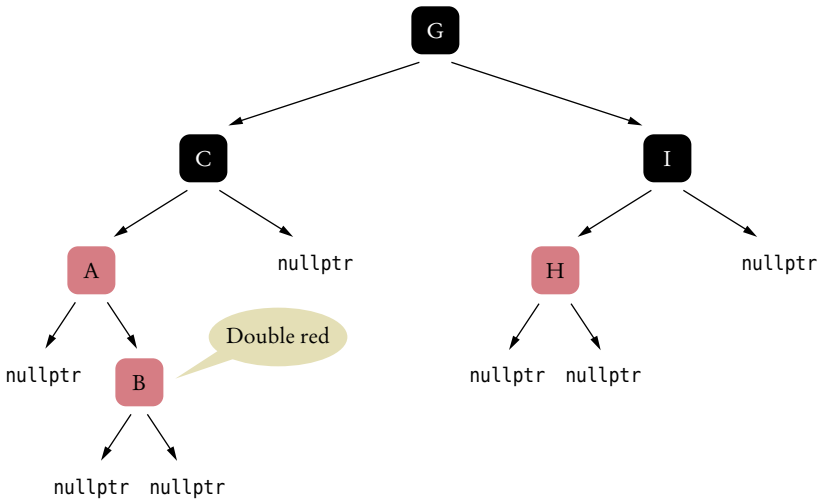


Figure 20 A Tree that Violates the "No Double Red" Rule

The "no double reds" rule gives some flexibility to add nodes without having to restructure the tree all the time. Some paths from the root to a `nullptr` can be a bit longer than others—by alternating red and black nodes—but none can be longer than twice the exit cost.

The cost of traveling on a path from a given node to a `nullptr` (that is, the number of black nodes on the path), is called the *black height* of the node. The cost of traveling from the root to a `nullptr` is called the black height of the tree.

A tree with given black height bh can't be too sparse—it must have at least $2^{bh} - 1$ nodes (see Exercise R16.18). Or, if we turn this relationship around,

$$2^{bh} - 1 \leq n$$

$$2^{bh} \leq n + 1$$

$$bh \leq \log_2(n + 1)$$

The "no double reds" rule says that the total height h of a tree is at most twice the black height:

$$h \leq 2 \cdot bh \leq 2 \cdot \log_2(n + 1)$$

Therefore, traveling from the root to a `nullptr` is $O(\log(n))$.

16.5.2 Insertion

To insert a new node into a red-black tree, first insert it as you would into a regular binary search tree (see Section 16.3.2). Note that the new node is a leaf.

If it is the first node of the tree, it must be black. Otherwise, color it red. If its parent is black, we still have a red-black tree, and we are done.

To rebalance a red-black tree after inserting an element, fix all double-red violations.

However, if the parent is also red, we have a "double red" and need to fix it by changing the color of one or more nodes. Because the rest of the tree is a proper red-black tree, we know that the grandparent is black.

There are four possible configurations of a "double red", shown in Figure 21.

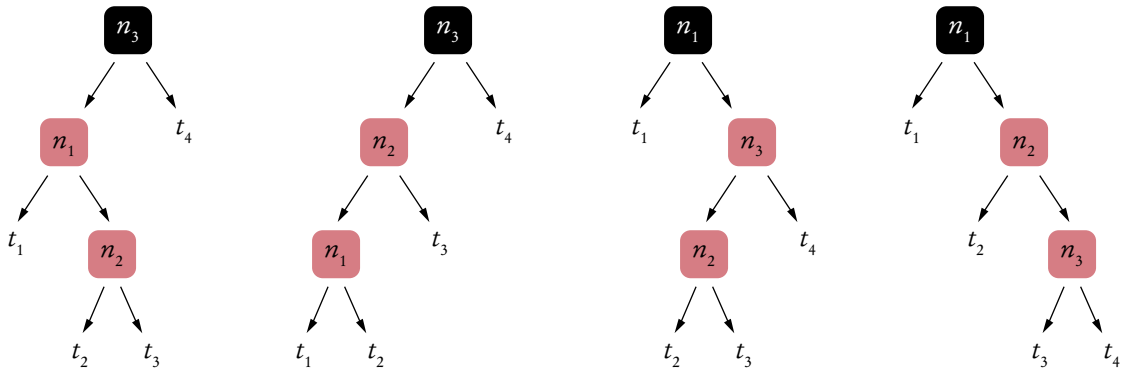


Figure 21 The Four Possible Configurations of a "Double Red"

Of course, our tree is a binary search tree, and we will now take advantage of that fact. In each tree of Figure 21, we labeled the smallest, middle, and largest of the three nodes as n_1, n_2, and n_3. We also labeled their children in sorted order, starting with t_1. To fix the "double red", rearrange the three nodes as shown in Figure 22, keeping their data values, but updating their left and right pointers.

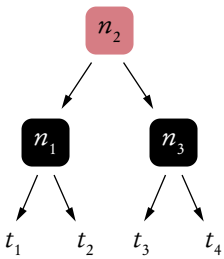


Figure 22 Fixing the "Double Red" Violation

Because the fix preserves the sort order, the result is a binary search tree. The fix does not change the number of black nodes on a path. Therefore, it preserves the "equal exit cost" rule.

If the parent of n_2 is black, we get a red-black tree, and we are done. If that parent is red, we have another "double red", but it is one level closer to the root. In that case, fix the double-red violation of n_2 and its parent. You may have to continue fixing double-red violations, moving closer to the root each time. If the red parent is the root, simply turn it black. This increments all path costs, preserving the "equal exit cost" rule.

Worked Example 16.2 has an implementation of this algorithm.

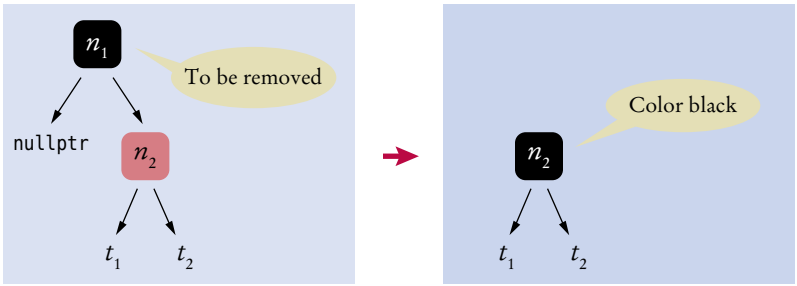
We can determine the efficiency with more precision than we were able to in Section 16.5.1. To find the insertion location requires at most h steps, where h is the height of the tree. To fix the "double red" violations takes at most $h/2$ steps. (Look carefully at Figures 21 and 22 to see that each fix pushes the violation up *two* nodes. If the top node of each subtree in Figure 21 has height t, then the nodes of the double-red violation have heights $t + 1$ and $t + 2$. In Figure 22, the top node also has height t. If there is a double-red violation, it is between that node and its parent at height $t - 1$.) We know from Section 16.5.1 that $h = O(\log(n))$. Therefore, insertion into a red-black tree is guaranteed to be $O(\log(n))$.

16.5.3 Removal

To remove a node from a red-black tree, you first use the removal algorithm for binary search trees (Section 16.3.3). Note that in that algorithm, the removed node has at most one child. We never remove a node with two children; instead, we fill it with the value of another node with at most one child and remove that node.

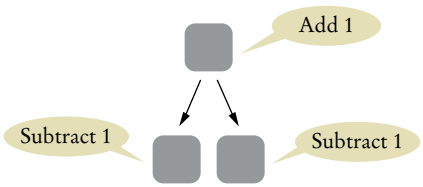
Two cases are easy. First, if the node to be removed is red, there is no problem with the removal—the resulting tree is still a red-black tree.

Next, assume that the node to be removed has a child. Because of the "equal exit cost" rule, the child must be red. Simply remove the parent and color the child black.



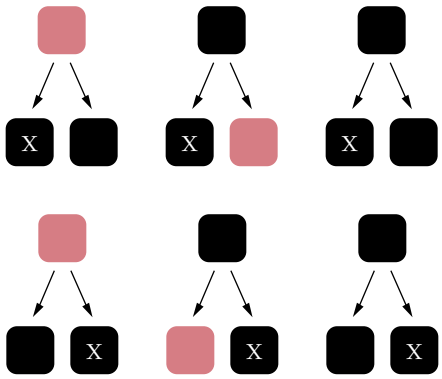
The troublesome case is the removal of a black leaf. We can't just remove it because the exit cost to the `nullptr` replacing it would be too low. Instead, we'll first turn it into a red node.

To turn a black node into a red one, we will temporarily "bubble up" the "toll charges", raising the charge of the parent by 1 and lowering the charge of the children by 1. (Recall that the charge is 1 for a black node and `0` for a red node.)



This process leaves all path costs unchanged, and it turns the black leaf into a red one which we can safely remove.

Now consider a black leaf that is to be removed. Because of the equal-exit rule, it must have a sibling. The sibling and the parent can be black or red, but they can't both be red. The leaf to be removed can be to the left or to the right. The figure below shows all possible cases.



In the first column, bubbling up will work perfectly—it simply turns the red node into a black one and the black ones into red ones. One of the red ones is removed. The other may cause a double-red violation with one of its children, which we fix if necessary.

Before removing a node in a red-black tree, turn it red and fix any double-black and double-red violations.

But in the other cases, a new problem arises. Adding 1 to a black parent yields a charge of 2, which we call *double-black*. Subtracting 1 from a red child yields a *negative-red* node with a charge of –1. These are not valid nodes in a red-black tree, and we need to eliminate them.

A negative-red node is always below a double-black one, and the pair can be eliminated by the transformation shown in Figure 23.

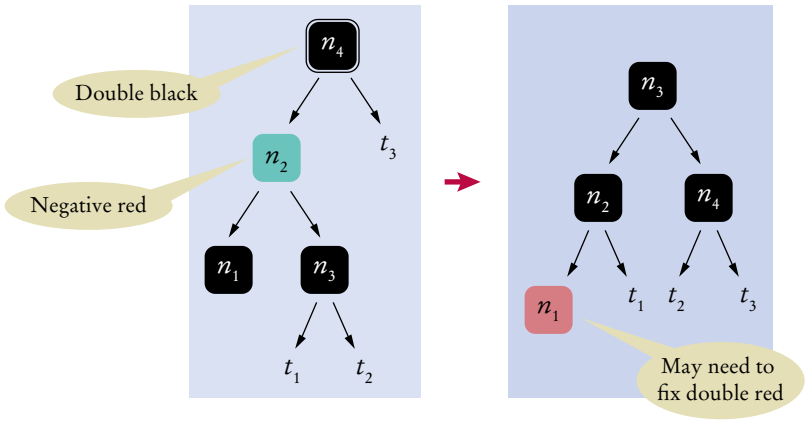


Figure 23 Eliminating a Negative-Red Node with a Double-Black Parent

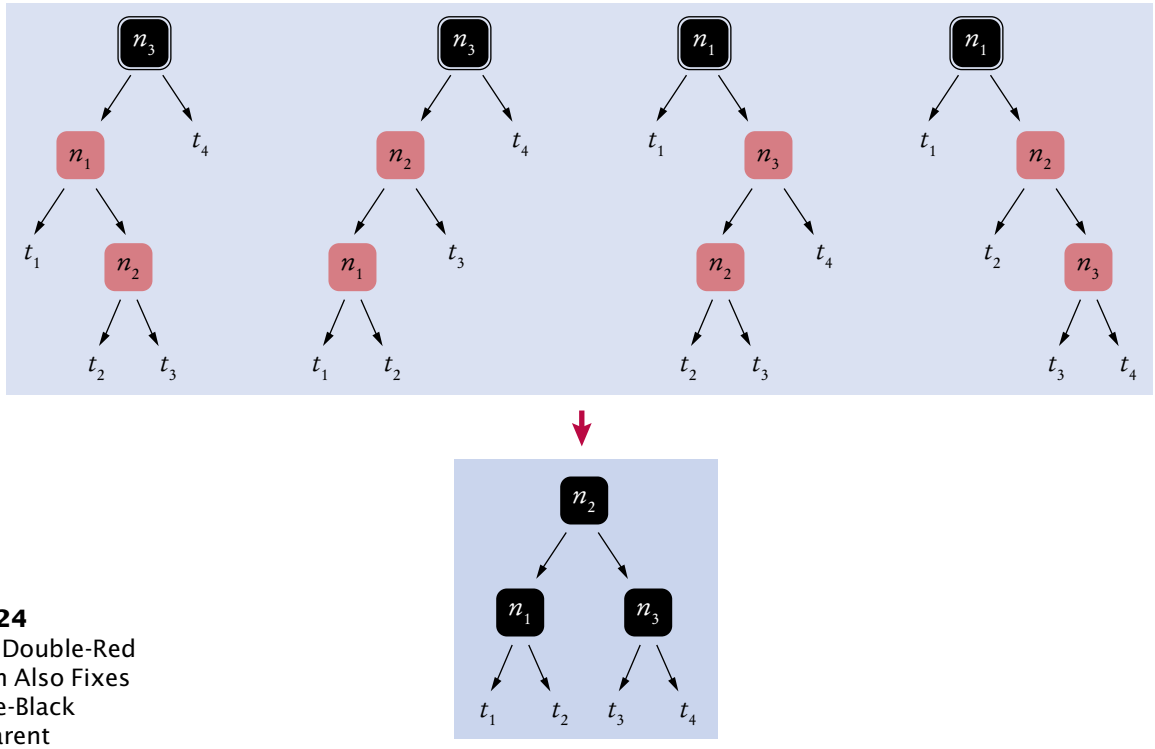


Figure 24
Fixing a Double-Red Violation Also Fixes a Double-Black Grandparent

Sometimes, the creation of a double-black node also causes a double-red violation below. We can fix the double-red violation as in the preceding section, but now we color the middle node black instead of red—see Figure 24.

To see that this transformation is valid, imagine a trip through one of the node sequences in Figure 24 from the top node to one of the trees below. The cost of that portion of the trip is 2 for each tree, both before and after the transformation.

Sometimes, neither of the two transformations applies, and then we need to "bubble up" again, which pushes the double-black node closer to the root. Figure 25 shows the possible cases.

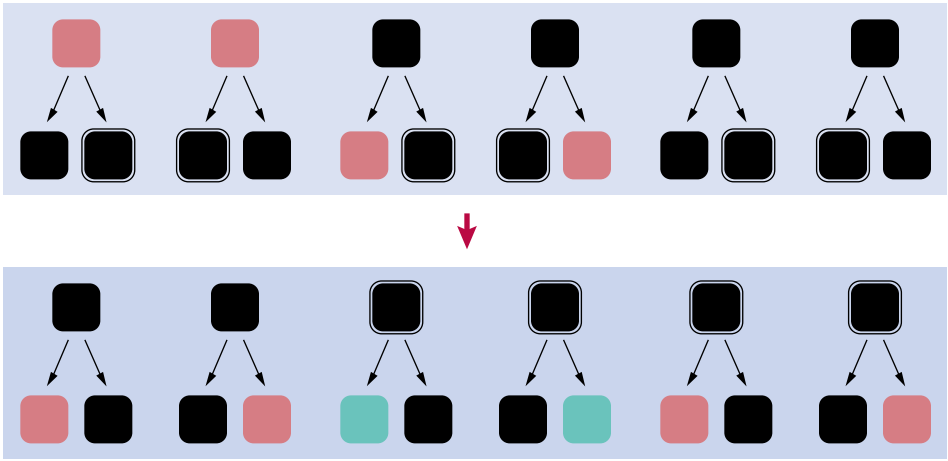


Figure 25
Bubbling Up a Double-Black Node

If the double-black node reaches the root, we can replace it with a regular black node. This reduces the cost of all paths by 1 and preserves the "equal exit cost" rule.

See Worked Example 16.2 for an implementation of node removal.

Adding or removing an element in a red-black tree is an $O(\log(n))$ operation.

Let us now determine the efficiency of this process. Removing a node from a binary search tree requires $O(h)$ steps, where h is the height of the tree. The double-black node may bubble up, perhaps all the way to the root. Bubbling up will happen at most h times, and its cost is constant—it only involves changing the costs of three nodes. If we generate a negative red, we remove it (as shown in Figure 23), and the bubbling stops. We may have to fix one double-red violation, which takes $O(h)$ steps. It is also possible that bubbling creates a double-red violation, but its fix will absorb the double-black node, and bubbling also stops. The entire process takes $O(h)$ steps. Because $h = O(\log(n))$, removal from a red-black tree is also guaranteed to be $O(\log(n))$.

Table 3 Efficiency of Red-Black Tree Operations

Find an element.	$O(\log(n))$
Add an element.	$O(\log(n))$
Remove an element.	$O(\log(n))$

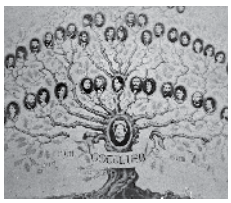
WORKED EXAMPLE 16.2

Implementing a Red-Black Tree

Learn how to implement a red-black tree as described in Section 16.5. See your E-Text or visit `wiley.com/go/bclo3`.

CHAPTER SUMMARY

Describe and implement general trees.



- A tree is composed of nodes, each of which can have child nodes.
- The root is the node with no parent. A leaf is a node with no children.
- A tree class uses a node class to represent nodes and has a data member for the root node.
- Many tree properties are computed with recursive functions.

Describe binary trees and their applications.

- A binary tree consists of nodes, each of which has at most two child nodes.
- In a Huffman tree, the left and right turns on the paths to the leaves describe binary encodings.
- An expression tree shows the order of evaluation in an arithmetic expression.
- In a balanced tree, all paths from the root to the leaves have approximately the same length.

Explain the implementation of a binary search tree and its performance characteristics.

- All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.
- To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a `nullptr` node.
- When removing a node with only one child from a binary search tree, the child replaces the node to be removed.
- When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.
- In a balanced tree, all paths from the root to the leaves have about the same length.
- If a binary search tree is balanced, then adding, locating, or removing an element takes $O(\log(n))$ time.

Describe preorder, inorder, and postorder tree traversal.

- To visit all elements in a tree, visit the root and recursively visit the subtrees.
- We distinguish between preorder, inorder, and postorder traversal.
- Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.
- Depth-first search uses a stack to track the nodes that it still needs to visit.
- Breadth-first search first visits all nodes on the same level before visiting the children.

Describe how red-black trees provide guaranteed $O(\log(n))$ operations.

- In a red-black tree, node coloring rules ensure that the tree is balanced.
- To rebalance a red-black tree after inserting an element, fix all double-red violations.
- Before removing a node in a red-black tree, turn it red and fix any double-black and double-red violations.
- Adding or removing an element in a red-black tree is an $O(\log(n))$ operation.

REVIEW EXERCISES

■ **R16.1** What are all possible shapes of trees of height h with one leaf? Of height 2 with k leaves?

■■ **R16.2** Describe a recursive algorithm for finding the maximum number of siblings in a tree.

■■■ **R16.3** Describe a recursive algorithm for finding the total path length of a tree. The total path length is the sum of the lengths of all paths from the root to the leaves. (The length of a path is the number of nodes on the path.) What is the efficiency of your algorithm?

■■ **R16.4** Show that a binary tree with l leaves has at least $l - 1$ interior nodes, and exactly $l - 1$ interior nodes if all of them have two children.

■ **R16.5** What is the difference between a binary tree and a binary search tree? Give examples of each.

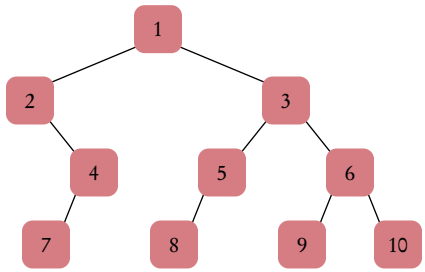
■ **R16.6** What is the difference between a balanced tree and an unbalanced tree? Give examples of each.

■ **R16.7** The following elements are inserted into a binary search tree. Make a drawing that shows the resulting tree after each insertion.

```
Adam
Eve
Romeo
Juliet
Tom
Diana
Harry
```

■■ **R16.8** Insert the elements of Exercise R16.7 in opposite order. Then determine how the `BinarySearchTree.print` function from Section 16.4.1 prints out both the tree from Exercise R16.7 and this tree. Explain how the printouts are related.

■■ **R16.9** Consider the following tree. In which order are the nodes printed by the `BinarySearchTree.print` function from Section 16.4.1? The numbers identify the nodes. The data stored in the nodes is not shown.



■■ **R16.10** Design an algorithm for finding the kth element (in sort order) of a binary search tree. How efficient is your algorithm?

■■ **R16.11** Design an $O(\log(n))$ algorithm for finding the kth element in a binary search tree, provided that each node has a data member containing the size of the subtree. Also describe how these data members can be maintained by the insertion and removal operations without affecting their big-Oh efficiency.

■■ **R16.12** Design an algorithm for deciding whether two binary trees have the same shape. What is the running time of your algorithm?

■ **R16.13** Insert the following eleven words into a binary search tree:

> Mary had a little lamb. Its fleece was white as snow.

Draw the resulting tree.

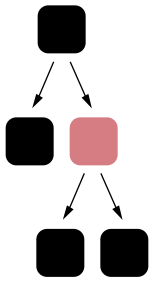
■ **R16.14** What is the result of printing the tree from Exercise R16.13 using preorder, inorder, and postorder traversal?

■■ **R16.15** Locate nodes with no children, one child, and two children in the tree of Exercise R16.13. For each of them, show the tree of size 10 that is obtained after removing the node.

■■ **R16.16** Repeat Exercise R16.13 for a red-black tree.

■■■ **R16.17** Repeat Exercise R16.15 for a red-black tree.

■■ **R16.18** Show that a red-black tree with black height bh has at least $2^{bh} - 1$ nodes. *Hint:* Look at the root. A black child has black height $bh - 1$. A red child must have two black children of black height $bh - 1$.



■■ **R16.19** Let $rbts(bh)$ be the number of red-black trees with black height bh. Give a recursive formula for $rbts(bh)$ in terms of $rbts(bh - 1)$. How many red-black trees have heights 1, 2, and 3? *Hint:* Look at the hint for Exercise R16.18.

■■ **R16.20** What is the maximum number of nodes in a red-black tree with black height bh?

■■ **R16.21** Show that any red-black tree must have fewer interior red nodes than it has black nodes.

■■■ **R16.22** Show that the "black root" rule for red-black trees is not essential. That is, if one allows trees with a red root, insertion and deletion still occur in $O(\log(n))$ time.

■■ **R16.23** Many textbooks use "dummy nodes"—black nodes with two `nullptr` children—instead of regular null pointers in red-black trees. In this representation, all non-dummy nodes of a red-black tree have two children. How does this simplify the description of the removal algorithm?

PRACTICE EXERCISES

■ **E16.1** Write a member function for the `Tree` class in Section 16.1 that counts the number of all leaves in a tree.

■ **E16.2** Add a member function `count_nodes_with_one_child` to the `BinaryTree` class.

■ **E16.3** Add a member function `swap_children` (that swaps all left and right children) to the `BinaryTree` class.

■■ **E16.4** Implement the animal guessing game described in Section 16.2.1. Start with the tree in Figure 4, but present the leaves as "Is it a(n) X?" If it wasn't, ask the user what the animal was, and ask for a question that is true for that animal but false for X. Then insert a new node for that animal. For example,

```
Is it a mammal? Y
Does it have stripes? N
Is it a pig? N
I give up. What is it? A hamster
Please give me a question that is true for a hamster and false for a pig.
Is it small and cuddly?
```

Now insert a node so that this question is used in future dialogs:

```
Is it a mammal? Y
Does it have stripes? N
Is it small and cuddly?
```

In this way, the program learns additional facts.

■■ **E16.5** Change the implementation of the binary search tree in Section 16.3 so that the private `add_node` member function is instead a member function of the `Node` class.

■■ **E16.6** Change the implementation of the binary search tree in Section 16.3 so that the private `print` member function is instead a member function of the `Node` class.

■ **E16.7** Write a member function of the `BinarySearchTree` class

```
string smallest()
```

that returns the smallest element of a tree.

■ **E16.8** Add member functions

```
void preorder(Visitor& v)
void inorder(Visitor& v)
void postorder(Visitor& v)
```

to the `BinaryTree` class of Section 16.2.

■■ **E16.9** Using a visitor, compute the average length of the elements in a binary tree filled with strings.

■ **E16.10** Add a member function `void depth_first(Visitor& v)` to the `Tree` class of Section 16.4.

■■ **E16.11** Implement a member function `void inorder(Visitor& v)` for the `BinaryTree` class of Section 16.2.

■ **E16.12** Change the `visit` member function of the `Visitor` class so that it returns a `bool` value that should be `true` if the visitation should continue or `false` to quit visiting further nodes. Change the `breadth_first` member function of the `Tree` class in Section 16.4 to accept such a visitor and to stop visitation when indicated. Provide an example that stops visiting a tree when an element starts with `"M"`.

■■ **E16.13** Change the `preorder` and `postorder` member functions of the `Tree` class in Section 16.4 so that visitation is stopped as soon as the `visit` function returns `false`, as described in Exercise E16.12. *Hint:* Have these functions return `false` when `visit` returns `false`.

■ **E16.14** Using an appropriate visitor, print all elements stored in leaf nodes of a given tree.

■■ **E16.15** Using an appropriate visitor, collect all elements stored in leaf nodes of a binary tree in a vector in increasing order.

■■ **E16.16** Write a member function for the `RedBlackTree` class of Worked Example 16.2 that checks that the tree fulfills the rules for a red-black tree.

■■ **E16.17** In Worked Example 16.2, run the removal test with a different tree template that has more nodes.

PROGRAMMING PROJECTS

■■■ **P16.1** A general tree (in which each node can have arbitrarily many children) can be implemented as a binary tree in this way: For each node with n children, use a chain of n binary nodes. Each `left` pointer points to a child and each `right` pointer points to the next node in the chain. Using the binary tree implementation of Section 16.2, implement a tree class with the same public interface as the one in Section 16.1.

■■■ **P16.2** Continue Exercise E16.4 and write the tree to a file when the program exits. Load the file when the program starts again.

■■■ **P16.3** Change the `BinarySearchTree.print` member function to print the tree as a tree shape. You can print the tree sideways. Extra credit if you instead display the tree with the root node centered on the top.

■■■ **P16.4** In the `BinarySearchTree` class, modify the `erase` member function so that a node with two children is replaced by the largest child of the left subtree.

■■ **P16.5** Reimplement the `BinarySearchTree` class to use a `vector<string> elements` instead of `Node` objects to store the nodes. Store the root element in `elements[0]` and the children of `elements[i]` in `elements[2 * i]` and `elements[2 * i + 1]`. If one of these elements is the empty string or the index is not valid, then the corresponding child does not exist.

■■ **P16.6** Reimplement the `remove` member function in the `RedBlackTree` class of Worked Example 16.2 so that the node is first removed using the binary search tree removal algorithm, and the tree is rebalanced after removal.

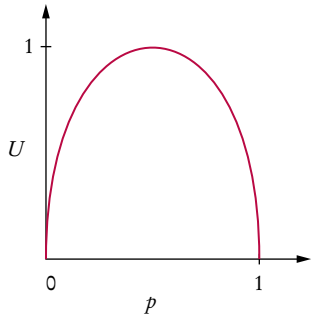
■■■ **P16.7** The ID3 algorithm describes how to build a decision tree for a given a set of sample facts. The tree asks the most important questions first. We have a set of criteria (such as "Is it a mammal?") and an objective that we want to decide (such as "Can it swim?"). Each fact has a value for each criterion and the objective. Here is a set of five facts about animals. (Each row is a fact.) There are four criteria and one objective (the columns of the table). For simplicity, we assume that the values of the criteria and objective are binary (Y or N).

Is it a mammal?	Does it have fur?	Does it have a tail?	Does it lay eggs?	Can it swim?
N	N	Y	Y	N
N	N	N	Y	Y
N	N	Y	Y	Y
Y	N	Y	N	Y
Y	Y	Y	N	N

We now need several definitions. Given any probability value p between 0 and 1, its uncertainty is

$$U(p) = -p\log_2(p) - (1-p)\log_2(1-p)$$

If p is 0 or 1, the outcome is certain, and the uncertainty $U(p)$ is 0. If $p = 1/2$, then the outcome is completely uncertain and $U(p) = 1$.



Let n be the number of facts and $n(c = \text{Y})$ be the number of facts for which the criterion c has the value Y. Then the uncertainty $U(c, o)$ that c contributes to the outcome o is the weighted average of two uncertainties:

$$U(c,o) = \frac{n(c = \text{Y})}{n} \cdot U\left(\frac{n(c = \text{Y}, o = \text{Y})}{n(c = \text{Y})}\right) + \frac{n(c = \text{N})}{n} \cdot U\left(\frac{n(c = \text{N}, o = \text{Y})}{n(c = \text{N})}\right)$$

Your program should read in a file that contains the criteria and objective, one per line, followed by a blank line and a sequence of lines, each containing a fact (that is, a sequence of Y and N for the criteria and objective). Then build a decision tree whose leaves indicate whether the objective is fulfilled or not. Allow the user to provide inputs for each criterion, and print the objective when it is determined.

Find the criterion c that minimizes the uncertainty $U(c, o)$. That question becomes the root of your tree. Recursively, repeat for the subsets of the facts for which c is Y (in the left subtree) and N (in the right subtree). If it happens that the uncertainty is constant, then you have a leaf with an answer, and the recursion stops.

In our example, we have

Two out of five are mammals.

One of those two swims.

Three out of five aren't mammals.

Two of those three swim.

Is it a mammal? $\frac{2}{5} \cdot U\left(\frac{1}{2}\right) + \frac{3}{5} \cdot U\left(\frac{2}{3}\right) = 0.95$

Does it have fur? $\frac{1}{5} \cdot U\left(\frac{0}{1}\right) + \frac{4}{5} \cdot U\left(\frac{3}{4}\right) = 0.65$

Does it have a tail? $\frac{4}{5} \cdot U\left(\frac{2}{4}\right) + \frac{1}{5} \cdot U\left(\frac{1}{1}\right) = 0.8$

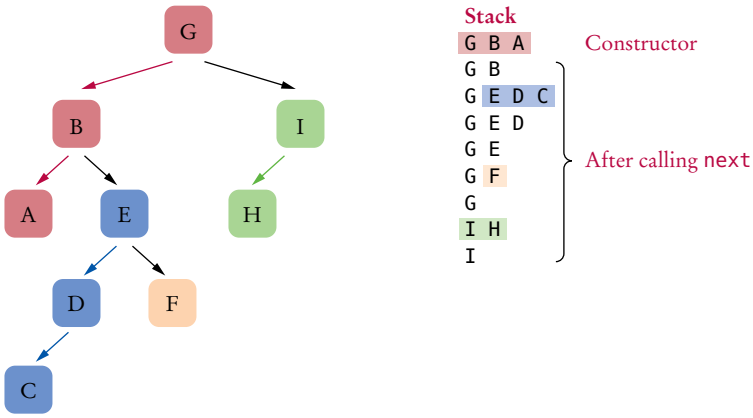
Does it lay eggs? $\frac{3}{5} \cdot U\left(\frac{2}{3}\right) + \frac{2}{5} \cdot U\left(\frac{1}{2}\right) = 0.95$

Therefore, we choose "Does it have fur?" as our first criterion.

In the left subtree, look at the animals with fur. There is only one, a non-swimmer, so you can declare "It doesn't swim." For the right subtree, you now have four facts (the animals without fur) and three criteria. Repeat the process.

■■■ **P16.8** Modify the expression evaluator from Section 11.5 to produce an expression tree. (Note that the resulting tree is a binary tree but not a binary search tree.) Then use postorder traversal to evaluate the expression, using a stack for the intermediate results.

■■■ **P16.9** Implement an iterator for the `BinarySearchTree` class that visits the nodes in sorted order. *Hint:* In the constructor, keep pushing left nodes on a stack until you reach a `nullptr`. In each call to `next`, deliver the top of the stack as the visited node, but first push the left nodes in its right subtree.



■■■ **P16.10** Implement an iterator for the `RedBlackTree` class in Worked Example 16.2 that visits the nodes in sorted order. *Hint:* Take advantage of the `parent` links.

■■■ **P16.11** Add a copy constructor, assignment operator, and destructor to the tree implementation of Section 16.1. You will need to rethink the `add_subtree` member function that currently copies the node pointers.

■■ **P16.12** Add a copy constructor, assignment operator, and destructor to the binary search tree implementation of Section 16.3.

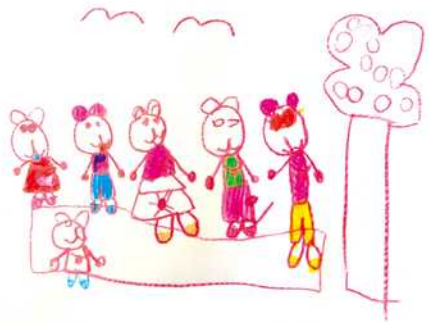
■■ **P16.13** Turn the binary search tree implementation of Section 16.3 into a template that works for any type with a `<` operator. You will need to turn around any comparisons with `>`, and change any equality `x == y` to `!(x < y || y < x)`.

WORKED EXAMPLE 16.1

Building a Huffman Tree

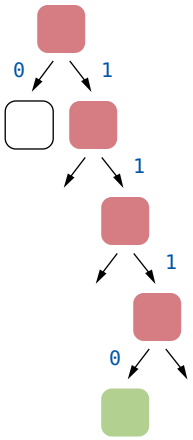
A Huffman code encodes symbols into sequences of bits (zeroes and ones), so that the most frequently occurring symbols have the shortest encodings. The symbols can be characters of the alphabet, but they can also be something else. For example, when images are compressed using a Huffman encoding, the symbols are the colors that occur in the image.

Problem Statement Encode a child's painting like the one below by building a Huffman tree with an optimal encoding. Most of the pixels are white (50%), there are lots of orange (20%) and pink (20%) pixels, and small amounts of yellow (5%), blue (3%), and green (2%).



Charlotte and Emily Horstmann.

We want a short code (perhaps 0) for white and a long one (perhaps 1110) for green. Such a variable-length encoding minimizes the overall length of the encoded data.



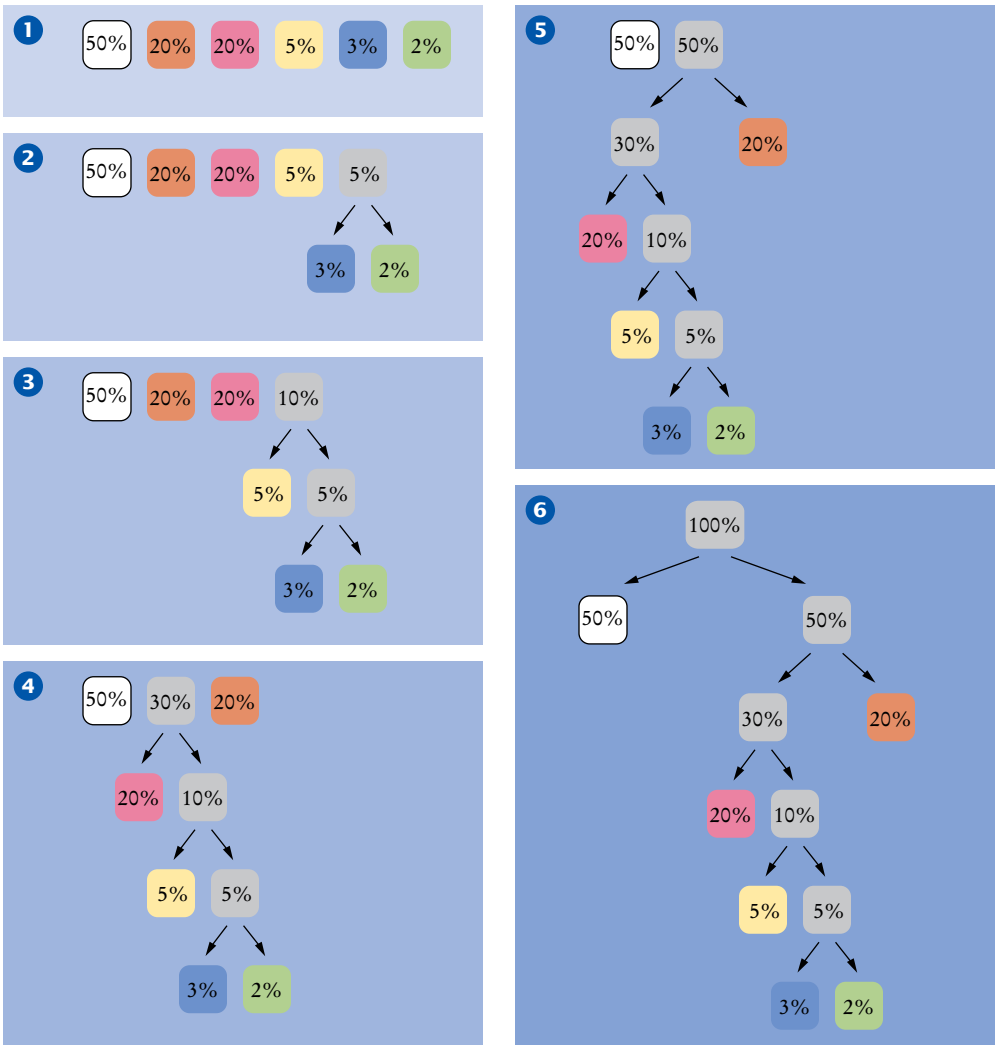
The challenge is to build a tree that yields an optimal encoding. The following algorithm, developed by David Huffman when he was a graduate student, achieves this task.

Make a tree node for each symbol to be encoded. Each node has a data member for the frequency.

```
Add all nodes to a vector.
While there are two nodes left
   Remove the two nodes with the smallest frequencies.
   Make them children of a parent whose frequency is the sum of the child frequencies.
   Add the parent to the node vector.
```

The remaining node is the root of the Huffman tree.

The following figure shows the algorithm applied to our sample data.



After the tree has been constructed, the frequencies are no longer needed.

The resulting code is

```
White       0
Pink        100
Yellow      1010
Blue        10110
Green       10111
Orange      11
```

Note that this is not a code for encrypting information. The code is known to all; its purpose is to compress data by using the shortest codes for the most common symbols. Also note that the code has the property that no codeword is the prefix of another codeword. For example, because white is encoded as `0`, no other codeword starts with `0`, and because orange is 11, no other codeword starts with 11.

The implementation is very straightforward. The `Node` class needs data members for holding the symbol to be encoded (which we assume to be a character) and its frequency.

```
class Node
{
private:
   char character;
   int frequency;
   Node* left;
   Node* right;
friend class HuffmanTree;
};
```

When constructing a tree, we need the frequencies for all characters. The tree constructor receives them in an `unordered_map<char, int>`. The frequencies need not be percentages. They can be counts from a sample text.

First, we make a node for each character to be encoded, and add each node to a vector:

```
vector<Node*> nodes;
for (auto iter = frequencies.begin(); iter != frequencies.end(); iter++)
{
   Node* new_node = new Node;
   new_node->character = iter->first;
   new_node->frequency = iter->second;
   new_node->left = nullptr;
   new_node->right = nullptr;
   nodes.push_back(new_node);
}
```

Then, following the algorithm, we keep combining the two nodes with the lowest frequencies:

```
while (nodes.size() > 1)
{
   Node* smallest = remove_min(nodes);
   Node* next_smallest = remove_min(nodes);
   Node* new_node = new Node;
   new_node->frequency
      = smallest->frequency + next_smallest->frequency;
   new_node->left = smallest;
   new_node->right = next_smallest;
   nodes.push_back(new_node);
}

root = nodes[0];
```

Removing the node with the lowest frequency can be made much more efficient by using a priority queue instead of a vector—see Chapter 17.

Decoding a sequence of zeroes and ones is very simple: just follow the links to the left or right until a leaf is reached. Note that each node has either two or no children, so we only need to check whether one of the children is `nullptr` to detect a leaf. Here we use strings of `0` or `1` characters, not actual bits, to keep the demonstration simple.

```
string HuffmanTree::decode(string input) const
{
   string result = "";
   Node* n = root;
   for (int i = 0; i < input.length(); i++)
   {
      char ch = input[i];
      if (ch == '0')
      {
         n = n->left;
```

```
         }
         else
         {
            n = n->right;
         }
         if (n->left == nullptr) // n is a leaf
         {
            result = result + n->character;
            n = root;
         }
      }
      return result;
   }
```

The tree is not useful for efficient encoding because we don't want to search through the leaves each time we encode a character. Instead, we will just compute a map that maps each character to its encoding. This can be done by recursively visiting the subtrees and remembering the current prefix, that is, the path to the root of the subtree. Follow the left or right children, adding a `0` or `1` to the end of that prefix, or, if the subtree is a leaf, simply add the character and the prefix to the map:

```
void HuffmanTree::fill_encoding_map(unordered_map<char, string>& map,
   string prefix, Node* n) const
{
   if (n == nullptr) return;
   if (n->left == nullptr) // It's a leaf
   {
      map[n->character] = prefix;
   }
   else
   {
      fill_encoding_map(map, prefix + "0", n->left);
      fill_encoding_map(map, prefix + "1", n->right);
   }
}
```

This recursive helper function, `fill_encoding_map`, is called from the member function `encoding_map` of the `HuffmanTree` class:

```
unordered_map<char, string> HuffmanTree::encoding_map() const
{
   unordered_map<char, string> map;
   fill_encoding_map(map, "", root);
   return map;
}
```

The demonstration program for this example computes the Huffman encoding for the Hawaiian language, which was chosen because it uses fewer letters than most other languages. The frequencies were obtained from a text sample on the Internet.

Here is the complete program:

worked_example_1/huffman_tree.h

```
#ifndef HUFFMAN_TREE_H
#define HUFFMAN_TREE

#include <string>
#include <vector>
#include <unordered_map>

using namespace std;

```

```
10 class Node
11 {
12 private:
13    char character;
14    int frequency;
15    Node* left;
16    Node* right;
17 friend class HuffmanTree;
18 };
19
20 /*
21    A tree for decoding Huffman codes.
22 */
23 class HuffmanTree
24 {
25 public:
26    /**
27       Constructs a Huffman tree from given character frequencies.
28       @param frequencies a map whose keys are the characters to be encoded
29       and whose values are the frequencies of the characters
30    */
31    HuffmanTree(const unordered_map<char, int>& frequencies);
32    /**
33       Decodes an encoded string.
34       @param input a string made up of 0 and 1
35    */
36    string decode(string input) const;
37
38    unordered_map<char, string> encoding_map() const;
39
40 private:
41    Node* remove_min(vector<Node*>& nodes) const;
42    void fill_encoding_map(unordered_map<char, string>& map,
43       string prefix, Node* n) const;
44
45    Node* root;
46 };
47
48 #endif
```

worked_example_1/huffman_tree.cpp

```
 1 #include "huffman_tree.h"
 2
 3 HuffmanTree::HuffmanTree(const unordered_map<char, int>& frequencies)
 4 {
 5    vector<Node*> nodes;
 6    for (auto iter = frequencies.begin(); iter != frequencies.end(); iter++)
 7    {
 8       Node* new_node = new Node;
 9       new_node->character = iter->first;
10       new_node->frequency = iter->second;
11       new_node->left = nullptr;
12       new_node->right = nullptr;
13       nodes.push_back(new_node);
14    }
15
16    while (nodes.size() > 1)
17    {
```

```
      Node* smallest = remove_min(nodes);
      Node* next_smallest = remove_min(nodes);
      Node* new_node = new Node;
      new_node->frequency
         = smallest->frequency + next_smallest->frequency;
      new_node->left = smallest;
      new_node->right = next_smallest;
      nodes.push_back(new_node);
   }

   root = nodes[0];
}

string HuffmanTree::decode(string input) const
{
   string result = "";
   Node* n = root;
   for (int i = 0; i < input.length(); i++)
   {
      char ch = input[i];
      if (ch == '0')
      {
         n = n->left;
      }
      else
      {
         n = n->right;
      }
      if (n->left == nullptr) // n is a leaf
      {
         result = result + n->character;
         n = root;
      }
   }
   return result;
}

unordered_map<char, string> HuffmanTree::encoding_map() const
{
   unordered_map<char, string> map;
   fill_encoding_map(map, "", root);
   return map;
}

void HuffmanTree::fill_encoding_map(unordered_map<char, string>& map,
   string prefix, Node* n) const
{
   if (n == nullptr) return;
   if (n->left == nullptr) // It's a leaf
   {
      map[n->character] = prefix;
   }
   else
   {
      fill_encoding_map(map, prefix + "0", n->left);
      fill_encoding_map(map, prefix + "1", n->right);
   }
}
```

```
77 Node* HuffmanTree::remove_min(vector<Node*>& nodes) const
78 {
79    int last = nodes.size() - 1;
80    if (last == -1) { return nullptr; }
81    int min_pos = 0;
82    for (int i = 1; i <= last; i++)
83    {
84       if (nodes[i]->frequency < nodes[min_pos]->frequency)
85       {
86          min_pos = i;
87       }
88    }
89    Node* result = nodes[min_pos];
90    nodes[min_pos] = nodes[last];
91    nodes.pop_back();
92    return result;
93 }
```

worked_example_1/huffman_demo.cpp

```
 1 #include <iostream>
 2 #include <string>
 3 #include <unordered_map>
 4 #include "huffman_tree.h"
 5
 6 using namespace std;
 7
 8 string encode(string to_encode,
 9    const unordered_map<char, string>& encoding_map)
10 {
11    string result = "";
12    for (int i = 0; i < to_encode.length(); i++)
13    {
14       char ch = to_encode[i];
15       string encoded = encoding_map.at(ch);
16       result = result + encoded;
17    }
18    return result;
19 }
20
21 int main()
22 {
23    unordered_map<char, int> frequency_map;
24    frequency_map['A'] = 2089;
25    frequency_map['E'] = 576;
26    frequency_map['H'] = 357;
27    frequency_map['I'] = 671;
28    frequency_map['K'] = 849;
29    frequency_map['L'] = 354;
30    frequency_map['M'] = 259;
31    frequency_map['N'] = 660;
32    frequency_map['O'] = 844;
33    frequency_map['P'] = 239;
34    frequency_map['U'] = 472;
35    frequency_map['W'] = 74;
36    frequency_map['\''] = 541;
37    HuffmanTree tree(frequency_map);
38
39    unordered_map<char, string> encoding_map = tree.encoding_map();
40    string encoded = encode("ALOHA", encoding_map);
```

```
41     cout << encoded << endl;
42     string decoded = tree.decode(encoded);
43     cout << decoded << endl;
44     return 0;
45  }
```

WORKED EXAMPLE 16.2

Implementing a Red-Black Tree

Problem Statement Implement a red-black tree using the algorithm for adding and removing elements from Section 16.5. Read that section first if you have not done so already.

The Node Implementation

The nodes of the red-black tree need to store the "color", which we represent as the cost of traversing the node:

```
const int BLACK = 1;
const int RED = 0;
const int NEGATIVE_RED = -1;
const int DOUBLE_BLACK = 2;

class Node
{
public:
   Node();
private:
   void add_node(Node* new_node);
public:
   void set_left_child(Node* child);
   void set_right_child(Node* child);
   string data;
   Node* left;
   Node* right;
   Node* parent;
   int color;
friend class RedBlackTree;
};
```

Several member functions of the `Node` class, as well as the data members, have public visibility to enable thorough testing.

Nodes in a red-black tree also have a link to the parent. When adding or moving a node, it is important that the parent and child links are synchronized. Because this synchronization is tedious and error-prone, we provide several helper functions:

```
void Node::set_left_child(Node* child)
{
   left = child;
   if (child != nullptr) { child->parent = this; }
}

void Node::set_right_child(Node* child)
{
   right = child;
   if (child != nullptr) { child->parent = this; }
}

void RedBlackTree::replace_with(Node* to_be_replaced, Node* replacement)
{
   if (to_be_replaced->parent == nullptr)
   {
```

```
      replacement->parent = nullptr;
      root = replacement;
   }
   else if (to_be_replaced == to_be_replaced->parent->left)
   {
      to_be_replaced->parent->set_left_child(replacement);
   }
   else
   {
      to_be_replaced->parent->set_right_child(replacement);
   }
}
```

Insertion

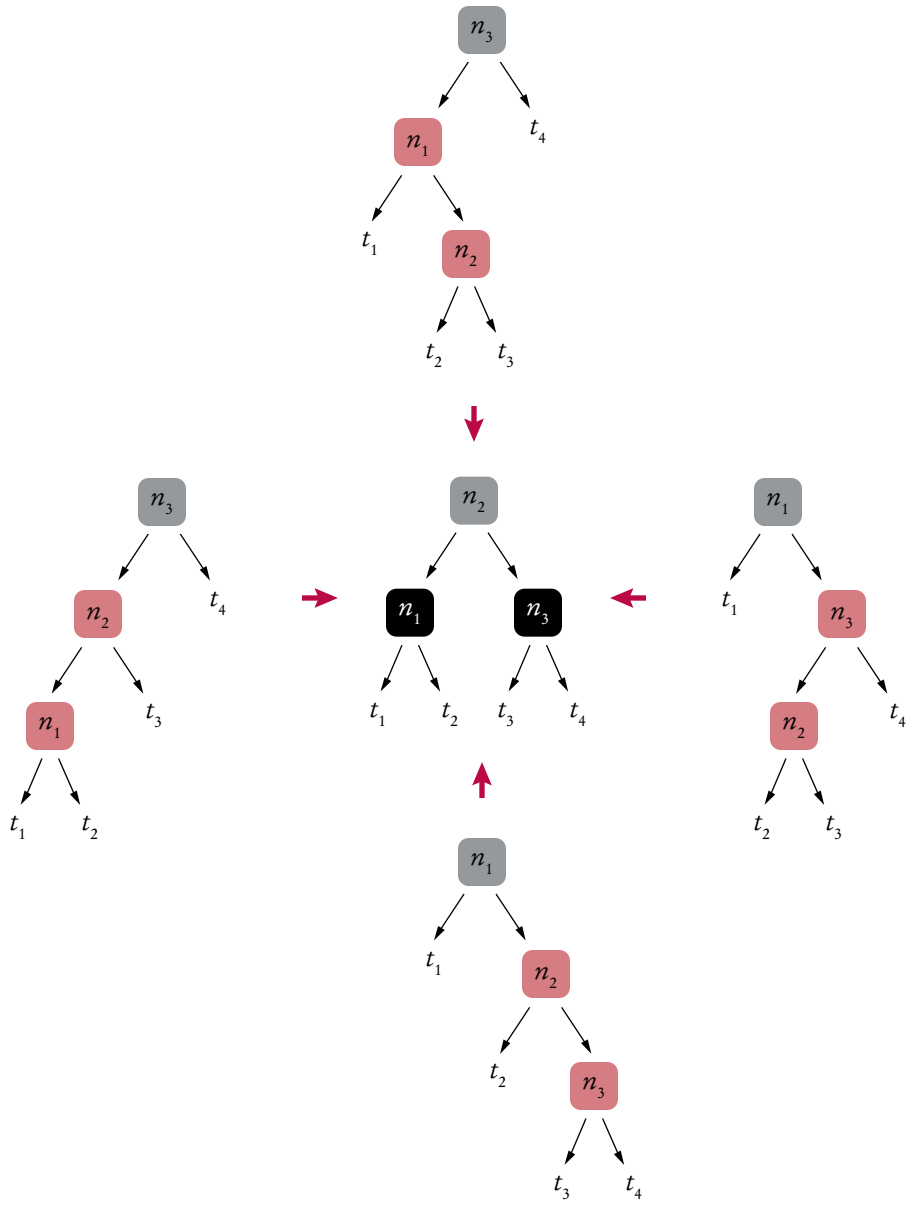
Insertion is handled as it is in a binary search tree. We insert a red node. Afterward, we call a function that fixes up the tree so it is a red-black tree again:

```
void RedBlackTree::insert(string element)
{
   Node* new_node = new Node;
   new_node->data = element;
   new_node->left = nullptr;
   new_node->right = nullptr;
   if (root == nullptr) { root = new_node; }
   else { root->add_node(new_node); }
   fix_after_add(new_node);
}
```

If the inserted node is the root, it is turned black. Otherwise, we fix up any double-red violations:

```
/**
   Restores the tree to a red-black tree after a node has been added.
   @param new_node the node that has been added
*/
void RedBlackTree::fix_after_add(Node* new_node)
{
   if (new_node->parent == nullptr)
   {
      new_node->color = BLACK;
   }
   else
   {
      new_node->color = RED;
      if (new_node->parent->color == RED) { fix_double_red(new_node); }
   }
}
```

The code for fixing up a double-red violation is quite long. Recall that there are four possible arrangements of the double-red nodes:



In each case, we must sort the nodes and their children. Once we have the seven references `n1`, `n2`, `n3`, `t1`, `t2`, `t3`, and `t4`, the remainder of the procedure is straightforward. We build the replacement tree, change the reds to black, and subtract one from the color of the grandparent (which might be a double-black node when this function is called during node removal).

If we find that we introduced another double-red violation, we continue fixing it. Eventually, the violation is removed, or we reach the root, in which case the root is simply colored black:

```
/**
   Fixes a "double red" violation.
   @param child the child with a red parent
*/
void RedBlackTree::fix_double_red(Node* child)
{
```

```
   Node* parent = child->parent;
   Node* grandparent = parent->parent;
   if (grandparent == nullptr) { parent->color = BLACK; return; }
   Node* n1;
   Node* n2;
   Node* n3;
   Node* t1;
   Node* t2;
   Node* t3;
   Node* t4;
   if (parent == grandparent->left)
   {
      n3 = grandparent; t4 = grandparent->right;
      if (child == parent->left)
      {
         n1 = child; n2 = parent;
         t1 = child->left; t2 = child->right; t3 = parent->right;
      }
      else
      {
         n1 = parent; n2 = child;
         t1 = parent->left; t2 = child->left; t3 = child->right;
      }
   }
   else
   {
      n1 = grandparent; t1 = grandparent->left;
      if (child == parent->left)
      {
         n2 = child; n3 = parent;
         t2 = child->left; t3 = child->right; t4 = parent->right;
      }
      else
      {
         n2 = parent; n3 = child;
         t2 = parent->left; t3 = child->left; t4 = child->right;
      }
   }

   replace_with(grandparent, n2);
   n1->set_left_child(t1);
   n1->set_right_child(t2);
   n2->set_left_child(n1);
   n2->set_right_child(n3);
   n3->set_left_child(t3);
   n3->set_right_child(t4);
   n2->color = grandparent->color - 1;
   n1->color = BLACK;
   n3->color = BLACK;

   if (n2 == root)
   {
      root->color = BLACK;
   }
   else if (n2->color == RED && n2->parent->color == RED)
   {
      fix_double_red(n2);
   }
}
```

Removal

We remove a node in the same way as in a binary search tree. However, before removing it, we want to make sure that it is colored red. There are two cases for removal: removing an element with one child and removing the successor of an element with two children. Both branches must be modified:

```
void RedBlackTree::erase(string element)
{
   // Find node to be removed

   Node* to_be_removed = root;
   bool found = false;
   while (!found && to_be_removed != nullptr)
   {
      if (element == to_be_removed->data)
      {
         found = true;
      }
      else if (element < to_be_removed->data)
      {
         to_be_removed = to_be_removed->left;
      }
      else
      {
         to_be_removed = to_be_removed->right;
      }
   }

   if (!found) { return; }

   // to_be_removed contains element

   // If one of the children is empty, use the other

   if (to_be_removed->left == nullptr || to_be_removed->right == nullptr)
   {
      Node* new_child;
      if (to_be_removed->left == nullptr) { new_child = to_be_removed->right; }
      else { new_child = to_be_removed->left; }

      fix_before_remove(to_be_removed);
      replace_with(to_be_removed, new_child);
      return;
   }

   // Neither subtree is empty

   // Find smallest element of the right subtree

   Node* smallest = to_be_removed->right;
   while (smallest->left != nullptr)
   {
      smallest = smallest->left;
   }

   // smallest contains smallest child in right subtree
```

```
   // Move contents, unlink child

   to_be_removed->data = smallest->data;
   fix_before_remove(smallest);
   replace_with(smallest, smallest->right);
}
```

The `replace_with` helper function, which was shown earlier, takes care of updating the parent, child, and root links. The `fix_before_remove` function has three cases. Removing a red leaf is safe. If a black node has a single child, that child must be red, and we can safely swap the colors. (We don't actually bother to color the node that is to be removed.) The case with a black leaf is the hardest. We need to initiate the "bubbling up" process:

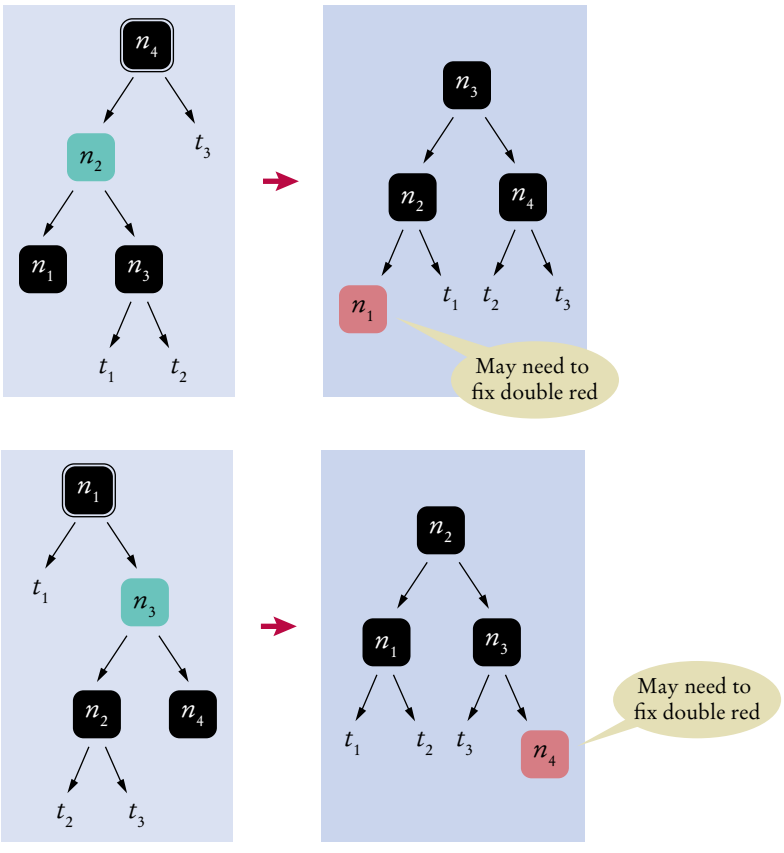
```
/**
   Fixes the tree so that it is a red-black tree after a node has been removed.
   @param to_be_removed the node that is to be removed
*/
void RedBlackTree::fix_before_remove(Node* to_be_removed)
{
   if (to_be_removed->color == RED) { return; }

   if (to_be_removed->left != nullptr
         || to_be_removed->right != nullptr) // It is not a leaf
   {
      // Color the child black
      if (to_be_removed->left == nullptr) { to_be_removed->right->color = BLACK; }
      else { to_be_removed->left->color = BLACK; }
   }
   else { bubble_up(to_be_removed->parent); }
}
```

To bubble up, we move a "toll charge" from the children to the parent. This may result in a negative-red or double-red child, which we fix. If neither fix was successful, and the parent node is still double-black, we bubble up again until we reach the root. The root color can be safely changed to black.

```
/**
   Move a charge from two children of a parent.
   @param parent a node with two children, or nullptr (in which case nothing is done)
*/
void RedBlackTree::bubble_up(Node* parent)
{
   if (parent == nullptr) { return; }
   parent->color++;
   parent->left->color--;
   parent->right->color--;

   if (bubble_up_fix(parent->left)) { return; }
   if (bubble_up_fix(parent->right)) { return; }

   if (parent->color == DOUBLE_BLACK)
   {
      if (parent->parent == nullptr) { parent->color = BLACK; }
      else { bubble_up(parent->parent); }
   }
}
```

```
/**
   Fixes a negative-red or double-red violation introduced by bubbling up.
   @param child the child to check for negative-red or double-red violations
   @return true if the tree was fixed
*/
bool RedBlackTree::bubble_up_fix(Node* child)
{
   if (child->color == NEGATIVE_RED)
   {
      fix_negative_red(child);
      return true;
   }
   else if (child->color == RED)
   {
      if (child->left != nullptr && child->left->color == RED)
      {
         fix_double_red(child->left); return true;
      }
      if (child->right != nullptr && child->right->color == RED)
      {
         fix_double_red(child->right); return true;
      }
   }
   return false;
}
```

We are left with the negative red removal. In the diagram in the book, we show only one of the two possible situations. In the code, we also need to handle the mirror image.



The implementation is not difficult, just long.

```
/**
   Fixes a "negative red" violation.
   @param neg_red the negative red node
*/
void RedBlackTree::fix_negative_red(Node* neg_red)
{
   Node* parent = neg_red->parent;
   Node* child;
   if (parent->left == neg_red)
   {
      Node* n1 = neg_red->left;
      Node* n2 = neg_red;
      Node* n3 = neg_red->right;
      Node* n4 = parent;
      Node* t1 = n3->left;
      Node* t2 = n3->right;
      Node* t3 = n4->right;
      n1->color = RED;
      n2->color = BLACK;
      n4->color = BLACK;

      replace_with(n4, n3);
      n3->set_left_child(n2);
      n3->set_right_child(n4);
      n2->set_left_child(n1);
      n2->set_right_child(t1);
      n4->set_left_child(t2);
      n4->set_right_child(t3);

      child = n1;
   }
   else // Mirror image
   {
      Node* n4 = neg_red->right;
      Node* n3 = neg_red;
      Node* n2 = neg_red->left;
      Node* n1 = parent;
      Node* t3 = n2->right;
      Node* t2 = n2->left;
      Node* t1 = n1->left;
      n4->color = RED;
      n3->color = BLACK;
      n1->color = BLACK;

      replace_with(n1, n2);
      n2->set_right_child(n3);
      n2->set_left_child(n1);
      n3->set_right_child(n4);
      n3->set_left_child(t3);
      n1->set_right_child(t2);
      n1->set_left_child(t1);

      child = n4;
   }

   if (child->left != nullptr && child->left->color == RED)
   {
      fix_double_red(child->left);
```

```
   }
   else if (child->right != nullptr && child->right->color == RED)
   {
      fix_double_red(child->right);
   }
}
```

Simple Tests

With such a complex implementation, it is extremely likely that some errors slipped in somewhere, and it is important to carry out thorough testing.

We can start with the test case used for the binary search tree from the book:

```
void test_from_book()
{
   RedBlackTree t;
   t.insert("D");
   t.insert("B");
   t.insert("A");
   t.insert("C");
   t.insert("F");
   t.insert("E");
   t.insert("I");
   t.insert("G");
   t.insert("H");
   t.insert("J");
   t.erase("A"); // Removing leaf
   t.erase("B"); // Removing element with one child
   t.erase("F"); // Removing element with two children
   t.erase("D"); // Removing root
   t.print();
   cout << "Expected: C E G H I J" << endl;
}
```

If this test fails (which it did for the author at the first attempt), it is fairly easy to debug. If it passes, it gives some confidence. But there are so many different configurations that more thorough tests are required.

For a more exhaustive test, we can insert all permutations of the ten letters A – J and check that the resulting tree has the desired contents. Here, we use the `next_permutation` function from the C++ standard library to visit all permutations of a string:

```
/**
   Inserts all permutations of a string into a red-black tree and checks that
   it contains the strings afterwards.
   @param letters a sorted string of letters without repetition
   @return the number of passing tests
*/
int insertion_test(string letters)
{
   int count = 0;
   string s = letters;
   do
   {
      RedBlackTree t;
      for (int i = 0; i < s.length(); i++)
      {
         t.insert(s.substr(i, 1));
      }
      bool pass = true;
      for (int i = 0; i < s.length(); i++)
      {
```

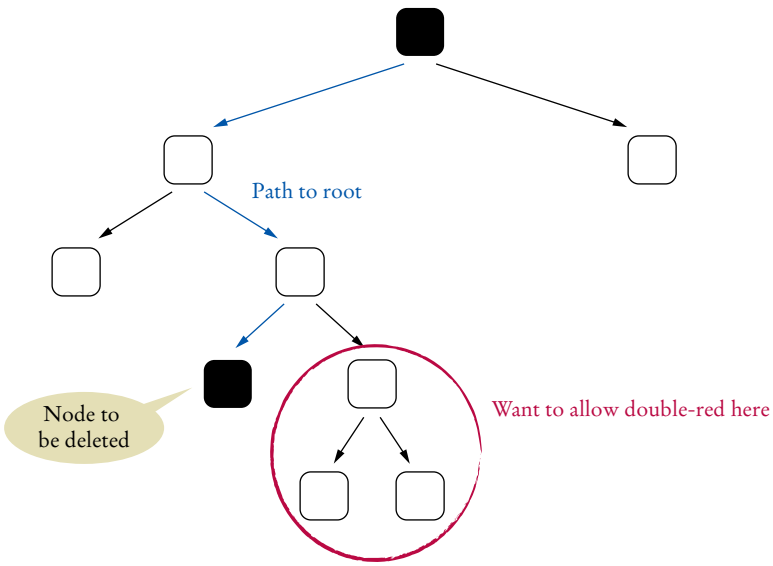
```
            string e = s.substr(i, 1);
            if (t.count(e) == 0)
            {
               cout << e << " not inserted" << endl;
               pass = false;
            }
         }
         if (pass)
         {
            count++;
         }
         else
         {
            cout << "Failing for letters " << letters << endl;
         }
      }
      while(next_permutation(s.begin(), s.end()));
      return count;
   }
```

This test runs through 10! = 3,628,800 permutations, which seems pretty exhaustive. But how do we really know that all possible configurations of red and black nodes have been covered? For example, it seems plausible that all four possible configurations of Figure 21 occur somewhere in these test cases, but how do we know for sure? We take up that question in the next section.

An Advanced Test

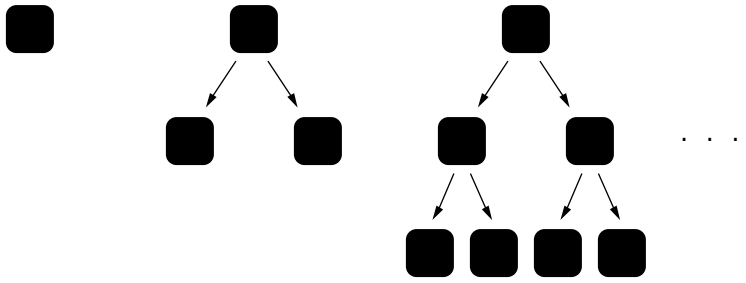
In the previous section, we reached the limits of what one can achieve with "black box" testing. For more exhaustive coverage, we need to manufacture red-black trees with all possible patterns of red and black nodes. At first, that seems hopeless. According to Exercise R16.19, there are 25,728,160,000 red-black trees with black height 3, far too many to generate and test.

Fortunately, we don't have to test them all. The algorithms for insertion and removal fix up nodes that form a direct path to the root. It is enough to fill this path, and its neighboring elements with all possible color combinations. Let us test the most complex case: removing a black leaf. We allow for two nodes between the leaf and the root.



Along the path to the root, we add siblings that can be red or black. We also add a couple of nodes to allow double-red violations. Each of the seven white nodes will be filled in with red or black, yielding 128 test cases. We also test all mirror images, for a total of 256 test cases.

Of course, if we fill in arbitrary combinations of red and black, the result may not be a red-black tree. First off, we add completely black subtrees



to each leaf so that the black height is constant (and equal to the black height of the node to be deleted). Then we remove trees with double-red violations. For the remaining trees, we fill in data values 1, 2, 3, so that we have a binary search tree. Then we remove the target node and check that the tree is still a proper red-black tree and that it contains the required values.

This seems like an ambitious undertaking, but it is better than the alternative—laboriously constructing a set of test cases by hand. It also provides good practice for working with trees.

In order to facilitate this style of testing, the data members of the `Node` and `RedBlackTree` classes are public.

The following function produces the template for testing:

```
/**
   Makes a template for testing removal.
   @return a partially complete red black tree for the test.
   The node to be removed is black.
*/
RedBlackTree removal_test_template()
{
   RedBlackTree result;

   /*
                           n7
                          /  \
                        n1   n8
                       /  \
                     n0    n3
                          /  \
                        n2*  n5
                             /\
                           n4  n6
   */

   Node* n[9];
   for (int i = 0; i < 9; i++) { n[i] = new Node; }
   result.root = n[7];
   n[7]->set_left_child(n[1]);
   n[7]->set_right_child(n[8]);
   n[1]->set_left_child(n[0]);
   n[1]->set_right_child(n[3]);
   n[3]->set_left_child(n[2]);
   n[3]->set_right_child(n[5]);
   n[5]->set_left_child(n[4]);
   n[5]->set_right_child(n[6]);
```

```
   n[2]->color = BLACK;
   return result;
}
```

Because each test changes the shape of the tree, we want to make a copy of the template in each test. The following recursive function makes a copy of a tree:

```
/**
   Copies all nodes of a red-black tree.
   @param n the root of a red-black tree
   @return the root node of a copy of the tree
*/
Node* copy(Node* n)
{
   if (n == nullptr) { return nullptr; }
   Node* new_node = new Node;
   new_node->set_left_child(copy(n->left));
   new_node->set_right_child(copy(n->right));
   new_node->data = n->data;
   new_node->color = n->color;
   return new_node;
}
```

To make a mirror image instead of a copy, just swap the left and right child:

```
/**
   Generates the mirror image of a red black tree.
   @param n the root of the tree to reflect
   @return the root of the mirror image of the tree
*/
Node* mirror(Node* n)
{
   if (n == nullptr) { return nullptr; }
   Node* new_node = new Node;
   new_node->set_left_child(mirror(n->right));
   new_node->set_right_child(mirror(n->left));
   new_node->data = n->data;
   new_node->color = n->color;
   return new_node;
}
```

We want to test all possible combinations of red and black nodes in the template. Each pattern of reds and blacks can be represented as a sequence of zeroes and ones, or a binary number between 0 and $2^n - 1$, where n is the number of nodes to be colored.

```
for (int k = 0; k < pow(2, nodes_to_color); k++)
{
   RedBlackTree rb; // The nodes to be colored
   if (m == 0) { rb.root = copy(t.root); }
   else { rb.root = mirror(t.root); }
   vector<Node*> nodes;
   get_nodes(rb.root, nodes);
   Node* to_delete = nullptr;

   // Color with the bit pattern of k
   int bits = k;
   for (Node* n : nodes)
   {
      if (n == rb.root)
      {
         n->color = BLACK;
```

```
      }
      else if (n->color == BLACK)
      {
         to_delete = n;
      }
      else
      {
         n->color = bits % 2;
         bits = bits / 2;
      }
   }

   // Now run a test with this tree
   . . .
}
```

We need to have a helper function to get all nodes of a tree into a vector. Here it is:

```
/**
   Gets all nodes of a subtree and fills them into a vector.
   @param n the root of the subtree
   @param nodes the vector into which to place the nodes
*/
void get_nodes(Node* n, vector<Node*>& nodes)
{
   if (n == nullptr) { return; }
   get_nodes(n->left, nodes);
   nodes.push_back(n);
   get_nodes(n->right, nodes);
}
```

Once the tree has been colored, we need to give it a constant black height. For each leaf, we compute the cost to the root:

```
/**
   Computes the cost from a node to a root.
   @param n a node of a red-black tree
   @return the number of black nodes between n and the root
*/
int cost_to_root(Node* n)
{
   int c = 0;
   while (n != nullptr) { c = c + n->color; n = n->parent; }
   return c;
}
```

If that cost is less than the black height of the node to be removed, we add a full tree of black nodes to make up the difference. This function makes these trees:

```
/**
   Makes a full tree of black nodes of a given depth.
   @param depth the desired depth
   @return the root node of a full black tree
*/
Node* full_tree(int depth)
{
   if (depth <= 0) { return nullptr; }
   Node* r = new Node;
   r->color = BLACK;
   r->set_left_child(full_tree(depth - 1));
   r->set_right_child(full_tree(depth - 1));
   return r;
}
```

This loop adds the full trees to the nodes:

```
int target_cost = cost_to_root(to_delete);
for (Node* n : nodes)
{
   int cost = target_cost - cost_to_root(n);
   if (n->left == nullptr)
   {
      n->set_left_child(full_tree(cost));
   }
   if (n->right == nullptr)
   {
      n->set_right_child(full_tree(cost));
   }
}
```

Now we need to fill the tree with values. Because `get_nodes` returns the nodes in sorted order, we just populate them with A, B, C, and so on.

```
/**
   Populates a tree with the values A, B, C, ... .
   @param t a red-black tree
   @return the number of nodes in t
*/
int populate(RedBlackTree t)
{
   vector<Node*> nodes;
   get_nodes(t.root, nodes);
   for (int i = 0; i < nodes.size(); i++)
   {
      string d = "A";
      d[0] = d[0] + i;
      nodes[i]->data = d;
   }
   return nodes.size();
}
```

The resulting tree might not be a valid red-black tree. It might have some leaves with greater black height than the node to be removed, or it might have double-red violations. We will develop a function to check that a red-black tree is valid and call it before and after the removal. We also want to verify that all the parent and child links are not corrupted. Because removal introduces colors other than red or black (e.g., double-black or negative-red), we want to check that those colors are no longer present after the operation has completed. Specifically, we need to check the following for each subtree with root `n`:

- The left and right subtree of `n` have the same black depth.
- `n` must be red or black.
- If `n` is red, its parent is not.
- If `n` has children, then their parent pointers must equal `n`.
- `n->parent` is `nullptr` if and only if `n` is the root of the tree.
- The root is black.

Moreover, because fixing double-red and negative-red violations reorders nodes, we will check that the tree is still a binary search tree. This can be tested by visiting the tree in order.

Here are the integrity check functions. The `report_errors` parameter is set to `false` when we want to test whether a tree is valid before removing an element. It is set to `true` when testing that it remains valid after removal.

```
/**
   Checks whether a red-black tree is valid and reports an error if not.
```

```
   @param t the tree to test
   @param report_errors whether error messages should be printed
   @return true if the tree passes, false if not
*/
bool check_red_black(RedBlackTree& t, bool report_errors)
{
   int result = check_red_black(t.root, true, report_errors);
   if (result == -1) { return false; }

   // Check that it's a BST
   vector<Node*> nodes;
   get_nodes(t.root, nodes);
   for (int i = 0; i < nodes.size() - 1; i++)
   {
      if (nodes[i]->data > nodes[i + 1]->data > 0)
      {
         if (report_errors)
         {
            cout << nodes[i]->data << " is larger than " <<
               nodes[i + 1]->data << endl;
         }
         return false;
      }
   }
   return true;
}

/**
   Checks that the tree with the given node is a red-black tree
   and prints an error message if a structural error is found.
   @param n the root of the subtree to check
   @param is_root true if this is the root of the tree
   @param report_errors whether error messages should be printed
   @return the black depth of this subtree, or -1 if this is not a
   valid red-black tree
*/
int check_red_black(Node* n, bool is_root, bool report_errors)
{
   if (n == nullptr) { return 0; }
   int nleft = check_red_black(n->left, false, report_errors);
   int nright = check_red_black(n->right, false, report_errors);
   if (nleft == -1 || nright == -1) return -1;
   if (nleft != nright)
   {
      if (report_errors)
      {
         cout << "Left and right children of " << n->data
            << " have different black depths" << endl;
      }
      return -1;
   }
   if (n->parent == nullptr)
   {
      if (!is_root)
      {
         if (report_errors)
         {
            cout << n->data << " is not root and has no parent" << endl;
         }
         return -1;
```

```
      }
      if (n->color != BLACK)
      {
         if (report_errors)
         {
            cout << "Root " << n->data << " is not black";
         }
         return -1;
      }
   }
   else
   {
      if (is_root)
      {
         if (report_errors)
         {
            cout << n->data << " is root and has a parent" << endl;
         }
         return -1;
      }
      if (n->color == RED && n->parent->color == RED)
      {
         if (report_errors)
         {
            cout << "Parent of red " << n->data << " is red" << endl;
         }
         return -1;
      }
   }
   if (n->left != nullptr && n->left->parent != n)
   {
      if (report_errors)
      {
         cout << "Left child of " << n->data
            << " has bad parent link" << endl;
      }
      return -1;
   }
   if (n->right != nullptr && n->right->parent != n)
   {
      if (report_errors)
      {
         cout << "Right child of " << n->data
            << " has bad parent link" << endl;
      }
      return -1;
   }
   if (n->color != RED && n->color != BLACK)
   {
      if (report_errors)
      {
         cout << n->data + " has color " << n->color;
      }
      return -1;
   }
   return n->color + nleft;
}
```

Now we have all the pieces together. Here is the complete function for testing removal. Note that the outer loop switches between copying and mirroring, and the inner loop iterates over all red/black colorings.

```
/**
   Tests removal, given a template for a tree with a black node that
   is to be deleted. All other nodes should be given all possible combinations
   of red and black.
   @param t the template for the test cases
   @return the number of passing tests
*/
int removal_test(const RedBlackTree& t)
{
   int count = 0;
   for (int m = 0; m <= 1; m++)
   {
      int nodes_to_color = size(t.root) - 2;
        // We don't recolor the root or to_delete
      for (int k = 0; k < pow(2, nodes_to_color); k++)
      {
         RedBlackTree rb;
         if (m == 0) { rb.root = copy(t.root); }
         else { rb.root = mirror(t.root); }
         vector<Node*> nodes;
         get_nodes(rb.root, nodes);
         Node* to_delete = nullptr;

         // Color with the bit pattern of k
         int bits = k;
         for (Node* n : nodes)
         {
            if (n == rb.root)
            {
               n->color = BLACK;
            }
            else if (n->color == BLACK)
            {
               to_delete = n;
            }
            else
            {
               n->color = bits % 2;
               bits = bits / 2;
            }
         }

         // Add children to make equal costs to nullptr
         int target_cost = cost_to_root(to_delete);
         for (Node* n : nodes)
         {
            int cost = target_cost - cost_to_root(n);
            if (n->left == nullptr)
            {
               n->set_left_child(full_tree(cost));
            }
            if (n->right == nullptr)
            {
               n->set_right_child(full_tree(cost));
            }
         }
```

```
            int filled_size = populate(rb);

            if (check_red_black(rb, false)) // Found a valid tree
            {
               string d = to_delete->data;
               rb.erase(d);
               bool pass = check_red_black(rb, true);
               for (int j = 0; j < filled_size; j++)
               {
                  string s = "A";
                  s[0] = s[0] + j;
                  if (rb.count(s) == 0 && d != s)
                  {
                     cout << s + " deleted" << endl;
                     pass = false;
                  }
               }
               if (rb.count(d) > 0)
               {
                  cout << d + " not deleted" << endl;
                  pass = false;
               }
               if (pass)
               {
                  count++;
               }
               else
               {
                  cout << "Failing tree: " << endl;
                  print_detailed(rb.root, 0);
               }
            }
         }
      }
      return count;
   }
```

In our `main` function, we run all three tests. There are 3,628,800 insertion tests and 32 removal tests. (Of the 256 trees generated, 224 are not valid red-black trees.) Because no error messages are displayed, we conclude that all tests pass.

```
int main()
{
   test_from_book();
   int passing = insertion_test("ABCDEFGHIJ");
   cout << passing << " insertion tests passed." << endl;
   passing = removal_test(removal_test_template());
   cout << passing << " removal tests passed." << endl;
   return 0;
}
```

worked_example_2/red_black_tree.h

```
1  #ifndef RED_BLACK_TREE_H
2  #define RED_BLACK_TREE_H
3
4  #include <string>
5
6  using namespace std;
7
```

```
 8  const int BLACK = 1;
 9  const int RED = 0;
10  const int NEGATIVE_RED = -1;
11  const int DOUBLE_BLACK = 2;
12
13  /**
14     A node of a red-black tree stores a data item and references
15     of the child nodes to the left and to the right. The color
16     is the "cost" of passing the node; 1 for black or 0 for red.
17     Temporarily, it may be set to 2 or -1.
18  */
19  class Node
20  {
21  public:
22     /**
23        Constructs a red node with no data.
24     */
25     Node();
26
27  private:
28     /**
29        Adds a node as a child of this node.
30        @param new_node the node to add
31     */
32     void add_node(Node* new_node);
33  public: // These members are public for testing
34     /**
35        Sets the left child and updates its parent reference.
36        @param child the new left child
37     */
38     void set_left_child(Node* child);
39
40     /**
41        Sets the right child and updates its parent reference.
42        @param child the new right child
43     */
44     void set_right_child(Node* child);
45
46     string data;
47     Node* left;
48     Node* right;
49     Node* parent;
50     int color;
51
52  friend class RedBlackTree;
53  };
54
55  /*
56     This class implements a red-black tree.
57  */
58  class RedBlackTree
59  {
60  public:
61     /**
62        Constructs an empty tree.
63     */
64     RedBlackTree();
65
66     /**
67        Inserts a new element into the tree.
```

```
68          @param element the element to insert
69       */
70       void insert(string element);
71
72       /**
73          Tries to find an element in the tree.
74          @param element the element to find
75          @return 1 if the element is contained in the tree, 0 otherwise
76       */
77       int count(string element) const;
78
79       /**
80          Tries to remove an element from the tree. Does nothing
81          if the element is not contained in the tree.
82          @param element the element to remove
83       */
84       void erase(string element);
85
86       /**
87          Prints the contents of the tree in sorted order.
88       */
89       void print() const;
90
91    private:
92       /**
93          Prints a node and all of its descendants in sorted order.
94          @param parent the root of the subtree to print
95       */
96       void print(Node* parent) const;
97
98       /**
99          Updates the parent's and replacement node's
100         links when this node is replaced.
101         Also updates the root reference if it is replaced.
102         @param to_be_replaced the node that is to be replaced
103         @param replacement the node that replaces that node
104      */
105      void replace_with(Node* to_be_replaced, Node* replacement);
106
107      /**
108         Restores the tree to a red-black tree after a node has been added.
109         @param new_node the node that has been added
110      */
111      void fix_after_add(Node* new_node);
112
113      /**
114        Fixes the tree so that it is a red-black tree after a node has been removed.
115        @param to_be_removed the node that is to be removed
116      */
117      void fix_before_remove(Node* to_be_removed);
118
119      /**
120         Move a charge from two children of a parent.
121         @param parent a node with two children, or
122         nullptr (in which case nothing is done)
123      */
124      void bubble_up(Node* parent);
125
126      /**
127         Fixes a negative-red or double-red violation introduced by bubbling up.
```

```
      @param child the child to check for negative-red
      or double-red violations
      @return true if the tree was fixed
   */
   bool bubble_up_fix(Node* child);

   /**
      Fixes a "double red" violation.
      @param child the child with a red parent
   */
   void fix_double_red(Node* child);

   /**
      Fixes a "negative red" violation.
      @param neg_red the negative red node
   */
   void fix_negative_red(Node* neg_red);

public: // for testing
   Node* root;
};

#endif
```

worked_example_2/red_black_tree.cpp

```
#include "red_black_tree.h"

#include <iostream>

using namespace std;

RedBlackTree::RedBlackTree()
{
   root = nullptr;
}

void RedBlackTree::insert(string element)
{
   Node* new_node = new Node;
   new_node->data = element;
   new_node->left = nullptr;
   new_node->right = nullptr;
   if (root == nullptr) { root = new_node; }
   else { root->add_node(new_node); }
   fix_after_add(new_node);
}

int RedBlackTree::count(string element) const
{
   Node* current = root;
   while (current != nullptr)
   {
      if (element < current->data)
      {
         current = current->left;
      }
      else if (element > current->data)
      {
         current = current->right;
```

```
      }
      else return 1;
   }
   return 0;
}

void RedBlackTree::erase(string element)
{
   // Find node to be removed

   Node* to_be_removed = root;
   bool found = false;
   while (!found && to_be_removed != nullptr)
   {
      if (element == to_be_removed->data)
      {
         found = true;
      }
      else if (element < to_be_removed->data)
      {
         to_be_removed = to_be_removed->left;
      }
      else
      {
         to_be_removed = to_be_removed->right;
      }
   }

   if (!found) { return; }

   // to_be_removed contains element

   // If one of the children is empty, use the other

   if (to_be_removed->left == nullptr || to_be_removed->right == nullptr)
   {
      Node* new_child;
      if (to_be_removed->left == nullptr)
      {
         new_child = to_be_removed->right;
      }
      else { new_child = to_be_removed->left; }

      fix_before_remove(to_be_removed);
      replace_with(to_be_removed, new_child);
      return;
   }

   // Neither subtree is empty

   // Find smallest element of the right subtree

   Node* smallest = to_be_removed->right;
   while (smallest->left != nullptr)
   {
      smallest = smallest->left;
   }

   // smallest contains smallest child in right subtree

```

```
95     // Move contents, unlink child
96
97     to_be_removed->data = smallest->data;
98     fix_before_remove(smallest);
99     replace_with(smallest, smallest->right);
100 }
101
102 void RedBlackTree::print() const
103 {
104    print(root);
105    cout << endl;
106 }
107
108 void RedBlackTree::print(Node* parent) const
109 {
110    if (parent == nullptr) { return; }
111    print(parent->left);
112    cout << parent->data << " ";
113    print(parent->right);
114 }
115
116 Node::Node()
117 {
118    left = nullptr;
119    right = nullptr;
120    parent = nullptr;
121    color = RED;
122 }
123
124 void Node::set_left_child(Node* child)
125 {
126    left = child;
127    if (child != nullptr) { child->parent = this; }
128 }
129
130 void Node::set_right_child(Node* child)
131 {
132    right = child;
133    if (child != nullptr) { child->parent = this; }
134 }
135
136 void Node::add_node(Node* new_node)
137 {
138    if (new_node->data < data)
139    {
140       if (left == nullptr)
141       {
142          left = new_node;
143          left->parent = this;
144       }
145       else { left->add_node(new_node); }
146    }
147    else if (new_node->data > data)
148    {
149       if (right == nullptr)
150       {
151          right = new_node;
152          right->parent = this;
153       }
154       else { right->add_node(new_node); }
```

```
155    }
156 }
157
158
159 void RedBlackTree::replace_with(Node* to_be_replaced, Node* replacement)
160 {
161    if (to_be_replaced->parent == nullptr)
162    {
163       replacement->parent = nullptr;
164       root = replacement;
165    }
166    else if (to_be_replaced == to_be_replaced->parent->left)
167    {
168       to_be_replaced->parent->set_left_child(replacement);
169    }
170    else
171    {
172       to_be_replaced->parent->set_right_child(replacement);
173    }
174 }
175
176 void RedBlackTree::fix_after_add(Node* new_node)
177 {
178    if (new_node->parent == nullptr)
179    {
180       new_node->color = BLACK;
181    }
182    else
183    {
184       new_node->color = RED;
185       if (new_node->parent->color == RED) { fix_double_red(new_node); }
186    }
187 }
188
189 void RedBlackTree::fix_before_remove(Node* to_be_removed)
190 {
191    if (to_be_removed->color == RED) { return; }
192
193    if (to_be_removed->left != nullptr
194          || to_be_removed->right != nullptr) // It is not a leaf
195    {
196       // Color the child black
197       if (to_be_removed->left == nullptr)
198       {
199          to_be_removed->right->color = BLACK;
200       }
201       else { to_be_removed->left->color = BLACK; }
202    }
203    else { bubble_up(to_be_removed->parent); }
204 }
205
206 void RedBlackTree::bubble_up(Node* parent)
207 {
208    if (parent == nullptr) { return; }
209    parent->color++;
210    parent->left->color--;
211    parent->right->color--;
212
213    if (bubble_up_fix(parent->left)) { return; }
214    if (bubble_up_fix(parent->right)) { return; }
```

```
215
216    if (parent->color == DOUBLE_BLACK)
217    {
218       if (parent->parent == nullptr) { parent->color = BLACK; }
219       else { bubble_up(parent->parent); }
220    }
221 }
222
223 bool RedBlackTree::bubble_up_fix(Node* child)
224 {
225    if (child->color == NEGATIVE_RED)
226    {
227       fix_negative_red(child);
228       return true;
229    }
230    else if (child->color == RED)
231    {
232       if (child->left != nullptr && child->left->color == RED)
233       {
234          fix_double_red(child->left); return true;
235       }
236       if (child->right != nullptr && child->right->color == RED)
237       {
238          fix_double_red(child->right); return true;
239       }
240    }
241    return false;
242 }
243
244 void RedBlackTree::fix_double_red(Node* child)
245 {
246    Node* parent = child->parent;
247    Node* grandparent = parent->parent;
248    if (grandparent == nullptr) { parent->color = BLACK; return; }
249    Node* n1;
250    Node* n2;
251    Node* n3;
252    Node* t1;
253    Node* t2;
254    Node* t3;
255    Node* t4;
256    if (parent == grandparent->left)
257    {
258       n3 = grandparent; t4 = grandparent->right;
259       if (child == parent->left)
260       {
261          n1 = child; n2 = parent;
262          t1 = child->left; t2 = child->right; t3 = parent->right;
263       }
264       else
265       {
266          n1 = parent; n2 = child;
267          t1 = parent->left; t2 = child->left; t3 = child->right;
268       }
269    }
270    else
271    {
272       n1 = grandparent; t1 = grandparent->left;
273       if (child == parent->left)
274       {
```

```
275            n2 = child; n3 = parent;
276            t2 = child->left; t3 = child->right; t4 = parent->right;
277         }
278         else
279         {
280            n2 = parent; n3 = child;
281            t2 = parent->left; t3 = child->left; t4 = child->right;
282         }
283      }
284
285      replace_with(grandparent, n2);
286      n1->set_left_child(t1);
287      n1->set_right_child(t2);
288      n2->set_left_child(n1);
289      n2->set_right_child(n3);
290      n3->set_left_child(t3);
291      n3->set_right_child(t4);
292      n2->color = grandparent->color - 1;
293      n1->color = BLACK;
294      n3->color = BLACK;
295
296      if (n2 == root)
297      {
298         root->color = BLACK;
299      }
300      else if (n2->color == RED && n2->parent->color == RED)
301      {
302         fix_double_red(n2);
303      }
304   }
305
306   void RedBlackTree::fix_negative_red(Node* neg_red)
307   {
308      Node* parent = neg_red->parent;
309      Node* child;
310      if (parent->left == neg_red)
311      {
312         Node* n1 = neg_red->left;
313         Node* n2 = neg_red;
314         Node* n3 = neg_red->right;
315         Node* n4 = parent;
316         Node* t1 = n3->left;
317         Node* t2 = n3->right;
318         Node* t3 = n4->right;
319         n1->color = RED;
320         n2->color = BLACK;
321         n4->color = BLACK;
322
323         replace_with(n4, n3);
324         n3->set_left_child(n2);
325         n3->set_right_child(n4);
326         n2->set_left_child(n1);
327         n2->set_right_child(t1);
328         n4->set_left_child(t2);
329         n4->set_right_child(t3);
330
331         child = n1;
332      }
333      else // Mirror image
334      {
```

```
      Node* n4 = neg_red->right;
      Node* n3 = neg_red;
      Node* n2 = neg_red->left;
      Node* n1 = parent;
      Node* t3 = n2->right;
      Node* t2 = n2->left;
      Node* t1 = n1->left;
      n4->color = RED;
      n3->color = BLACK;
      n1->color = BLACK;

      replace_with(n1, n2);
      n2->set_right_child(n3);
      n2->set_left_child(n1);
      n3->set_right_child(n4);
      n3->set_left_child(t3);
      n1->set_right_child(t2);
      n1->set_left_child(t1);

      child = n4;
   }

   if (child->left != nullptr && child->left->color == RED)
   {
      fix_double_red(child->left);
   }
   else if (child->right != nullptr && child->right->color == RED)
   {
      fix_double_red(child->right);
   }
}
```

worked_example_2/treedemo.cpp

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include "red_black_tree.h"

using namespace std;

/*
   This program tests the red-black tree class.
*/

/**
   Runs the simple test from the textbook.
*/
void test_from_book()
{
   RedBlackTree t;
   t.insert("D");
   t.insert("B");
   t.insert("A");
   t.insert("C");
   t.insert("F");
   t.insert("E");
   t.insert("I");
   t.insert("G");
   t.insert("H");
```

```
   t.insert("J");
   t.erase("A"); // Removing leaf
   t.erase("B"); // Removing element with one child
   t.erase("F"); // Removing element with two children
   t.erase("D"); // Removing root
   t.print();
   cout << "Expected: C E G H I J" << endl;
}

/**
   Inserts all permutations of a string into a red-black tree and checks that
   it contains the strings afterward.
   @param letters a sorted string of letters without repetition
   @return the number of passing tests
*/
int insertion_test(string letters)
{
   int count = 0;
   string s = letters;
   do
   {
      RedBlackTree t;
      for (int i = 0; i < s.length(); i++)
      {
         t.insert(s.substr(i, 1));
      }
      bool pass = true;
      for (int i = 0; i < s.length(); i++)
      {
         string e = s.substr(i, 1);
         if (t.count(e) == 0)
         {
            cout << e << " not inserted" << endl;
            pass = false;
         }
      }
      if (pass)
      {
         count++;
      }
      else
      {
         cout << "Failing for letters " << letters << endl;
      }
   }
   while(next_permutation(s.begin(), s.end()));
   return count;
}

/**
   Counts the nodes in a binary tree.
   @param n the root of a binary tree
   @return the number of nodes in the tree
*/
int size(Node* n)
{
   if (n == nullptr) { return 0; }
   else { return 1 + size(n->left) + size(n->right); }
}
```

```
87  /**
88     Computes the cost from a node to a root.
89     @param n a node of a red-black tree
90     @return the number of black nodes between n and the root
91  */
92  int cost_to_root(Node* n)
93  {
94     int c = 0;
95     while (n != nullptr) { c = c + n->color; n = n->parent; }
96     return c;
97  }
98
99  /**
100    Copies all nodes of a red-black tree.
101    @param n the root of a red-black tree
102    @return the root node of a copy of the tree
103 */
104 Node* copy(Node* n)
105 {
106    if (n == nullptr) { return nullptr; }
107    Node* new_node = new Node;
108    new_node->set_left_child(copy(n->left));
109    new_node->set_right_child(copy(n->right));
110    new_node->data = n->data;
111    new_node->color = n->color;
112    return new_node;
113 }
114
115 /**
116    Generates the mirror image of a red-black tree.
117    @param n the root of the tree to reflect
118    @return the root of the mirror image of the tree
119 */
120 Node* mirror(Node* n)
121 {
122    if (n == nullptr) { return nullptr; }
123    Node* new_node = new Node;
124    new_node->set_left_child(mirror(n->right));
125    new_node->set_right_child(mirror(n->left));
126    new_node->data = n->data;
127    new_node->color = n->color;
128    return new_node;
129 }
130
131 /**
132    Makes a full tree of black nodes of a given depth.
133    @param depth the desired depth
134    @return the root node of a full black tree
135 */
136 Node* full_tree(int depth)
137 {
138    if (depth <= 0) { return nullptr; }
139    Node* r = new Node;
140    r->color = BLACK;
141    r->set_left_child(full_tree(depth - 1));
142    r->set_right_child(full_tree(depth - 1));
143    return r;
144 }
145
146 /**
```

```
147     Gets all nodes of a subtree and fills them into a vector.
148     @param n the root of the subtree
149     @param nodes the vector into which to place the nodes
150  */
151  void get_nodes(Node* n, vector<Node*>& nodes)
152  {
153     if (n == nullptr) { return; }
154     get_nodes(n->left, nodes);
155     nodes.push_back(n);
156     get_nodes(n->right, nodes);
157  }
158
159  /**
160     Prints a detailed view of a binary tree.
161     @param n the root of the tree to print
162     @param level the indentation level for the root
163  */
164  void print_detailed(Node* n, int level)
165  {
166     if (n == nullptr) { return; }
167     print_detailed(n->left, level + 1);
168     for (int i = 0; i < level; i++) { cout << "  "; }
169     cout << n->data << " " << n->color << endl;
170     print_detailed(n->right, level + 1);
171  }
172
173  /**
174     Populates a tree with the values A, B, C, ... .
175     @param t a red-black tree
176     @return the number of nodes in t
177  */
178  int populate(RedBlackTree t)
179  {
180     vector<Node*> nodes;
181     get_nodes(t.root, nodes);
182     for (int i = 0; i < nodes.size(); i++)
183     {
184        string d = "A";
185        d[0] = d[0] + i;
186        nodes[i]->data = d;
187     }
188     return nodes.size();
189  }
190
191  /**
192     Checks that the tree with the given node is a red-black tree
193     and prints an error message if a structural error is found.
194     @param n the root of the subtree to check
195     @param is_root true if this is the root of the tree
196     @param report_errors whether error messages should be printed
197     @return the black depth of this subtree, or -1 if this is not a
198     valid red-black tree
199  */
200  int check_red_black(Node* n, bool is_root, bool report_errors)
201  {
202     if (n == nullptr) { return 0; }
203     int nleft = check_red_black(n->left, false, report_errors);
204     int nright = check_red_black(n->right, false, report_errors);
205     if (nleft == -1 || nright == -1) return -1;
```

```
206    if (nleft != nright)
207    {
208       if (report_errors)
209       {
210          cout << "Left and right children of " << n->data
211             << " have different black depths" << endl;
212       }
213       return -1;
214    }
215    if (n->parent == nullptr)
216    {
217       if (!is_root)
218       {
219          if (report_errors)
220          {
221             cout << n->data << " is not root and has no parent" << endl;
222          }
223          return -1;
224       }
225       if (n->color != BLACK)
226       {
227          if (report_errors)
228          {
229             cout << "Root " << n->data << " is not black";
230          }
231          return -1;
232       }
233    }
234    else
235    {
236       if (is_root)
237       {
238          if (report_errors)
239          {
240             cout << n->data << " is root and has a parent" << endl;
241          }
242          return -1;
243       }
244       if (n->color == RED && n->parent->color == RED)
245       {
246          if (report_errors)
247          {
248             cout << "Parent of red " << n->data << " is red" << endl;
249          }
250          return -1;
251       }
252    }
253    if (n->left != nullptr && n->left->parent != n)
254    {
255       if (report_errors)
256       {
257          cout << "Left child of " << n->data
258             << " has bad parent link" << endl;
259       }
260       return -1;
261    }
262    if (n->right != nullptr && n->right->parent != n)
263    {
264       if (report_errors)
265       {
```

```
266          cout << "Right child of " << n->data
267             << " has bad parent link" << endl;
268       }
269       return -1;
270    }
271    if (n->color != RED && n->color != BLACK)
272    {
273       if (report_errors)
274       {
275          cout << n->data + " has color " << n->color;
276       }
277       return -1;
278    }
279    return n->color + nleft;
280 }
281
282 /**
283    Checks whether a red-black tree is valid and reports an error if not.
284    @param t the tree to test
285    @param report_errors whether error messages should be printed
286    @return true if the tree passes, false if not
287 */
288 bool check_red_black(RedBlackTree& t, bool report_errors)
289 {
290    int result = check_red_black(t.root, true, report_errors);
291    if (result == -1) { return false; }
292
293    // Check that it's a BST
294    vector<Node*> nodes;
295    get_nodes(t.root, nodes);
296    for (int i = 0; i < nodes.size() - 1; i++)
297    {
298       if (nodes[i]->data > nodes[i + 1]->data > 0)
299       {
300          if (report_errors)
301          {
302             cout << nodes[i]->data << " is larger than " <<
303                nodes[i + 1]->data << endl;
304          }
305          return false;
306       }
307    }
308    return true;
309 }
310
311 /**
312    Tests removal, given a template for a tree with a black node that
313    is to be deleted. All other nodes should be given all possible combinations
314    of red and black.
315    @param t the template for the test cases
316    @return the number of passing tests
317 */
318 int removal_test(const RedBlackTree& t)
319 {
320    int count = 0;
321    for (int m = 0; m <= 1; m++)
322    {
323       int nodes_to_color = size(t.root) - 2;
324         // We don't recolor the root or to_delete
325       for (int k = 0; k < pow(2, nodes_to_color); k++)
```

```
326        {
327           RedBlackTree rb;
328           if (m == 0) { rb.root = copy(t.root); }
329           else { rb.root = mirror(t.root); }
330           vector<Node*> nodes;
331           get_nodes(rb.root, nodes);
332           Node* to_delete = nullptr;
333
334           // Color with the bit pattern of k
335           int bits = k;
336           for (Node* n : nodes)
337           {
338              if (n == rb.root)
339              {
340                 n->color = BLACK;
341              }
342              else if (n->color == BLACK)
343              {
344                 to_delete = n;
345              }
346              else
347              {
348                 n->color = bits % 2;
349                 bits = bits / 2;
350              }
351           }
352
353           // Add children to make equal costs to nullptr
354           int target_cost = cost_to_root(to_delete);
355           for (Node* n : nodes)
356           {
357              int cost = target_cost - cost_to_root(n);
358              if (n->left == nullptr)
359              {
360                 n->set_left_child(full_tree(cost));
361              }
362              if (n->right == nullptr)
363              {
364                 n->set_right_child(full_tree(cost));
365              }
366           }
367
368           int filled_size = populate(rb);
369
370           if (check_red_black(rb, false)) // Found a valid tree
371           {
372              string d = to_delete->data;
373              rb.erase(d);
374              bool pass = check_red_black(rb, true);
375              for (int j = 0; j < filled_size; j++)
376              {
377                 string s = "A";
378                 s[0] = s[0] + j;
379                 if (rb.count(s) == 0 && d != s)
380                 {
381                    cout << s + " deleted" << endl;
382                    pass = false;
383                 }
384              }
385              if (rb.count(d) > 0)
```

```
            {
               cout << d + " not deleted" << endl;
               pass = false;
            }
            if (pass)
            {
               count++;
            }
            else
            {
               cout << "Failing tree: " << endl;
               print_detailed(rb.root, 0);
            }
         }
      }
   }
   return count;
}

/**
   Makes a template for testing removal.
   The node to be removed is black.
   @return a partially complete red-black tree for the test.
*/
RedBlackTree removal_test_template()
{
   RedBlackTree result;

   /*
                            n7
                           /  \
                          n1   n8
                         /  \
                       n0    n3
                            /  \
                           n2*  n5
                                /\
                               n4  n6
   */

   Node* n[9];
   for (int i = 0; i < 9; i++) { n[i] = new Node; }
   result.root = n[7];
   n[7]->set_left_child(n[1]);
   n[7]->set_right_child(n[8]);
   n[1]->set_left_child(n[0]);
   n[1]->set_right_child(n[3]);
   n[3]->set_left_child(n[2]);
   n[3]->set_right_child(n[5]);
   n[5]->set_left_child(n[4]);
   n[5]->set_right_child(n[6]);
   n[2]->color = BLACK;
   return result;
}
```

```
440
441 int main()
442 {
443    test_from_book();
444    int passing = insertion_test("ABCDEFGHIJ");
445    cout << passing << " insertion tests passed." << endl;
446    passing = removal_test(removal_test_template());
447    cout << passing << " removal tests passed." << endl;
448    return 0;
449 }
```

CHAPTER 17

PRIORITY QUEUES AND HEAPS

© tomazl/iStockphoto.

CHAPTER GOALS

- To become familiar with the priority queue data type
- To understand the implementation and efficiency of the heap data structure
- To use heaps for implementing priority queues and the heapsort algorithm

CHAPTER CONTENTS

With a binary search tree, you can efficiently sort a set of elements. Sometimes, you don't need to sort the entire set, but you are just interested in the largest one; for example, the event with the highest priority in a set of events. In this chapter, you will study an ingenious tree structure, called a *heap*, that lets you efficiently find and remove the largest element. By repeatedly removing the maximum, you can sort a sequence. This is the basis of an efficient sorting algorithm called *heapsort*.

17.1 Priority Queues

When removing an element from a priority queue, the element with the highest priority is retrieved.

A **priority queue** is a container optimized for one special task; quickly locating the element with highest priority. Prioritization is a weaker condition than ordering. In a priority queue the order of the remaining elements is irrelevant, it is only the highest priority element that is important.

Consider this example, where a priority queue contains strings denoting tasks:

```
priority_queue<string> tasks;
tasks.push("2 - Shampoo carpets");
tasks.push("9 - Fix overflowing sink");
tasks.push("5 - Order cleaning supplies");
```

The strings are formatted so that they start with a priority number. When it comes time to do work, we will want to retrieve and remove the task with the top priority:

```
string most_important = tasks.top(); // Returns "9 - Fix overflowing sink"
tasks.pop();
```

The term priority *queue* is actually a misnomer, because the priority queue does not have the "first in/first out" behavior as a true queue does. In fact, the interface for the priority queue is more similar to a stack than to a queue. The basic three operations are `push`, `pop`, and `top`. The `push` operation places a new element into the priority queue. `top` returns the element with highest priority; `pop` removes this element.

One obvious implementation for a priority queue is a sorted set. Then it is an easy matter to locate and remove the largest element. However, another data structure, called a **heap**, is even more suitable for implementing priority queues. Heaps store all elements in a single array, which is more efficient than storing each element in a tree node. We will describe heaps in the next section.

When you retrieve an item from a priority queue, you always get the most urgent one.

Table 1 Working with Priority Queues

`priority_queue<int> q;`	This priority queue holds integer objects. In practice, you would use objects that describe tasks.

`q.push(3);`
`q.push(1);`
`q.push(2);`
	Adds values to the priority queue.

`int first = q.top();`	Yields the largest element in the priority queue.
`q.pop();`	Removes the largest element.

`while (q.size() > 0)`
`{`
`  cout << q.top() << endl;`
`  q.pop();`
`}`
	Prints and removes all elements in descending order.

Here is a simple program that demonstrates a priority queue. Instead of storing strings, we use a `Task` class. As described in Section 13.1.2, we supply an `operator<` function that compares tasks so that the priority queue can find the most important one.

sec01/task.h

```
1  #ifndef TASK_H
2  #define TASK_H
3
4  #include <string>
5
6  using namespace std;
7
8  class Task
9  {
10 public:
11    Task(string description, int priority);
12    string get_description() const;
13    int get_priority() const;
14 private:
15    string description;
16    int priority;
17 };
18
19 bool operator<(const Task& a, const Task& b);
20
21 #endif
```

sec01/task.cpp

```
1  #include "task.h"
2
3  Task::Task(string description, int priority)
4  {
5     this->description = description;
6     this->priority = priority;
7  }
```

```
 8
 9  string Task::get_description() const
10  {
11     return description;
12  }
13
14  int Task::get_priority() const
15  {
16     return priority;
17  }
18
19  bool operator<(const Task& a, const Task& b)
20  {
21     return a.get_priority() < b.get_priority();
22  }
```

sec01/pqueue.cpp

```
 1  #include <iostream>
 2  #include <queue>
 3  #include "task.h"
 4
 5  using namespace std;
 6
 7  int main()
 8  {
 9     priority_queue<Task> tasks;
10     tasks.push(Task("Shampoo carpets", 2));
11     tasks.push(Task("Empty trash", 3));
12     tasks.push(Task("Water plants", 2));
13     tasks.push(Task("Remove pencil sharpener shavings", 1));
14     tasks.push(Task("Replace light bulb", 4));
15     tasks.push(Task("Fix overflowing sink", 9));
16     tasks.push(Task("Clean coffee maker", 1));
17     tasks.push(Task("Order cleaning supplies", 5));
18
19     while (tasks.size() > 0)
20     {
21        Task most_important = tasks.top();
22        tasks.pop();
23        cout << most_important.get_priority() << " - "
24           << most_important.get_description() << endl;
25     }
26     return 0;
27  }
```

Program Run

```
 9 - Fix overflowing sink
 5 - Order cleaning supplies
 4 - Replace light bulb
 3 - Empty trash
 2 - Water plants
 2 - Shampoo carpets
 1 - Remove pencil sharpener shavings
 1 - Clean coffee maker
```

WORKED EXAMPLE 17.1

Simulating a Queue of Waiting Customers

Learn how to use a queue to simulate a line of waiting customers. See your E-Text or visit `wiley.com/go/bclo3`.

Jack Hollingsworth/Photodisc/Getty Images.

17.2 Heaps

A heap is an almost complete tree in which the values of all nodes are at least as large as those of their descendants.

A **heap** (or, for greater clarity, *max-heap*) is a binary tree with two special properties:

1. A heap is *almost complete:* all nodes are filled in, except the last level may have some nodes missing toward the right (see Figure 1).

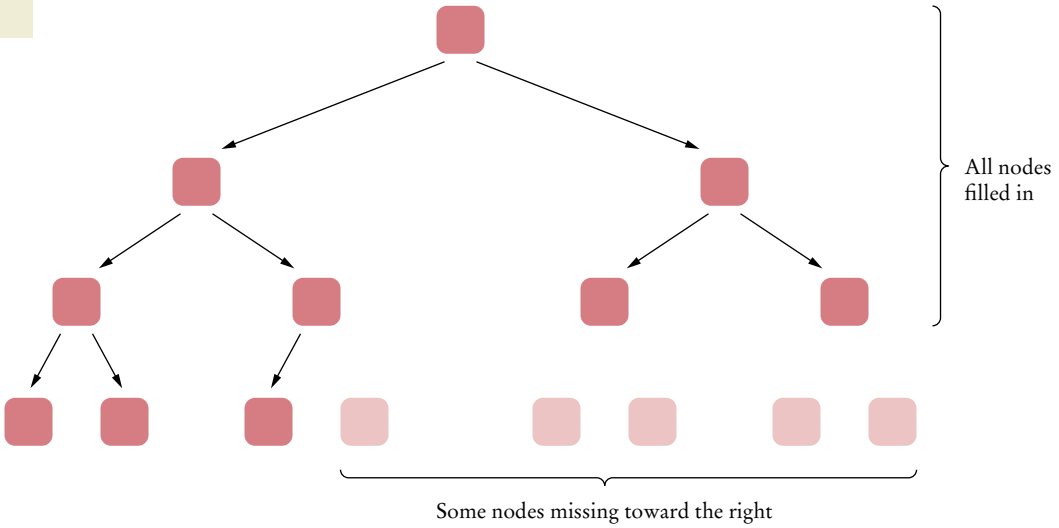


Figure 1 An Almost Complete Tree

2. The tree fulfills the *heap property:* all nodes store values that are at least as large as the values stored in their descendants (see Figure 2).

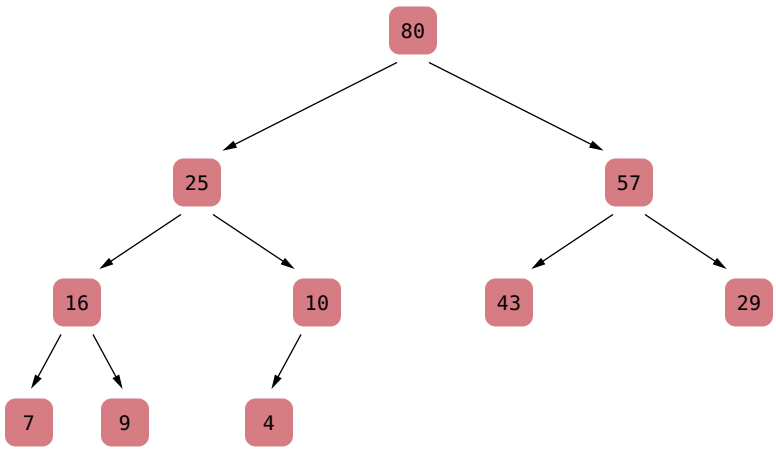


Figure 2 A Heap

It is easy to see that the heap property ensures that the tree's largest element is stored in the root.

A heap is superficially similar to a binary search tree, but there are two important differences:

1. The shape of a heap is very regular. Binary search trees can have arbitrary shapes.
2. In a heap, the left and right subtrees both store elements that are smaller than the root element. In contrast, in a binary search tree, smaller elements are stored in the left subtree and larger elements are stored in the right subtree.

In an almost complete tree, all layers but one are completely filled.

Suppose we have a heap and want to insert a new element. After the insertion, the heap property should again be fulfilled. The following algorithm carries out the insertion (see Figure 3).

1. First, add a vacant slot to the end of the tree.
2. Next, demote the parent of the empty slot if it is smaller than the element to be inserted. That is, move the parent value into the vacant slot, and move the vacant slot up. Repeat this demotion as long as the parent of the vacant slot is smaller than the element to be inserted.
3. At this point, either the vacant slot is at the root, or the parent of the vacant slot is larger than the element to be inserted. Insert the element into the vacant slot.

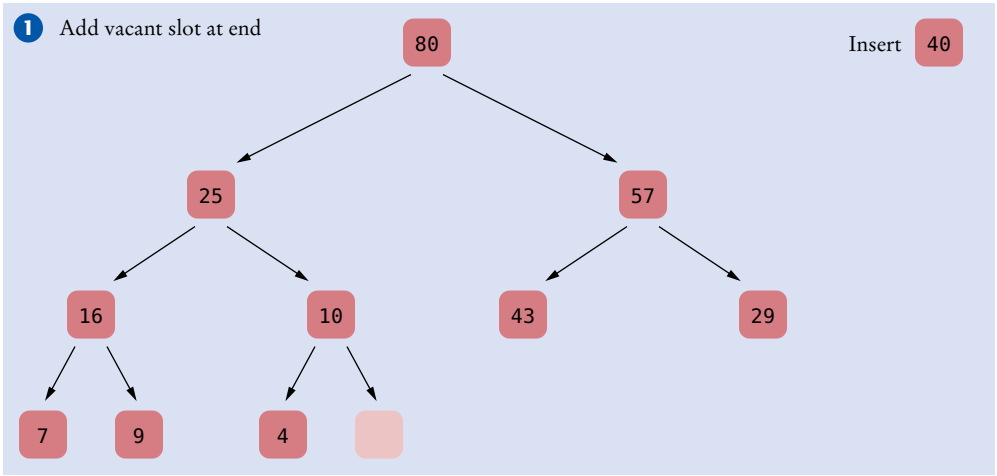


Figure 3 Inserting an Element into a Heap

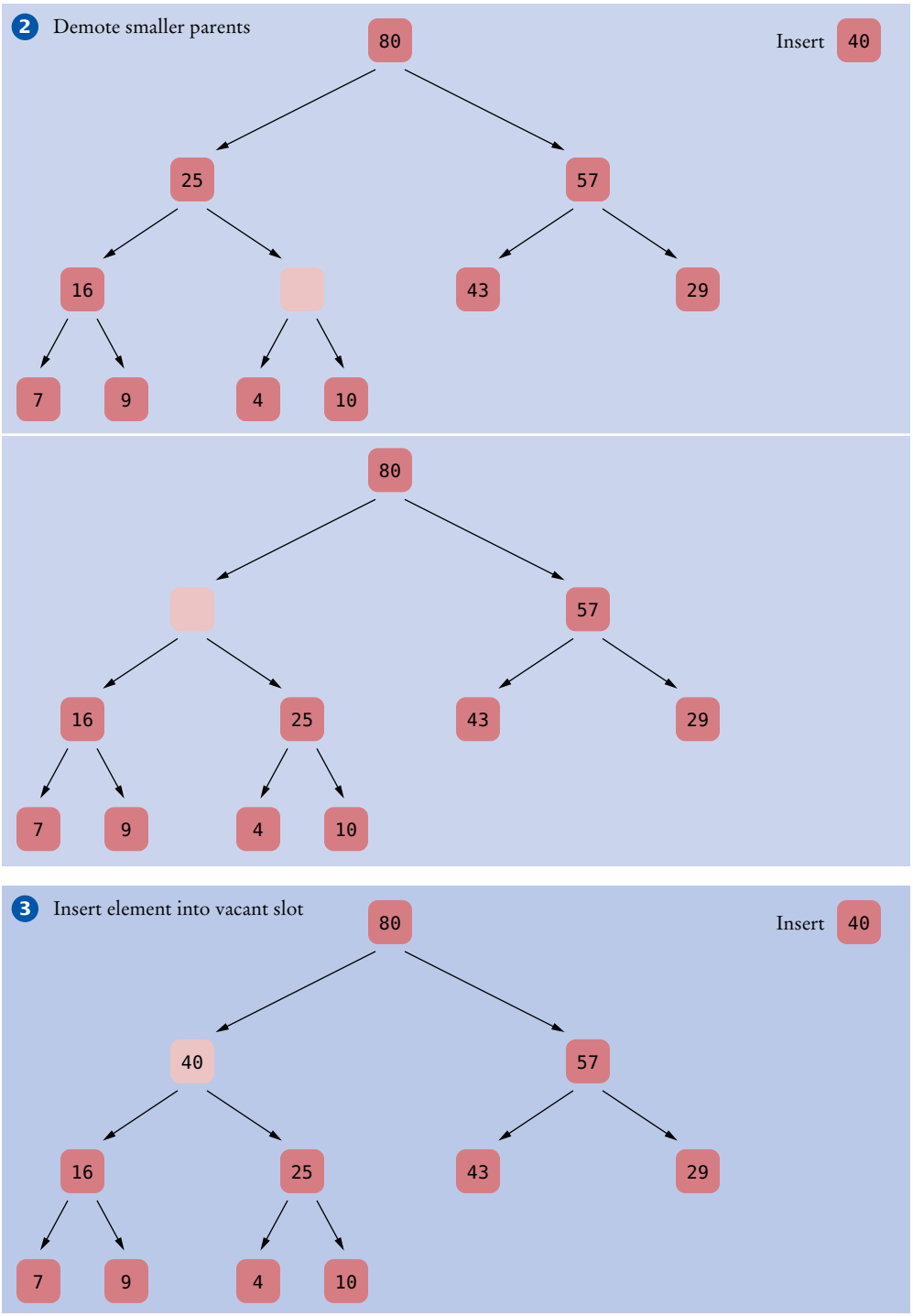


Figure 3 (continued) Inserting an Element into a Heap

We will not consider an algorithm for removing an arbitrary node from a heap. The only node that we will remove is the root node, which contains the maximum of all of the values in the heap.

Figure 4 shows the algorithm in action.

1. Extract the root node value.
2. Move the value of the last node of the heap into the root node, and remove the last node. Now the heap property may be violated for the root node, because one or both of its children may be larger.
3. Promote the larger child of the root node. (See Figure 4 (continued).) Now the root node again fulfills the heap property. Repeat this process with the demoted child. That is, promote the larger of its children. Continue until the demoted child has no larger children. The heap property is now fulfilled again. This process is called "fixing the heap".

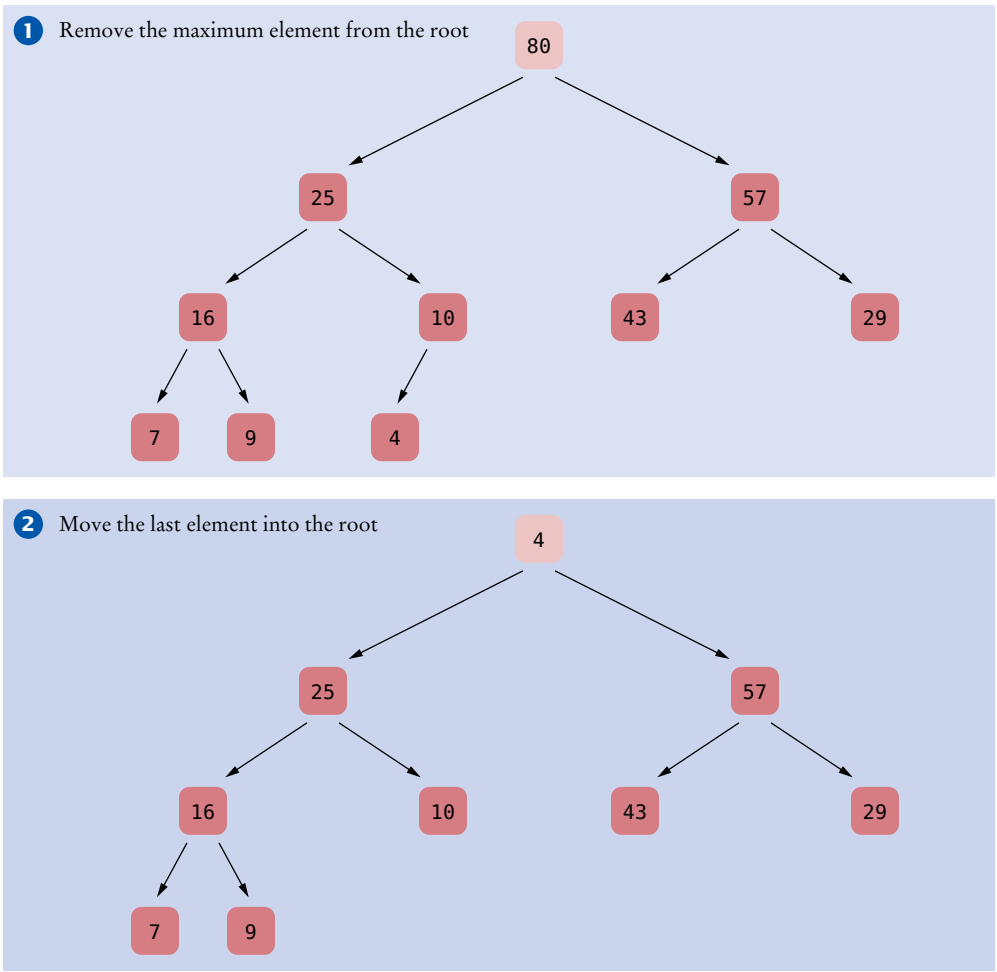


Figure 4 Removing the Maximum Element from a Heap

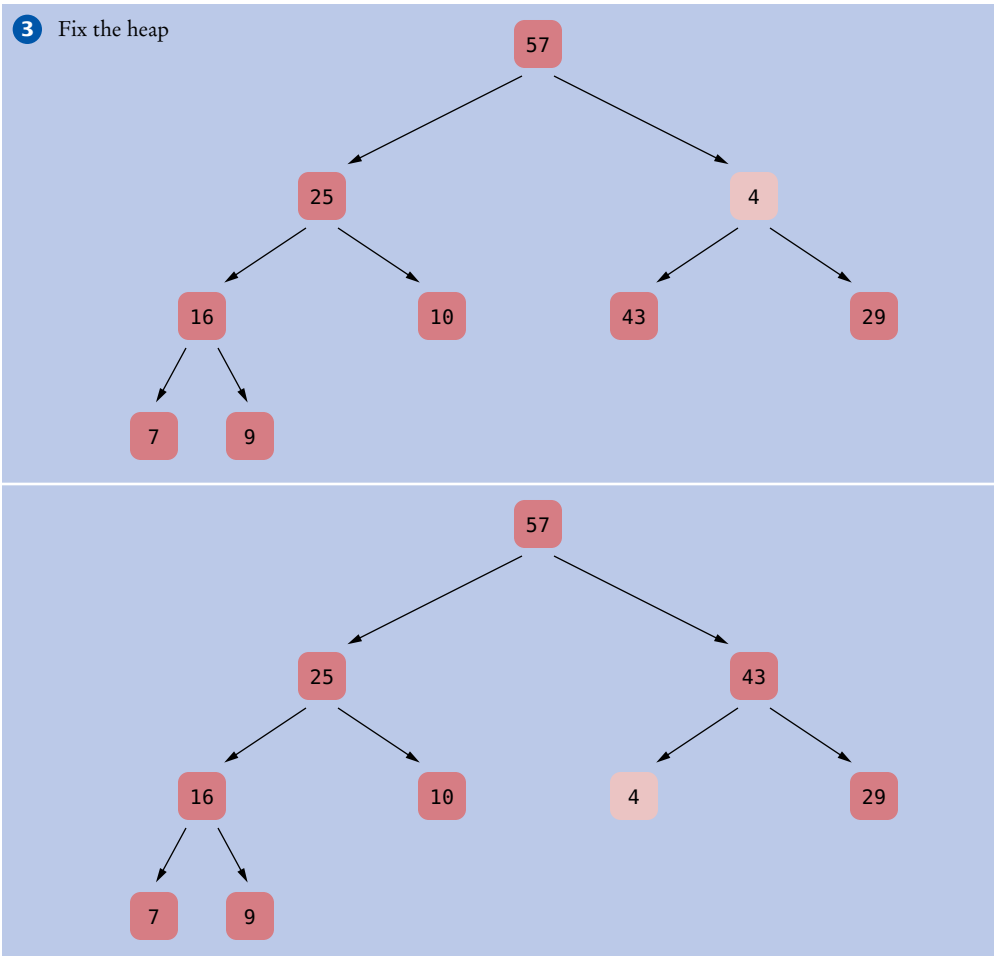


Figure 4 (continued) Removing the Maximum Element from a Heap

Inserting and removing heap elements is very efficient. The reason lies in the balanced shape of a heap. The insertion and removal operations visit at most h nodes, where h is the height of the tree. A heap of height h contains at least 2^{h-1} elements, but less than 2^h elements. In other words, if n is the number of elements, then

$$2^{h-1} \leq n < 2^h$$

or

$$h - 1 \leq \log_2(n) < h$$

Inserting or removing a heap element is an $O(\log(n))$ operation.

This argument shows that the insertion and removal operations in a heap with n elements take $O(\log(n))$ steps.

Contrast this finding with the situation of binary search trees. When a binary search tree is unbalanced, it can degenerate into a linked list, so that in the worst case insertion and removal are $O(n)$ operations.

The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

Heaps have another major advantage. Because of the regular layout of the heap nodes, it is easy to store the node values in an array. First store the first layer, then the second, and so on (see Figure 5). For convenience, we leave the 0 element of the array empty. Then the child nodes of the node with index i have index $2 \cdot i$ and $2 \cdot i + 1$, and the parent node of the node with index i has index $i/2$. For example, as you can see in Figure 5, the children of node 4 are nodes 8 and 9, and the parent is node 2.

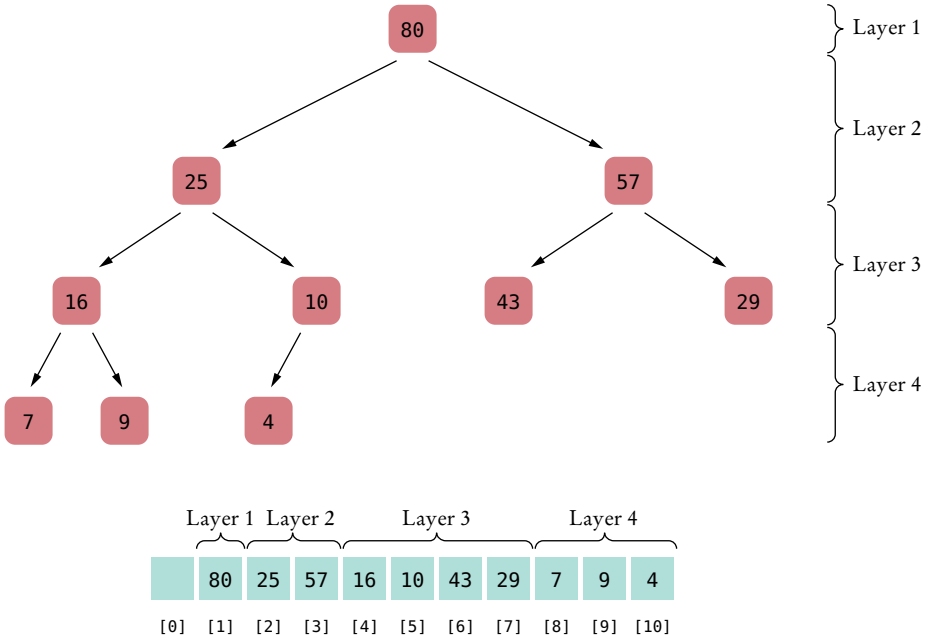


Figure 5 Storing a Heap in an Array

Storing the heap values in an array may not be intuitive, but it is very efficient. There is no need to allocate individual nodes or to store the links to the child nodes. Instead, child and parent positions can be determined by very simple computations.

Here is an implementation of a template class `MaxHeap<T>` that you can use with any type that has a < operator defined.

sec02/maxheap.h

```
#ifndef MAXHEAP_H
#define MAXHEAP_H

#include <vector>

using namespace std;

/**
   This class implements a heap.
*/
template<typename T>
class MaxHeap
{
public:
```

```
15     /**
16        Constructs an empty heap.
17     */
18     MaxHeap();
19
20     /**
21        Adds a new element to this heap.
22        @param element the element to add
23     */
24     void push(T element);
25
26     /**
27        Gets the maximum element stored in this heap.
28        @return the maximum element
29     */
30     T top() const;
31
32     /**
33        Removes the maximum element from this heap.
34     */
35     void pop();
36
37     /**
38        Gets the size of this heap.
39        @return the size
40     */
41     int size() const;
42  private:
43     /**
44        Turns the tree back into a heap, provided only the root
45        node violates the heap condition.
46     */
47     void fix_heap();
48
49     /**
50        Returns the index of the left child.
51        @param index the index of a node in this heap
52        @return the index of the left child of the given node
53     */
54     int get_left_child_index(int index) const;
55
56     /**
57        Returns the index of the right child.
58        @param index the index of a node in this heap
59        @return the index of the right child of the given node
60     */
61     int get_right_child_index(int index) const;
62
63     /**
64        Returns the index of the parent.
65        @param index the index of a node in this heap
66        @return the index of the parent of the given node
67     */
68     int get_parent_index(int index) const;
69
70     /**
71        Returns the value of the left child.
72        @param index the index of a node in this heap
73        @return the value of the left child of the given node
74     */
```

```
   T get_left_child(int index) const;

   /**
      Returns the value of the right child.
      @param index the index of a node in this heap
      @return the value of the right child of the given node
   */
   T get_right_child(int index) const;

   /**
      Returns the value of the parent.
      @param index the index of a node in this heap
      @return the value of the parent of the given node
   */
   T get_parent(int index) const;

   vector<T> elements;
};

template<typename T>
MaxHeap<T>::MaxHeap()
{
   T dummy;
   elements.push_back(dummy);
}

template<typename T>
void MaxHeap<T>::push(T new_element)
{
   // Add a new leaf
   T dummy;
   elements.push_back(dummy);
   int index = elements.size() - 1;

   // Demote parents that are smaller than the new element
   while (index > 1 && get_parent(index) < new_element)
   {
      elements[index] = get_parent(index);
      index = get_parent_index(index);
   }

   // Store the new element into the vacant slot
   elements[index] = new_element;
}

template<typename T>
T MaxHeap<T>::top() const
{
   return elements[1];
}

template<typename T>
void MaxHeap<T>::pop()
{
   // Remove last element
   int last_index = elements.size() - 1;
   T last = elements[last_index];
   elements.pop_back();
   if (last_index > 1)
   {
```

```
135       elements[1] = last;
136       fix_heap();
137    }
138 }
139
140 template<typename T>
141 void MaxHeap<T>::fix_heap()
142 {
143    T root = elements[1];
144
145    int last_index = elements.size() - 1;
146    // Promote children of removed root while they are smaller than root
147    int index = 1;
148    bool done = false;
149    while (!done)
150    {
151       int child_index = get_left_child_index(index);
152       if (child_index <= last_index)
153       {
154          // Get larger child
155
156          // Get left child first
157          T child = get_left_child(index);
158
159          // Use right child instead if it is larger
160          if (get_right_child_index(index) <= last_index
161             && child < get_right_child(index))
162          {
163             child_index = get_right_child_index(index);
164             child = get_right_child(index);
165          }
166
167          // Check if larger child is larger than root
168          if (root < child)
169          {
170             // Promote child
171             elements[index] = child;
172             index = child_index;
173          }
174          else
175          {
176             // Root is larger than both children
177             done = true;
178          }
179       }
180       else
181       {
182          // No children
183          done = true;
184       }
185    }
186
187    // Store root element in vacant slot
188    elements[index] = root;
189 }
190
191 template<typename T>
192 int MaxHeap<T>::size() const
193 {
194    return elements.size() - 1;
```

```
195 }
196
197 template<typename T>
198 int MaxHeap<T>::get_left_child_index(int index) const
199 {
200    return 2 * index;
201 }
202
203 template<typename T>
204 int MaxHeap<T>::get_right_child_index(int index) const
205 {
206    return 2 * index + 1;
207 }
208
209 template<typename T>
210 int MaxHeap<T>::get_parent_index(int index) const
211 {
212    return index / 2;
213 }
214
215 template<typename T>
216 T MaxHeap<T>::get_left_child(int index) const
217 {
218    return elements[2 * index];
219 }
220
221 template<typename T>
222 T MaxHeap<T>::get_right_child(int index) const
223 {
224    return elements[2 * index + 1];
225 }
226
227 template<typename T>
228 T MaxHeap<T>::get_parent(int index) const
229 {
230    return elements[index / 2];
231 }
232
233 #endif
```

sec02/heapdemo.cpp

```
1  #include <iostream>
2  #include "maxheap.h"
3  #include "task.h"
4
5  using namespace std;
6
7  int main()
8  {
9     MaxHeap<Task> tasks;
10    tasks.push(Task("Shampoo carpets", 2));
11    tasks.push(Task("Empty trash", 3));
12    tasks.push(Task("Water plants", 2));
13    tasks.push(Task("Remove pencil sharpener shavings", 1));
14    tasks.push(Task("Replace light bulb", 4));
15    tasks.push(Task("Fix overflowing sink", 9));
16    tasks.push(Task("Clean coffee maker", 1));
17    tasks.push(Task("Order cleaning supplies", 5));
18
```

```
19    while (tasks.size() > 0)
20    {
21       Task most_important = tasks.top();
22       tasks.pop();
23       cout << most_important.get_priority() << " - "
24          << most_important.get_description() << endl;
25    }
26    return 0;
27 }
```

EXAMPLE CODE See sec02 of your companion code for the Task class that completes this program.

Program Run

```
9 - Fix overflowing sink
5 - Order cleaning supplies
4 - Replace light bulb
3 - Empty trash
2 - Water plants
2 - Shampoo carpets
1 - Remove pencil sharpener shavings
1 - Clean coffee maker
```

17.3 The Heapsort Algorithm

The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.

Heapsort is an $O(n \log(n))$ algorithm.

Heaps are not only useful for implementing priority queues, they also give rise to an efficient sorting algorithm, heapsort. In its simplest form, the **heapsort algorithm** works as follows. First insert all elements to be sorted into the heap, then keep extracting the maximum.

This algorithm is an $O(n \log(n))$ algorithm: each insertion and removal is $O(\log(n))$, and these steps are repeated n times, once for each element in the sequence that is to be sorted.

The algorithm can be made a bit more efficient. Rather than inserting the elements one at a time, we will start with a sequence of values in an array. Of course, that array does not represent a heap. We will use the procedure of "fixing the heap" that you encountered in the preceding section as part of the element removal algorithm. "Fixing the heap" operates on a binary tree whose child trees are heaps but whose root value may not be larger than the descendants. The procedure turns the tree into a heap, by repeatedly promoting the largest child value, moving the root value to its proper location.

Of course, we cannot simply apply this procedure to the initial sequence of unsorted values—the child trees of the root are not likely to be heaps. But we can first fix small subtrees into heaps, then fix larger trees. Because trees of size 1 are automatically heaps, we can begin the fixing procedure with the subtrees whose roots are located in the next-to-last level of the tree.

The sorting algorithm uses a generalized `fix_heap` function that fixes a subtree:

```
void fix_heap(int a[], int root_index, int last_index)
```

The subtree is specified by the index of its root and of its last node.

The `fix_heap` function needs to be invoked on all subtrees whose roots are in the next-to-last level. Then the subtrees whose roots are in the next level above are fixed, and so on. Finally, the fixup is applied to the root node, and the tree is turned into a heap (see Figure 6).

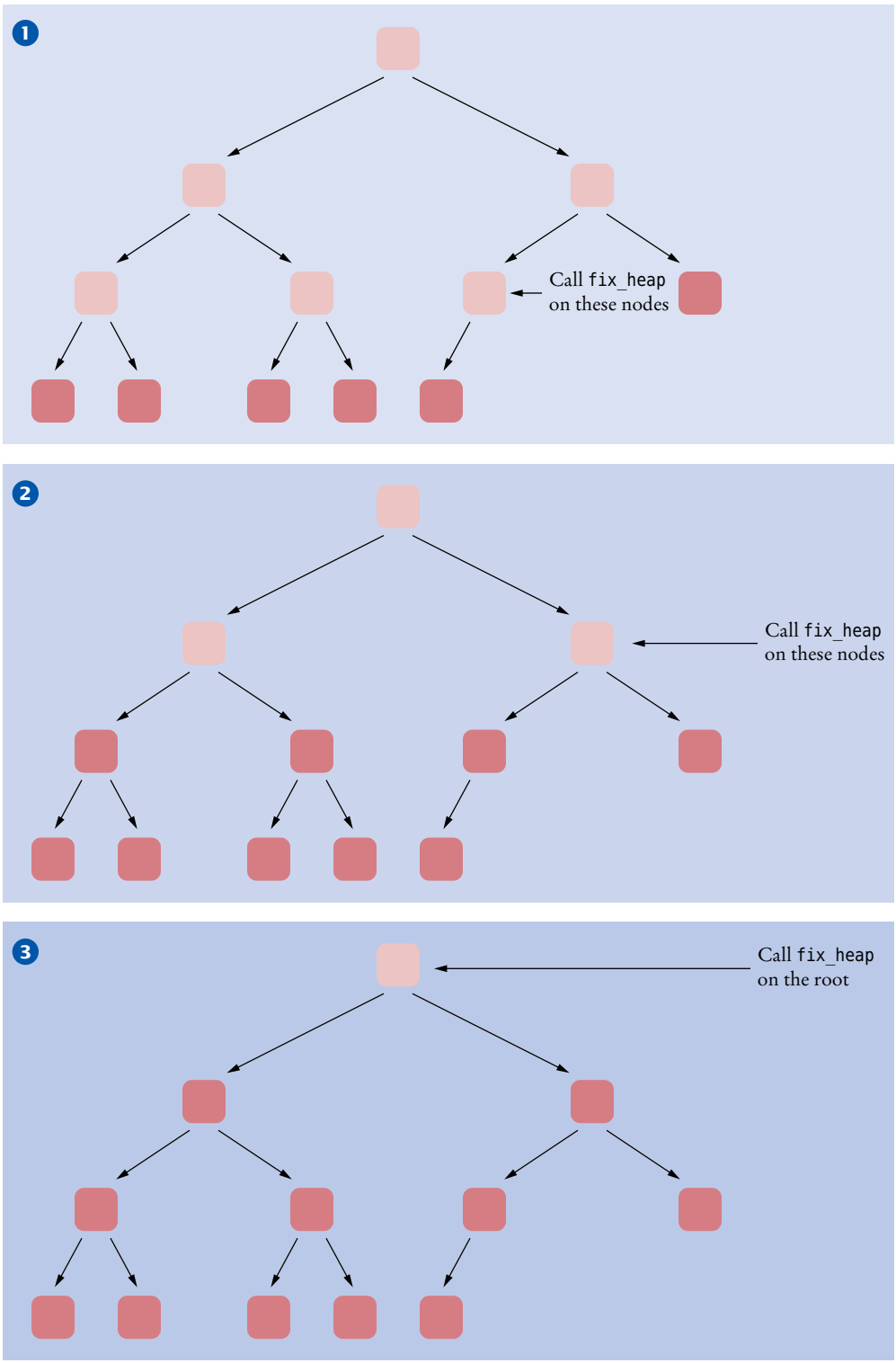


Figure 6 Turning a Tree into a Heap

That repetition can be programmed easily. Start with the *last* node on the next-to-lowest level and work toward the left. Then go to the next higher level. The node index values then simply run backward from the index of the last node to the index of the root.

```
int n = size - 1;
for (int i = (n - 1) / 2; i >= 0; i--)
{
   fix_heap(a, i, n);
}
```

It can be shown that this procedure turns an arbitrary array into a heap in $O(n)$ steps.

Note that the loop ends with index `0`. When working with a given array, we don't have the luxury of skipping the `0` entry. We consider the `0` entry the root and adjust the formulas for computing the child and parent index values. The child nodes of the node with index i (starting at 1) have index $2 \cdot i + 1$ and $2 \cdot i + 2$.

After the array has been turned into a heap, we repeatedly remove the root element. Recall from the preceding section that removing the root element is achieved by placing the last element of the tree in the root and calling the `fix_heap` function. Because we call the $O(\log(n))$ `fix_heap` function n times, this process requires $O(n \log(n))$ steps.

Rather than moving the root element into a separate array, we can *swap* the root element with the last element of the tree and then reduce the tree size. Thus, the removed root ends up in the last position of the array, which is no longer needed by the heap. In this way, we can use the same array both to hold the heap (which gets shorter with each step) and the sorted sequence (which gets longer with each step).

```
while (n > 0)
{
   // Swap root with the last element
   int temp = a[0];
   a[0] = a[n];
   a[n] = temp;
   n--;
   fix_heap(a, 0, n);
}
```

After the first step, the largest value is placed at the end of the array. After the next step, the next-largest value is swapped from the heap root to the second position from the end, and so on (see Figure 7).

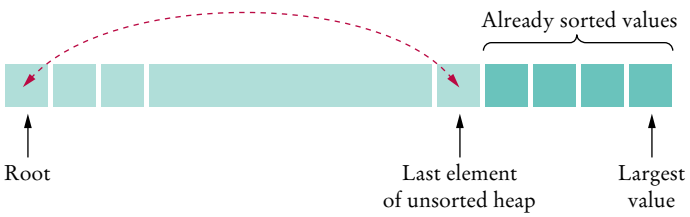


Figure 7 Using Heapsort to Sort an Array

The following class implements the heapsort algorithm:

sec03/heapsort.cpp

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

using namespace std;

/**
   Returns the index of the left child.
   @param index the index of a node in this heap
   @return the index of the left child of the given node
*/
int get_left_child_index(int index)
{
   return 2 * index + 1;
}

/**
   Returns the index of the right child.
   @param index the index of a node in this heap
   @return the index of the right child of the given node
*/
int get_right_child_index(int index)
{
   return 2 * index + 2;
}

/**
   Ensures the heap property for a subtree, provided its
   children already fulfill the heap property.
   @param a the array to sort
   @param root_index the index of the subtree to be fixed
   @param last_index the last valid index of the tree that
   contains the subtree to be fixed
*/
void fix_heap(int a[], int root_index, int last_index)
{
   // Remove root
   int root_value = a[root_index];

   // Promote children while they are larger than the root

   int index = root_index;
   bool done = false;
   while (!done)
   {
      int child_index = get_left_child_index(index);
      if (child_index <= last_index)
      {
         // Use right child instead if it is larger
         int right_child_index = get_right_child_index(index);
         if (right_child_index <= last_index
            && a[child_index] < a[right_child_index])
         {
            child_index = right_child_index;
         }

```

```
57          if (root_value < a[child_index])
58          {
59             // Promote child
60             a[index] = a[child_index];
61             index = child_index;
62          }
63          else
64          {
65             // Root value is larger than both children
66             done = true;
67          }
68       }
69       else
70       {
71          // No children
72          done = true;
73       }
74    }
75
76    // Store root value in vacant slot
77    a[index] = root_value;
78 }
79
80 /**
81    Sorts an array, using heap sort.
82    @param a the array to sort
83    @param size the number of elements in a
84 */
85 void heap_sort(int a[], int size)
86 {
87    int n = size - 1;
88    for (int i = (n - 1) / 2; i >= 0; i--)
89    {
90       fix_heap(a, i, n);
91    }
92    while (n > 0)
93    {
94       // Swap root with the last element
95       int temp = a[0];
96       a[0] = a[n];
97       a[n] = temp;
98       n--;
99       fix_heap(a, 0, n);
100   }
101 }
102
103 /**
104    Prints all elements in an array.
105    @param a the array to print
106    @param size the number of elements in a
107 */
108 void print(int a[], int size)
109 {
110    for (int i = 0; i < size; i++)
111    {
112       cout << a[i] << " ";
113    }
114    cout << endl;
115 }
116
```

```
117 int main()
118 {
119    srand(time(0));
120    const int SIZE = 20;
121    int values[SIZE];
122    for (int i = 0; i < SIZE; i++)
123    {
124       values[i] = rand() % 100;
125    }
126    print(values, SIZE);
127    heap_sort(values, SIZE);
128    print(values, SIZE);
129    return 0;
130 }
```

CHAPTER SUMMARY

Describe the behavior of the priority queue data type.

- When removing an element from a priority queue, the element with the highest priority is retrieved.

Describe the heap data structure and the efficiency of its operations.

- A heap is an almost complete tree in which the values of all nodes are at least as large as those of their descendants.
- Inserting or removing a heap element is an $O(\log(n))$ operation.
- The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

Describe the heapsort algorithm and its run-time performance.

- The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.
- Heapsort is an $O(n \log(n))$ algorithm.

REVIEW EXERCISES

R17.1 Could a priority queue be implemented efficiently as a binary search tree? Give a detailed argument for your answer.

R17.2 Will preorder, inorder, or postorder traversal print a heap in sorted order? Why or why not?

R17.3 Prove that a heap of height h contains at least 2^{h-1} elements but less than 2^h elements.

R17.4 Suppose the heap nodes are stored in an array, starting with index 1. Prove that the child nodes of the heap node with index i have index $2 \cdot i$ and $2 \cdot i + 1$, and the parent node of the heap node with index i has index $i/2$.

R17.5 Show the heaps that are obtained by inserting the following numbers, one at a time:

```
50 25 10 30 60 90 75 40 80
```

Then show the heaps that result from repeatedly removing the largest element until none are left.

R17.6 Simulate the heapsort algorithm manually to sort the array

```
11 27 8 14 45 6 24 81 29 33
```

Show all steps.

R17.7 What are the steps that heapsort takes if the input is already in increasing or decreasing order? What is the big-Oh efficiency in these cases?

PRACTICE EXERCISES

E17.1 Change the `Task` class in Section 17.1 so that it has a due date instead of a priority, and change the sample program so that the tasks with the earliest due date are executed first. Represent the date as a string in year-month-day format such as `"2018-06-25"`.

E17.2 Write a program that reads a set of floating-point numbers and prints out the ten largest numbers. As you process the inputs, do not store more than eleven numbers. Insert the numbers into a priority queue. When it holds more than ten values, remove the largest value, as it cannot be a part of the answer.

E17.3 Reimplement the Huffman coding algorithm in Worked Example 16.1 using a priority queue.

E17.4 Modify the implementation of the `MaxHeap` class so that the parent and child index positions and elements are computed directly, without calling helper methods.

E17.5 Time the results of heapsort and merge sort. Which algorithm behaves better in practice?

E17.6 When you use the `MaxHeap` template in Section 17.2 with a class, the class must have a default constructor. Reimplement the template so that this is not necessary. Verify that you can form a `MaxHeap<Task>` after removing the default constructor for the `Task` class.

E17.7 Implement a heap sort function template that can sort arrays of any type with a `<` operator.

■■ **E17.8** In Worked Example 17.1, we had to specialize `std::less<Event*>`, which is not very intuitive. In this exercise, you will use an alternate approach. Provide a function

```
bool event_less(Event* a, Event* b)
```

and construct the priority queue by passing the name of this function to the constructor. You now need to specify additional template arguments, namely the types of the container and the comparator:

```
priority_queue<Event*, vector<Event*>,
   bool (*)(Event*, Event*)> event_queue(event_less);
```

PROGRAMMING PROJECTS

■■ **P17.1** Write a member function of the `BinaryTree` class in Section 16.2 that tests whether a given binary tree is a heap.

■■ **P17.2** In Worked Example 17.1, we had to specialize `std::less<Event*>`. Exercise E17.8 provides a workaround, but it is also complex. In this exercise, redefine the `Event` class so that it has a time stamp and a pointer to an `Action` object. Make the `Event::process` function call a `process` function of the `Action` class. Provide derived classes `ArrivalAction` and `DepartureAction`. Then use a `priority_queue<Event>` and define an `operator<` for `Event` objects.

■■ **P17.3** Implement the event simulation in Worked Example 17.1 using shared pointers for events and customers (see Special Topic 13.7).

■■ **P17.4** This problem illustrates the use of a discrete event simulation, as described in Worked Example 17.1. Imagine you are planning to open a small hot dog stand. You need to determine how many stools your stand should have. Too few stools and you will lose customers; too many and your stand will look empty most of the time.

There are two types of events in this simulation. An arrival event signals the arrival of a customer. If seated, the customer stays a randomly generated amount of time then leaves. A departure event frees the seat the customer was occupying.

Simulate a hot dog stand with three seats. To initialize the simulation a random number of arrival events are scheduled for the period of one hour. The output shows what time each customer arrives and whether they stay or leave. The following is the beginning of a typical run:

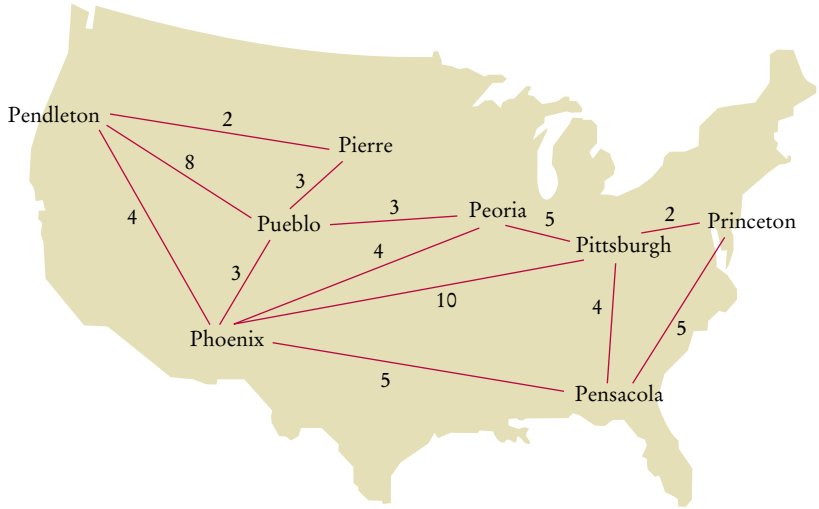
```
time 0.13 Customer is seated
time 0.14 Customer is seated
time 0.24 Customer is seated
time 0.29 Customer finishes eating, leaves
time 0.31 Customer is seated
time 0.38 Customer finishes eating, leaves
time 0.41 Customer is seated
time 0.42 Customer is unable to find a seat, leaves
time 0.48 Customer is unable to find a seat, leaves
time 0.63 Customer is unable to find a seat, leaves
time 0.64 Customer is unable to find a seat, leaves
time 0.68 Customer finishes eating, leaves
time 0.71 Customer is seated
```

■■ **P17.5** In most banks, customers enter a single waiting queue, but most supermarkets have a separate queue for each cashier. Modify Worked Example 17.1 so that each teller has

a separate queue. An arriving customer picks the shortest queue. What is the effect on the average time spent in the bank?

■■■ **P17.6** Modify the implementation of the `MaxHeap` class in Section 17.2 so that the `0` element of the array is not wasted.

■■■ **Business P17.7** Consider the problem of finding the least expensive routes to all cities in a network from a given starting point. For example, in the network shown on the map below, the least expensive route from Pendleton to Peoria has cost 8 (going through Pierre and Pueblo).



The following helper class expresses the distance to another city:

```
class DistanceTo
{
public:
   DistanceTo(string city, int dist);
   bool operator<(const DistanceTo& other) const;
   . . .
};
```

All direct connections between cities are stored in an `unordered_map<string, unordered_set<DistanceTo>>`. The algorithm now proceeds as follows:

```
Let from be the starting point.
Add DistanceTo(from, 0) to a priority queue.
Construct a map shortest_known_distance from city names to distances.
While the priority queue is not empty
   Get its smallest element.
   If its target is not a key in shortest_known_distance
      Let d be the distance to that target.
      Put (target, d) into shortest_known_distance.
      For all cities c that have a direct connection from target
         Add DistanceTo(c, d + distance from target to c) to the priority queue.
```

Here, you need a priority queue that returns the smallest, not the largest value. You can achieve that by defining `operator<` so that it returns the reverse of the normal comparison.

When the algorithm has finished, *shortest_known_distance* contains the shortest distance from the starting point to all reachable targets.

Your task is to write a program that implements this algorithm. Your program should read in lines of the form $city_1$ $city_2$ *distance*. The starting point is the first city in the first line. Print the shortest distances to all other cities.

WORKED EXAMPLE 17.1

Simulating a Queue of Waiting Customers

Jack Hollingsworth/Photodisc/Getty Images.

Problem Statement A good application of object-oriented programming is simulation. In fact, the first object-oriented language, Simula, was designed with this application in mind. One can simulate the activities of air molecules around an aircraft wing, of customers in a supermarket, or of vehicles on a road system. The goal of a simulation is to observe how changes in the design affect the behavior of a system. Modifying the shape of a wing, the location and staffing of cash registers, or the synchronization of traffic lights has an effect on turbulence in the air stream, customer satisfaction, or traffic throughput. Modeling these systems in the computer is far cheaper than running actual experiments.

Kinds of Simulation

Simulations fall into two broad categories. A *continuous simulation* constantly updates all objects in a system. A simulated clock advances in seconds or some other suitable constant time interval. At every clock tick, each object is moved or updated in some way. Consider the simulation of traffic along a road. Each car has some position, velocity, and acceleration. Its position needs to be updated with every clock tick. If the car gets too close to an obstacle, it must decelerate. The new position may be displayed on the screen.

In contrast, in a *discrete event simulation*, time advances in chunks. All interesting events are kept in a priority queue, sorted by the time in which they are to happen. As soon as one event has completed, the clock jumps to the time of the next event to be executed.

To see the contrast between these two simulation styles, consider the updating of a traffic light. Suppose the traffic light just turned red, and it will turn green again in 30 seconds. In a continuous model, the traffic light is visited every second, and a counter variable is decremented. Once the counter reaches `0`, the color changes. In a discrete model, the traffic light schedules an event to be notified 30 seconds from now. For 29 seconds, the traffic light is not bothered at all, and then it receives a message to change its state. Discrete event simulation avoids "busy waiting".

In this Worked Example, you will see how to use queues and priority queues in a discrete event simulation of customers at a bank. The simulation makes use of two base classes, `Event` and `Simulation`, that are useful for any discrete event simulation. We use inheritance to extend these classes to make classes that simulate the bank.

Events

A discrete event simulation generates, stores, and processes events. Each event has a time stamp indicating when it is to be executed. Each event has some action associated with it that must be carried out at that time. Beyond these properties, the scheduler has no concept of what an event represents. Of course, actual events must carry with them some information. For example, the event notifying a traffic light of a state change must know which traffic light to notify.

To do so, we will have all events extend a common base class, `Event`. An `Event` object has a data member to indicate at which time it should be processed. When that time has arrived, the event's `process` member function is called. This function may move objects around, update information, and schedule additional events.

```
class Event
{
public:
   Event(double event_time);
   virtual void process(Simulation& sim) const;
   double get_time() const;
   virtual ~Event() {}
private:
   double time;
};

Event::Event(double event_time)
{
   time = event_time;
}

void Event::process(Simulation& sim) const {}

double Event::get_time() const { return time; }
```

Comparing Events

We store the events in a `priority_queue<Event*>`. This poses a technical problem. The priority queue compares its elements with the < operator. However, for pointers, the < operator is defined to compare memory addresses, and you cannot change that definition by defining an `operator<(Event*, Event*)`. Fortunately, the `priority_queue` template does not call the < operator directly, but it uses a `less<T>` template that you can redefine. The process is similar to redefining the `hash<T>` template in Special Topic 15.2.

You define how a `less<Event*>` should compare two `Event*` pointers. In our case, we want to extract the earlier events first, so we reverse the time comparison:

```
namespace std
{
   template<>
   class less<Event*>
   {
   public:
      bool operator()(Event* a, Event* b)
      {
         return a->get_time() > b->get_time();
      }
   };
}
```

The Simulation Class

In any discrete event simulation, events are kept in a priority queue. After initialization, the simulation enters an event loop in which events are retrieved from the priority queue in the order specified by their time stamp. The simulated time is advanced to the time stamp of the event, and the event is processed according to its `process` member function. To simulate a specific activity, such as customer activity in a bank, extend the `Simulation` base class and provide member functions for printing the current state after each event, and a summary after the completion of the simulation.

```
class Simulation
{
public:
   . . .
```

```
   /**
      Displays intermediate results after each event.
   */
   virtual void print() const;

   /**
      Displays summary results after the end of the simulation.
   */
   virtual void print_summary() const;

private:
   priority_queue<Event*> event_queue;
   double current_time;
};
```

Here is the event loop in the `Simulation` class:

```
void Simulation::run(double start_time, double end_time)
{
   current_time = start_time;

   while (event_queue.size() > 0 && current_time <= end_time)
   {
      Event* event = event_queue.top();
      event_queue.pop();
      current_time = event->get_time();
      event->process(*this);
      delete event;
      print();
   }
   print_summary();
}
```

In the `Simulation` class, we provide a utility function for generating reasonable random values for the time between two independent events. These random time differences can be modeled with an "exponential distribution", as follows: Let m be the mean time between arrivals. Let u be a random value that can, with equal probability, assume any floating-point value between 0 inclusive and 1 exclusive. Then inter-arrival times can be generated as

$$a = -m \log(1 - u)$$

where log is the natural logarithm. The utility function `expdist` computes these random values:

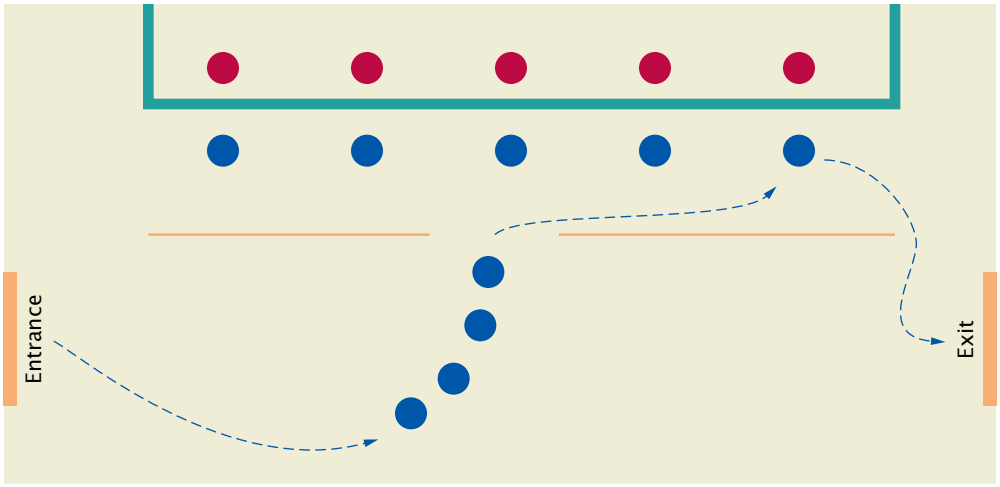
```
double Simulation::expdist(double mean) const
{
   return -mean * log(1 - rand() * 1.0 / RAND_MAX);
}
```

If a customer arrives at time t, the program can schedule the next customer arrival at t + `expdist(`m`)`.

Processing time is also exponentially distributed, with a different average. In this simulation we assume that, on average, one minute elapses between customer arrivals, and customer transactions require an average of five minutes.

The Bank

The following figure shows the layout of the bank. Customers enter the bank. If there is a queue, they join the queue; otherwise they move up to a teller. When a customer has completed a teller transaction, the time spent in the bank is logged, the customer is removed, and the next customer in the queue moves up to the teller.



The `BankSimulation` class keeps an array of tellers as well as a queue to hold waiting customers. The queue is not a priority queue but a regular FIFO (first-in, first-out) queue:

```
class BankSimulation : public Simulation
{
   . . .
private:
   . . .
   vector<Customer*> tellers;
   queue<Customer*> customer_queue;
   int total_customers;
   double total_time;

   const double INTERARRIVAL = 1;
      // average of 1 minute between customer arrivals
   const double PROCESSING = 5;
      // average of 5 minutes processing time per customer
};
```

It also keeps track of the total number of customers that have been serviced, and the total amount of time they spent in the bank (both in the waiting queue and in front of a teller.)

Teller `i` is busy if `tellers[i]` holds a pointer to a `Customer` object and available if it is `nullptr`.

When a customer is added to the bank, the program first checks whether a teller is available to handle the customer. If not, the customer is added to the waiting queue:

```
void BankSimulation::add(Customer* c)
{
   bool added_to_teller = false;
   for (int i = 0; !added_to_teller && i < tellers.size(); i++)
   {
      if (tellers[i] == nullptr)
      {
         add_to_teller(i, c);
         added_to_teller = true;
      }
   }
   if (!added_to_teller) { customer_queue.push(c); }

   add_event(new Arrival(get_current_time() + expdist(INTERARRIVAL)));
}
```

In addition, the simulation must ensure that customers keep coming. We know the next customer will arrive in about one minute, but it may be a bit earlier or, occasionally, a lot later. To obtain a random time, we call `expdist(INTERARRIVAL)`. Of course, we cannot wait around for that to happen, because other events will be going on in the meantime. Therefore when a customer is added, another arrival event is scheduled to occur when this random time has elapsed.

Similarly, when a customer steps up to a teller, the average transaction will be five minutes. We need to schedule a departure event that removes the customer from the bank. This happens in the `add_to_teller` function:

```
void BankSimulation::add_to_teller(int i, Customer* c)
{
   tellers[i] = c;
   add_event(new Departure(get_current_time() + expdist(PROCESSING), i));
}
```

When the departure event is processed, it will notify the bank to remove the customer. The bank simulation removes the customer and keeps track of the total amount of time the customer spent in the waiting queue and with the teller. This makes the teller available to service the next customer from the waiting queue. If there is a queue, we add the first customer to this teller:

```
void BankSimulation::remove(int i)
{
   Customer* c = tellers[i];
   tellers[i] = nullptr;

   // Update statistics
   total_customers++;
   total_time = total_time + get_current_time() - c->get_arrival_time();
   delete c;

   if (customer_queue.size() > 0)
   {
      add_to_teller(i, customer_queue.front());
      customer_queue.pop();
   }
}
```

Event Classes

The classes `Arrival` and `Departure` are derived classes of `Event`.

When a new customer is to arrive at the bank, an arrival event is processed. The processing action of that event has the responsibility of making a customer and adding it to the bank.

```
class Arrival : public Event
{
public:
   Arrival(double time);
   virtual void process(Simulation& sim) const;
};

Arrival::Arrival(double time) : Event(time) {}

void Arrival::process(Simulation& sim) const
{
   double now = sim.get_current_time();
   BankSimulation& bank = dynamic_cast<BankSimulation&>(sim);
   Customer* c = new Customer(now);
   bank.add(c);
}
```

Departures remember not only the departure time but also the teller from whom a customer is to depart. To process a departure event, we remove the customer from the teller.

```
class Departure : public Event
{
public:
   Departure(double time, int teller);
   virtual void process(Simulation& sim) const;
private:
   int teller;
};

Departure::Departure(double time, int teller) : Event(time)
{
   this->teller = teller;
}

void Departure::process(Simulation& sim) const
{
   BankSimulation& bank = dynamic_cast<BankSimulation&>(sim);
   bank.remove(teller);
}
```

Running the Simulation

To run the simulation, we first construct a `BankSimulation` object with five tellers. The most important task in setting up the simulation is to get the flow of events going. At the outset, the event queue is empty. We will schedule the arrival of a customer at the start time (9 a.m.). Because the processing of an arrival event schedules the arrival of each successor, the insertion of the arrival event for the first customer takes care of the generation of all arrivals. Once customers arrive at the bank, they are added to tellers, and departure events are generated.

Here is the `main` function:

```
int main()
{
   const double START_TIME = 9 * 60; // 9 a.m.
   const double END_TIME = 17 * 60; // 5 p.m.

   const int NTELLERS = 5;

   BankSimulation sim(NTELLERS);
   sim.add_event(new Arrival(START_TIME));
   sim.run(START_TIME, END_TIME);
}
```

Here is a typical program run. The bank starts out with empty tellers, and customers start dropping in:

```
.....<
C....<
CC...<
CCC..<
CCCC.<
C.CC.<
CCCC.<
CCCCC<
CCCCC<C
CCCCC<
C.CCC<
```

Due to the random fluctuations of customer arrival and processing, the queue can get quite long:

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCCC
CCCCC<CCCCCCCCCCCC
CCCCC<CCCCCCCCCC

CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCCC
CCCCC<CCCCCCCCCCCC
CCCCC<CCCCCCCCCCCCC
CCCCC<CCCCCCCCCCCC
```

At other times, the bank is empty again:

```
CCC.C<
CCC..<
CC...<
.C...<
.....<
C....<
```

This particular run of the simulation ends up with the following statistics:

```
457 customers. Average time 15.28 minutes.
```

If you are the bank manager, this result is quite depressing. You hired enough tellers to take care of all customers. (Every hour, you need to serve, on average, 60 customers. Their transactions take an average of 5 minutes each; that is 300 teller-minutes, or 5 teller-hours. Hence, hiring five tellers should be just right.) Yet the average customer had to wait in line more than 10 minutes, twice as long as their transaction time. This is an average, so some customers had to wait even longer. If disgruntled customers hurt your business, you may have to hire more tellers and pay them for being idle some of the time.

Here's the complete bank simulation program:

worked_example_1/banksim.cpp

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <queue>
#include <string>

using namespace std;

class Simulation;

class Event
{
public:
   Event(double event_time);
   virtual void process(Simulation& sim) const;
   double get_time() const;
   virtual ~Event() {}
private:
   double time;
};

// Redefine < for Event* pointers
namespace std
{
```

```
25    template<>
26    class less<Event*>
27    {
28    public:
29       bool operator()(Event* a, Event* b)
30       {
31          return a->get_time() > b->get_time();
32       }
33    };
34 }
35
36 class Simulation
37 {
38 public:
39    /**
40       Constructs a discrete event simulation.
41    */
42    Simulation();
43
44    double get_current_time() const;
45
46    /**
47       Compute exponentially distributed random numbers.
48       @param mean the mean of the number sequence
49       @return a random number
50    */
51    double expdist(double mean) const;
52
53    /**
54       Adds an event to the event queue.
55       @param evt the event to add
56    */
57    void add_event(Event* evt);
58
59    /**
60       Displays intermediate results after each event.
61    */
62    virtual void print() const;
63
64    /**
65       Displays summary results after the end of the simulation.
66    */
67    virtual void print_summary() const;
68
69    /**
70       Runs this simulation for a given time interval.
71       @param start_time the start time
72       @param end_time the end time
73    */
74    void run(double start_time, double end_time);
75    virtual ~Simulation() {}
76 private:
77    priority_queue<Event*> event_queue;
78    double current_time;
79 };
80
81 class Arrival : public Event
82 {
```

```
83  public:
84     /**
85        @param time the arrival time
86     */
87     Arrival(double time);
88     virtual void process(Simulation& sim) const;
89  };
90
91  class Departure : public Event
92  {
93  public:
94     /**
95        @param time the departure time
96        @param teller the teller holding the customer
97     */
98     Departure(double time, int teller);
99     virtual void process(Simulation& sim) const;
100 private:
101    int teller;
102 };
103
104 class Customer
105 {
106 public:
107    /**
108       Constructs a customer.
109       @param the time at which the customer entered the bank
110    */
111    Customer(double time);
112
113    /**
114       Gets the time at which the customer entered the bank.
115       @return the arrival time
116    */
117    double get_arrival_time() const;
118 private:
119    double arrival_time;
120 };
121
122 class BankSimulation : public Simulation
123 {
124 public:
125    /**
126       Constructs a bank simulation.
127       @param number_of_tellers the number of tellers
128    */
129    BankSimulation(int number_of_tellers);
130    /**
131       Adds a customer to the simulation.
132       @param c the customer
133    */
134    void add(Customer* c);
135    /**
136       Removes a customer from a teller.
137       @param i teller position
138    */
139    void remove(int i);
140    virtual void print() const;
141    virtual void print_summary() const;
```

```
142 private:
143    /**
144       Adds a customer to a teller and schedules the departure event.
145       @param i the teller number
146       @param c the customer
147    */
148    void add_to_teller(int i, Customer* c);
149    vector<Customer*> tellers;
150    queue<Customer*> customer_queue;
151    int total_customers;
152    double total_time;
153
154    const double INTERARRIVAL = 1;
155       // average of 1 minute between customer arrivals
156    const double PROCESSING = 5;
157       // average of 5 minutes processing time per customer
158 };
159
160 Simulation::Simulation() {}
161
162 double Simulation::get_current_time() const { return current_time; }
163
164 double Simulation::expdist(double mean) const
165 {
166    return -mean * log(1 - rand() * 1.0 / RAND_MAX);
167 }
168
169 void Simulation::add_event(Event* evt)
170 {
171    event_queue.push(evt);
172 }
173
174 void Simulation::print() const {}
175
176 void Simulation::print_summary() const {}
177
178 void Simulation::run(double start_time, double end_time)
179 {
180    current_time = start_time;
181
182    while (event_queue.size() > 0 && current_time <= end_time)
183    {
184       Event* event = event_queue.top();
185       event_queue.pop();
186       current_time = event->get_time();
187       event->process(*this);
188       delete event;
189       print();
190    }
191    print_summary();
192 }
193
194 Event::Event(double event_time)
195 {
196    time = event_time;
197 }
198
199 void Event::process(Simulation& sim) const {}
200
201 double Event::get_time() const { return time; }
```

```
202
203 Arrival::Arrival(double time) : Event(time) {}
204
205 void Arrival::process(Simulation& sim) const
206 {
207    double now = sim.get_current_time();
208    BankSimulation& bank = dynamic_cast<BankSimulation&>(sim);
209    Customer* c = new Customer(now);
210    bank.add(c);
211 }
212
213 Departure::Departure(double time, int teller) : Event(time)
214 {
215    this->teller = teller;
216 }
217
218 void Departure::process(Simulation& sim) const
219 {
220    BankSimulation& bank = dynamic_cast<BankSimulation&>(sim);
221    bank.remove(teller);
222 }
223
224 Customer::Customer(double time) { arrival_time = time; }
225
226 double Customer::get_arrival_time() const { return arrival_time; }
227
228 BankSimulation::BankSimulation(int number_of_tellers)
229 {
230    for (int i = 0; i < number_of_tellers; i++)
231    {
232       tellers.push_back(nullptr);
233    }
234    total_customers = 0;
235    total_time = 0;
236 }
237
238 void BankSimulation::add(Customer* c)
239 {
240    bool added_to_teller = false;
241    for (int i = 0; !added_to_teller && i < tellers.size(); i++)
242    {
243       if (tellers[i] == nullptr)
244       {
245          add_to_teller(i, c);
246          added_to_teller = true;
247       }
248    }
249    if (!added_to_teller) { customer_queue.push(c); }
250
251    add_event(new Arrival(get_current_time() + expdist(INTERARRIVAL)));
252 }
253
254 void BankSimulation::add_to_teller(int i, Customer* c)
255 {
256    tellers[i] = c;
257    add_event(new Departure(get_current_time() + expdist(PROCESSING), i));
258 }
259
260 void BankSimulation::remove(int i)
261 {
```

```
262      Customer* c = tellers[i];
263      tellers[i] = nullptr;
264
265      // Update statistics
266      total_customers++;
267      total_time = total_time + get_current_time() - c->get_arrival_time();
268      delete c;
269
270      if (customer_queue.size() > 0)
271      {
272         add_to_teller(i, customer_queue.front());
273         customer_queue.pop();
274      }
275   }
276
277   void BankSimulation::print() const
278   {
279      for (int i = 0; i < tellers.size(); i++)
280      {
281         if (tellers[i] == nullptr)
282         {
283            cout << ".";
284         }
285         else
286         {
287            cout << "C";
288         }
289      }
290      cout << "<";
291      int q = customer_queue.size();
292      for (int j = 1; j <= q; j++) { cout << "C"; }
293      cout << endl;
294   }
295
296   void BankSimulation::print_summary() const
297   {
298      double average_time = 0;
299      if (total_customers > 0)
300      {
301         average_time = total_time / total_customers;
302      }
303      cout << total_customers << " customers. Average time "
304         << average_time << " minutes." << endl;
305   }
306
307   int main()
308   {
309      const double START_TIME = 9 * 60; // 9 a.m.
310      const double END_TIME = 17 * 60; // 5 p.m.
311
312      const int NTELLERS = 5;
313
314      BankSimulation sim(NTELLERS);
315      sim.add_event(new Arrival(START_TIME));
316      sim.run(START_TIME, END_TIME);
317   }
```

APPENDIX A

RESERVED WORD SUMMARY

Reserved Word	Description	Reference Location
`auto`	A type that is automatically inferred	Special Topic 2.3 (C++ 11)
`bool`	The Boolean type	Section 3.7
`break`	Break out of a loop or switch	Special Topic 3.3, 4.2
`case`	A label in a `switch` statement	Special Topic 3.3
`char`	The character type	Section 7.3
`class`	Definition of a class	Section 9.2
`const`	Definition of a constant value, reference, member function, or pointer	Section 2.1.5, Special Topic 5.2, Special Topic 6.4, Programming Tip 9.2
`default`	The default case of a `switch` statement	Special Topic 3.3
`delete`	Return a memory block to the free store	Section 7.4
`do`	A loop that is executed at least once	Section 4.4
`double`	The double-precision, floating-point type	Section 2.1.2
`else`	The alternative clause in an `if` statement	Section 3.1
`false`	The false Boolean value	Section 3.7
`float`	The single-precision, floating-point type	Special Topic 2.1
`for`	A loop that is intended to initialize, test, and update a variable	Section 4.3
`friend`	Allows another class or function to access the private features of this class	Section 14.2
`if`	The conditional branch statement	Section 3.1
`int`	The integer type	Section 2.1.1
`long`	A modifier for the `int` and `double` types that indicates that the type may have more bytes	Special Topic 2.1
`namespace`	A name space for disambiguating names	Section 1.5
`new`	Allocate a memory block from the free store	Section 7.4
`nullptr`	A pointer that does not point to any value	Section 7.1.3 (C++ 11)

Reserved Word	Description	Reference Location
`operator`	An overloaded operator	Section 13.1
`override`	Declares that a member function is intended to override a virtual function	Common Error 10.5 (C++ 11)
`private`	Features of a class that can only be accessed by this class and its friends	Section 9.3
`public`	Features of a class that can be accessed by all functions	Section 9.3
`return`	Returns a value from a function	Section 5.4
`short`	A modifier for the `int` type that indicates that the type may have fewer bytes	Special Topic 2.1
`static_cast`	Convert from one type to another	Special Topic 2.4
`struct`	A construct for aggregating items of arbitrary types into a single value	Section 7.7
`switch`	A statement that selects among multiple branches, depending upon the value of an expression	Special Topic 3.3
`template`	Defines a parameterized type or function	Section 13.3
`this`	The pointer to the implicit parameter of a member function	Section 9.10.3
`true`	The true value of the Boolean type	Section 3.7
`typename`	A type parameter in a template, or a denotation that the following name is a type name	Section 13.3, Worked Example 14.2
`unsigned`	A modifier for the `int` and `char` types that indicates that values of the type cannot be negative	Special Topic 2.1
`using`	Importing a name space	Section 1.5
`virtual`	A member function with dynamic dispatch	Section 10.4
`void`	The empty type of a function or pointer	Section 5.5
`while`	A loop statement that is controlled by a condition	Section 4.1

The following reserved words are not covered in this book:

```
alignas          const_cast       inline           signed           try
alignof          continue         mutable          sizeof           typedef
asm              decltype         noexcept         static           typeid
catch            dynamic_cast     protected        static_assert    union
char16_t         explicit         register         static_cast      volatile
char32_t         enum             reinterpret_cast thread_local     wchar_t
constexpr        goto             short            throw
```

APPENDIX **B**

OPERATOR SUMMARY

The operators are listed in groups of decreasing precedence in the table below. The heavy horizontal lines in the table indicate a change in operator precedence. For example, `z = x - y;` means `z = (x - y);` because `=` has a lower precedence than `-`.

The prefix unary operators and the assignment operators associate right-to-left. All other operators associate left-to-right. For example, `x - y - z` means `(x - y) - z` because `-` associates left-to-right, but `x = y = z` means `x = (y = z)` because `=` associates right-to-left.

Operator	Description	Reference Location
`::`	Scope resolution	Section 9.5.1
`.`	Access member	Section 2.5.4, Section 7.7.1
`->`	Dereference and access member	Section 9.10.2
`[]`	Vector or array index	Section 6.1, Section 6.7.1
`()`	Function call	Section 5.1
`++`	Increment	Section 2.2.2
`--`	Decrement	Section 2.2.2
`!`	Boolean *not*	Section 3.7
`~`	Bitwise *not*	Appendix F
`+` (unary)	Positive	Section 2.2.1
`-` (unary)	Negative	Section 2.2.1
`*` (unary)	Pointer dereferencing	Section 7.1
`&` (unary)	Address of variable	Section 5.9, Section 7.1
`new`	Free store allocation	Section 7.4
`delete`	Free store recycling	Section 7.4
`sizeof`	Size of variable or type	Appendix F
(type)	Cast	not covered
`.*`	Access pointer to member	not covered

Operator	Description	Reference Location
`->*`	Dereference and access pointer to member	not covered
`*`	Multiplication	Section 2.2.1
`/`	Division or integer division	Section 2.2.1, Section 2.2.3
`%`	Integer remainder	Section 2.2.3
`+`	Addition	Section 2.2.1
`-`	Subtraction	Section 2.2.1
`<<`	Output	Section 1.5, Section 2.3.2, Appendix F
`>>`	Input	Section 2.3.1, Appendix F
`<`	Less than	Section 3.2
`<=`	Less than or equal	Section 3.2
`>`	Greater than	Section 3.2
`>=`	Greater than or equal	Section 3.2
`==`	Equal	Section 3.2
`!=`	Not equal	Section 3.2
`&`	Bitwise *and*	Appendix F
`^`	Bitwise *xor*	Appendix F
`|`	Bitwise *or*	Appendix F
`&&`	Boolean *and*	Section 3.7
`||`	Boolean *or*	Section 3.7
`? :`	Conditional operator	Special Topic 3.1
`=`	Assignment	Section 2.1.4
`+= -= *= /= %= &= |= ^= >>= <<=`	Combined operator and assignment	Special Topic 2.5
`,`	Sequencing of expressions	not covered

APPENDIX C

CHARACTER CODES

These escape sequences can occur in strings (for example, "\n") and characters (for example, '\'') However, the \U escape sequences should only be used inside strings that have an encoding prefix, such as

```
string message = u8"San Jos\U000000e9 \U0001f684";
```

A string is a sequence of bytes. In the UTF-8 encoding, bytes between 0 and 127 correspond to the ASCII character range (see Table 2). Other Unicode characters are encoded into multiple bytes, where the first byte is ≥ 192, and subsequent bytes are between 128 and 191. Table 3 shows the encoding.

Table 1 Escape Sequences

Escape Sequence	Description
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\b	Backspace
\f	Form feed
\a	Alert
\\	Backslash
\"	Double quote
\'	Single quote
\?	Question mark
$\texttt{\textbackslash U}h_4h_3h_2h_1h_0$	A Unicode character, which will be encoded into a sequence of bytes determined by a character encoding

Table 2 ASCII Code Table

Dec. Code	Hex Code	Char-acter	Dec. Code	Hex Code	Char-acter	Dec. Code	Hex Code	Char-acter	Dec. Code	Hex Code	Char-acter
0	00	\0	32	20	Space	64	40	@	96	60	‘
1	01		33	21	!	65	41	A	97	61	a
2	02		34	22	"	66	42	B	98	62	b
3	03		35	23	#	67	43	C	99	63	c
4	04		36	24	$	68	44	D	100	64	d
5	05		37	25	%	69	45	E	101	65	e
6	06		38	26	&	70	46	F	102	66	f
7	07	\a	39	27	'	71	47	G	103	67	g
8	08	\b	40	28	(	72	48	H	104	68	h
9	09	\t	41	29	)	73	49	I	105	69	i
10	0A	\n	42	2A	*	74	4A	J	106	6A	j
11	0B	\v	43	2B	+	75	4B	K	107	6B	k
12	0C	\f	44	2C	,	76	4C	L	108	6C	l
13	0D	\r	45	2D	-	77	4D	M	109	6D	m
14	0E		46	2E	.	78	4E	N	110	6E	n
15	0F		47	2F	/	79	4F	O	111	6F	o
16	10		48	30	0	80	50	P	112	70	p
17	11		49	31	1	81	51	Q	113	71	q
18	12		50	32	2	82	52	R	114	72	r
19	13		51	33	3	83	53	S	115	73	s
20	14		52	34	4	84	54	T	116	74	t
21	15		53	35	5	85	55	U	117	75	u
22	16		54	36	6	86	56	V	118	76	v
23	17		55	37	7	87	57	W	119	77	w
24	18		56	38	8	88	58	X	120	78	x
25	19		57	39	9	89	59	Y	121	79	y
26	1A		58	3A	:	90	5A	Z	122	7A	z
27	1B		59	3B	;	91	5B	[	123	7B	{
28	1C		60	3C	<	92	5C	\	124	7C	|
29	1D		61	3D	=	93	5D	]	125	7D	}
30	1E		62	3E	>	94	5E	^	126	7E	~
31	1F		63	3F	?	95	5F	_	127	7F	

Table 3 UTF-8 Encoding

UTF-8	Unicode
0xxxxxxx	00000 00000000 0xxxxxxx
110xxxxx 10xxxxxx	00000 00000xxx xxxxxxxx
1110xxxx 10xxxxxx 10xxxxxx	00000 xxxxxxxx xxxxxxxx
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	xxxxx xxxxxxxx xxxxxxxx

APPENDIX D

C++ LIBRARY SUMMARY

Standard Code Libraries

<algorithm>

- `bool binary_search(I begin, I end, T x)`
  Function: Checks whether the value `x` is contained in the sorted range `[begin, end)`.
- `T max(T x, T y)`
  Function: The maximum of `x` and `y`.
- `T min(T x, T y)`
  Function: The minimum of `x` and `y`.
- `void sort(I begin, I end)`
  Function: Sorts the elements in the range `[begin, end)`.

<cctype>

- `bool isalnum(char c)`
  Function: Test whether `c` is a letter or a number.
- `bool isalpha(char c)`
  Function: Tests whether `c` is a letter.
- `bool isdigit(char c)`
  Function: Tests whether `c` is a digit.
- `bool islower(char c)`
  Function: Tests whether `c` is lowercase.
- `bool isspace(char c)`
  Function: Tests whether `c` is white space.
- `bool isupper(char c)`
  Function: Tests whether `c` is uppercase.
- `char tolower(char c)`
  Function: Returns the lowercase of `c`.
- `char toupper(char c)`
  Function: Returns the uppercase of `c`.

<cmath>

- `int abs(int x)`
- `double abs(double x)`
  Function: Absolute value, $|x|$
- `double cos(double x)`
  Function: Cosine, $\cos x$ (x in radians)
- `double log(double x)`
  Function: Natural log, $\log(x)$, $x > 0$
- `double log10(double x)`
  Function: Decimal log, $\log_{10}(x)$, $x > 0$
- `double pow(double x, double y)`
  Function: Power, x^y. If $x > 0$, y can be any value. If x is 0, y must be > 0. If $x < 0$, y must be an integer.
- `double sin(double x)`
  Function: Sine, $\sin x$ (x in radians)
- `double sqrt(double x)`
  Function: Square root, $\sqrt{x}$
- `double tan(double x)`
  Function: Tangent, $\tan x$ (x in radians)

<cstdlib>

- `int abs(int x)`
  Function: Absolute value, $|x|$
- `void exit(int n)`
  Function: Exits the program with status code n.
- `int rand()`
  Function: Random integer
- `void srand(int n)`
  Function: Sets the seed of the random number generator to n.

<ctime>

- `time_t time(time_t* p)`
  Function: Returns the number of seconds since January 1, 1970, 00:00:00 GMT.
  If `p` is not `nullptr`, the return value is also stored in the location to which `p` points.

<string>

- `istream& getline(istream& in, string& s)`
  Function: Gets the next input line from the input stream `in` and stores it in the string `s`.
- `int stoi(const string& s)` (C++ 11)
- `double stod(const string& s)` (C++ 11)
  Function: Converts the string to an integer or floating-point number.

Class string

- `const char* string::c_str() const`
  Member function: Returns a char array with the characters in this string.
- `int string::length() const`
  Member function: Returns the length of the string.
- `string string::substr(int i) const`
  Member function: Returns the substring from index i to the end of the string.
- `string string::substr(int i, int n) const`
  Member function: Returns the substring of length n starting at index i.

<iomanip>

- **boolalpha**
  Manipulator: Causes Boolean values to be displayed as true and false instead of the default 1 and 0.
- **defaultfloat**
  Manipulator: Selects the default floating-point format, which uses scientific format for very large or very small values, and fixed format for all other values.
- **fixed**
  Manipulator: Selects fixed floating-point format, with trailing zeroes.
- **left**, **right**
  Manipulator: Left- or right-justifies values if they are shorter than the field width.
- **scientific**
  Manipulator: Selects scientific floating-point format, such as 1.729000e+03.
- **setfill**(char c)
  Manipulator: Sets the fill character to the character c.
- **setprecision**(int n)
  Manipulator: Sets the precision of floating-point values to n digits after the decimal point in fixed and scientific formats.
- **setw**(int n)
  Manipulator: Sets the width of the next field.

<iostream>

Class istream

- `bool istream::fail() const`
  Member function: True if input has failed.
- `istream& istream::get(char& c)`
  Member function: Gets the next character and places it into c.
- `istream& istream::seekg(long p)`
  Member function: Moves the get position to position p.
- `long istream::tellg()`
  Member function: Returns the get position.
- `istream& istream::unget()`
  Member function: Puts the last character read back into the stream, to be read again in the next input operation; only one character can be put back at a time.

Class ostream

- `ostream& ostream::seekp(long p)`
  Member function: Moves the put position to position p.
- `long ostream::tellp()`
  Member function: Returns the put position.

<fstream>

Class ifstream

- `void ifstream::open(const string& n)` (C++ 11)
- `void ifstream::open(const char n[])`
  Member function: Opens a file with name n for reading.

Class ofstream

- `void ofstream::open(const string& n)` (C++ 11)
- `void ofstream::open(const char n[])`
  Member function: Opens a file with name n for writing.

Class fstream

- `void fstream::open(const string& n)` (C++ 11)
- `void fstream::open(const char n[])`
  Member function: Opens a file with name n for reading and writing.

Class fstreambase

- `void fstreambase::close()`
  Member function: Closes the file stream.

  Notes: fstreambase is the common base class of ifstream, ofstream, and fstream.
  To open a binary file n both for input and output, use f.open(n, ios::in | ios::out | ios::binary)

<strstream>

Class istringstream

- `istringstream::istringstream(string s)`
  Constructs a string stream that reads from the string s.

Class ostringstream

- `string ostringstream::str() const`
  Member function: Returns the string that was collected by the string stream.
  Notes: Call `s = string(out.str())` to get a string object that contains the characters collected by `ostringstream out`.

<list>

Class list<T>

- `list<T>::iterator list<T>::erase(list<T>::iterator p)`
  Member function: Erases the element to which `p` points. Returns an iterator that points to the next element.
- `list<T>::iterator list<T>::insert(list<T>::iterator p, const T& x)`
  Member function: Inserts `x` before `p`. Returns an iterator that points to the inserted value.
- `void list<T>::pop_back()`
  Member function: Removes (but does not return) the last element.
- `void list<T>::pop_front()`
  Member function: Removes (but does not return) the first element.
- `void list<T>::push_back(const T& x)`
  Member function: Inserts `x` after the last element.
- `void list<T>::push_front(const T& x)`
  Member function: Inserts `x` before the first element.
- `list<T>::iterator list<T>::begin()`
  Member function: Gets an iterator that points to the first element in the container.
- `list<T>::iterator list<T>::end()`
  Member function: Gets an iterator that points past the last element in the container.
- `int list<T>::size() const`
  Member function: The number of elements in the container.

<set>

Class set<T>

- `int set<T>::count(const T& x) const`
  Member function: Returns 1 if `x` occurs in the set; returns `0` otherwise.
- `int set<T>::erase(const T& x)`
  Member function: Removes `x` and returns 1 if it occurs in the set; returns `0` otherwise.
- `void set<T>::erase(set<T>::iterator p)`
  Member function: Erases the element at the given position.
- `pair< set<T>::iterator, bool > set<T>::insert(const T& x)`
  Member function: If `x` is not present in the list, inserts it and returns an iterator that points to the newly inserted element and the Boolean value `true`. If `x` is present, returns an iterator pointing to the existing set element and the Boolean value `false`.
- `set<T>::iterator set<T>::begin()`
  Member function: Gets an iterator that points to the first element in the container.
- `set<T>::iterator set<T>::end()`
  Member function: Gets an iterator that points past the last element in the container.
- `int set<T>::size() const`
  Member function: The number of elements in the container.

Class unordered_set<T>

Has the same interface as `set<T>` and works with any type `T` that has a hash function.

<map>

Class map<K, V>

- `const V& map<K, V>::at(const K& k) const`
  Gets the value associated with `k`, or the default for type `V` if the key is not present, without adding the key to the map. Use the `at` member function when you have a constant reference to a map.
- `int map<K, V>::count(const K& k) const`
  Member function: Counts the elements with key `k`.
- `int map<K, V>::erase(const K& k)`
  Member function: Removes all occurrences of elements with key `k`. Returns the number of removed elements.
- `void map<K, V>::erase(map<K, V>::iterator p)`
  Member function: Erases the element at the given position.
- `map<K, V>::iterator map<K, V>::find(const K& k)`
  Member function: Returns an iterator to an element with key `k`, or `end()` if no such element exists.
  Notes: The key type `K` must be totally ordered by a `<` comparison operator.
  A map iterator points to `pair<K, V>` entries.
- `V& map<K, V>::operator[](const K& k)`
  Member function: Accesses the value with key `k`.

- `int map<K, V>::size() const`
  Member function: The number of elements in the container.

Class `unordered_map<K, V>`

Has the same interface as `map<K, V>` and works with any type `K` that has a hash function.

<multiset>

Class `multiset<T>`

- `int multiset<T>::count(const T& x) const`
  Member function: Counts the elements equal to `x`.
- `int multiset<T>::erase(const T& x)`
  Member function: Removes all occurrences of `x`. Returns the number of removed elements.
- `void multiset<T>::erase(multiset<T>::iterator p)`
  Member function: Erases the element at the given position.
- `multiset<T>::iterator multiset<T>::insert(const T& x)`
  Member function: Inserts `x` into the container. Returns an iterator that points to the inserted value.
- `multiset<T>::iterator multiset<T>::find(const T& x)`
  Member function: Returns an iterator to an element equal to `x`, or `end()` if no such element exists.

  Notes: The type `T` must be totally ordered by a `<` comparison operator.

<queue>

Class `queue<T>`

- `T& queue<T>::back()`
  Member function: The value at the back of the queue.
- `T& queue<T>::front()`
  Member function: The value at the front of the queue.
- `void queue<T>::pop()`
  Member function: Removes (but does not return) the front value of the queue.
- `void queue<T>::push(const T& x)`
  Member function: Adds `x` to the back of the queue.
- `int queue<T>::size() const`
  Member function: The number of elements in the container.

Class `priority_queue<T>`

- `void priority_queue<T>::pop()`
  Member function: Removes (but does not return) the largest value in the container.
- `void priority_queue<T>::push(const T& x)`
  Member function: Adds `x` to the container.
- `int priority_queue<T>::size() const`
  Member function: The number of elements in the container.
- `T& priority_queue<T>::top()`
  Member function: The largest value in the container.

<stack>

Class `stack<T>`

- `void stack<T>::pop()`
  Member function: Removes (but does not return) the top value of the stack.
- `void stack<T>::push(const T& x)`
  Member function: Adds `x` to the top of the stack.
- `int stack<T>::size() const`
  Member function: The number of elements in the container.
- `T& stack<T>::top()`
  Member function: The value at the top of the stack.

<utility>

Class `pair`

- `pair<F, S>::pair(const F& f, const S& s)`
  Constructs a pair from a first and second value.
- `F pair<F, S>::first`
  The public member holding the first value of the pair.
- `S pair<F, S>::second`
  The public member holding the second value of the pair.

<vector>

Class `vector<T>`

- `vector<T>::vector()`
  Constructs an empty vector.
- `vector<T>::vector(int n)`
  Constructs a vector with `n` elements.
- `T& vector<T>::operator[](int n)`
  Member function: Accesses the element at index `n`.
- `void vector<T>::pop_back()`
  Member function: Removes (but does not return) the last element.
- `void vector<T>::push_back(const T& x)`
  Member function: Inserts `x` after the last element.
- `int vector<T>::size() const`
  Member function: Returns the number of elements in the container.

APPENDIX E

C++ LANGUAGE CODING GUIDELINES

Introduction

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses. It lays down rules that you must follow for your programming assignments.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate the consistency if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it *reduces gratuitous choice*. If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines a number of constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are of marginal utility and can be expressed just as well or even better with other language constructs.

If you have already programmed in C or C++, you may be initially uncomfortable about giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat long and dull. They also mention features that you may not yet have seen in class. Here are the most important highlights:

- Tabs are set every three spaces.
- Variable and function names are lowercase.
- Constant names are uppercase. Class names start with an uppercase letter.
- There are spaces after reserved words and between binary operators.
- Braces must line up.
- No magic numbers may be used.
- Every function must have a comment.
- At most 30 lines of code may be used per function.
- No `goto`, `continue`, or `break` is allowed.
- At most two global variables may be used per file.

A note to the instructor: Of course, many programmers and organizations have strong feelings about coding style. If this style guide is incompatible with your own preferences or with local custom, please feel free to modify it. For that purpose, this coding style guide is available in electronic form on the companion web site for this book.

Source Files

Each program is a collection of one or more files or modules. The executable program is obtained by compiling and linking these files. Organize the material in each file as follows:

- Header comments
- `#include` statements
- Constants
- Classes
- Functions

It is common to start each file with a comment block. Here is a typical format:

```
/**
   @file invoice.cpp
   @author Jenny Koo
   @date 2022-01-24
   @version 3.14
*/
```

You may also want to include a copyright notice, such as

```
/* Copyright 2022 Jenny Koo */
```

A valid copyright notice consists of

- the copyright symbol © or the word "Copyright" or the abbreviation "Copr."
- the year of first publication of the work
- the name of the owner of the copyright

(Note: To save space, this header comment has been omitted from the programs in this book as well as the programs on disk so that the actual line numbers match those that are printed in the book.)

Next, list all included header files.

```
#include <iostream>
#include "question.h"
```

Do not embed absolute path names, such as

```
#include "c:\me\my_homework\widgets.h"  // Don't !!!
```

After the header files, list constants that are needed throughout the program file.

```
const int GRID_SIZE = 20;
const double CLOCK_RADIUS = 5;
```

Then supply the definitions of all classes.

```
class Product
{
   . . .
};
```

Order the class definitions so that a class is defined before it is used in another class.

Finally, list all functions, including member functions of classes and nonmember functions. Order the nonmember functions so that a function is defined before it is called. As a consequence, the `main` function will be the last function in your file.

Functions

Supply a comment of the following form for every function.

```
/**
   Explanation.
   @param parameter variable1   explanation
   @param parameter variable2   explanation
   . . .
   @return explanation
*/
```

The introductory explanation is required for all functions except `main`. It should start with an uppercase letter and end with a period. Some documentation tools extract the first sentence of the explanation into a summary table. Thus, if you provide an explanation that consists of multiple sentences, formulate the explanation such that the first sentence is a concise explanation of the function's purpose.

Omit the `@param` comment if the function has no parameter variables. Omit the `@return` comment for `void` functions. Here is a typical example:

```
/**
   Converts calendar date into Julian day. This algorithm is from Press
   et al., Numerical Recipes in C, 2nd ed., Cambridge University Press, 1992.
   @param year the year of the date to be converted
   @param month the month of the date to be converted
   @param day the day of the date to be converted
   @return the Julian day number that begins at noon of the given
   calendar date
*/
long dat2jul(int year, int month, int day)
{
   . . .
}
```

Parameter variable names must be explicit, especially if they are integers or Boolean.

```
Employee remove(int d, double s); // Huh?
Employee remove(int department, double severance_pay); // OK
```

Of course, for very generic functions, short names may be very appropriate.

Do not write `void` functions that return exactly one answer through a reference. Instead, make the result into a return value.

```
void find(vector<Employee> c, bool& found); // Don't!
bool find(vector<Employee> c); // OK
```

Of course, if the function computes more than one value, some or all results can be returned through reference parameters.

Functions must have at most 30 lines of code. (Comments, blank lines, and lines containing only braces are not included in this count.) Functions that consist of one long `if/else/else` statement sequence may be longer, provided each branch is 10 lines or less. This rule forces you to break up complex computations into separate functions.

Local Variables

Do not define all local variables at the beginning of a block. Define each variable just before it is used for the first time.

Every variable must be either explicitly initialized when defined or set in the immediately following statement (for example, through a >> instruction).

```
int pennies = 0;
```

or

```
int pennies;
cin >> pennies;
```

Move variables to the innermost block in which they are needed:

```
while (. . .)
{
   double xnew = (xold + a / xold) / 2;
   . . .
}
```

Do not define two variables in one statement:

```
int dimes = 0, nickels = 0; // Don't
```

When defining a pointer variable, place the * with the type, not the variable:

```
Link* p; // OK
```

not

```
Link *p; // Don't
```

Use `auto` only with types that are complex and not very informative, such as iterators. For example, prefer

```
unordered_map<string, int> scores = . . .;
auto pos = scores.find("Harry");
```

over

```
unordered_map<string, double>::iterator pos = scores.find("Harry");
```

Constants

In C++, do not use `#define` to define constants:

```
#define CLOCK_RADIUS 5 // Don't
```

Use `const` instead:

```
const double CLOCK_RADIUS = 5; // The radius of the clock face
```

You may not use magic numbers in your code. (A magic number is an integer constant embedded in code without a constant definition.) Any number except 0, 1, or 2 is considered magic:

```
if (p.get_x() < 10) // Don't
```

Use a `const` variable instead:

```
const double WINDOW_XMAX = 10;
if (p.get_x() < WINDOW_XMAX) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice.

Make a constant

```
const int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365's, 364's, 366's, 367's, and so on in your code.

Classes

Lay out the items of a class as follows:

```
class ClassName
{
public:
   constructors
   mutators
   accessors
private:
   data
};
```

All data members of classes must be private.

Control Flow

The for Statement

Use `for` loops only when a variable runs from somewhere to somewhere else with some constant increment/decrement.

```
for (i = 0; i < a.size(); i++)
{
   cout << a[i] << endl;
}
```

Do not use the `for` loop for weird constructs such as

```
for (xnew = a / 2; count < ITERATIONS; cout << xnew) // Don't
{
   xold = xnew;
   xnew = xold + a / xold;
   count++;
}
```

Make such a loop into a `while` loop, so the sequence of instructions is much clearer.

```
xnew = a / 2;
while (count < ITERATIONS) // OK
{
   xold = xnew;
   xnew = xold + a / xold;
   count++;
   cout << xnew;
}
```

Consider using the range-based `for` loop when traversing a container such as a `vector` or `list`:

```
vector<int> values = . . .;
for (int v : values)
{
   cout << v << " ";
}
```

Nonlinear Control Flow

Don't use the `switch` statement. Use `if/else` instead.

Do not use the `break`, `continue`, or `goto` statement. Use a `bool` variable to control the execution flow.

Lexical Issues

Naming Conventions

The following rules specify when to use upper- and lowercase letters in identifier names.

1. All variable and function names and all data members of classes are in lowercase, sometimes with an underscore in the middle. For example, `first_player`.
2. All constants are in uppercase, with an occasional underscore. For example, `CLOCK_RADIUS`.
3. All class names start with uppercase and are followed by lowercase letters, with an occasional uppercase letter in the middle. For example, `BankTeller`.

Names must be reasonably long and descriptive. Use `first_player` instead of `fp`. No drppng f vwls. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for five counter variables in your function. Surely each of these variables has a specific purpose and can be named to remind the reader of it (for example, `ccurrent`, `cnext`, `cprevious`, `cnew`, `cresult`).

Indentation and White Space

Use tab stops every three columns. Save your file so that it contains no tabs at all. That means you will need to change the tab stop setting in your editor! In the editor, make sure to select "3 spaces per tab stop" and "save all tabs as spaces". Every programming editor has these settings. If yours doesn't, don't use tabs at all but type the correct number of spaces to achieve indentation.

Use blank lines freely to separate logically distinct parts of a function.

Use a blank space around every binary operator:

```
x1 = (-b - sqrt(b * b - 4 * a * c)) / (2 * a); // Good
x1=(-b-sqrt(b*b-4*a*c))/(2*a); // Bad
```

Leave a blank space after (and not before) each comma, semicolon, and reserved word, but not after a function name.

```
if (x == 0) . . .
f(a, b[i]);
```

Every line must fit in 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] = ..............................................
   + ................;
```

Braces

Opening and closing braces must line up, either horizontally or vertically.

```
while (i < n) { cout << a[i] << endl; i++; }  // OK
while (i < n)
{
   cout << a[i] << endl;
   i++;
}  // OK
```

Some programmers don't line up vertical braces but place the `{` *behind* the `while`:

```
while (i < n) {   // Don't
   cout << a[i] << endl;
   i++;
}
```

This style saves a line, but it is difficult to match the braces.

Always use braces with `if`, `while`, `do`, and `for` statements, even if the body is only a single statement.

```
if (floor > 13)
{             // OK
   floor--;
}
if (floor > 13)
   floor--;   // Don't
```

Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```
class Employee
{
   . . .
private:
  string  name;
     int  age;
  double  hourly_wage;
};
```

This is undeniably neat, and we recommend it if your editor does it for you, but *don't* do it manually. The layout is not *stable* under change. A data type that is longer than the pre-allotted number of columns requires that you move *all* entries around.

Some programmers like to start every line of a multiline comment with **:

```
/* This is a comment
** that extends over
** three source lines
*/
```

Again, this is neat if your editor has a command to add and remove the asterisks, and if you know that all programmers who will maintain your code also have such an editor. Otherwise, it can be a powerful method of *discouraging* programmers from editing the comment. If you have to choose between pretty comments and comments that reflect the current facts of the program, facts win over beauty.

APPENDIX F

NUMBER SYSTEMS

Binary Numbers

Decimal notation represents numbers as powers of 10, for example

$$1729_{\text{decimal}} = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

There is no particular reason for the choice of 10, except that several historical number systems were derived from people's counting with their fingers. Other number systems, using a base of 12, 20, or 60, have been used by various cultures throughout human history. However, computers use a number system with base 2 because it is far easier to build electronic components that work with two values, which can be represented by a current being either off or on, than it would be to represent 10 different values of electrical signals. A number written in base 2 is also called a *binary* number.

For example,

$$1101_{\text{binary}} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

For digits after the "decimal" point, use negative powers of 2.

$$\begin{aligned} 1.101_{\text{binary}} &= 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 + \frac{1}{2} + \frac{1}{8} \\ &= 1 + 0.5 + 0.125 = 1.625 \end{aligned}$$

In general, to convert a binary number into its decimal equivalent, simply evaluate the powers of 2 corresponding to digits with value 1, and add them up. Table 1 shows the first powers of 2.

To convert a decimal integer into its binary equivalent, keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the *last* one. For example,

$$100 \div 2 = 50 \text{ remainder } 0$$

$$50 \div 2 = 25 \text{ remainder } 0$$

$$25 \div 2 = 12 \text{ remainder } 1$$

$$12 \div 2 = 6 \text{ remainder } 0$$

$$6 \div 2 = 3 \text{ remainder } 0$$

$$3 \div 2 = 1 \text{ remainder } 1$$

$$1 \div 2 = 0 \text{ remainder } 1$$

Therefore, $100_{\text{decimal}} = 1100100_{\text{binary}}$.

Table 1 Powers of Two

Power	Decimal Value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

Conversely, to convert a fractional number less than 1 to its binary format, keep multiplying by 2. If the result is greater than 1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the *first* one. For example,

$$0.35 \cdot 2 = 0.7$$

$$0.7 \cdot 2 = 1.4$$

$$0.4 \cdot 2 = 0.8$$

$$0.8 \cdot 2 = 1.6$$

$$0.6 \cdot 2 = 1.2$$

$$0.2 \cdot 2 = 0.4$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately.

Long, Short, Signed, and Unsigned Integers

There are two important properties that characterize integer values in computers. These are the number of bits used in the representation, and whether the integers are considered to be signed or unsigned.

Most computers you are likely to encounter use a 32-bit integer. However, the C++ language does not require this, and there have been machines that used 16-, 20-, 36-, or even 64-bit integers. There are times when it is useful to have integers of different sizes. The C++ language provides two modifiers that are used to declare such integers. A `short int` (or simply a `short`) is an integer that, on most implementations, has fewer bits than an `int`. (The phrase "on most implementations" is necessary because the language definition only requires that a short integer have no more bits than a standard integer.) On most platforms that use a 32-bit integer, a short is 16 bits. At the other extreme are long integers. As you might expect, a `long int` (or simply a `long`) contains no fewer bits than a standard integer. At the present time most personal computers still use a 32-bit long, but processors that provide 64-bit longs have started to appear and will likely be more common in the future. A character (or `char`) is sometimes used as a very short (8-bit) integer. The C++ programmer therefore has the following hierarchy of integer sizes:

Type	Typical Size
`char`	8-bit
`short`	16-bit
`int`	32-bit
`long`	32- or 64-bit

The `sizeof` operator can be used to tell how many bits your compiler assigns to each type. This operator takes a type as argument and returns the number of bytes each type requires. Multiplying the number of bytes by 8 will tell you the number of bits:

```
cout << "Number of bytes for char " << sizeof(char)
   << " number of bits " << 8 * sizeof(char) << "\n";
cout << "Number of bytes for short " << sizeof(short)
   << " number of bits " << 8 * sizeof(short) << "\n";
cout << "Number of bytes for int " << sizeof(int)
   << " number of bits "   << 8 * sizeof(int) << "\n";
cout << "Number of bytes for long " << sizeof(long)
   << " number of bits " << 8 * sizeof(long) << "\n";
```

If the only numbers you need are positive, then the preceding discussion would be everything you needed to know. However, in most applications it is more useful to allow both positive and negative values, and so a more complicated encoding is necessary. This characteristic of an integer is declared using the modifiers `signed` and `unsigned`.

An `unsigned` integer holds only positive values. An `unsigned short int` that is represented using 16 bits can maintain the values between 0 and 65,535 (that is, between zero

and 2^{16}–1). A 32-bit `unsigned int` can represent values between 0 and 4,294,967,295. If no modifier is provided, an integer is assumed to be signed.

Allowing both positive and negative values requires changing the representation of an integer value. The details of this representation are described in the next section. However, an important feature is that allowing both positive and negative numbers requires setting aside one bit (the so-called sign bit) to indicate whether the number is positive or negative. This reduces the largest value that can be represented. The following table shows the range of values that can be represented using signed and unsigned integers of 8, 16, 32, and 64 bits.

Integer Type	Range of Values
8-bit signed	–128 to 127
8-bit unsigned	0 to 255
16-bit signed	–32,768 to 32,767
16-bit unsigned	0 to 65,535
32-bit signed	–2,147,483,648 to 2,147,483,647
32-bit unsigned	0 to 4,294,967,295
64-bit signed	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
64-bit unsigned	0 to 18,446,744,073,709,551,615

Two's Complement Integers

To represent negative integers, there are two common representations, called "signed magnitude" and "two's complement". Signed magnitude notation is simple: use the leftmost bit for the sign (0 = positive, 1 = negative). For example, when using 8-bit numbers,

$$-13 = 10001101_{\text{signed magnitude}}$$

However, building circuitry for adding numbers gets a bit more complicated when one has to take a sign bit into account. The two's complement representation solves this problem. To form the two's complement of a number,

- Flip all bits.
- Then add 1.

For example, to compute –13 as an 8-bit value, first flip all bits of 00001101 to get 11110010. Then add 1:

$$-13 = 11110011_{\text{two's complement}}$$

Now no special circuitry is required for adding two numbers. Just follow the normal rule for addition, with a carry to the next position if the sum of the digits and the prior carry is 2 or 3. For example,

```
      1 1111 111
+13     0000 1101
-13     1111 0011
      ----------
      1 0000 0000
```

But only the last 8 bits count, so +13 and –13 add up to 0, as they should.

In particular, –1 has two's complement representation 1111 . . . 1111, with all bits set.

The leftmost bit of a two's complement number is 0 if the number is positive or zero, 1 if it is negative.

Two's complement notation with a given number of bits can represent one more negative number than positive numbers. For example, the 8-bit two's complement numbers range from –128 to +127.

This phenomenon is an occasional cause for a programming error. For example, consider the following code:

```
short b = ...;
if (b < 0) { b = -b; }
```

This code does not guarantee that `b` is nonnegative afterwards. If short values are 16 bits and `b` happens to be –32,768, then computing its negative again yields –32,768. (Try it out—take 100 . . . 00 (15 zeros), flip all bits, and add 1.)

IEEE Floating-Point Numbers

The Institute for Electrical and Electronics Engineering (IEEE) defines standards for floating-point representations in the IEEE-754 standard. Figure 1 shows how single-precision (`float`) and double-precision (`double`) values are decomposed into

- A sign bit
- An exponent
- A mantissa

Floating-point numbers use scientific notation, in which a number is represented as

$$b_0.b_1b_2b_3 \ldots \times 2^e$$

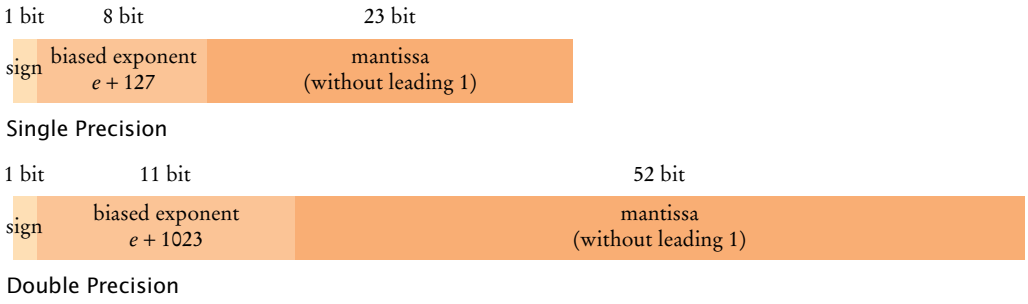


Figure 1 IEEE Floating-Point Representation

In this representation, e is the exponent, and the digits $b_0.b_1b_2b_3$ … form the mantissa. The normalized representation is the one where $b_0 \neq 0$. For example,

$$100_{\text{decimal}} = 1100100_{\text{binary}} = 1.100100_{\text{binary}} \times 2^6$$

Because in the binary number system the first bit of a normalized representation must be 1, it is not actually stored in the mantissa. Therefore, you always need to add it on to represent the actual value. For example, the mantissa 1.100100 is stored as 100100.

The exponent part of the IEEE representation uses neither signed magnitude nor two's complement representation. Instead, a *bias* is added to the actual exponent. The bias is 127 for single-precision numbers and 1023 for double-precision numbers. For example, the exponent $e = 6$ would be stored as 133 in a single-precision number.

Thus,

$$100_{\text{decimal}} = \texttt{0 10000101 10010000000000000000000}_{\text{single-precision IEEE}}$$

In addition, there are several special values. Among them are:

- *Zero:* biased exponent = 0, mantissa = 0.
- *Infinity:* biased exponent = 11. . .1, mantissa = ±0.
- *NaN* (not a number): biased exponent = 11 . . . 1, mantissa ≠ ±0.

Hexadecimal Numbers

Because binary numbers can be hard to read for humans, programmers often use the hexadecimal number system, with base 16. The digits are denoted as 0, 1, …, 9, A, B, C, D, E, F. (See Table 2.)

Four binary digits correspond to one hexadecimal digit. That makes it easy to convert between binary and hexadecimal values. For example,

$$11|1011|0001_{\text{binary}} = 3B1_{\text{hexadecimal}}$$

In C++, hexadecimal integers are denoted with a `0x` prefix, such as `0x3B1`.

Table 2 Hexadecimal Digits

Hexadecimal	Decimal	Binary	Hexadecimal	Decimal	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Bit and Shift Operations

There are four bit operations in C++: the unary negation (~) and the binary *and* (&), *or* (|), and *exclusive or* (^), often called *xor*.

The tables below show the truth tables for the bit operations in C++. When a bit operation is applied to integer values, the operation is carried out on corresponding bits.

Table 3 The Unary Negation Operation

a	~a
0	1
1	0

Table 4 The Binary And, Or, and Xor Operations

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

For example, suppose you want to compute `46 & 13`. First convert both values to binary. $46_{\text{decimal}} = 101110_{\text{binary}}$ (actually `00000000000000000000000000101110` as a 32-bit integer), and $13_{\text{decimal}} = 1101_{\text{binary}}$. Now combine corresponding bits:

```
  0.....0101110
& 0.....0001101
  0.....0001100
```

The answer is $1100_{\text{binary}} = 12_{\text{decimal}}$.

You sometimes see the `|` operator being used to combine two bit patterns. For example, the symbolic constant `BOLD` is the value 1, and the symbolic constant `ITALIC` is 2. The binary *or* combination `BOLD | ITALIC` has both the bold and the italic bit set:

```
  0.....0000001
| 0.....0000010
  0.....0000011
```

Don't confuse the `&` and `|` bit operators with the `&&` and `||` operators. The latter should be thought of as operating only on `bool` values, not on bits of numbers. However, through the accident of history (C++ did not originally have Boolean values), these operators also work with integer values. To see the difference yourself, try assigning the value of `1 & 2` to an integer variable and printing the result. Then try the same with `1 && 2`. Whether they are working with integers or Booleans, another important difference is that the `&&` and `||` operators evaluate their result using lazy evaluation. This means that if the result can be determined using the left operand by itself, then the right operand is not even considered.

In addition to the operators that work on individual bits, there are shift operators that take the bit pattern of a number and shift it to the left or right by a given number of positions.

The left shift (<<) moves all bits to the left, filling in zeroes in the least significant bits (see Figure 2). Shifting to the left by n bits yields the same result as multiplication by 2^n. The expression

```
1 << n
```

yields a bit pattern in which the nth bit is set (where the `0` bit is the least significant bit).

To set the nth bit of a number, carry out the operation

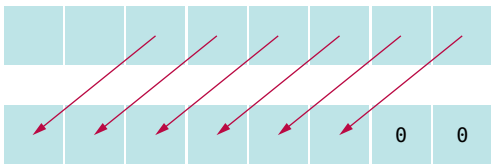
```
x = x | 1 << n
```

To check whether the nth bit is set, execute the test

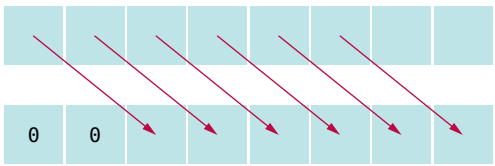
```
if ((x & 1 << n) != 0) . . .
```

Note that the parentheses around the `&` are required—the `&` operator has a lower precedence than the relational operators.

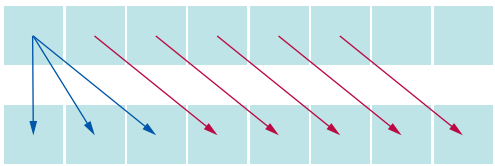
The right shift (>>) moves bits to the right. An important question for right shifts is the bit value that is assigned to the high-order positions as the bits are shifted. For unsigned integers, zero bits are used. Therefore, the result is the same as integer division by 2^n. For signed integers the language specification does not specify which bit values should be used. Many platforms will duplicate the sign bit, however this behavior is not guaranteed, so right shifts using anything other than unsigned integers should not be used. Figure 2 shows both variations of the right shift operator.



Left shift (<<)



Logical right shift (>>)



Arithmetic right shift (>>)

Figure 2 The Shift Operations

GLOSSARY

Abstract data type A specification of the fundamental operations that characterize a data type, without supplying an implementation.

Access specifier A reserved word that indicates the accessibility of a feature, such as private or public.

Accessor function A member function that accesses an object but does not change it.

Address A value that specifies the location of a variable in memory.

Aggregation The "*has-a*" relationship between classes.

Algorithm An unambiguous, executable, and terminating specification of a way to solve a problem.

ANSI/ISO C++ Standard The standard for the C++ language that was developed by the American National Standards Institute and the International Standards Organization.

API (Application Programming Interface) A code library for building programs.

Argument A value supplied in a function call, or one of the values combined by an operator.

Array A collection of values of the same type stored in contiguous memory locations, each of which can be accessed by an integer index.

Arrow operator The `->` operator. `p->m` is the same as `(*p).m`.

ASCII code The American Standard Code for Information Interchange, which associates code values between `0` and 127 to letters, digits, punctuation marks, and control characters.

Assignment Placing a new value into a variable.

Assignment statement A statement that places a new value into an existing variable.

Asymmetric bounds Bounds that include the starting index but not the ending index.

Balanced tree A tree in which each subtree has the property that the number of descendants to the left is approximately the same as the number of descendants to the right.

Base class A class from which another class is derived.

Behavior (of an object) The actions taken by an object when its functions are invoked.

Big-Oh notation The notation $g(n) = O(f(n))$, which denotes that the function g grows at a rate that is bounded by the growth rate of the function f with respect to n . For example, $10n^2 + 100n - 1000 = O(n^2)$.

Big three The three management functions that are essential for classes that manage free store memory or other resources: copy constructor, destructor, and assignment operator.

Binary file A file in which values are stored in their binary representation and cannot be read as text.

Binary operator An operator that takes two arguments, for example + in $x + y$.

Binary search A fast algorithm for finding a value in a sorted array. It narrows the search down to half of the array in every step.

Binary search tree A binary tree in which *each* subtree has the property that all left descendants are smaller than the value stored in the root, and all right descendants are larger.

Binary tree A tree in which each node has at most two child nodes.

Bit Binary digit; the smallest unit of information, having two possible values: `0` and 1. A data element consisting of n bits has 2^n possible values.

Black Box A device with a given specification but unknown implementation.

Block A group of statements bracketed by `{}`.

Body All statements of a function or block.

Boolean operator An operator that can be applied to Boolean values. C++ has three Boolean operators: `&&`, `||`, and `!`.

Boolean type A type with two possible values: `true` and `false`.

Boundary test case A test case involving values that are at the outer boundary of the set of legal values. For example, if a function is expected to work for all nonnegative integers, then `0` is a boundary test case.

Bounds error Trying to access an array element that is outside the legal range.

break statement A statement that terminates a loop or `switch` statement.

Breakpoint A point in a program, specified in a debugger, at which the debugger stops executing the program and lets the user inspect the program state.

Buffer A storage location for holding values that can expand and contract as needed (for example, to hold characters typed by the user) until they are consumed.

Bug A programming error.

Byte A number made up of eight bits. Essentially all currently manufactured computers use a byte as the smallest unit of storage in memory.

Call stack The ordered set of all functions that currently have been called but not yet terminated, starting with the current function and ending with `main`.

Capacity The number of values that a data structure such as an array can potentially hold, in contrast to the size (the number of elements it currently holds).

Case sensitive Distinguishing upper- and lowercase characters.

Cast Explicitly converting a value from one type to a different type. For example, the cast from a floating-point number `x` to an integer is expressed in C++ by the static cast notation `static_cast<int>(x)`.

Central processing unit (CPU) The part of a computer that executes the machine instructions.

Character A single letter, digit, or symbol.

Class A programmer-defined data type.

Class template A specification describing how a class will be constructed once template parameters are provided.

Command line The line the user types to start a program in DOS or UNIX or a command window in Windows. It consists of the program name followed by any necessary arguments.

Command line arguments Additional strings of information provided at the command line that the program can use.

Comment An explanation to help the human reader understand a section of a program; ignored by the compiler.

Compiler A program that translates code in a high-level language (such as C++) to machine instructions.

Compile-time error An error that is detected when a program is compiled.

Compound statement A statement such as `if` or `for` that is made up of several parts (for example, condition, body).

Computer program A sequence of instructions that is executed by a computer.

Concatenate To place one string after another to form a new string.

Constant A value that cannot be changed by a program. In C++, constants are marked with the reserved word `const`.

Construction Setting a newly allocated object to an initial state.

Constructor A sequence of statements for initializing a newly allocated object.

Container A data structure, such as a list, that can hold a collection of objects and present them individually to a program.

Conversion operator A member function operator used to convert from one type to another.

Copy constructor A function that initializes an object as a copy of another.

CPU (Central Processing Unit) The part of a computer that executes the machine instructions.

Dangling pointer A pointer that does not point to a valid location.

Data member A variable that is present in every object of a class.

Delimiter A character or sequence of characters used to specify the beginning or end of a text segment.

De Morgan's Law A law about logical operations that describes how to negate expressions formed with *and* and *or* operations.

Debugger A program that lets a user run another program one or a few steps at a time, stop execution, and inspect the variables in order to analyze it for bugs.

Declaration A statement that announces the existence of a variable, function, or class but does not define it.

Default constructor A constructor that can be invoked with no parameters.

`#define` directive A directive that defines constant values and macros for the preprocessor. Values can be queried during the preprocessing phase with the `#if` and `#ifndef` directives. Macros are replaced by the preprocessor when they are encountered in the program file.

Definition A statement or series of statements that fully describes a variable, a function and its implementation, a type, or a class and its properties.

`delete` operator The operator that recycles memory to the free store.

Deque A sequential container that permits efficient insertion and removal of elements from either end.

Dereferencing Locating an object when a pointer to the object is given.

Derived class A class that modifies a base class by adding data members, adding member functions, or redefining member functions.

Destructor A function that is executed whenever an object goes out of scope.

Directory A structure on a disk that can hold files or other directories; also called a folder.

Documentation comment A comment in a source file that can be automatically extracted into the program documentation by a program such as `javadoc`.

Dot notation The notation *object.function*(*parameters*) used to invoke a member function on an object.

Doubly-linked list A linked list in which each link has a reference to both its predecessor and successor links.

Dynamic memory allocation Allocating memory as a program runs as required by the program's needs.

Editor A program for writing and modifying text files.

Element A storage location in an array.

Embedded system The processor, software, and supporting circuitry that is included in a device other than a computer.

Encapsulation The hiding of implementation details.

Escape character A character in text that is not taken literally but has a special meaning when combined with the character or characters that follow it. The `\\` character is an escape character in C++ strings.

Escape sequence A sequence of characters that starts with an escape character, such as `\\n` or `\\&#x0022;`.

Exception A class that signals a condition that prevents the program from continuing normally. When such a condition occurs, an object of the exception class is thrown.

Executable file The file that contains a program's machine instructions.

Explicit parameter A parameter of a member function other than the object on which the function is invoked.

Expression A syntactical construct that is made up of constants, variables, function calls, and the operators combining them.

Extension The last part of a file name, which specifies the file type. For example, the extension `.cpp` denotes a C++ file.

Failed stream state The state of a stream after an invalid operation has been attempted, such as reading a number when the next stream position yielded a nondigit, or reading after the end of file was reached.

Fibonacci numbers The sequence of numbers 1, 1, 2, 3, 5, 8, 13, . . . , in which every term is the sum of its two predecessors.

File A sequence of bytes that is stored on disk.

File pointer The position within a random-access file of the next byte to be read or written. It can be moved so as to access any byte in the file.

Flag A type with two possible values: `true` and `false`.

Floating-point number A number that can have a fractional part.

Folder A structure on a disk that can hold files or other folders; also called a directory.

Free store A reservoir of storage from which memory can be allocated when a program runs.

Function A sequence of statements that can be invoked multiple times, with different values for its parameter variables.

Function call operator An operator that is invoked using the same syntax as a function call. Defining this operator in a class produces a function object.

Function signature The name of a function and the types of its parameters.

Garbage collection Automatic reclamation of memory occupied by objects that are no longer referenced. C++ does not have garbage collection

Global variable A variable whose scope is not restricted to a single function.

`grep` The "global regular expression print" search program, useful for finding all strings matching a pattern in a set of files.

Hard disk A device that stores information on rotating platters with magnetic coating.

Hardware The physical equipment for a computer or another device.

Hash code A value that is computed by a hash function.

Hash collision Two different objects for which a hash function computes identical values.

Hash function A function that computes an integer value from an object in such a way that different objects are likely to yield different values.

Hash table A data structure in which elements are mapped to array positions according to their hash function values.

Hashing Applying a hash function to a set of objects.

Header file A file that informs the compiler of features that are available in another module or library.

Heap A balanced binary tree that is used for implementing sorting algorithms and priority queues.

Heapsort algorithm A sorting algorithm that inserts the values to be sorted into a heap.

High-level programming language A programming language that provides an abstract view of a computer and allows programmers to focus on their problem domain.

HTML (Hypertext Markup Language) The language in which web pages are described.

IDE (Integrated Development Environment) A programming environment that includes an editor, compiler, and debugger.

`#if` directive A directive to the preprocessor to include the code contained between the `#if` and the matching `#endif` if a condition is true.

Implicit parameter The object on which a member function is invoked. For example, in the call `x.f(y)`, the object `x` is the implicit parameter of the function `f`.

`#include` directive An instruction to the preprocessor to include a header file.

Index The position of an element in an array.

Inheritance The "*is-a*" relationship between a general base class and a specialized derived class.

Initialization Setting a variable to a well-defined value when it is created.

Input stream An abstraction for a sequence of bytes from which data can be read.

Instance of a class An object whose type is that class.

Integer A number that cannot have a fractional part.

Integer division Taking the quotient of two integers and discarding the remainder. In C++ the `/` symbol denotes integer division if both arguments are integers. For example, `11/4` is `2`, not `2.75`.

Integrated Development Environment (IDE) A programming environment that includes an editor, compiler, and debugger.

Internet A worldwide collection of networks, routing equipment, and computers using a common set of protocols that define how participants interact with each other.

Iterator An object that represents a position in a container such as a linked list and that can be used to read or update the element at that position.

Lexicographic ordering Ordering strings in the same order as in a dictionary, by skipping all matching characters and comparing the first non-matching characters of both strings. For example, "orbit" comes before "orchid" in lexicographic ordering. Note that in C++, unlike a dictionary, the ordering is case sensitive: `Z` comes before `a`.

Library A set of precompiled classes and functions that can be included in programs.

Linear search Searching a container (such as an array or list) for an object by inspecting each element in turn.

Linked list A data structure that can hold an arbitrary number of objects, each of which is stored in a link object, which contains a pointer to the next link.

Linker The program that combines object and library files into an executable file.

Literal A constant value in a program that is explicitly written as a number, such as –2 or 6.02214115E23, or as a character sequence, such as `"Harry"`.

Local variable A variable whose scope is a block.

Logic error An error in a syntactically correct program that causes it to act differently from its specification. (A form of run-time error.)

Logical operator An operator that can be applied to Boolean values. C++ has three Boolean operators: `&&`, `||`, and `!`.

Loop A sequence of instructions that is executed repeatedly.

Loop and a half A loop whose termination decision is neither at the beginning nor at the end.

Machine code Instructions that can be executed directly by the CPU.

Magic number A number that appears in a program without explanation.

`main` function The function that is first called when a program executes.

Map A data structure that keeps associations between keys and values.

Member function A function that is defined by a class and operates on objects of that class.

Memory The circuitry that stores code and data in a computer.

Memory leak Memory that is dynamically allocated but never returned to the free store manager. A succession of memory leaks can cause the free store manager to run out of memory.

Memory location A value that specifies the location of data in computer memory.

Merge sort A sorting algorithm that first sorts two halves of a data structure and then merges the sorted subarrays together.

Modulus The `%` operator that computes the remainder of an integer division.

Mutator function A member function that changes the state of an object.

Mutual recursion Cooperating functions that call each other.

Nested block A block that is contained inside another block.

Nested loop A loop that is contained in another loop.

`new` operator The operator that allocates new memory from the free store.

Newline The `'\n'` character, which indicates the end of a line.

null pointer The value `nullptr` that indicates that a pointer does not point to any object.

Number literal A constant value in a program that is explicitly written as a number, such as –2 or 6.02214115E23.

Object A value of a class type.

Object-oriented programming Designing a program by discovering objects, their properties, and their relationships.

Off-by-one error A common programming error in which a value is one larger or smaller than it should be.

Operating system The software that launches application programs and provides services (such as a file system) for those programs.

Operator A symbol denoting a mathematical or logical operation, such as `+` or `&&`.

Operator associativity The rule that governs in which order operators of the same precedence are executed. For example, in C++ the `-` operator is left-associative because `a - b - c` is interpreted as `(a - b) - c`, and `=` is right-associative because `a = b = c` is interpreted as `a = (b = c)`.

Operator precedence The rule that governs which operator is evaluated first. For example, in C++ the `&&` operator has a higher precedence than the `||` operator. Hence `a || b && c` is interpreted as `a || (b && c)`.

Output stream An abstraction for a sequence of bytes to which data can be written.

Overloading Giving more than one meaning to a function name or operator.

Overriding Redefining a function from a base class in a derived class.

Parallel arrays Arrays of the same length, in which corresponding elements are logically related.

Parallel vectors Vectors of the same length, in which corresponding elements are logically related.

Parameter An item of information that is specified to a function when the function is called.

Parameter passing Specifying expressions to be arguments for a function when it is called.

Parameter variable A variable of a function that is initialized with a value when the function is called.

Partially filled array An array that is not filled to capacity, together with a companion variable that indicates the number of elements actually stored.

Path (to a file or directory) The sequence of directory names and, for a file, a file name at the end, that describes how to reach the file or directory from a given starting point.

Permutation A rearrangement of a set of values.

Pointer A value that denotes the memory location of an object.

Polymorphism Selecting a function among several functions that have the same name on the basis of the actual type of the implicit parameter.

Postfix operator A unary operator that is written after its argument.

Prefix operator A unary operator that is written before its argument.

Priority queue An abstract data type that enables efficient insertion of elements and efficient removal of the element with the highest priority.

Programming The act of designing and implementing computer programs.

Prompt A string that tells the user to provide input.

Prototype The declaration of a function, including its parameter types and return type.

Pseudocode A high-level description of the actions of a program or algorithm, using a mixture of English and informal programming language syntax.

Pseudorandom number A number that appears to be random but is generated by a mathematical formula.

Public interface The features (functions, variables, and nested types) of a class that are accessible to all clients.

Queue A collection of items with "first in, first out" retrieval.

Quicksort A generally fast sorting algorithm that picks an element, called the pivot, partitions the sequence into the elements smaller than the pivot and those larger than the pivot, and then recursively sorts the subsequences.

RAM (random-access memory) Electronic circuits in a computer that can store code and data of running programs.

Random access The ability to access any value directly without having to read the values preceding it.

Recursion A technique for computing a result by decomposing the inputs into simpler values and applying the same function to them.

Recursive function A function that can call itself with simpler values. It must handle the simplest values without calling itself.

Red-black tree A kind of binary search tree that rebalances itself after each insertion and removal.

Redirection Linking the input or output of a program to a file instead of the keyboard or display.

Reference A value that denotes the location of an object in memory.

Reference parameter A parameter that is bound to a variable supplied in the call. Changes made to the parameter within the function affect the variable outside the function.

Regular expression A string that defines a set of matching strings according to their content. Each part of a regular expression can be a specific required character; one of a set of permitted characters such as `[abc]`, which can be a range such as `[a-z]`; any character not in a set of forbidden characters, such as `[^0-9]`; a repetition of one or more matches, such as `[0-9]+`, or zero or more, such as `[ACGT]`; one of a set of alternatives, such as `and|et|und`; or various other possibilities. For example, `[A-Za-z][0-9]+` matches `Cloud9` or `007` but not `Jack`.

Relational operator An operator that compares two values, yielding a Boolean result.

Reserved word A word that has a special meaning in a programming language and therefore cannot be used as a name by the programmer.

Return value The value returned by a function through a `return` statement.

Reverse Polish notation A style of writing expressions in which the operators are written following the operands, such as `2 3 4 * +` for `2 + 3 * 4`.

Roundoff error An error introduced by the fact that the computer can store only a finite number of digits of a floating-point number.

Run-time error An error in a syntactically correct program that causes it to act differently from its specification.

Run-time stack The data structure that stores the local variables of all called functions as a program runs.

Scope The part of a program in which a variable is defined.

Secondary storage Storage that persists without electricity, e.g., a hard disk.

Selection sort A sorting algorithm in which the smallest element is repeatedly found and removed until no elements remain.

Sentinel A value in input that is not to be used as an actual input value but to signal the end of input.

Sequential access Accessing values one after another without skipping over any of them.

Sequential search Searching a container (such as an array or list) for an object by inspecting each element in turn.

Set An unordered collection that allows efficient addition, location, and removal of elements.

Shadowing Hiding a variable by defining another one with the same name.

Shell window A window for interacting with an operating system through textual commands.

Short-circuit evaluation Evaluating only a part of an expression if the remainder cannot change the result.

Sign bit The bit of a binary number that indicates whether the number is positive or negative.

Slicing an object Copying an object of a derived class into a variable of the base class, thereby losing the derived-class data.

Software The intangible instructions and data that are necessary for operating a computer or another device.

Source code Instructions in a programming language that need to be translated before execution on a computer.

Source file A file containing instructions in a programming language such as C++.

Stack A data structure with "last-in, first-out" retrieval. Elements can be added and removed only at one position, called the top of the stack.

Stack trace A printout of the call stack, listing all currently pending function calls.

State The current value of an object, which is determined by the cumulative action of all functions that were invoked on it.

Statement A syntactical unit in a program. In C++ a statement is either a simple statement, a compound statement, or a block.

Stepwise refinement The process of solving a problem that starts out with a subdivision into steps, then continues by further subdividing those steps.

Stream An abstraction for a sequence of bytes from which data can be read or to which data can be written.

String A sequence of characters.

Structure A construct for aggregating items of arbitrary types into a single value.

Stub A function with no or minimal functionality.

Subclass A synonym for derived class.

Substitution principle The principle that a derived-class object can be used in place of any base-class object.

Superclass A synonym for base class.

Symmetric bounds Bounds that include the starting index and the ending index.

Syntax Rules that define how to form instructions in a particular programming language.

Syntax error An instruction that does not follow the programming language rules and is rejected by the compiler. (A form of compile-time error.)

Tab character The `'\t'` character, which advances the next character on the line to the next one of a set of fixed positions known as tab stops.

Ternary operator An operator with three arguments. C++ has one ternary operator, `a ? b : c`.

Test coverage The instructions of a program that are executed when a set of test cases are run.

Test suite A set of test cases for a program.

Text file A file in which values are stored in their text representation.

Token A sequence of consecutive characters from an input source that belongs together for the purpose of analyzing the input. For example, a token can be a sequence of characters other than white space.

Trace message A message that is printed during a program run for debugging purposes.

Tree A data structure consisting of nodes, each of which has a list of child nodes, and one of which is distinguished as the root node.

Turing machine A very simple model of computation that is used in theoretical computer science to explore computability of problems.

Two-dimensional array A tabular arrangement of elements in which an element is specified by a row and a column index.

Type A named set of values and the operations that can be carried out with them.

Type parameter A parameter in a class template or function template that can be replaced with an actual type.

Unary operator An operator with one argument.

Unicode A standard code that assigns code values consisting of two bytes to characters used in scripts around the world.

Unified Modeling Language (UML) A notation for specifying, visualizing, constructing, and documenting the artifacts of software systems.

Uninitialized variable A variable that has not been set to a particular value. It is filled with whatever "random" bytes happen to be present in the memory location that the variable occupies.

Unit test A test of a function by itself, isolated from the remainder of the program.

Value parameter A function parameter whose value is copied into a parameter variable of a function. If a variable is passed as a value parameter, changes made to the parameter inside the function do not affect the original variable outside the program.

Variable A symbol in a program that identifies a storage location that can hold different values.

Vector The standard C++ template for a dynamically-growing array.

Virtual function A function that can be redefined in a derived class. The actual function called depends on the type of the object on which it is invoked at run time.

`void` A reserved word indicating no type or an unknown type.

Walkthrough A step-by-step manual simulation of a computer program.

White space Any sequence of only space, tab, and newline characters.

INDEX

Note: Page numbers followed by f indicate figures; those followed by t indicate tables. Functions and classes from the C++ library are listed by name. For functions and classes in example programs, see "applications: programs and simulations."

Symbols

A

B

C

D

E

F

G

H

I

K

L

N

O

P

Q

R

S

T

U

V

W

Z

ILLUSTRATION CREDITS

Icons

Common Error icon: © Scott Harms/iStockphoto.
How To icon: © Steve Simzer/iStockphoto.
Paperclip: © Yvan Dubé/iStockphoto.
Programming Tip icon: © Macdaddy/Dreamstime.com.
Computing and Society icon: © Mishella/Dreamstime.com.
Self Check icon: © Nicholas Homrich/iStockphoto.
Special Topic icon: ©Nathan Winter/iStockphoto.
Worked Example icon: © Tom Horyn/iStockphoto.

Chapter 4

Page 130-133 (left to right, top to bottom):
Gogh, Vincent van *The Olive Orchard:* Chester Dale Collection 1963.10.152/National Gallery of Art.
Degas, Edgar *The Dance Lesson:* Collection of Mr. and Mrs. Paul Mellon 1995.47.6/National Gallery of Art.
Fragonard, Jean-Honoré *Young Girl Reading:* Gift of Mrs. Mellon Bruce in memory of her father, Andrew W. Mellon 1961.16.1/National Gallery of Art.
Gauguin, Paul *Self-Portrait:* Chester Dale Collection 1963.10.150/National Gallery of Art.
Gauguin, Paul *Breton Girls Dancing, Pont-Aven:* Collection of Mr. and Mrs. Paul Mellon 1983.1.19/National Gallery of Art.
Guigou, Paul *Washerwomen on the Banks of the Durance:* Chester Dale Fund 2007.73.1/National Gallery of Art.
Guillaumin, Jean-Baptiste-Armand *The Bridge of Louis Philippe:* Chester Dale Collection 1963.10.155/National Gallery of Art.
Manet, Edouard *The Railway:* Gift of Horace Havemeyer in memory of his mother, Louisine W. Havemeyer 1956.10.1/National Gallery of Art.
Manet, Edouard *Masked Ball at the Opera:* Gift of Mrs. Horace Havemeyer in memory of her mother-in-law, Louisine W. Havemeyer 1982.75.1/National Gallery of Art.
Manet, Edouard *The Old Musician:* Chester Dale Collection 1963.10.162/National Gallery of Art.
Monet, Claude *The Japanese Footbridge:* Gift of Victoria Nebeker Coberly, in memory of her son John W. Mudd, and Walter H. and Leonore Annenberg 1992.9.1/National Gallery of Art.
Monet, Claude *Woman with a Parasol—Madame Monet and Her Son:* Collection of Mr. and Mrs. Paul Mellon 1983.1.29/National Gallery of Art.
Monet, Claude *The Bridge at Argenteuil:* Collection of Mr. and Mrs. Paul Mellon 1983.1.24/National Gallery of Art.
Monet, Claude *The Artist's Garden in Argenteuil (A Corner of the Garden with Dahlias):* Gift of Janice H. Levin, in Honor of the 50th Anniversary of the National Gallery of Art 1991.27.1/National Gallery of Art.

Variable and Constant Definitions

Type *Name* *Initial value*

```
int cans_per_pack = 6;

const double CAN_VOLUME = 0.335;
```

Mathematical Operations

```
#include <cmath>
```

`pow(x, y)`	Raising to a power x^y
`sqrt(x)`	Square root $\sqrt{x}$
`log10(x)`	Decimal log $\log_{10}(x)$
`abs(x)`	Absolute value $\lvert x \rvert$
`sin(x)`, `cos(x)`, `tan(x)`	Sine, cosine, tangent of x (x in radians)

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

`[]`	Array element access
`++ -- !`	Increment, decrement, Boolean *not*
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`< <= > >=`	Comparisons
`== !=`	Equal, not equal
`&&`	Boolean *and*
`||`	Boolean *or*
`=`	Assignment

Loop Statements

Condition

```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
```

Executed while condition is true

Initialization *Condition* *Update*

```
for (int i = 0; i < 10; i++)
{
   cout << i << endl;
}
```

Loop body executed at least once

```
do
{
   cout << "Enter a positive integer: ";
   cin >> input;
}
while (input <= 0);
```

Conditional Statement

Condition

```
if (floor >= 13)
{
   actual_floor = floor - 1;
}
```

Executed when condition is true

```
else if (floor >= 0)
```

Second condition (optional)

```
{
   actual_floor = floor;
}
else
{
   cout << "Floor negative" << endl;
}
```

Executed when all conditions are false (optional)

String Operations

```
#include <string>
string s = "Hello";
int n = s.length(); // 5
string t = s.substr(1, 3); // "ell"
string c = s.substr(2, 1); // "l"
char ch = s[2]; // 'l'
for (int i = 0; i < s.length(); i++)
{
   string c = s.substr(i, 1);
   or char ch = s[i];
   Process c or ch
}
```

Function Definitions

Return type *Parameter type and name*

```
double cube_volume(double side_length)
{
   double vol = side_length * side_length * side_length;
   return vol;
}
```

Exits function and returns result.

Reference parameter

```
void deposit(double& balance, double amount)
{
   balance = balance + amount;
}
```

Modifies supplied argument

Arrays

Element type *Length*

```
int numbers[5];
int squares[] = { 0, 1, 4, 9, 16 };
int magic_square[4][4] =
{
   { 16, 3, 2, 13 },
   { 5, 10, 11, 8 },
   { 9, 6, 7, 12 },
   { 4, 15, 14, 1 }
};

for (int i = 0; i < size; i++)
{
   Process numbers[i]
}
```

Vectors

```
#include <vector>

vector<int> values = { 0, 1, 4, 9, 16 };

vector<string> names;

names.push_back("Ann");
names.push_back("Cindy"); // names.size() is now 2

names.pop_back(); // Removes last element

names[0] = "Beth"; // Use [] for element access
```

Element type

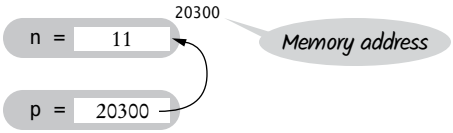
Initial values (C++ 11)

Initially empty

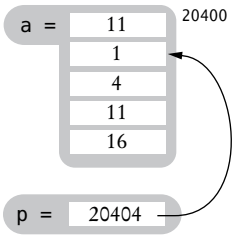
Add elements to the end

Pointers

```
int n = 10;
int* p = &n; // p set to address of n
*p = 11; // n is now 11
```




```
int a[5] = { 0, 1, 4, 9, 16 };
p = a; // p points to start of a
*p = 11; // a[0] is now 11
p++; // p points to a[1]
p[2] = 11; // a[3] is now 11
```



Input and Output

```
#include <iostream>
cin >> x; // x can be int, double, string
cout << x;

while (cin >> x) { Process x }
if (cin.fail()) // Previous input failed


#include <fstream>
string filename = ...;
ifstream in(filename);
ofstream out("output.txt");

string line; getline(in, line);
char ch; in.get(ch);
```

Range-based for Loop

```
for (int v : values)
{
   cout << v << endl;
}
```

An array, vector, or other container (C++ 11)

Output Manipulators

```
#include <iomanip>
```

`endl`	Output new line
`fixed`	Fixed format for floating-point
`setprecision(`*n*`)`	Number of digits after decimal point for fixed format
`setw(`*n*`)`	Field width for the next item
`left`	Left alignment (use for strings)
`right`	Right alignment (default)
`setfill(`*ch*`)`	Fill character (default: space)

Class Definition

```
class BankAccount
{
public:
   BankAccount(double amount);
   void deposit(double amount);
   double get_balance() const;
   ...
private:
   double balance;
};

void BankAccount::deposit(double amount)
{
   balance = balance + amount;
}
```

Constructor declaration

Member function declaration

Accessor member function

Data member

Member function definition

Inheritance

```
class CheckingAccount : public BankAccount
{
public:
   void deposit(double amount);
private:
   int transactions;
};

void CheckingAccount::deposit(double amount)
{
   BankAccount::deposit(amount);
   transactions++;
}
```

Derived class

Base class

Member function overrides base class

Added data member in derived class

Calls base class member function

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.