Big C++
Cay Horstmann
Late Objects
3/e
WILEY

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*.

Copyright © 2018, 2012, 2009 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website *www.copyright.com*. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201) 748-6011, fax (201) 748-6008, website *http://www.wiley.com/go/permissions*.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return shipping label are available at *www.wiley.com/go/returnlabel*. If you have chosen to adopt this textbook for use in your course, please accept this book as your complimentary desk copy. Outside of the United States, please contact your local representative.

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.

© Terraxplorer/iStockphoto.

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

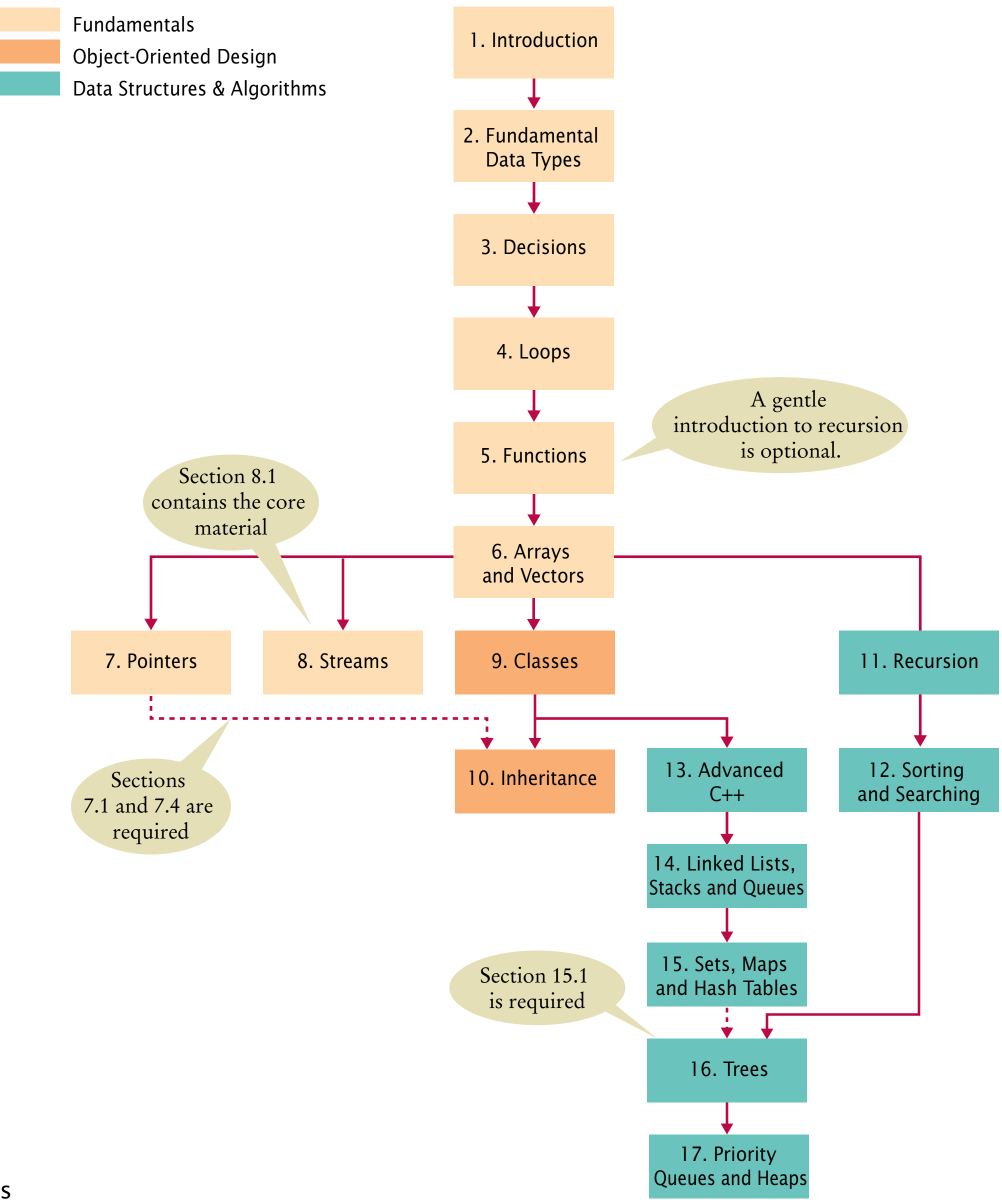


Figure 1
Chapter Dependencies

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.

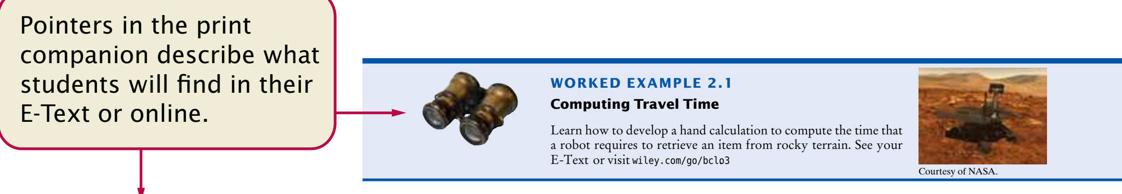
Appendix E contains a programming style guide. Using a style guide for programming assignments benefits students by directing them toward good habits and reducing gratuitous choice. The style guide is available in electronic form on the book's companion web site so that instructors can modify it to reflect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



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

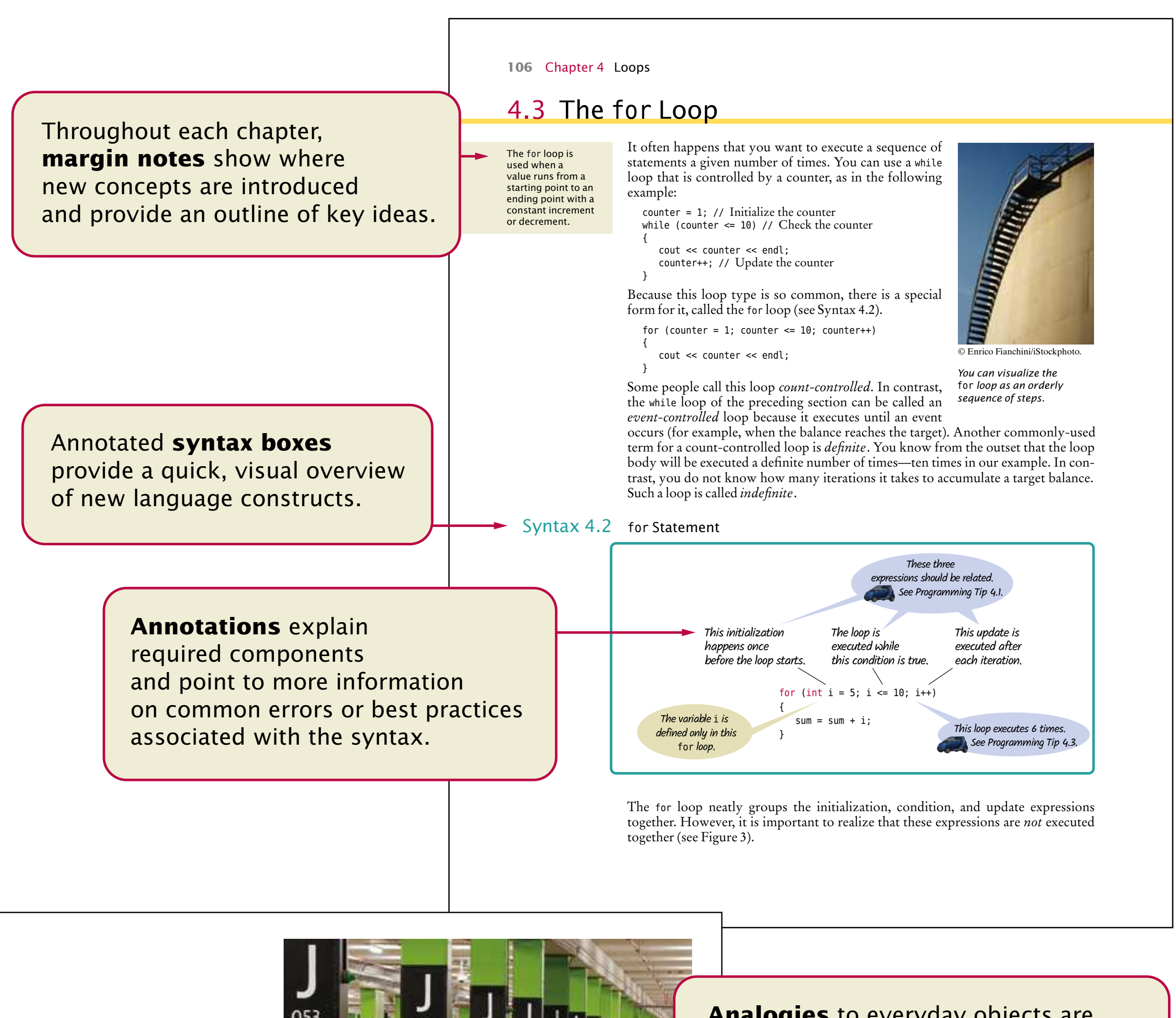
A Walkthrough of the Learning Aids

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.

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

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



106 Chapter 4 Loops

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

```
for (int i = 5; i <= 10; i++)
{
   sum = sum + i;
}
```

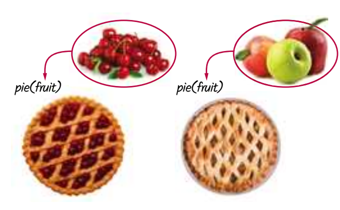
These three expressions should be related. See Programming Tip 4.1.
This initialization happens once before the loop starts.
The loop is executed while this condition is true.
This update is executed after each iteration.
The variable i is defined only in this for loop.
This loop executes 6 times. See Programming Tip 4.3.

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.



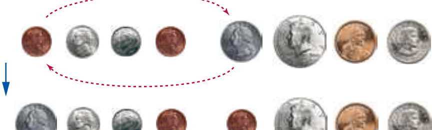
A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.

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.

6.5 Problem Solving: Discovering Algorithms by Manipulating Physical Objects 277

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

❶ Function call

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

result1 =

side_length =

❷ Initializing function parameter variable

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

result1 =

side_length = 2

❸ About to return to the caller

```
double volume = side_length * side_length * side_length;
return volume;
```

result1 =

side_length = 2

volume = 8

❹ After function call

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

result1 = 8

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.

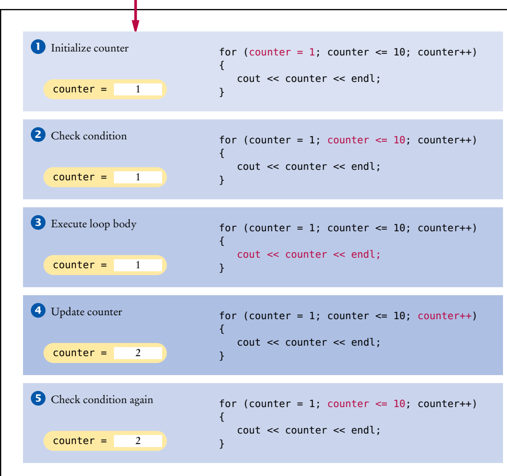
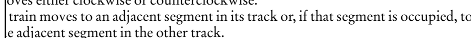


Figure 3 Execution of a `for` Loop

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

```
 1  #include <iostream>
 2
 3  using namespace std;
 4
 5  /**
 6     Computes the volume of a cube.
 7     @param side_length the side length of the cube
 8     @return the volume
 9  */
10  double cube_volume(double side_length)
11  {
12     double volume = side_length * side_length * side_length;
13     return volume;
14  }
15
16  int main()
17  {
18     double result1 = cube_volume(2);
19     double result2 = cube_volume(10);
20     cout << "A cube with side length 2 has volume " << result1 << endl;
21     cout << "A cube with side length 10 has volume " << result2 << endl;
22
23     return 0;
24  }
```

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 r 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 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

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.

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

Trace through a code segment

• 1. In this activity, trace through the code by clicking on the line that will be executed next. Observe the inputs as they appear in the table below. They denote hours in "military time" between 0 and 23. For each input, click on the line inside the if statement that will be executed when the hour variable has that value.

Please click on the next line.

```
cin >> hour;
if (hour < 12)
{
   greeting = "Good morning";
}
else
{
   greeting = "Good afternoon";
}
cout << greeting << endl;
```

hour	greeting
11	Good morning
13	

2 correct, 0 errors

Start over

Build an example table

• 2. Consider the following statement:

```
if (hour < 21)
{
   response = "Goodbye";
}
else
{
   response = "Goodnight";
}
```

Determine the value of response when hour has the values given in the table below.

Complete the second column. Press Enter to submit each entry.

hour	response	Explanation
20	"Goodbye"	20 < 21, and the first branch of the statement executes.
	"Goodnight"	It is not true that 22 < 21, so the else clause executes.

Explore common algorithms

Play with the following activity to learn how to find the largest value in an array. You visit each element in turn, incrementing the position i. When you find an element that is larger than the largest one that you have seen so far, store it as the largest. When you have visited all elements, you have found the largest value.

Press the Start button to see the array. Press the Start over button to try the activity again with different values.

Select the next action.

```
   i
a: 20 30 34 60 21 55 70

largest:
```

Increment i | Store as largest | Done

Arrange code to fulfill a task

•• 1. Assume that weekdays are coded as 0 = Monday, 1 = Tuesday, ..., 4 = Friday, 5 = Saturday, 6 = Sunday. Rearrange the lines of code so that weekday is set to the next working day (Monday through Friday). Not all lines are useful.

Order the statements by dragging them into the left window. Use the guidelines for proper indenting.

```
if (weekday < 4)
{
   weekday++;
}
else
{
}
```

```
weekday = 5;
weekday = 0;
weekday = 1;
if (weekday <= 5)
```

0 correct, 0 errors

Create a memory diagram

•• 2. What happens when the following code is executed?

```
BankAccount* p = new BankAccount(2000);
BankAccount* q = new BankAccount(1000);
q->withdraw(500); ❶
p = q; ❷
p->withdraw(200); ❸
```

❷ Update the p variable.

Draw the arrow.

p = → BankAccount, balance = 2000

q = → BankAccount, balance = 500

One correct, 0 errors

Start over

Complete a program and get immediate feedback

•• 2. Write a program that reads a word and prints whether

- it is short (fewer than 5 letters).
- it is long (at least 10 letters).
- it ends with the letter y.
- has the same first and last character.

▸ Show Code to be Completed

Complete the code in your IDE or go to CODE CHECK to complete the code and evaluate your

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

PREFACE **iii**

SPECIAL FEATURES **xxiv**

1 INTRODUCTION **1**

1.1 What Is Programming? **2**

1.2 The Anatomy of a Computer **3**

C&S Computers Are Everywhere 5

1.3 Machine Code and Programming Languages **5**

C&S Standards Organizations 7

1.4 Becoming Familiar with Your Programming Environment **7**

PT1 Backup Copies 10

1.5 Analyzing Your First Program **11**

CE1 Omitting Semicolons 13

ST1 Escape Sequences 13

1.6 Errors **14**

CE2 Misspelling Words 15

1.7 PROBLEM SOLVING Algorithm Design **16**

The Algorithm Concept 16

An Algorithm for Solving an Investment Problem 17

Pseudocode 18

From Algorithms to Programs 19

HT1 Describing an Algorithm with Pseudocode 19

WE1 Writing an Algorithm for Tiling a Floor 21

2 FUNDAMENTAL DATA TYPES **25**

2.1 Variables **26**

Variable Definitions 26

Number Types 28

Variable Names 29

The Assignment Statement 30

Constants 31

Comments 31

CE1 Using Undefined Variables 33

CE2 Using Uninitialized Variables 33

PT1 Choose Descriptive Variable Names 33

PT2 Do Not Use Magic Numbers 34

ST1 Numeric Types in C++ 34

ST2 Numeric Ranges and Precisions 35

ST3 Defining Variables with `auto` 35

2.2 Arithmetic **36**

Arithmetic Operators 36

Increment and Decrement 36

Integer Division and Remainder 36

Converting Floating-Point Numbers to Integers 37

Powers and Roots 38

CE3 Unintended Integer Division 39

CE4 Unbalanced Parentheses 40

CE5 Forgetting Header Files 40

CE6 Roundoff Errors 41

PT3 Spaces in Expressions 42

ST4 Casts 42

ST5 Combining Assignment and Arithmetic 42

C&S The Pentium Floating-Point Bug 43

2.3 Input and Output **44**

Input 44

Formatted Output 45

2.4 PROBLEM SOLVING First Do It By Hand **47**

WE1 Computing Travel Time 48

HT1 Carrying out Computations 48

WE2 Computing the Cost of Stamps 51

2.5 Strings **51**

The `string` Type 51

Concatenation 52

String Input 52

String Functions 52

C&S International Alphabets and Unicode 55

3 DECISIONS **59**

3.1 The `if` Statement **60**

CE1 A Semicolon After the `if` Condition 63

PT1 Brace Layout 63

PT2 Always Use Braces 64

PT3 Tabs 64

PT4 Avoid Duplication in Branches 65

ST1 The Conditional Operator 65

3.2 Comparing Numbers and Strings 66
CE2 Confusing = and == 68
CE3 Exact Comparison of Floating-Point Numbers 68
PT5 Compile with Zero Warnings 69
ST2 Lexicographic Ordering of Strings 69
HT1 Implementing an if Statement 70
WE1 Extracting the Middle 72
C&S Dysfunctional Computerized Systems 72

3.3 Multiple Alternatives 73
ST3 The switch Statement 75

3.4 Nested Branches 76
CE4 The Dangling else Problem 79
PT6 Hand-Tracing 79

3.5 PROBLEM SOLVING Flowcharts 81

3.6 PROBLEM SOLVING Test Cases 83
PT7 Make a Schedule and Make Time for Unexpected Problems 84

3.7 Boolean Variables and Operators 85
CE5 Combining Multiple Relational Operators 88
CE6 Confusing && and || Conditions 88
ST4 Short-Circuit Evaluation of Boolean Operators 89
ST5 De Morgan's Law 89

3.8 APPLICATION Input Validation 90
C&S Artificial Intelligence 92

4 LOOPS 95

4.1 The while Loop 96
CE1 Infinite Loops 100
CE2 Don't Think "Are We There Yet?" 101
CE3 Off-by-One Errors 101
C&S The First Bug 102

4.2 PROBLEM SOLVING Hand-Tracing 103

4.3 The for Loop 106
PT1 Use for Loops for Their Intended Purpose Only 109
PT2 Choose Loop Bounds That Match Your Task 110
PT3 Count Iterations 110

4.4 The do Loop 111
PT4 Flowcharts for Loops 111

4.5 Processing Input 112
Sentinel Values 112
Reading Until Input Fails 114
ST1 Clearing the Failure State 115
ST2 The Loop-and-a-Half Problem and the break Statement 116
ST3 Redirection of Input and Output 116

4.6 PROBLEM SOLVING Storyboards 117

4.7 Common Loop Algorithms 119
Sum and Average Value 119
Counting Matches 120
Finding the First Match 120
Prompting Until a Match is Found 121
Maximum and Minimum 121
Comparing Adjacent Values 122
HT1 Writing a Loop 123
WE1 Credit Card Processing 126

4.8 Nested Loops 126
WE2 Manipulating the Pixels in an Image 129

4.9 PROBLEM SOLVING Solve a Simpler Problem First 130

4.10 Random Numbers and Simulations 134
Generating Random Numbers 134
Simulating Die Tosses 135
The Monte Carlo Method 136
C&S Digital Piracy 138

5 FUNCTIONS 141

5.1 Functions as Black Boxes 142

5.2 Implementing Functions 143
PT1 Function Comments 146

5.3 Parameter Passing 146
PT2 Do Not Modify Parameter Variables 148

5.4 Return Values 148
CE1 Missing Return Value 149
ST1 Function Declarations 150
HT1 Implementing a Function 151
WE1 Generating Random Passwords 152
WE2 Using a Debugger 152

5.5 Functions Without Return Values 153

5.6 PROBLEM SOLVING Reusable Functions 154

5.7 PROBLEM SOLVING Stepwise Refinement **156**
- PT3 Keep Functions Short 161
- PT4 Tracing Functions 161
- PT5 Stubs 162
- WE3 Calculating a Course Grade 163

5.8 Variable Scope and Global Variables **163**
- PT6 Avoid Global Variables 165

5.9 Reference Parameters **165**
- PT7 Prefer Return Values to Reference Parameters 169
- ST2 Constant References 170

5.10 Recursive Functions (Optional) **170**
- HT2 Thinking Recursively 173
- C&S The Explosive Growth of Personal Computers 174

6 ARRAYS AND VECTORS 179

6.1 Arrays **180**
- Defining Arrays 180
- Accessing Array Elements 182
- Partially Filled Arrays 183
- CE1 Bounds Errors 184
- PT1 Use Arrays for Sequences of Related Values 184
- C&S Computer Viruses 185

6.2 Common Array Algorithms **185**
- Filling 186
- Copying 186
- Sum and Average Value 186
- Maximum and Minimum 187
- Element Separators 187
- Counting Matches 187
- Linear Search 188
- Removing an Element 188
- Inserting an Element 189
- Swapping Elements 190
- Reading Input 191
- ST1 Sorting with the C++ Library 192
- ST2 A Sorting Algorithm 192
- ST3 Binary Search 193

6.3 Arrays and Functions **194**
- ST4 Constant Array Parameters 198

6.4 PROBLEM SOLVING Adapting Algorithms **198**
- HT1 Working with Arrays 200
- WE1 Rolling the Dice 203

6.5 PROBLEM SOLVING Discovering Algorithms by Manipulating Physical Objects **203**

6.6 Two-Dimensional Arrays **206**
- Defining Two-Dimensional Arrays 207
- Accessing Elements 207
- Locating Neighboring Elements 208
- Computing Row and Column Totals 208
- Two-Dimensional Array Parameters 210
- CE2 Omitting the Column Size of a Two-Dimensional Array Parameter 212
- WE2 A World Population Table 213

6.7 Vectors **213**
- Defining Vectors 214
- Growing and Shrinking Vectors 215
- Vectors and Functions 216
- Vector Algorithms 216
- Two-Dimensional Vectors 218
- PT2 Prefer Vectors over Arrays 219
- ST5 The Range-Based `for` Loop 219

7 POINTERS AND STRUCTURES 223

7.1 Defining and Using Pointers **224**
- Defining Pointers 224
- Accessing Variables Through Pointers 225
- Initializing Pointers 227
- CE1 Confusing Pointers with the Data to Which They Point 228
- PT1 Use a Separate Definition for Each Pointer Variable 229
- ST1 Pointers and References 229

7.2 Arrays and Pointers **230**
- Arrays as Pointers 230
- Pointer Arithmetic 230
- Array Parameter Variables Are Pointers 232
- ST2 Using a Pointer to Step Through an Array 233
- CE2 Returning a Pointer to a Local Variable 234
- PT2 Program Clearly, Not Cleverly 234
- ST3 Constant Pointers 235

7.3 C and C++ Strings 235
The char Type 235
C Strings 236
Character Arrays 237
Converting Between C and C++ Strings 237
C++ Strings and the [ ] Operator 238
ST4 Working with C Strings 238
7.4 Dynamic Memory Allocation 240
CE3 Dangling Pointers 242
CE4 Memory Leaks 243
7.5 Arrays and Vectors of Pointers 243
7.6 PROBLEM SOLVING Draw a Picture 246
HT1 Working with Pointers 248
WE1 Producing a Mass Mailing 249
C&S Embedded Systems 250
7.7 Structures 250
Structured Types 250
Structure Assignment and Comparison 251
Functions and Structures 252
Arrays of Structures 252
Structures with Array Members 253
Nested Structures 253
7.8 Pointers and Structures 254
Pointers to Structures 254
Structures with Pointer Members 255
ST5 Smart Pointers 256

8 STREAMS 259

8.1 Reading and Writing Text Files 260
Opening a Stream 260
Reading from a File 261
Writing to a File 262
A File Processing Example 262
8.2 Reading Text Input 265
Reading Words 265
Reading Characters 266
Reading Lines 267
CE1 Mixing >> and getline Input 268
ST1 Stream Failure Checking 269
8.3 Writing Text Output 270
ST2 Unicode, UTF-8, and C++ Strings 272
8.4 Parsing and Formatting Strings 273
8.5 Command Line Arguments 274
C&S Encryption Algorithms 277
HT1 Processing Text Files 278
WE1 Looking for for Duplicates 281
8.6 Random Access and Binary Files 281
Random Access 281
Binary Files 282
Processing Image Files 282
C&S Databases and Privacy 286

9 CLASSES 289

9.1 Object-Oriented Programming 290
9.2 Implementing a Simple Class 292
9.3 Specifying the Public Interface of a Class 294
CE1 Forgetting a Semicolon 296
9.4 Designing the Data Representation 297
9.5 Member Functions 299
Implementing Member Functions 299
Implicit and Explicit Parameters 299
Calling a Member Function from a Member Function 301
PT1 All Data Members Should Be Private; Most Member Functions Should Be Public 303
PT2 const Correctness 303
9.6 Constructors 304
CE2 Trying to Call a Constructor 306
ST1 Overloading 306
ST2 Initializer Lists 307
ST3 Universal and Uniform Initialization Syntax 308
9.7 PROBLEM SOLVING Tracing Objects 308
HT1 Implementing a Class 310
WE1 Implementing a Bank Account Class 314
C&S Electronic Voting Machines 314
9.8 PROBLEM SOLVING Discovering Classes 315
PT3 Make Parallel Vectors into Vectors of Objects 317
9.9 Separate Compilation 318
9.10 Pointers to Objects 322
Dynamically Allocating Objects 322
The -> Operator 323
The this Pointer 324

9.11 PROBLEM SOLVING Patterns for Object Data 324
Keeping a Total 324
Counting Events 325
Collecting Values 326
Managing Properties of an Object 326
Modeling Objects with Distinct States 327
Describing the Position of an Object 328
C&S Open Source and Free Software 329

10 INHERITANCE 333

10.1 Inheritance Hierarchies 334
10.2 Implementing Derived Classes 338
CE1 Private Inheritance 341
CE2 Replicating Base-Class Members 341
PT1 Use a Single Class for Variation in Values, Inheritance for Variation in Behavior 342
ST1 Calling the Base-Class Constructor 342
10.3 Overriding Member Functions 343
CE3 Forgetting the Base-Class Name 345
10.4 Virtual Functions and Polymorphism 346
The Slicing Problem 346
Pointers to Base and Derived Classes 347
Virtual Functions 348
Polymorphism 349
PT2 Don't Use Type Tags 352
CE4 Slicing an Object 352
CE5 Failing to Override a Virtual Function 353
ST2 Virtual Self-Calls 354
HT1 Developing an Inheritance Hierarchy 354
WE1 Implementing an Employee Hierarchy for Payroll Processing 359
C&S Who Controls the Internet? 360

11 RECURSION 363

11.1 Triangle Numbers 364
CE1 Tracing Through Recursive Functions 367
CE2 Infinite Recursion 368
HT1 Thinking Recursively 369
WE1 Finding Files 372
11.2 Recursive Helper Functions 372
11.3 The Efficiency of Recursion 373
11.4 Permutations 377
11.5 Mutual Recursion 380
11.6 Backtracking 383
WE2 Towers of Hanoi 389
C&S The Limits of Computation 390

12 SORTING AND SEARCHING 393

12.1 Selection Sort 394
12.2 Profiling the Selection Sort Algorithm 397
12.3 Analyzing the Performance of the Selection Sort Algorithm 398
ST1 Oh, Omega, and Theta 399
ST2 Insertion Sort 400
12.4 Merge Sort 402
12.5 Analyzing the Merge Sort Algorithm 405
ST3 The Quicksort Algorithm 407
12.6 Searching 408
Linear Search 408
Binary Search 410
PT1 Library Functions for Sorting and Binary Search 412
ST4 Defining an Ordering for Sorting Objects 413
12.7 PROBLEM SOLVING Estimating the Running Time of an Algorithm 413
Linear Time 413
Quadratic Time 414
The Triangle Pattern 415
Logarithmic Time 417
WE1 Enhancing the Insertion Sort Algorithm 418
C&S The First Programmer 418

13 ADVANCED C++ 421

13.1 Operator Overloading 422
Operator Functions 422
Overloading Comparison Operators 425
Input and Output 425
Operator Members 426
ST1 Overloading Increment and Decrement Operators 427
ST2 Implicit Type Conversions 428
ST3 Returning References 429
WE1 A Fraction Class 430

13.2 Automatic Memory Management **430**
Constructors That Allocate Memory 430
Destructors 432
Overloading the Assignment Operator 433
Copy Constructors 437
PT1 Use Reference Parameters To Avoid Copies 441
CE1 Defining a Destructor Without the Other Two Functions of the “Big Three” 442
ST4 Virtual Destructors 443
ST5 Suppressing Automatic Generation of Memory Management Functions 443
ST6 Move Operations 444
ST7 Shared Pointers 445
WE2 Tracing Memory Management of Strings 446

13.3 Templates **446**
Function Templates 447
Class Templates 448
ST8 Non-Type Template Parameters 450

14 LINKED LISTS, STACKS, AND QUEUES 453

14.1 Using Linked Lists **454**

14.2 Implementing Linked Lists **459**
The Classes for Lists, Node, and Iterators 459
Implementing Iterators 460
Implementing Insertion and Removal 462
WE1 Implementing a Linked List Template 472

14.3 The Efficiency of List, Array, and Vector Operations **472**

14.4 Stacks and Queues **476**

14.5 Implementing Stacks and Queues **479**
Stacks as Linked Lists 479
Stacks as Arrays 482
Queues as Linked Lists 482
Queues as Circular Arrays 483

14.6 Stack and Queue Applications **484**
Balancing Parentheses 484
Evaluating Reverse Polish Expressions 485
Evaluating Algebraic Expressions 487
Backtracking 490
ST1 Reverse Polish Notation 492

15 SETS, MAPS, AND HASH TABLES 495

15.1 Sets **496**

15.2 Maps **499**
PT1 Use the auto Type for Iterators 503
ST1 Multisets and Multimaps 503
WE1 Word Frequency 504

15.3 Implementing a Hash Table **504**
Hash Codes 504
Hash Tables 505
Finding an Element 507
Adding and Removing Elements 508
Iterating over a Hash Table 508
ST2 Implementing Hash Functions 514
ST3 Open Addressing 516

16 TREE STRUCTURES 519

16.1 Basic Tree Concepts **520**

16.2 Binary Trees **524**
Binary Tree Examples 524
Balanced Trees 526
A Binary Tree Implementation 527
WE1 Building a Huffman Tree 528

16.3 Binary Search Trees **528**
The Binary Search Property 529
Insertion 530
Removal 532
Efficiency of the Operations 533

16.4 Tree Traversal **538**
Inorder Traversal 539
Preorder and Postorder Traversals 540
The Visitor Pattern 541
Depth-First and Breadth-First Search 542
Tree Iterators 543

16.5 Red-Black Trees **544**
Basic Properties of Red-Black Trees 544
Insertion 546
Removal 548
WE2 Implementing a Red-Black Tree 551

17 PRIORITY QUEUES AND HEAPS **553**

17.1 Priority Queues **554**
WE 1 Simulating a Queue of Waiting Customers 557

17.2 Heaps **557**

17.3 The Heapsort Algorithm **567**

APPENDIX A RESERVED WORD SUMMARY A-1

APPENDIX B OPERATOR SUMMARY A-3

APPENDIX C CHARACTER CODES A-5

APPENDIX D C++ LIBRARY SUMMARY A-8

APPENDIX E C++ LANGUAGE CODING GUIDELINES A-12

APPENDIX F NUMBER SYSTEMS AND BIT AND SHIFT OPERATIONS A-19

GLOSSARY **G-1**

INDEX **I-1**

CREDITS **C-1**

QUICK REFERENCE **C-2**

ALPHABETICAL LIST OF SYNTAX BOXES

Assignment 30

C++ Program 12
Class Definition 295
Class Template 449
Comparisons 67
Constructor with Base-Class Initializer 342
Copy Constructor 440

Defining an Array 181
Defining a Structure 251
Defining a Vector 213
Derived-Class Definition 340
Destructor Definition 433
Dynamic Memory Allocation 240

`for` Statement 106
Function Definition 145
Function Template 448

`if` Statement 61
Input Statement 44

Member Function Definition 301

Output Statement 13
Overloaded Assignment Operator 437
Overloaded Operator Definition 424

Pointer Syntax 226

Two-Dimensional Array Definition 207

Variable Definition 27

`while` Statement 97
Working with File Streams 262

Chapter	Common Errors		How Tos and Worked Examples	
1 Introduction	Omitting Semicolons	13	Describing an Algorithm with Pseudocode	19
	Misspelling Words	15	Writing an Algorithm for Tiling a Floor	21
2 Fundamental Data Types	Using Undefined Variables	33	Computing Travel Time	48
	Using Uninitialized Variables	33	Carrying out Computations	48
	Unintended Integer Division	39	Computing the Cost of Stamps	51
	Unbalanced Parentheses	40		
	Forgetting Header Files	40		
	Roundoff Errors	41		
3 Decisions	A Semicolon After the `if` Condition	63	Implementing an `if` Statement	70
	Confusing = and ==	68	Extracting the Middle	72
	Exact Comparison of Floating-Point Numbers	68		
	The Dangling `else` Problem	79		
	Combining Multiple Relational Operators	88		
	Confusing && and || Conditions	88		
4 Loops	Infinite Loops	100	Writing a Loop	123
	Don't Think "Are We There Yet?"	101	Credit Card Processing	126
	Off-by-One Errors	101	Manipulating the Pixels in an Image	129
5 Functions	Missing Return Value	149	Implementing a Function	151
			Generating Random Passwords	152
			Using a Debugger	152
			Calculating a Course Grade	163
			Thinking Recursively	173

Programming Tips		Special Topics		Computing & Society	
Backup Copies	10	Escape Sequences	13	Computers Are Everywhere Standards Organizations	5 7
Choose Descriptive Variable Names Do Not Use Magic Numbers Spaces in Expressions	33 34 42	Numeric Types in C++ Numeric Ranges and Precisions Defining Variables with `auto` Casts Combining Assignment and Arithmetic	34 35 35 42 42	The Pentium Floating-Point Bug International Alphabets and Unicode	43 55
Brace Layout Always Use Braces Tabs Avoid Duplication in Branches Compile with Zero Warnings Hand-Tracing Make a Schedule and Make Time for Unexpected Problems	63 64 64 65 69 79 84	The Conditional Operator Lexicographic Ordering of Strings The `switch` Statement Short-Circuit Evaluation of Boolean Operators De Morgan's Law	65 69 75 89 89	Dysfunctional Computerized Systems Artificial Intelligence	72 92
Use `for` Loops for Their Intended Purpose Only Choose Loop Bounds That Match Your Task Count Iterations Flowcharts for Loops	109 110 110 111	Clearing the Failure State The Loop-and-a-Half Problem and the `break` Statement Redirection of Input and Output	115 116 116	The First Bug Digital Piracy	102 138
Function Comments Do Not Modify Parameter Variables Keep Functions Short Tracing Functions Stubs Avoid Global Variables Prefer Return Values to Reference Parameters	146 148 161 161 162 165 169	Function Declarations Constant References	150 170	The Explosive Growth of Personal Computers	174

Chapter	Common Errors		How Tos and Worked Examples	
6 Arrays and Vectors	Bounds Errors	184	Working with Arrays	200
	Omitting the Column Size of a Two-Dimensional Array Parameter	212	Rolling the Dice	203
			A World Population Table	213
7 Pointers and Structures	Confusing Pointers with the Data to Which They Point	228	Working with Pointers	248
	Returning a Pointer to a Local Variable	234	Producing a Mass Mailing	249
	Dangling Pointers	242		
	Memory Leaks	243		
8 Streams	Mixing >> and getline Input	268	Processing Text Files	278
			Looking for for Duplicates	281
9 Classes	Forgetting a Semicolon	296	Implementing a Class	310
	Trying to Call a Constructor	306	Implementing a Bank Account Class	314
10 Inheritance	Private Inheritance	341	Developing an Inheritance Hierarchy	354
	Replicating Base-Class Members	341	Implementing an Employee Hierarchy for Payroll Processing	359
	Forgetting the Base-Class Name	345		
	Slicing an Object	352		
	Failing to Override a Virtual Function	353		

Programming Tips		Special Topics		Computing & Society	
Use Arrays for Sequences of Related Values	184	Sorting with the C++ Library	192	Computer Viruses	185
Prefer Vectors over Arrays	219	A Sorting Algorithm	192		
		Binary Search	193		
		Constant Array Parameters	198		
		The Range-Based `for` Loop	219		
Use a Separate Definition for Each Pointer Variable	229	Pointers and References	229	Embedded Systems	250
Program Clearly, Not Cleverly	234	Using a Pointer to Step Through an Array	233		
		Constant Pointers	235		
		Working with C Strings	238		
		Smart Pointers	256		
		Stream Failure Checking	269	Encryption Algorithms	277
		Unicode, UTF-8, and C++ Strings	272	Databases and Privacy	286
All Data Members Should Be Private; Most Member Functions Should Be Public	303	Overloading	306	Electronic Voting Machines	314
`const` Correctness	303	Initializer Lists	307	Open Source and Free Software	329
Make Parallel Vectors into Vectors of Objects	317	Universal and Uniform Initialization Syntax	308		
Use a Single Class for Variation in Values, Inheritance for Variation in Behavior	342	Calling the Base-Class Constructor	342	Who Controls the Internet?	360
		Virtual Self-Calls	354		

Chapter	Common Errors	How Tos and Worked Examples
11 Recursion	Tracing Through Recursive Functions 367 Infinite Recursion 368	Thinking Recursively 369 Finding Files 372 Towers of Hanoi 389
12 Sorting and Searching		Enhancing the Insertion Sort Algorithm 418
13 Advanced C++	Defining a Destructor Without the Other Two Functions of the “Big Three” 442	A Fraction Class 430 Tracing Memory Management of Strings 446
14 Linked Lists, Stacks, and Queues		Implementing a Linked List Template 472
15 Set, Maps, and Hash Tables		Word Frequency 504
16 Tree Structures		Building a Huffman Tree 528 Implementing a Red-Black Tree 551
17 Priority Queues and Heaps		Simulating a Queue of Waiting Customers 557

Programming Tips		Special Topics		Computing & Society	
				The Limits of Computation	390
Library Functions for Sorting and Binary Search	412	Oh, Omega, and Theta	399	The First Programmer	418
		Insertion Sort	400		
		The Quicksort Algorithm	407		
		Defining an Ordering for Sorting Objects	413		
Use Reference Parameters To Avoid Copies	441	Overloading Increment and Decrement Operators	427		
		Implicit Type Conversions	428		
		Returning References	429		
		Virtual Destructors	443		
		Suppressing Automatic Generation of Memory Management Functions	443		
		Move Operations	444		
		Shared Pointers	445		
		Non-Type Template Parameters	450		
		Reverse Polish Notation	492		
Use the auto Type for Iterators	503	Multisets and Multimaps	503		
		Implementing Hash Functions	514		
		Open Addressing	516		

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

1.1 WHAT IS PROGRAMMING? 2

1.2 THE ANATOMY OF A COMPUTER 3

C&S Computers Are Everywhere 5

1.3 MACHINE CODE AND PROGRAMMING LANGUAGES 5

C&S Standards Organizations 7

1.4 BECOMING FAMILIAR WITH YOUR PROGRAMMING ENVIRONMENT 7

PT 1 Backup Copies 10

1.5 ANALYZING YOUR FIRST PROGRAM 11

SYN C++ Program 12

SYN Output Statement 13

CE 1 Omitting Semicolons 13

ST 1 Escape Sequences 13

1.6 ERRORS 14

CE 2 Misspelling Words 15

1.7 PROBLEM SOLVING: ALGORITHM DESIGN 16

HT 1 Describing an Algorithm with Pseudocode 19

WE 1 Writing an Algorithm for Tiling a Floor 21

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.

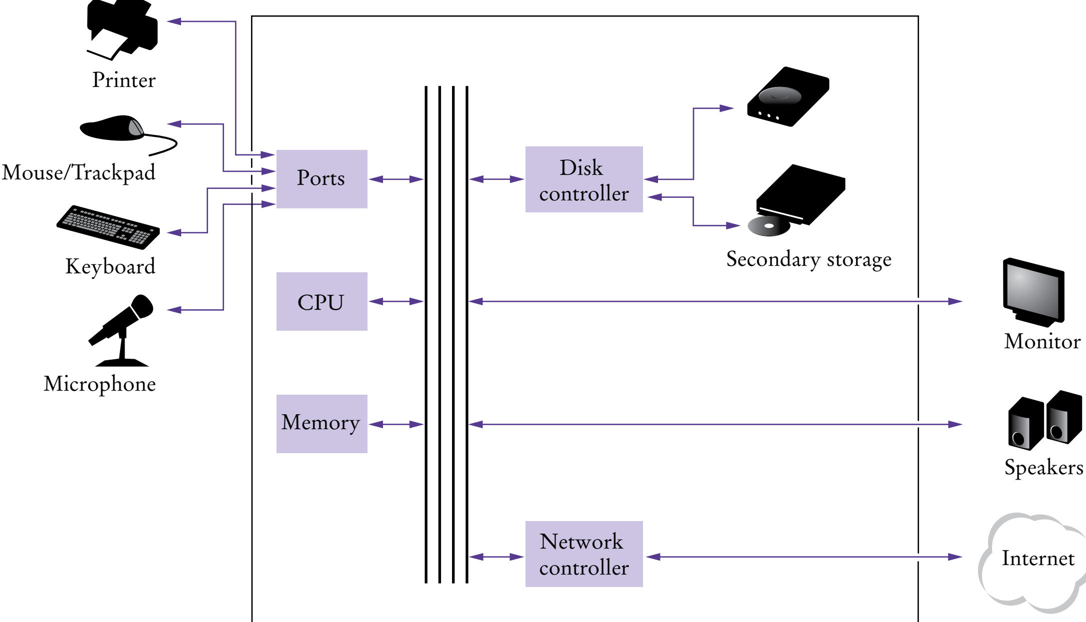


Figure 3 Schematic Design of a Personal Computer

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.

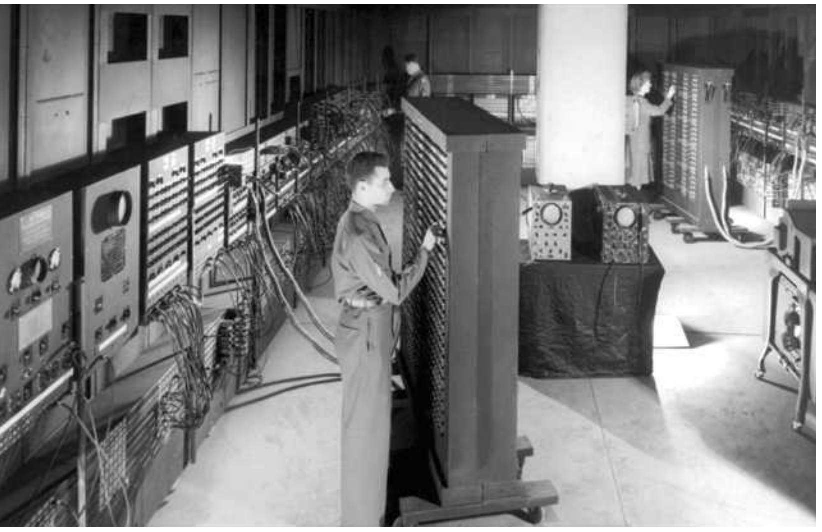
The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are today carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies nowadays are often 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

```
Terminal
File Edit View Terminal Help
~$ cd cs1/bookcode/ch01
~/cs1/bookcode/ch01$ g++ -o hello hello.cpp
~/cs1/bookcode/ch01$ ./hello
Hello, World!
~/cs1/bookcode/ch01$
```

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`.

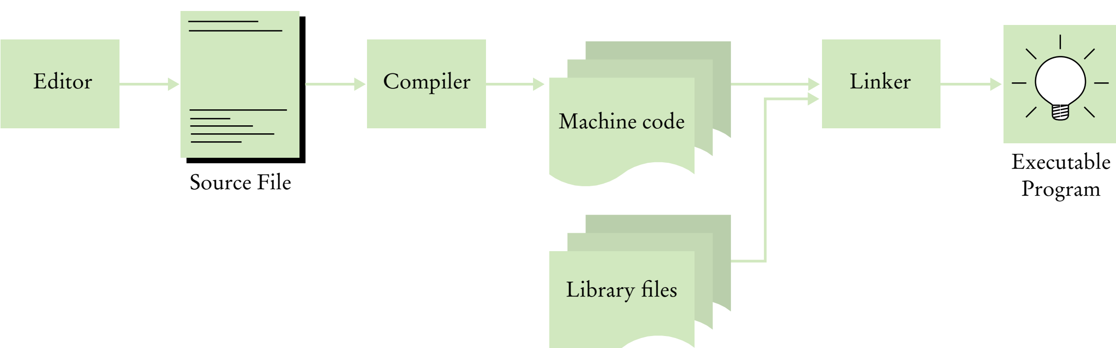


Figure 7 From Source Code to Executable Program