PRENTICE HALL

A Beginner's Guide That Makes You Feel **SMART**

C++ WITHOUT FEAR

THIRD EDITION

- **Learn** programming basics fast
- **Understand** with well-illustrated figures and examples
- **Practice** with games, exercises, and puzzles
- **Write** your first C++ program
- **Refer** to summaries, appendices, and C++14 notes

BRIAN OVERLAND

Updated for C++14!

C++ Without Fear

Third Edition

C++ Without Fear

Third Edition

A Beginner's Guide That Makes You Feel Smart

Brian Overland

PRENTICE HALL

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Cataloging-in-Publication Data

Overland, Brian, 1958– author.
C++ without fear : a beginner's guide that makes you feel smart / Brian Overland.—Third edition.
pages cm
Includes index.
ISBN 978-0-13-431430-3 (pbk. : alk. paper)—ISBN 0-13-431430-1 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.
QA76.73.C153O838 2016
005.13'3—dc23

2015033385

ISBN-13: 978-0-13-431430-3
ISBN-10: 0-13-431430-1
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Once more, for Colin

This page intentionally left blank

Contents

This page intentionally left blank

Preface

It's safe to say that C++ is the most important programming language in the world today.

This language is widely used to create commercial applications, ranging from operating systems to word processors. There was a time when big applications had to be written in machine code because there was little room in a computer for anything else. But that time has long passed. Gone are the days in which Bill Gates had to squeeze all of BASICA into 64K!

C++, the successor to the original C language, remains true to the goal of producing efficient programs while maximizing programmer productivity. It typically produces executable files second in compactness only to machine code, but it enables you to get far more done. More often than not, C++ is the language of choice for professionals.

But it sometimes gets a reputation for not being the easiest to learn. That's the reason for this book.

We'll Have Fun, Fun, Fun...

Anything worth learning is worth a certain amount of effort. But that doesn't mean it can't be fun, which brings us to this book.

I've been programming in C since the 1980s and in C++ since the 1990s, and have used them to create business- and systems-level applications. The pitfalls are familiar to me—things like uninitialized pointers and using one equal sign (=) instead of two (==) in an "if" condition. I can steer you past the errors that caused me hours of debugging and sweat, years ago.

But I also love logic problems and games. Learning a programming language doesn't have to be dull. In this book, we'll explore the Tower of Hanoi and the Monty Hall paradox, among other puzzles.

Learning to program is a lot more fun and easy when you can visualize concepts. This book makes heavy use of diagrams and illustrations.

Why C and C++?

There's nothing wrong with other programming languages. I was one of the first people in the world to write a line of code in Visual Basic (while a project lead at Microsoft), and I admire Python as a high-level scripting tool.

But with a little care, you'll find C++ almost as easy to learn. Its syntax is slightly more elaborate than Visual Basic's or Python's, but C++ has long been seen as a clean, flexible, elegant language, which was why its predecessor, C, caught on with so many professionals.

From the beginning, C was designed to provide shortcuts for certain lines of code you'll write over and over; for example, you can use "++n" to add 1 to a variable rather than "n = n + 1." The more you program in C or C++, the more you'll appreciate its shortcuts, its brevity, and its flexibility.

C++: How to "Think Objects"

A systems programmer named Dennis Ritchie created C as a tool to write operating systems. (He won the Turing Award in 1983.) He needed a language that was concise and flexible, and could manipulate low-level things like physical addresses when needed. The result, C, quickly became popular for other uses as well.

Later, Bjarne Stroustrup created C++, originally as a kind of "C with classes." It added the ability to do object orientation, a subject I'll devote considerable space to, starting in Chapter 10. Object orientation is a way of building a program around intelligent data types. A major goal of this edition is to showcase object orientation as a superior, more modular way to program, and how to "think objects."

Ultimately, C++ became far more than just "C with classes." Over the years, support was added for many new features, notably the Standard Template Library (STL). The STL is not difficult to learn and this book shows you how to use it to simplify a lot of programming work. As time goes on, this library is becoming more central to the work of C++ programmers.

Purpose of the Third Edition

The purpose of the third edition is simple: double down on the strengths of past editions and correct limitations.

In particular, this edition aims at being more fun and easier to use than ever. Most of the features of the previous edition remain, but the focus is more on the practical (and entertaining) use of C++ and object orientation, and not as much on esoteric features that see little use. For example, I assume you won't want to write your own **string** class, because all up-to-date C++ compilers have provided this feature for a long time now.

In this edition, I also put more stress on "correct" programming practices that have become standard, or nearly so, in the C++ community.

This edition of the book starts out by focusing on successful installation and usage of the Microsoft C++ compiler, Community Edition. If you have another C++ compiler you're happy with, fine. You can use that because the great majority of examples are written in generic C++. The first chapter, however, guides you through the process of using the Microsoft compiler with Visual Studio, if you've never used them before.

Other features of this edition include:

- **Coverage of new features in C++11 and C++14:** This edition brings you up to date on many of the newest features introduced since C++11, as well as introducing some brand-new features in C++14. It's assumed you have a C++ compiler at least as up to date as the Microsoft Community Edition, so I've purged this edition of the book of some out-of-date programming practices.
- **Even more puzzles, games, exercises, and figures:** These features, all a successful part of the second edition, show up even more frequently in this edition.
- **More focus on the "whys" and "how tos" of object orientation:** The class and object features of C++ have always held great promise. A major goal in revising this edition was to put greater emphasis on the practical value of classes and objects, and how to "think objects."
- **More on the STL:** The Standard Template Library, far from being difficult to learn, can make your life much easier and make you more productive as a programmer. This edition explores more of the STL.
- **Useful reference:** This edition maintains the quick-reference appendixes in the back of the book and even expands on them.

Where Do I Begin?

This edition assumes you know little or nothing about programming. If you can turn on a computer and use a menu system, keyboard, and mouse, you can begin with Chapter 1. I'll lead you through the process of installing and using Microsoft C++ Community version.

You should note that this version of C++ runs on recent versions of Microsoft Windows. If you use another system, such as a Macintosh, you'll need to download different tools. But the rules of generic C++ still apply, so you should be able to use most of the book without change.

Icons and More Icons

Building on the helpful icons in the first two editions, this edition provides even more—as signposts on the pages to help you find what you need. Be sure to look for these symbols because they call out sections to which you'll want to pay special attention.

How It Works

These sections take apart program examples and explain, line by line, how and why the examples work. You don't have to wade through long programming examples—I do that for you! (Or, rather, we go through the examples together.)

Exercises

After each full programming example, I provide at least one exercise, and usually several, that build on the example in some way. These encourage you to alter and extend the programming code you've just seen. This is the best way to learn. The answers to the exercises can be found on my Web site (brianoverland.com).

Optimizing

These sections develop an example by showing how it can be improved, made shorter, or made more efficient.

Variation

As with "Optimizing," these sections take the example in new directions, helping you learn by showing how the example can be varied or modified to do other things.

Keyword

This icon indicates a place where a keyword of the language is introduced and its usage clearly defined. These places in the text summarize how a given keyword can be used.

Key Syntax

The purpose of this icon is similar to "Keyword," but instead it calls attention to a piece of C++ syntax that does not involve a keyword.

Pseudocode

"Pseudocode" is a program, or a piece of a program, in English-language form. By reading a pseudocode summary, you understand what a program needs to do. It then remains only to translate English-language statements into C++ statements.

This book also uses “Interludes,” which are side topics that—while highly illuminating and entertaining—aren’t always crucial to the flow of the discussion. They can be read later.

Note ▶ Finally, some important ideas are sometimes called out with notes; these notes draw your attention to special issues and occasional “gotchas.” For example, one of the most common types of notes deals with version issues, pointing out that some features require a recent compiler:

C++14 ▶ This note is used to indicate sections that apply only to versions of C++ compliant with the more recent C++ specifications.

Anything Not Covered?

Nothing good in life is free—except maybe love, sunsets, breathing air, and puppies. (Well actually, puppies aren’t free. Not long ago I looked at some Great Dane puppies costing around $3,000 each. But they were cute.)

To focus more on topics important to the beginner-to-intermediate programmer, this edition has slightly reduced coverage of some of the more esoteric subjects. For example, operator overloading (a feature you might never get around to actually programming into your classes) is still present but moved to the last chapter.

Most other topics—even relatively advanced topics such as bit manipulation—are at least touched upon. But the focus is on fundamentals.

C++ is perhaps the largest programming language on earth, much as English has the largest vocabulary of natural languages. It’s a mistake for an introductory text to try to cover absolutely everything in a language of this size. But once you want to learn more about advanced topics in C++, there are plenty of resources.

Two of the books I’d recommend are Bjarne Stroustrup’s *The C++ Programming Language*, *Fourth Edition* (Addison-Wesley, 2013), which is by the original author of the C++ language. This is a huge, sophisticated, and exhaustive text, and I recommend it after you’ve learned to be comfortable writing C++ code. As for an easy-to-use reference, I recommend my own *C++ for the Impatient* (Addison-Wesley, 2013), which covers nearly the whole language and almost every part of the Standard Template Library.

Graphical-user-interface (GUI) programming is specific to this or that platform and is deserving of its own—or rather many—books. This book introduces you to the core C++ language, plus its libraries and templates, which are platform independent.

A Final Note: Have Fun!

There's nothing to fear about C++. There are a few potholes here and there, but I'm going to steer you around them. Occasionally, C++ can be a little harder on you if you're not careful or don't know what you're doing, but you'll be better off in the long run by being made to think about these issues.

C++ doesn't have to be intimidating. I hope you use the practical examples and find the puzzles and games entertaining. This is a book about learning and about taking a road to new knowledge, but more than that, it's about enjoying the ride.

Acknowledgments

This edition is largely the result of a conversation between editor Kim Boedigheimer and myself while we had tea in a shop next to Seattle's Pike Place Market. So I think of this book as being as much hers as mine. She brought in an editorial and production team that made life easy for me, including Kesel Wilson, Deborah Thompson, Chris Zahn, Susan Brown Zahn, and John Fuller.

I'm especially indebted to Leor Zolman (yes, that's "Leor"), who provided the single finest technical review I've ever seen. Also providing useful input were John R. Bennett, a software developer emeritus from Microsoft, and online author David Jack ("the logic junkie"), who suggested some useful diagrams.

This page intentionally left blank

About the Author

Brian Overland published his first article in a professional math journal at age 14.

After graduating from Yale, he began working on large commercial projects in C and Basic, including an irrigation-control system used all over the world. He also tutored students in math, computer programming, and writing, as well as lecturing to classes at Microsoft and at the community-college level. On the side, he found an outlet for his lifelong love of writing by publishing film and drama reviews in local newspapers. His qualifications as an author of technical books are nearly unique because they involve so much real programming and teaching experience, as well as writing.

In his 10 years at Microsoft, he was a tester, author, programmer, and manager. As a technical writer, he became an expert on advanced utilities, such as the linker and assembler, and was the "go-to" guy for writing about new technology. His biggest achievement was probably organizing the entire documentation set for Visual Basic 1.0 and having a leading role in teaching the "object-based" way of programming that was so new at the time. He was also a member of the Visual C++ 1.0 team.

Since then, he has been involved with the formation of new start-up companies (sometimes as CEO). He is currently working on a novel.

1 Start Using C++

Nothing succeeds like success. This chapter focuses on successfully installing and using the C++ compiler—the tool that translates C++ statements into an executable program (or *application*).

I'm going to assume at first that you're using Microsoft Visual Studio, Community Edition. This includes an excellent C++ compiler—it's powerful, fast, and has nearly all of the up-to-date features. However, the Microsoft compiler raises some special issues, and one of the purposes of this chapter is to acquaint you with those issues so you can successfully use C++.

If you're not using this compiler, skip ahead to the section, "If You're Not Using Microsoft."

I'll get into the more abstract aspects of C++ later, but first let's get that compiler installed.

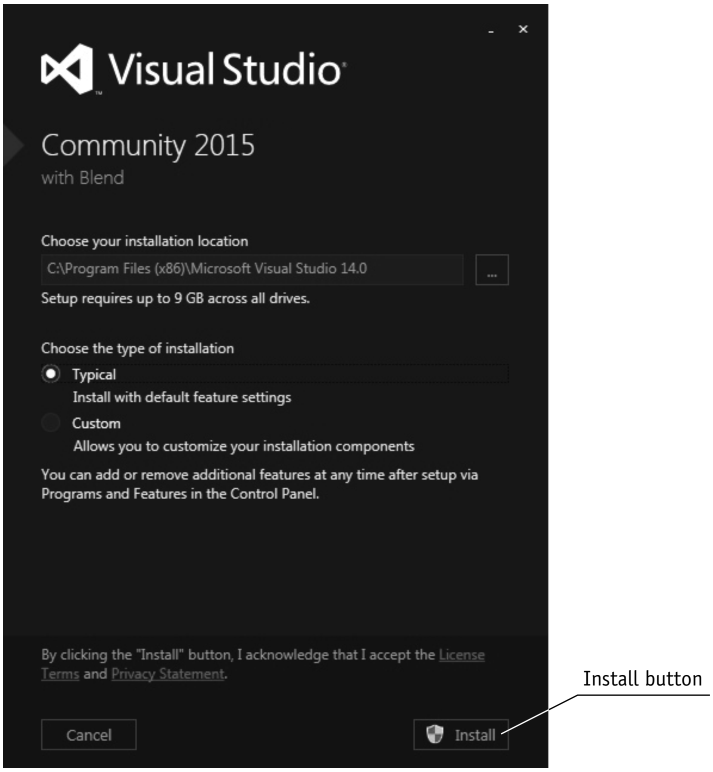
Install Microsoft Visual Studio

Even if you have an older version of Microsoft Visual Studio, you should consider updating to the current Community Edition, because it has nearly all the up-to-date features presented in this book. If you're already running Enterprise Edition, congratulations, but make sure it's up to date.

Here are the steps for installing Microsoft Visual Studio Community Edition:

1 Regardless of whether you're downloading from the Internet (you can use a search engine to look up "Visual Studio download") or, using the CD accompanying this book's Barnes & Noble Special Edition, get a copy of the file **vc_community** on your computer. If you're downloading, this will be found in your Download folder after using the site.

2 Double click the file **vc_community**. This launches the installation program. The following screen appears:


Visual Studio

Community 2015
with Blend

Choose your installation location
C:\Program Files (x86)\Microsoft Visual Studio 14.0
Setup requires up to 9 GB across all drives.

Choose the type of installation
(●) Typical
Install with default feature settings
( ) Custom
Allows you to customize your installation components

You can add or remove additional features at any time after setup via Programs and Features in the Control Panel.

By clicking the "Install" button, I acknowledge that I accept the License Terms and Privacy Statement.

Used with permission from Microsoft.

3 Click the Install button in the lower-right corner. Installation should begin right away.

4 If you're downloading from the Internet, be prepared for a long wait! If you're using the CD, installation will be many, many times faster.

If all goes well, Microsoft Visual Studio, which includes the Microsoft C++ compiler, should be installed on your computer, and you're ready to start programming. First, however, you need to create a project.

Create a Project with Microsoft

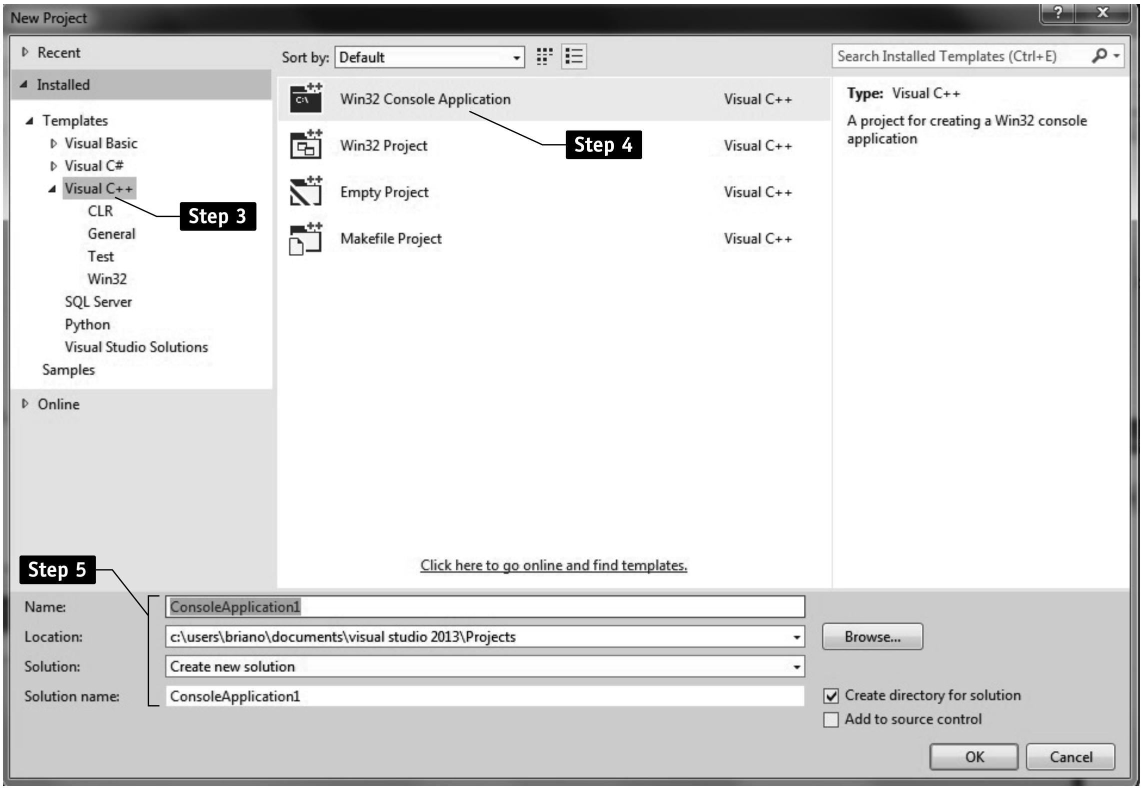
There are some files and settings you need for even the simplest program, but Visual Studio puts all the items you need into something called a *project*.

With Visual Studio, Microsoft makes things easy by providing everything you need when you create a project. Note that you will need to create a new project for each program you work on.

So let's create a project.

1 Launch Visual Studio. After you've installed it, you should find that Visual Studio is available on the Start menu (assuming you're running Windows). Visual Studio should then appear onscreen.

2 From the File menu (the first menu on the menu bar), choose the New Project command. The New Project window then appears.



Name: ConsoleApplication1
Location: c:\users\briano\documents\visual studio 2013\Projects
Solution: Create new solution
Solution name: ConsoleApplication1
☑ Create directory for solution
☐ Add to source control

Used with permission from Microsoft.

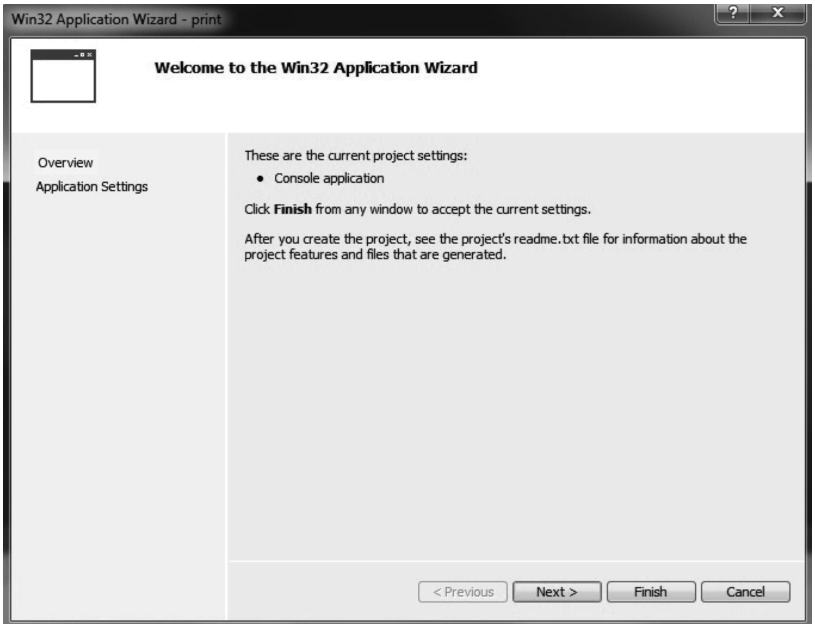
3 In the left pane, select Visual C++.

4 In the central windowpane, select Win32 Console Application.

5 There are four text boxes at the bottom of the window. You need only fill out one. In the Name box, type the name of the program: in this case, "print1." The Solution name box will automatically display the same text.

6 Click OK in the bottom right corner or just press ENTER.

The Application Wizard appears, asking if you're ready to go ahead. (Of course you are.) Click the Finish button at the bottom of the window.


Win32 Application Wizard - print

Welcome to the Win32 Application Wizard

These are the current project settings:

- Console application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

Used with permission from Microsoft

After you complete these steps, a new project is opened for you. The major area on the screen is a text window into which you can enter a program. Visual Studio provides a skeleton (or boilerplate) for a new program containing the following:

```
// print1.cpp: Defines the entry point...
//

#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
     return 0;
}
```

You're probably asking, what is all this stuff? The first thing to be aware of is that any line that begins with double slashes (//) is a *comment* and is ignored by the compiler.

Comments exist for the benefit of the programmer, presumably to help a human read and understand the program better, but the C++ compiler completely ignores comments. For now, we're going to ignore them as well.

So the part you care about is just:

```
#include "stdafx.h"

int _tmain(int arg, _TCHAR* argv[])
{
     return 0;
}
```

Writing a Program in Microsoft Visual Studio

Now—again, assuming you're using Microsoft Visual Studio—you're ready to write your first program. The previous section showed the skeleton (or boilerplate) that's already provided. Your task is to insert some new statements.

In the following example, I've added the new lines and placed them in bold—so you know exactly what to type:

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
     cout << "Never fear, C++ is here!";
     return 0;
}
```

For now, just leave **#include "stdafx.h"** and **t_main** alone, but add new statements where I've indicated. These lines are Microsoft specific, and I'll have more to say about them in the section "Compatibility Issue #1: stdafx.h." First, however, let's just run the program.

Running a Program in Visual Studio

Now you need to translate and run the program. In Visual Studio, all you do is press Ctrl+F5 or else choose the Start Without Debugging command from the Debug menu.

Visual Studio will say that the program is out of date and ask if you want to rebuild it. Say yes by clicking the Yes button.

Note You can also build and run the program by pressing F5, but the output of the program will "flash" and not stay on the screen. So use Ctrl+F5 instead.

If you received error messages, you probably have mistyped something. One of the intimidating aspects of C++, until you get used to it, is that even a single mistyped character can result in a series of "cascading errors." So, don't get upset, just check your spelling. In particular, check the following:

- The two C++ statements (and most lines of code you type in *will* be C++ statements), end with a semicolon (;), so be careful not to forget those semis.
- But make sure the **#include** directives do *not* end with semicolons(;).
- Case sensitivity absolutely matters in C++ (although spacing, for the most part, does not). Make sure you did not type any capital letters except for text enclosed in quotation marks.

After you're sure you've typed everything correctly, you can rebuild the program by pressing Ctrl+F5 again.

Compatibility Issue #1: stdafx.h

If you're like me, you'd prefer not to deal with compatibility issues but get right to programming. However, there are a couple of things you need to keep in mind to make sure you succeed with Microsoft Visual Studio.

In order to support something called "precompiled headers," Microsoft Visual Studio inserts the following line at the beginning of your programs. There's nothing wrong with this, unless you paste sample code over it and then wonder why nothing works.

```
#include "stdafx.h"
```

The problem is that other compilers will not work with this line of code, but programs built with Microsoft Visual Studio require it, unless you make the changes described in this section.

You can adopt one of several strategies to make sure your programs compile inside Microsoft Visual Studio.

- The easiest thing to do is to make sure this line of code is always the first line in any program created with Visual Studio. So, if you copy generic C++ code listings into a Visual Studio project, make sure you do not erase the directive **#include** "**stdafx.h**".
- If you want to compile generic C++ code (nothing Microsoft-specific), then, when creating a project, do not click the Finish button when the Application Wizard window appears. Instead, click Next. Then, in the Application Settings window, click the "Precompiled Headers" button to de-select it.
- After a project is created, you can still change settings by doing the following: First, from the Project menu, choose the Properties command (Alt + F7). Then, in the left pane, select Precompiled Headers. (You may first have to expand "Configuration Properties" and then expand "C/C++" by clicking on these words.) Finally, in the right pane, choose "Not Using Precompiled Headers" from the top drop-down list box.

With the last two options, Microsoft-specific lines such as **#include** "**stdafx.h**" still appear! However, after the Precompiled Headers option box is de-selected, the Microsoft-specific lines can be replaced with generic C++ code.

Also note that Visual Studio uses the following skeleton for the main function:

```
int _tmain(int arg, _TCHAR* argv[])
{

}
```

instead of:

```
int main()
{

}
```

Both of these work fine with Visual Studio, but if you keep the version that features the word **_tmain**, remember that it requires **#include stdafx.h** as well.

The items inside the parentheses, just after **_tmain**, support access to command-line arguments. But since this book does not address command-line arguments, you won't need them for the examples in this book. Just leave them as they are.

Compatibility Issue #2: Pausing the Screen

As stated earlier, if you build and run the program by pressing Ctrl+F5, your results should be satisfactory, but if you press F5, you'll get the problem of the program output flashing on the screen and disappearing.

If you're using Microsoft Visual Studio, the easiest solution is to just press Ctrl+F5 (Start Without Debugging) every time you build and run the program. However, not all compilers offer this option.

Another way to deal with the problem of output flashing on the screen and disappearing is to add the following line of code, just above "return 0;":

```
system("PAUSE");
```

When this statement is executed, it has roughly the same effect as pressing Ctrl+F5. It causes the program to pause and print "Press any key to continue."

The problem with this statement is that it is system specific. It does what you want in Windows, but it might not work on another platform. Only put this statement in if you're reasonably sure you want your program to run just on Windows-based systems.

If you're working on another platform, you'll need to look for another solution. Check your compiler documentation for more information.

Now, if you're using Microsoft Visual Studio, skip ahead to Exercise 1.1.

If You're Not Using Microsoft

If you're not using Microsoft Visual Studio as your compiler, most of the steps described in the previous sections won't apply. If any documentation comes with your compiler, make sure you read it in case, like Microsoft Visual Studio, it has idiosyncrasies of its own.

With compilers other than Visual Studio, do not put in the line **#include "stdafx.h"** and make sure you use the simpler program skeleton:

```
int main() {

}
```

Beginning with the next section, this book is going to adhere fairly closely to generic C++, which has nothing that is platform or vendor specific. But in this chapter, I'll keep reminding you of what you need to do for Visual Studio.

Example 1.1. *Print a Message*

Here is the program introduced earlier, written in generic C++ (except for the comment, which indicates what you have to do to run it in Visual Studio).

print1.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     cout << "Never fear, C++ is here! ";
     return 0;
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

Also remember that if and only if you are working with Microsoft Visual Studio, then, at the beginning of the program, you must leave in the following line:

```
#include "stdafx.h"
```

After entering the program, build and run it (from within Microsoft Visual Studio, press Ctrl+F5). Here's what the program prints when correctly entered and run:

```
Never fear, C++ is here!
```

However, this output may be run together with the message "Press any key to continue." In the upcoming sections, we're going to correct that.

How It Works

Believe it or not, this simple program has only one real statement. You can think of the rest as "boilerplate" for now—stuff you have to include but can safely ignore. (If you're interested in the details, the upcoming "Interlude" discusses the **#include** directive.)

Except for the one line in italics, the lines below are "boilerplate": these are items that always have to be present, even if the program doesn't do anything. For now, don't worry about why these lines are necessary; their usage will become clearer as you progress with this book. In between the braces ({}), you insert the actual lines of the program—which in this case consist of just one important statement.

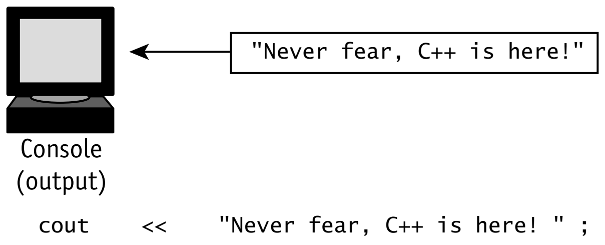
```
#include <iostream>
using namespace std;

int main()
{
     Enter_your_statements_here!
     return 0;
}
```

This program has one only real statement. Don't forget the semicolon (;) at the end!

```
cout << "Never fear, C++ is here!";
```

What is **cout**? This is an *object*—that's a concept I'll discuss a lot more in the second half of the book. In the meantime, all you have to know is that **cout** stands for "console output." In other words, it represents the computer screen. When you send something to the screen, it gets printed, just as you'd expect.

In C++, you print output by using **cout** and a leftward stream operator (<<) that shows the flow of data from a value (in this case, the text string "Never fear, C++ is here!") to the console. You can visualize it this way:



Don't forget the semicolon (;). Every C++ statement must end with a semicolon, with few exceptions.

For technical reasons, **cout** must always appear on the left side of the line of code whenever it's used. Data in this case flows to the left. Use the leftward "arrows," which are actually a pair of less-than signs (<<).

The following table shows other simple uses of **cout**:

STATEMENT	ACTION
`cout << "Do you C++?";`	Prints the words "Do you C++?"
`cout << "I think,";`	Prints the words "I think,"
`cout << "Therefore I program.";`	Prints the words "Therefore I program."

EXERCISES

Exercise 1.1.1. Write a program that prints the message "Get with the program!" If you want, you can work on the same source file used for the featured example and alter it as needed. (Hint: Alter only the text inside the quotation marks; otherwise, reuse all the same programming code.)

Exercise 1.1.2. Write a program that prints your own name.

Exercise 1.1.3. Write a program that prints "Do you C++?"

Interlude

What about the #include and using?

I said that the fifth line of the program is the first "real" statement of the program. I glossed over the first line:

```
#include <iostream>
```

This is an example of a C++ *preprocessor directive*, a general instruction to the C++ compiler. A directive of the form

```
#include <filename>
```

loads declarations and definitions that support part of the C++ standard library. Without this directive, you couldn't use **cout**.

If you've used older versions of C++ and C, you may wonder why no specific file (such as an .h file) is named. The filename iostream is a *virtual include file*, which has information in a precompiled form.

If you're new to C++, just remember you have to use **#include** to turn on support for specific parts of the C++ standard library. Later, when we start using math functions such as **sqrt** (square root), you'll need to switch on support for the math library:

```
#include <cmath>
```

▼ *continued on next page*

Interlude

▼ continued

Is this extra work? A little, yes. Include files originated because of a distinction between the C language and the standard runtime library. (Professional C/C++ programmers sometimes avoid the standard library and use their own.) Library functions and objects—although they are indispensable to beginners—are treated just like user-defined functions, which means (as you'll learn in Chapter 4) that they have to be declared. That's what include files do.

You also need to put in a **using** statement. This enables you to refer directly to objects such as **std::cout**. Without this statement, you'd have to print messages this way:

```
std::cout << "Never fear, C++ is here!";
```

We're going to be using **cout** (and its cousin, **cin**) quite a lot, so for now it's easier just to put a **using** statement at the beginning of every program.

Advancing to the Next Print Line

With C++, text sent to the screen does not automatically advance to the next physical line. You have to print a *newline* character to do that. (Exception: If you never print a newline, the text may automatically "wrap" when the current physical line fills up, but this produces an ugly result.)

The easiest way to print a newline is to use the predefined constant **endl**. For example:

```
cout << "Never fear, C++ is here!" << endl;
```

Note ▶ The endl name is short for "end line"; it is therefore spelled "end ELL," not "end ONE." Also note that **endl** is actually **std::endl**, but the **using** statement saves you from having to type **std::**.

Another way to print a newline is to insert the characters \n. This is an escape sequence, which C++ interprets as having a special meaning rather than interpreting it literally. The following statement has the same effect as the previous example:

```
cout << "Never fear, C++ is here!\n";
```

Example 1.2. *Print Multiple Lines*

The program in this section prints messages across several lines. If you're following along and entering the programs, remember once again to use uppercase and lowercase letters exactly as shown—although you can change the capitalization of the text inside quotation marks and the program will still run.

If you're working with Visual Studio, the only lines you should add are the ones shown here in bold. Leave **#include stdafx.h** and **_tmain** alone. If you're working with another compiler, the code should look as follows, minus the comments (//).

print2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     cout << "I am Blaxxon," << endl;
     cout << "the godlike computer." << endl;
     cout << "Fear me!" << endl;
     return 0;
}
```

Remember that exact spacing does not matter, but case-sensitivity does.

The resulting program, if you're working with Visual Studio, should be as follows. The lines in bold are what you need to add to the code Visual Studio provides for you.

```
#include "stdafx.h"

#include <iostream>
using namespace std;

int _tmain(int arg, _TCHAR* argv[])
{
     cout << "I am Blaxxon," << endl;
     cout << "the godlike computer." << endl;
```

```
        cout << "Fear me!" << endl;
        return 0;
    }
```

After entering the program, compile and run it. Here's what the program prints when correctly entered and run:

```
I am Blaxxon,
the godlike computer.
Fear me!
```

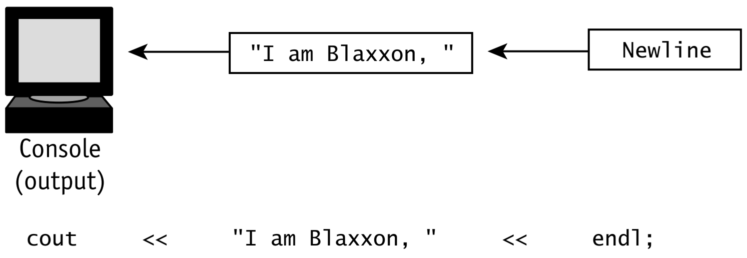
How It Works

This example is similar to the first one I introduced. The main difference is this example uses newline characters. If these characters were omitted, the program would print

```
I am Blaxxon, the godlike computer.Fear me!
```

which is not what we wanted.

Conceptually, here's how the statements in the program work:



You can print any number of separate items this way, though again, they won't advance to the next physical line without a newline character (**endl**). You could send several items to the console with one statement

```
cout << "This is a " << "nice " << "C++ program.";
```

which prints the following when run:

```
This is a nice C++ program.
```

Or, you can embed a newline, like this

```
cout << "This is a" << endl << "C++ program.";
```

which prints the following:

```
This is a
C++ program.
```

The example, like the previous one, returns a value. "Returning a value" is the process of sending back a signal—in this case to the operating system or development environment.

You return a value by using the **return** statement:

```
return 0;
```

The return value of **main** is a code sent to the operating system, in which 0 indicates success. The examples in this book return 0, but they could return an error code sometimes (–1 for example) if you found that to be useful. However, I would ignore that for now.

EXERCISES

Exercise 1.2.1. Remove the newlines from the example in this section, but put in extra spaces so that none of the words are crammed together. (Hint: Remember that C++ doesn't automatically insert a space between output strings.) The resulting output should look like this:

```
I am Blaxxon, the godlike computer. Fear me!
```

Exercise 1.2.2. Alter the example so that it prints a blank line between each two lines of output—in other words, make the results double-spaced rather than single-spaced. (Hint: Print *two* newline characters after each text string.)

Exercise 1.2.3. Alter the example so that it prints two blank lines between each of the lines of output.

Interlude

What Is a String?

From the beginning, I've made use of text inside of quotes, as in this statement:

```
cout << "Never fear, C++ is here!";
```

Everything outside of the quotes is part of C++ syntax. What's inside the quotes is data.

▼ continued on next page

Interlude

▼ *continued*

In actuality, all the data stored on a computer is numeric, but depending on how data is used, it can be interpreted as a string of printable characters. That's the case here.

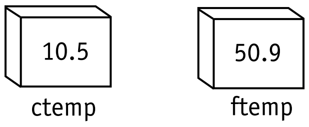
You may have heard of "ASCII code." That's what kind of data "Never fear, C++ is here!" is in this example. The characters "N", "e", "v", "e", "r", and so on, are stored in individual bytes, each of which is a numeric code corresponding to a printable character.

I'll talk a lot more about this kind of data in Chapter 8. The important thing to keep in mind is that text enclosed in quotes is considered raw data, as opposed to a command. This kind of data is considered a string of text or, more commonly, just a *string*.

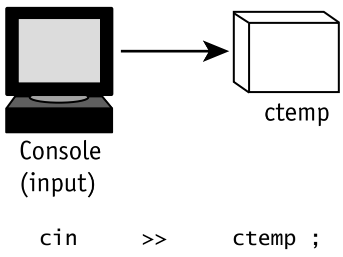
Storing Data: C++ Variables

If all you could do was print messages, C++ wouldn't be useful. The fundamental purpose of nearly any computer program is usually to get data from somewhere—such as end-user input—and then do something interesting with it.

Such operations require *variables*. These are locations into which you can place data. You can think of variables as magic boxes that hold values. As the program proceeds, it can read, write, or alter these values as needed. The upcoming example uses variables named ctemp and ftemp to hold Celsius and Fahrenheit values, respectively.



How are values put into variables? One way is through console input. In C++, you can input values by using the **cin** object, representing (appropriately enough) console input. With **cin**, you use a stream operator showing data flowing to the right (>>):



Here's what happens in response to this statement. (The actual process is a little more complicated, but don't worry about that for now.)

1 The program suspends running and waits for the user to enter a number.

2 The user types a number and presses ENTER.

3 The number is accepted and placed in the variable ctemp (in this case).

4 The program resumes running.

So, if you think about it, a lot happens in response to this statement:

```
cin >> ctemp;
```

But before you can use a variable in C++, you must declare it. This is an absolute rule and it makes C++ different from Basic, which is sloppy in this regard and doesn't require declaration (but generations of Basic programmers have banged their heads against their terminals as they discovered errors cropping up as a result of Basic's laxness about variables).

This is important enough to justify restating, so I'll make it a cardinal rule:

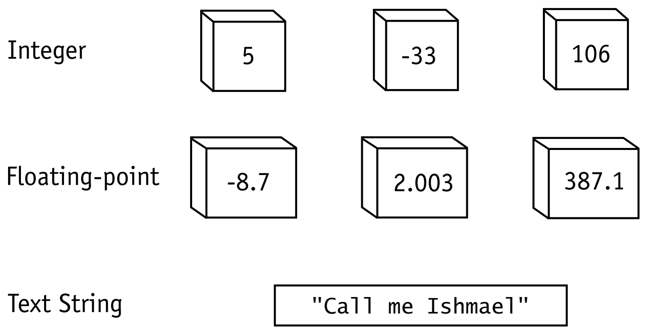
* **In C++, you must declare a variable before using it.**

To declare a variable, you first have to know what *data type* to use. This is a critical concept in C++ as in most other languages.

Introduction to Data Types

A variable is something you can think of as a magic box into which you can place information—or, rather, *data*. But what kind of data?

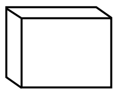
All data on a computer is ultimately numeric, but it is organized into one of three basic formats: integer, floating-point, and text string.



There are several differences between floating-point and integer format. But the main rule is simple:

If you need to store numbers with fractional portions, use a floating-point variable; otherwise, use integer types.

The principal floating-point data type in C++ is **double**. This may seem like a strange name, but it stands for "double-precision floating point." There is also a single-precision type (**float**), but its use is relatively infrequent. When you need the ability to retain fractional portions, you'll get better results—and fewer error messages—if you stick to **double**.



A **double** declaration has the following syntax. Note that this statement is terminated with a semicolon (;), just as most kinds of statements are.

```
double variable_name;
```

You can also use a **double** declaration to create a series of variables:

```
double variable_name1, variable_name2, ...;
```

For example, this statement creates a **double** variable named aFloat:

```
double  aFloat;
```

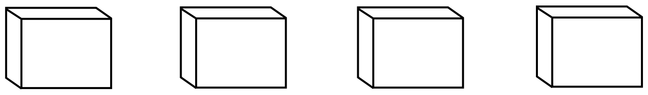
This statement creates a variable of type **double**.
The next statement declares four **double** variables named b, c, d, and amount:

```
double  b, c, d, amount;
```

The effect of this statement is equivalent to the following:

```
double  b;
double  c;
double  d;
double  amount;
```

The result of these declarations is to create four variables of type **double**.



An important rule of good programming style is that variables should usually be initialized, which means giving them a value as soon as you declare them. The declarations just shown should really be:

```
double  b  = 0.0;
double  c  = 0.0;
double  d  = 0.0;
double  amount = 0.0;
```

Starting in the next chapter, I'll have a lot more to say about issues such as data types and initialization. But for the next program, I'll keep the code simple. We'll worry about initialization in Chapter 2 onward.

Interlude

Why Double Precision, Not Single?

Double precision is like single precision, except better. Double precision supports a greater range of values, with better accuracy: It uses 8 bytes rather than 4.

C++ converts all data to double precision when doing calculations, which makes sense given that today's PCs include 8-byte co-processors. C++ also stores floating-point constants in double precision unless you specify otherwise (for example, by using the notation 12.5F instead of 12.5).

Double precision has one drawback: it requires more space. This is a factor only when you have large amounts of floating-point values to be stored in a disk file. Then, and only then, should you consider using the single-precision type, **float**.

Example 1.3. *Convert Temperatures*

Every time I go to Canada, I have to convert Celsius temperatures to Fahrenheit in my head. If I had a handheld computer, it would be nice to tell it to do this conversion for me; computers are good at that sort of thing.

Here's the conversion formula. The asterisk (*), when used to combine two values, means "multiply by."

```
Fahrenheit = (Celsius * 1.8) + 32
```

Now, a useful program will take *any* value input for Celsius and then convert it. This requires the use of some new features:

- Getting user input
- Storing the value input in a variable

Here is the complete program. Create a new project called "convert." Then enter the new program, and compile and run (press Ctrl + F5 if you're using Microsoft).

convert.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
     double  ctemp, ftemp;

     cout << "Input a Celsius temp and press ENTER: ";
     cin >> ctemp;
     ftemp = (ctemp * 1.8) + 32;
     cout << "Fahrenheit temp is: " << ftemp;
     return 0;
}
```

Remember, yet again (!), that if and only if you're working with Microsoft Visual Studio, you must leave the following line in at the beginning of the program:

```
#include "stdafx.h"
```

Programs are easier to follow when you add comments, which in C++ are notated by double slashes (//). Comments are ignored by the compiler (they have no effect on program behavior), but they are useful for humans. Here is the more heavily commented version:

convert2.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

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

convert2.cpp, cont.

```
int main()
{

     double ctemp;   // Celsius temperature
     double ftemp;   // Fahrenheit temperature

     // Get value of ctemp (Celsius temp).

     cout << "Input a Celsius temp and press ENTER: ";
     cin >> ctemp;


     // Calculate ftemp (Fahrenheit temp) and output.

     ftemp = (ctemp * 1.8) + 32;
     cout << "Fahrenheit temp is: " << ftemp << endl;

     return 0;
}
```

This commented version, although it's easier for humans to read, takes more work to enter. While following the examples in this book, you can always omit the comments or choose to add them later. Remember this cardinal rule for comments:

C++ code beginning with double slashes (//) is a comment and is ignored by the C++ compiler to the end of the line.

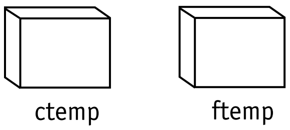
Using comments is always optional, although it is a good idea, especially if any humans (including you) are going to ever look at the C++ code.

How It Works

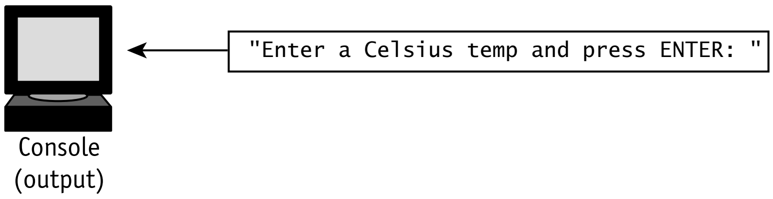
The first statement inside **main** declares variables of type **double**, ctemp and ftemp, which store Celsius temperature and Fahrenheit temperature, respectively.

```
double  ctemp, ftemp;
```

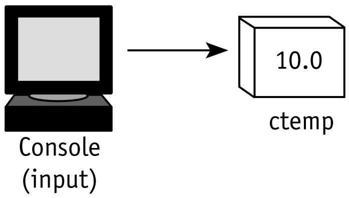
This gives us two locations at which we can store numbers. Because they have type **double**, they can contain fractional portions. Remember that **double** stands for "double-precision floating point."



The next two statements prompt the user and then store input in the variable ctemp. Assume that the user types 10. Then the numeric value 10.0 is put into ctemp.



```
cout    <<    "Enter a Celsius temp and press ENTER: " ;
```



```
cin     >>      ctemp;
```

In general, you can use similar statements in your own programs to print a prompting message and then store the input. The prompt is very helpful because otherwise the user may not know when he or she is supposed to do something.

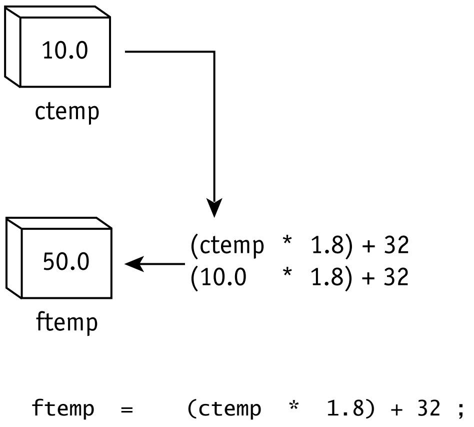
Note ▶ Although the number entered in this case was 10, it is stored as 10.0. In purely mathematical terms, 10 and 10.0 are equivalent, but in C++ terms, the notation 10.0 indicates that the value is stored in floating-point format rather than integer format. This turns out to have important consequences.

The next statement performs the actual conversion, using the value stored in ctemp to calculate the value of ftemp:

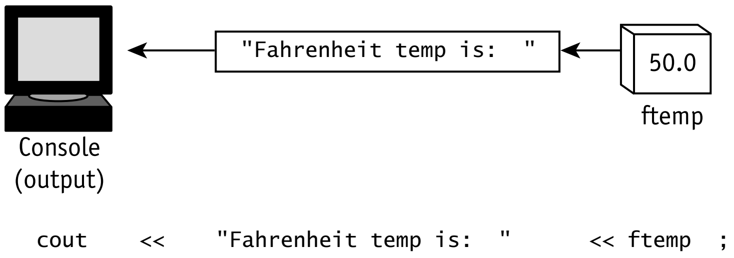
```
ftemp = (ctemp * 1.8) + 32;
```

This statement features an *assignment*: the value on the right side of the equal sign (=) is evaluated and then copied to the variable on the left side. This is one of the most common operations in C++.

Again, assuming that the user input 10, this is how data would flow in the program:



Finally, the program prints the result—in this case, 50.



Optimizing the Program

If you look at the previous example carefully, you might ask yourself, was it really necessary to declare two variables instead of one?

Actually, it wasn't. Welcome to the task of optimization. The following version improves on the first version of the program by getting rid of ftemp and combining the conversion and output steps:

convert3.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double  ctemp;   // Celsius temperature

    // Prompt and input value of ctemp.

    cout << "Input a Celsius temp and press ENTER: ";
    cin >> ctemp;


    // Convert ctemp and output results.

    cout << "Fahr. temp is: " << (ctemp * 1.8) + 32;
    cout << endl;

    return 0;
}
```

Do you detect a pattern by now? With the simplest programs, the pattern is usually as follows:

1. Declare variables.
2. Get input from the user (after printing a prompt).
3. Perform calculations and output results.

For example, the next program does something different but should look familiar. This program prompts for a number and then prints the square. The statements are similar to those in the previous example but use a different variable (x) and a different calculation.

square.cpp

```
// If you're using Microsoft V.S. leave in this line:
// #include "stdafx.h"

#include <iostream>
using namespace std;

int main()
{
    double  x = 0.0;

    // Prompt and input value of x.

    cout << "Input a number and press ENTER: ";
    cin >> x;


    // Calculate and output the square.

    cout << "The square is: " << x * x << endl;
    return 0;
}
```

EXERCISES

Exercise 1.3.1. Rewrite the example so it performs the reverse conversion: Input a value into ftemp (Fahrenheit) and convert to ctemp (Celsius). Then print the results. (Hint: The reverse conversion formula is ctemp = (ftemp – 32) / 1.8.)

Exercise 1.3.2. Write the Fahrenheit-to-Celsius program using only one variable, ftemp. This is an optimization of Exercise 1.3.1.

Exercise 1.3.3. Write a program that inputs a value into a variable x and outputs the cube (x * x * x). Make sure the output statement uses the word *cube* rather than *square*.

Exercise 1.3.4. Rewrite the example square.cpp using the variable name num rather than x. Make sure you change the name everywhere “x” appears.

A Word about Variable Names and Keywords

This chapter has featured the variables ctemp, ftemp, and n. Exercise 1.3.4 suggested that you could replace "x" with "num," as long as you do the substitution consistently throughout the program. So "num" is a valid name for a variable as well.

There is an endless variety of variable names I could have used instead. I could, for example, give some variables the names killerRobot or GovernorOfCalifornia.

What variable names are permitted, and what ones are not? You can use any name you want, as long as you follow these rules:

- The first character should be a letter. It cannot be a number. The first character can be an underscore (_), but the C++ library uses that naming convention internally, so it's best to avoid starting a name that way.
- The rest of the name can be a letter, a number, or an underscore (_).
- You must avoid words that already have a special, predefined meaning in C++, such as the keywords.

It isn't necessary to sit down and memorize all the C++ keywords. You need to know only that if you try using a name that conflicts with one of the C++ keywords, the compiler will respond with an error message. In that case, try a different name.

EXERCISE

Exercise 1.3.5. In the following list, which of the words are legal variable names in C++, and which are not? Review the rules just mentioned as needed.

x1

EvilDarkness

PennslyvaniaAve1600

1600PennsylvaniaAve

Bobby_the_Robot

Bobby+the+Robot

whatThe???

amount

count2

count2five

5count

main

main2

Chapter 1 *Summary*

Here are the main points of Chapter 1:

- Creating a program begins with writing C++ source code. This consists of C++ statements, which bear some resemblance to English. (Machine code, by contrast, is completely incomprehensible unless you look up the meaning of each combination of 1s and 0s.) Before the program can be run, it must be translated into machine code, which is all the computer really understands.
- The process of translating C++ statements into machine code is called *compiling*.
- After compiling, the program also has to be linked to standard functions stored in the C++ library. This process is called *linking*. After this step is successfully completed, you have an executable program.
- If you have a development environment, the process of compiling and linking a program (*building*) is automated so you need only press a function key. With Microsoft Visual Studio, press Ctrl+F5 to build programs.
- If you're working with Microsoft Visual Studio, make sure you leave **#include "stdafx"** at the beginning of every program. If you start a project by going through the New Project command, the environment will always put this in for you. Just make sure you don't delete **#include "stdafx"** when pasting code into the environment.

```
#include "stdafx.h"
```

- Simple C++ programs have the following general form:

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

int main()
{
        Enter_your_statements_here!
        return 0;
    }
```

- To print output, use the **cout** object. For example:

```
    cout << "Never fear, C++ is here!";
```

- To print output and advance to the next line, use the **cout** object and send a newline character (**endl**). For example:

```
    cout << "Never fear, C++ is here!" << endl;
```

- Most C++ statements are terminated by a semicolon (;). Directives—lines beginning with a pound sign (#)—are a major exception.
- Double slashes (//) indicate a comment; all text to the end of the line is ignored by the compiler itself. But comments can be read by humans who have to maintain the program.
- Before using a variable, you must declare it. For example:

```
    double  x;    // Declare x as a floating-pt variable.
```

- Variables that may store a fractional portion should have type **double**. This stands for "double-precision floating point." The single-precision type (**float**) should be used only when storing large amounts of floating-point data on disk.
- To get keyboard input into a variable, you can use the **cin** object. For example:

```
    cin >> x;
```

- You can also put data into a variable by using assignment (=). This operation evaluates the expression on the right side of the equal sign (=) and places the value in the variable on the left side. For example:

```
    x = y * 2;  // Multiply y times 2, place result in x.
```

2 Decisions, Decisions

If you've worked through Chapter 1, you've already begun to do real programming: getting input, crunching numbers, and printing output. But to really do anything interesting, you have to give the program the ability to make decisions. IF this, THEN do that, and so on.

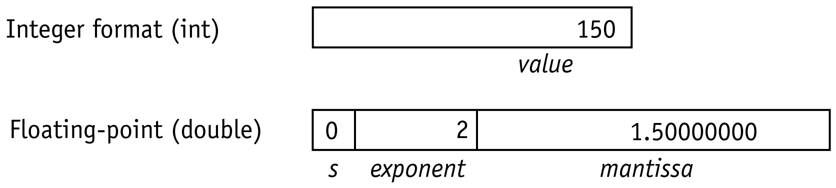
Computers don't really make the decisions the way humans do (not without Artificial Intelligence, a topic I discuss on page 31). Like everything else in a program, decisions must be absolutely clear and precise, and depend on the result of comparing two numbers. And yet, upon such simple decisions, it's possible to build complex, interesting, and rich behavior.

But First, a Few Words about Data Types

You can think of a variable as a magic box or a bucket to hold data. But it can hold only so much information—information being a precious commodity. A variable is not infinite.

Chapter 1 developed examples using floating-point data. This chapter uses integer data, which (unlike floating point) cannot hold fractional portions. You might think that you should just use **double**, the floating-point type, for all situations. But it's inefficient to use **double** when you don't really need it.

Under the covers, integer and floating-point formats look nothing alike. Here's how a value—150—is stored in different formats. I've made a number of simplifying assumptions here: for example, floating-point format uses binary, not decimal, representation (and integer format usually uses something called *two's complement*, which I describe in Appendix B).



The range of type **double** is much greater than integer; it can also store fractional portions of a number. But its use is more taxing on computer resources. Furthermore, its precision for very high integers is limited, and rounding errors can occur.

So, use the right data type for the job. It is definitely better to use the integer format when you're working with whole numbers only.

Here's the syntax for declaring an **int** (integer) variable. You can declare a variable with or without giving it an initial value, although good programming practice strongly favors initialization. If you fail to initialize a variable and it's local to a function—a concept to be discussed later—it will be initialized with "garbage," that is, a meaningless random value.

```
int variable_name;

int variable_name = initial_value;
```

You can also declare multiple **int** variables together:

```
int variable1, variable2, variable3,...;
```

Each variable (*variable1*, *variable2*, and so on) can be initialized, whether or not the others are (but remember that initialization is always a good idea). For example:

```
int a = 0, b = 1, c, d, e, f, g = 20;
```

You can assign values between integer and floating-point variables, but if you assign a floating-point value to an integer, the compiler complains about loss of data.

```
int n = 5;
double x = n;  // Ok: convert 5 to floating-pt
n = 3.7;       // Warning: convert from double to int
n = 3.0;       // This also gets a warning.
```

If you assign 3.7 to n, the fractional portion, 0.7, is dropped, so that the value 3 is stored in the variable n. But assigning 3.0 is also problematic, because the presence of a decimal point (.) causes a value to be stored in floating-point (**double**) format.

C++14 ▶ Beginning with C++11, the specification supports the **long long int** type, which has a comparatively immense large range. I'll have more to say about **long long int** in Chapter 10, "Classes and Objects." (Note: Some compilers have already been supporting this type for several years.) Most implementations that support this type use 64 bits rather than the default 32-bit integer size.

Decision Making in Programs

Remember, a computer can carry out only those instructions that are clear and precise. The computer always does exactly what you say. It follows instructions even if they are absurd; it has no judgment with which to question anything. Once again, this is one of the cardinal rules of programming—maybe *the* cardinal rule:

* **A computer can carry out only those instructions that are absolutely clear.**

A computer has no such thing as discretion or judgment. It can follow only those rules that are mathematically precise, such as comparing two values to see whether they are equal.

Interlude

What about Artificial Intelligence (AI)?

"But," I hear you say, "computers are smart! They *can* use judgment. What about that program Big Blue that beat Gary Kasparov, the world's chess champion, in 1997?"

Artificial intelligence (AI) is the exception that proves the rule. It might seem that an AI computer program is exercising judgment, but only because of a complex program made up of thousands, even millions, of individual little decisions, each of which is simple and mathematically clear.

Consider the human brain. The operation of each neuron is simple: given sufficient stimulus, it fires; otherwise, it doesn't. It's the *network* of billions of neurons, analogous to the millions of lines in a big computer program, that creates a host for consciousness.

Is that so different from a computer? This question leads us to Very Big Dilemmas. If a computer can be conscious, is it murder to shut it off or junk it? But if a computer cannot be conscious, then what is it about the brain that's so special? Answering such questions is far outside the scope of this book, but I consider this issue—could robots or computers theoretically become conscious?—to be the most central problem in philosophy today.

▼ continued on next page

Interlude

▼ continued

It remains a super-controversial question as to whether human brains, at the deepest level, are any more intelligent than computer circuits, or whether, as physicist Roger Penrose contends, the brain is "non-computational" and therefore cannot be captured by computer software. That question is outside the scope of this book!

if and if-else

The simplest way to program behavior is to say, "If A is true, then do B." That's what the C++ **if** statement does. Here is the simple form of the **if** statement syntax:

Keyword

```
if (condition)
    statement
```

There are more complex forms of this statement, which we'll get to soon. But first consider an **if** statement that compares two variables, x and y.

```
if (x == y)
     cout << "x and y are equal.";
```

This is strange. There are two equal signs (==) here instead of one (=). This is not a typo. C++ has two separate operators in this regard: One equal sign means assignment, which copies a value into a variable. Two equal signs (==) means test for equality.

Note ▶ Using assignment (=) where you meant to use test for equality (==) is one of the most common C++ programming errors. If you're going to program in any of the C family of programming languages (C#, C++, Java, etc.), this is an important rule you'll need to get used to as soon as possible!

What if, instead of executing one statement in response to a condition, you want to do a series of things? The answer is you can use a *compound statement* (or *block*):

```
if (x == y) {
     cout << "x and y are equal." << endl;
     cout << "Isn't that nice?";
     they_are_equal = true;
}
```

The significance of the block is that either all these statements are executed or none of them is executed. If the condition is not true, the program jumps past the end of the block. A block can be plugged into the **if** statement syntax because of another cardinal rule:

Anywhere you can use a statement in C++ syntax, you can use a compound statement (or *block*).

The block itself is not terminated by a semicolon (;)—only the statements inside are terminated by a semicolon. This is one of the few exceptions to the general rule that statements end with a semicolon in C++.

Here's the **if** statement syntax again:

```
if (condition)
    statement
```

Applying the cardinal rule I just stated, we can insert a block for the *statement*

```
if (condition) {
    statements
}
```

where *statements* is zero or more individual C++ statements.

You can also specify actions to take if the condition is *not* true. This is optional. As you might guess, this variation uses the **else** keyword:

Keyword

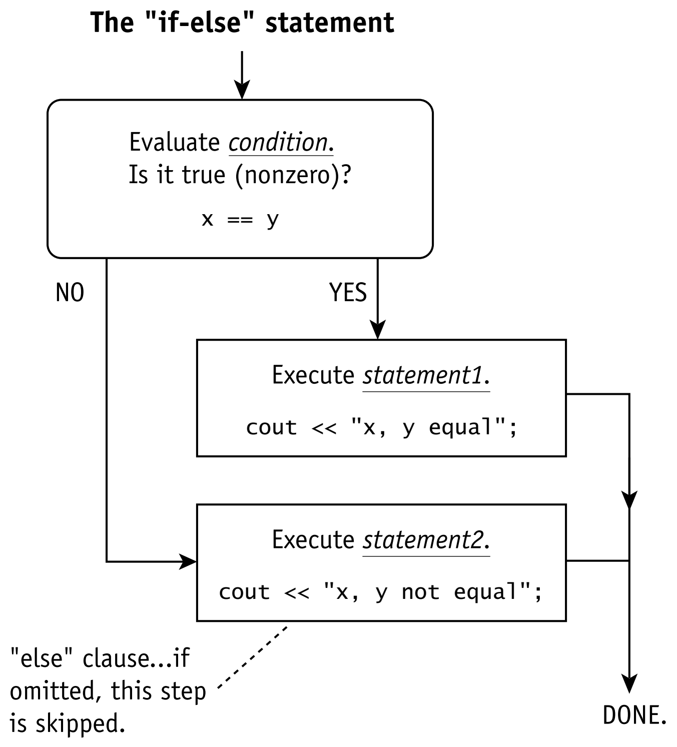
```
if (condition)
    statement1
else
    statement2
```

Either *statement1* or *statement2*, or both, can be a compound statement (that is, a block). And, in fact, many programming teachers and professionals insist that you should always use this "block style," even when the blocks have single statements, because the block style makes it easy to go back and add statements without inadvertently causing an error.

Here's an example, using statement-block style:

```
if (x == y) {
    cout << "x and y are equal";
} else {
    cout << "x and y are NOT equal";
}
```

The following diagram illustrates the flow of control:



The **if-else** statement is highly flexible. Although, technically speaking, there is no "else-if" keyword, you can achieve the same effect by simply adding new if clauses inside the else clauses. For example:

```
if (x == 1) {
    cout << "x equals 1";
} else if (x == 2) {
    cout << "x equals 2";
} else if (x == 3) {
    cout << "x equals 3";
} else if (x == 4) {
    cout << "x equals 4";
}
```

This is actually a poor way of printing out the value of x, but it does illustrate cascading **if** and **else** clauses, creating the same effect as "else-if." In this case, it would be much easier to write the following:

```
cout << "x equals " << x;
```

Interlude

Why Two Operators (= and ==)?

If you've used another programming language such as Pascal or Basic, you may wonder why = and == are two separate operators. After all, Basic uses a single equal sign (=) for both assignment and test for equality, using context to tell them apart.

In C and C++, the following code is freely permitted. Yet it's almost always wrong.

```
int x, y;
...
if (x = y)

         // ERROR! Assignment!
    cout << "x and y are equal";
```

What this example does is assign the value of y to x and use that value as the test condition. If this value is nonzero, it is considered "true." Consequently, if y is any value other than zero, the previous condition is "true" and the statement is always executed!

Here is the correct version, which will do what you want:

```
if (x == y)
    // CORRECT: test for equality
    cout << "x and y are equal";
```

Here, `x == y` is an operation that tests for equality and evaluates as true or false. The important thing to remember is not to confuse test for equality with assignment (x = y).

Why allow this potential source of problems? Well, most expressions in C++ (the main exception being calls to **void** functions) produce a value, and this includes assignment (=), which is an expression with a side effect. So, you can initialize three variables at once by doing this,

```
x = y = z = 0;   // Set all vars to 0.
```

which is equivalent to this:

```
x = (y = (z = 0));   // Set all vars to 0.
```

Each assignment, beginning with the rightmost one (z = 0), produces the value that was assigned (0), which is then used in the next assignment (y = 0). In other words, 0 is passed along three times, each time to a new variable.

▼ *continued on next page*

Interlude

▼ continued

C++ treats "x = y" as an expression that produces a value. And there'd be nothing wrong with that, except that *almost any valid numeric expression can be used as a condition.* Therefore, the compiler does not complain if you write this, even though it's usually wrong:

```
if (x = y)
    // Do action... (Probably should have used ==)
```

Example 2.1. *Odd or Even?*

OK, enough preliminaries. It's time to look at a complete program that uses decision making. This is a simple example, but it introduces a new operator (%) and shows the **if-else** syntax in action.

The program takes a number from the keyboard and reports whether it is odd or even.

even1.cpp

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

int main()
{
    int  n = 0, remainder = 0;

    // Get a number from the keyboard.

    cout << "Enter a number and press ENTER: ";
    cin >> n;

    // Get remainder after dividing by 2.

    remainder = n % 2;

    // If remainder is 0, the number input is even.

    if (remainder == 0) {
        cout << "The number is even." << endl;
```

even1.cpp, cont.

```
        } else {
            cout << "The number is odd." << endl;
        }
        return 0;
    }
```

If you're following along and want to enter this example by hand, the comments—lines beginning with double slashes (//)—are optional. You don't have to enter those.

Remember that comments are ignored by the C++ compiler itself. They are "no-ops." But if you plan to ever come back to the program, possibly to revise it or to show it to someone else, comments can have great value. You can use them to remind yourself what specific parts of the program do. Although, in theory, you can put anything you want inside a comment, presumably you'll write comments that are helpful to a human reading the program.

How It Works

The first statement of the program declares two integer variables, n and remainder:

```
int  n = 0, remainder = 0;
```

The next thing the program does is get a number and store it in the variable n. This should look familiar by now:

```
cout << "Enter a number and press ENTER: ";
cin >> n;
```

After a number is input this way, all the program needs to do is test n to see whether it is odd or even. How do you do that? Answer: You divide the number by 2 and look at the remainder. If the remainder is 0, the number is even (in other words, the number is perfectly divisible by 2). Otherwise, the number is odd.

That's exactly what this program does. The following statement divides by 2 and gets the remainder. This is called *modulus* or *remainder* division. The result is stored in a variable named (appropriately enough) "remainder."

```
int remainder = n % 2;
```

Again, if the remainder is 0, that means n divides evenly into 2. The numbers 2, 4, 6, 8, and 10 are all even numbers, and when you divide by 2 you get a remainder of 0. The numbers 1, 3, 5, and 7 are all odd, and when you divide by 2 you get a remainder of 1.

The percent sign (%) loses its ordinary meaning in C++ and instead signifies "remainder division." Here are some sample results:

EXAMPLE	REMAINDER FROM DIVISION	REMARKS
`3 % 2`	1	Odd
`4 % 2`	0	Even
`25 % 2`	1	Odd
`60 % 2`	0	Even
`25 % 5`	0	Divisible by 5
`13 % 5`	3	Not divisible by 5; 3 is left over after division

This special kind of division—remainder division—is restricted to integer data types. Dividing two integers produces a quotient and a remainder.

```
int my_quotient  = 17/3;   // Result is 5.
int my_remainder = 17%3;   // Result is 2.
```

In contrast, division of two floating-point numbers (such as 4.5 divided by 2.0) results in a single floating-point result… in this case, 2.25.

After dividing n by 2 and getting the remainder, we get a result of either 0 (even) or 1 (odd). The **if** statement compares the remainder to 0 and prints the appropriate message.

```
    if (remainder == 0) {
        cout << "The number is even." << endl;
    } else {
        cout << "The number is odd." << endl;
    }
```

Notice the double equal signs (==). As I mentioned earlier, test for equality uses double equal signs; a single equal sign (=) would mean assignment. If I'm getting repetitive on this subject, it's because when I was first learning C, I made this mistake too many times myself!

Optimizing

Optimizing the Code

The version of the odd-or-even program I just introduced is not as efficient as it could be. The remainder variable is not necessary in this case. This version is a little better:

even2.cpp

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

int main()
{
    int  n;

    // Get a number n from the keyboard.

    cout << "Enter a number and press ENTER: ";
    cin >> n;


    // Get remainder after dividing by 2.
    // If remainder is 0, then n is even.

    if (n % 2 == 0) {
        cout << "The number is even.";
    } else {
        cout << "The number is odd.";
    }
    return 0;
}
```

This version performs modulus division inside the condition, comparing the result to 0. Because the operation is done this way, there is no need to make use of an additional variable called "remainder."

EXERCISE

Exercise 2.1.1. Write a program that reports whether a number input is divisible by 7.

Introducing Loops

One of the most powerful concepts in any programming language is that of *loops*. Looping is one of the things that enables a program to do far more than perform a simple calculation and quit. In this section, you'll see how a few lines of C++ can set up an operation to be performed dozens of times or (if you choose) thousands of times or even millions.

When a program is in a loop, it performs an operation over and over as long as a condition is true. The simplest form is the classic **while** statement.

```
while (condition)
    statement
```

As with **if**, you can replace *statement* with a compound statement or block, which in turns lets you put as many statements inside the loop as you want.

```
while (condition) {
    statements
}
```

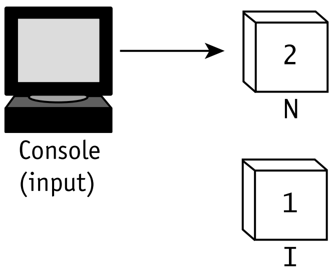
The **while** keyword creates a loop that evaluates the *condition* and executes the *statement* if the condition is true. Then the loop repeats this operation until the condition is false.

Probably the simplest program example is a loop that prints the numbers 1 to n, where n is a number input at the keyboard. We'll look first at this program in *pseudocode* form. For the next page or so, I use the variable names I and N, because they make the pseudocode easier to follow. Assume that the variables have already been declared.

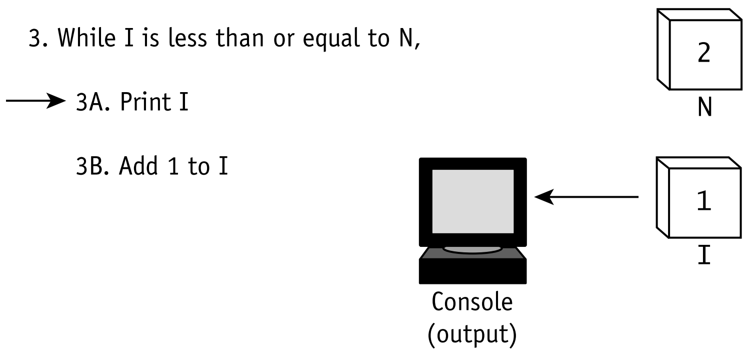
Here's how to print the numbers from 1 to N:

```
1  Get a number from the keyboard and store in N.
2  Set I to 1.
3  While I is less than or equal to N,
     3A  Output I to the console.
     3B  Add 1 to I.
```

The first two steps initialize the variables I and N, which are integers. I is set directly to 1. N is set by keyboard input. Assume that the user inputs 2.

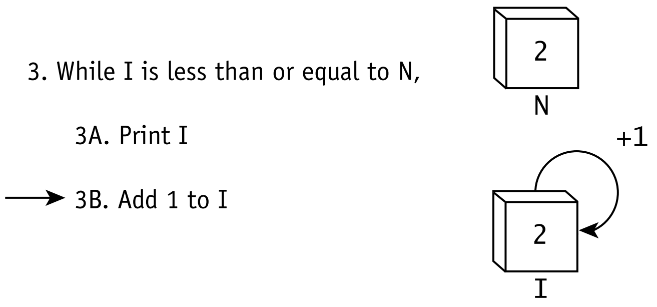


Step 3 is the interesting one. The program first considers whether I (which is 1) is less than or equal to N (which is 2). Since I *is* less than N, the program carries out steps 3A and 3B. First, it prints the value of I.



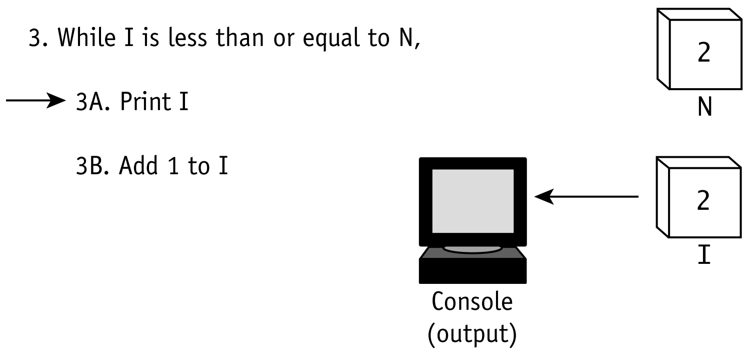
```
3. While I is less than or equal to N,
→   3A. Print I
    3B. Add 1 to I
```

Then it increases the value of I by 1. (This is also called *incrementing*.)



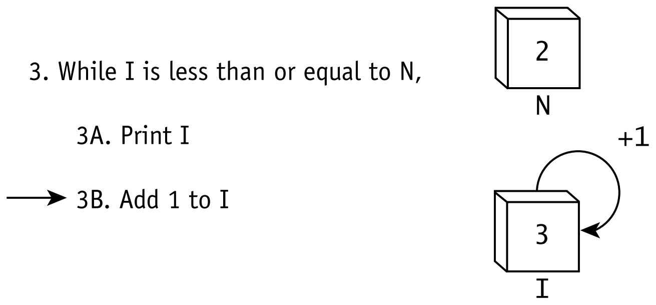
```
3. While I is less than or equal to N,
    3A. Print I
→   3B. Add 1 to I
```

Once it has carried out these steps, the program performs the comparison again. Because this is a **while** statement, not an **if**, the program continues to perform steps 3A and 3B until the condition is no longer true.



```
3. While I is less than or equal to N,
→   3A. Print I
    3B. Add 1 to I
```

So far, the condition is still true (because I is still less than or equal to 2), so the program continues.



After printing the new value of I, the program increments I again.

The program performs the test once more. Because I is now greater than N, the condition (is I less than or equal to N?) fails, causing the program to finally end. 3 is never printed. The output of the program is:

```
1 2
```

Because the user had input 2, the loop executed twice. But with a large input for N (say, 1024), the loop would continue many more times.

Think about it: Here's a program a few steps long that could (depending on the value input for n) print millions of numbers just as easily as it could print two numbers. The theoretical value of n has no limit, except for the maximum integer size; the largest number that can be stored in an **int** variable on a 32-bit system is approximately 2 billion (that is, 2,000 million). But note that the new **long long int type** (mandated in the C++11 and C++14 specs) is far larger still as it is usually based on 64 bits.

Interlude

Infinite Loopiness

Can you set the loop condition in such a way that it will *always* be true? And if so, what happens? The answer is yes, and the loop will run until something interrupts it. (One such way to set the loop condition is to use the **break** keyword, as you'll see in upcoming material.)

Unless you have a mechanism for escaping from a loop, it's important to make sure it doesn't run forever. Otherwise, the program will seem to reach a state where it sits there and does nothing. It *is* doing something, but it's stuck in a pointless loop that, left to itself, threatens to go on till the end of time.

Therefore, always make sure there's some way to exit (even if it involves the **break** keyword, which I'll introduce in the "Optimizing the Program" section on page 58).

Example 2.2. *Print 1 to N*

Now let's use C++ code to implement the loop described in the last section. This uses a simple **while** loop with compound-statement syntax. Here and in the remainder of the book, I stick to the C++ convention of using lowercase letters for variable names.

count1.cpp

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

int main()
{
    int  i = 0, n = 0;

    // Get number from the keyboard and initialize i.

    cout << "Enter a number and press ENTER: ";
    cin >> n;
    i = 1;

    while (i <= n) {  // While i less than or equal n,
        cout << i << " ";   //   Print i,
        i = i + 1;          //   Add 1 to i.
    }
    return 0;
}
```

Some of the comments are on the same line as the C++ statements. This works because a comment begins with the double slashes (//) and continues to the end of line. Comments can be on their own lines or to the right of the statements. Note that these to-the-end-of-line comments make line breaks significant. Make sure you have enough space to put your comments in without causing lines to wrap.

This program, when run, counts to the specified number. For example, if the user inputs 6, the program prints the following:

```
1 2 3 4 5 6
```

How It Works

This example introduces a new operator—although I'm sure you've surmised what it does. This is the less-than-or-equal-to test.

```
i <= n
```

The less-than-or-equal operator (<=) is one of several relational operators, all of which return true or false.

OPERATOR	MEANING
`==`	Test for equality
`!=`	Test for inequality (greater than or less than)
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

If you followed the logic in the "Introducing Loops" section, you know the loop itself is straightforward. The braces ({}) create a statement block so that the **while** loop executes two statements each time through, rather than one.

```
while (i <= n) {      // While i less than or equal n,
    cout << i << " "; //   Print i,
    i = i + 1;        //   Add 1 to i.
}
```

If you think about it, you'll see that the last number to be printed is n, which is what we want. As soon as i becomes greater than n, the loop ends and the output statement never executes for that case. The first statement inside the loop is as follows:

```
    cout << i << " ";   //   Print i,
```

This statement adds a space after i is printed. This is so the output is spaced like this:

```
1 2 3 4 5
```

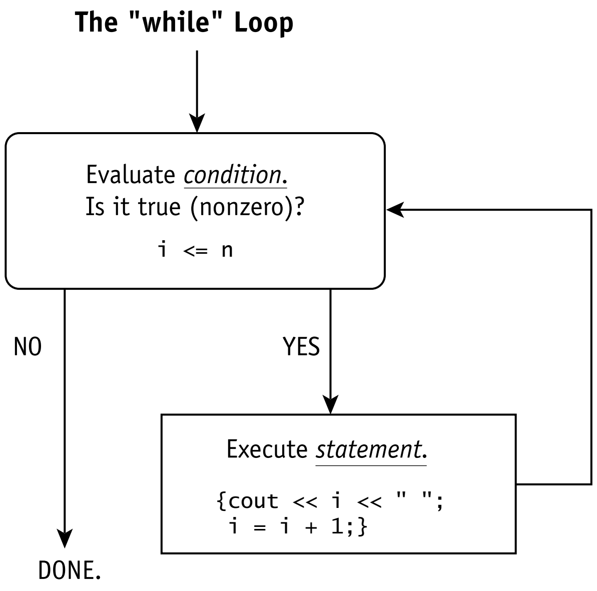
rather than this:

```
12345
```

The loop then adds 1 to i before continuing to the next cycle. This ensures that the loop eventually ends, because i will sooner or later become greater than n (at which point the loop condition will fail).

```
    i = i + 1;          //   Add 1 to i.
```

It may be easier to see how a **while** loop works in the following diagram, which shows the flow of control in the loop.



Optimizing the Program

The program can be made more efficient by combining a couple of the statements. You can do this by initialization, in which you assign a value as you declare it. You can use the equal sign (=) to give any numeric or string variable a value.

```
int variable = value;
```

In this revised program, i is initialized to 1 so it doesn't need to be assigned a value later. In this example, n is also initialized, although strictly speaking, it doesn't need to be. But getting into the habit of initializing variables is a good idea.

count2.cpp

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

int main()
{
    int  i = 1, n = 0;

    // Get number from the keyboard and initialize i.

    cout << "Enter a number and press ENTER: ";
    cin >> n;


    while (i <= n) {  // While i less than or equal n,
        cout << i << " ";   //   Print i,
        i = i + 1;          //   Add 1 to i.
    }

    return 0;
}
```

EXERCISES

Exercise 2.2.1. Write a program to print all the numbers from n1 to n2, where n1 and n2 are two numbers specified by the user. (Hint: You'll need to prompt for two values n1 and n2; then, initialize i to n1 and use n2 in the loop condition.)

Exercise 2.2.2. Alter the example so that it prints all the numbers from n to 1 in reverse order, as in 5 4 3 2 1. (Hint: To decrement a value inside the loop, use the statement "i = i – 1;".)

Exercise 2.2.3. Alter the example so that it prints only even numbers, as in 0, 2, 4. (Hint: One of the things you'll need to do is initialize i to 0.)

True and False in C++

What exactly are "true" and "false"? Are these values stored in the computer in numeric form, like any other? Yes. Every Boolean (relational) operator returns 1 or 0:

IF THE CONDITION EVALUATED IS...	THE EXPRESSION RETURNS...
True	1
False	0

Also, any nonzero value fed to a condition is interpreted as true. So, in this example, the statements are always executed:

```
if (true) {            // ALWAYS EXECUTED
    // Do some stuff.
}
```

This next case creates an infinite loop, which is normally a program-logic error, unless you provide some way to break out of it (such as a **break** or **return** statement):

```
// INFINITE LOOP!

while (true) {
    // Do some stuff.
}
```

Interlude

The bool Data Type

For quite a few years now, C++ compilers have supported the **bool** ("Boolean") type, which is similar to the integer type but holds only two values: true (1) or false (0). Any nonzero numeric input is converted to 1 ("true"). You should use the **bool** type if it is supported.

```
bool  is_less_than;

is_less_than = (i < n);  // Store the value
                         //   true (1) if
                         //   i is less than n.
```

Only the very oldest compilers lack support for the **bool** data type. If your computer lacks it, you'll need to use the **int** type in place of **bool** and use the number 1 in place of **true**.

The Increment Operator (++)

If you look at many lines of computer programs, you see certain statements come up over and over—so much so that you start to wonder if there isn't some kind of "shorthand" or shortcut that would reduce the work of programming. The C and C++ languages are extremely good at this, which helps explain the immense popularity of the C-language family in today's computing world.

One of the most common things to do is to add 1 to a variable. In C++, you can do this by putting double plus signs (++) in front of or after a variable name. For example:

```
n++;
++n;
```

The effect of the statement, in either case, is to increase n by 1. Consider the loop of the previous section:

```
while (i <= n) {    // While i less than or equal n,
    cout << i << " ";   //   Print i,
    i = i + 1;          //   Add 1 to i.
}
```

The second statement inside the loop can be replaced with a statement using the increment operator, producing the following:

```
while (i <= n) {    // While i less than or equal n,
    cout << i << " ";   //   Print i,
    ++i;                //   Add 1 to i.
}
```

So far, this substitution has saved only a few keystrokes. But it gets better. The item ++i is "an expression with a side effect," meaning it produces a value *and* performs an action. This is the prefix version, which increments the variable and then gets its value.

C++ also supports postfix expressions such as "i++": This is an expression with the same value as i; but after i++ is evaluated, it adds 1 to i. So, the loop can be shortened to this:

```
while (i <= n) {    // While i less than or equal n,
    cout << i++ << " ";  // Print i, then add 1 to i.
}
```

Do you see what this does? The statement prints out the current value of i and *then* increments it.

However, you need to show care here. Including the expression "i++" several times in a single statement causes multiple incrementing and can have unpredictable results. A good rule is to use it at most once per statement.

You might wonder whether there is a corresponding operator for subtraction. In fact, there are four increment/decrement operators. Here, *var* means any numeric variable: Generally, it should be an integer to work with these operators reliably.

OPERATOR	ACTION
var++	Pass along the current value of *var*; then add 1 to var.
++*var*	Add 1 to *var*; then pass along the result.
var--	Pass along the current value of *var*; then subtract 1 from it.
--*var*	Subtract 1 from *var*; then pass along the result.

In many contexts, you can use either "i++" or "++i" and the effect will be the same. For example, there is no difference in results when the expression appears alone:

```
++i;
```

But while the postfix version (i++) used to be the preferred style for cases like this, the prefix version (++i) is now preferred by most C++ experts. When you're working with integers, the difference between postfix and prefix increment is not significant, but when you work with more complex types, the prefix version potentially results in a more efficient program. Many experts feel it's important to get people into the right habits. This is why the prefix version (++i) is now favored in those cases where either would suffice.

So, even though it may seem to be a "Mrs. Grundy" rule, I will prefer the prefix version of increment (++) and decrement (−−) for the remainder of the book, wherever possible.

Statements versus Expressions

Until now, I've gone along using the terms *statement* and *expression*. These are fundamental terms in C++, so it's important to clarify them. In general, you recognize a statement by its terminating semicolon (;).

```
cout << ++i << " ";
```

A simple statement such as this is usually one line of a C++ program. But remember that a semicolon terminates a statement, so it's legal (though not recommended) to put two statements on a line:

```
cout << i << " ";  ++i;
```

Fine, you say—a statement is (usually) one line of a C++ program, terminated with a semicolon. So, what's an expression? An expression usually produces a value (with a few notable exceptions). You terminate an expression to get a simple statement. Here's a sample list of expressions, along with descriptions of what value each produces:

```
x               // Produces value of x
12              // Produces 12
x + 12          // Produces x + 12
x == 33         // Test for equality: true or false
x = 33          // Assignment: produces value assigned
++num           // Produces value before incrementing
i = num++ + 2   // Complex expression; produces
                // new value of i
```

Because these are expressions, any of them can be used as part of a larger expression, including assignment (=). The last three have *side effects*. x = 33 alters the value of x, and num++ alters the value of num. The last example changes the value of both num and i. Remember, any expression can be turned into a statement by using a semicolon (;).

```
++num;
```

The fact that any expression can be turned into a statement this way makes some strange statements possible. You could, for example, turn a literal constant into a statement, but such a statement would do exactly nothing.

```
12;
```

This is a legal statement in C++. Why would anyone write such a statement? The answer is they wouldn't. It just illustrates the point that any expression in C++ (with a few exceptions) can be turned into a statement.

But that's not to say that statements aren't significant. By putting expressions into different statements, you guarantee that everything in one statement is evaluated before everything in the next statement. The danger of putting everything into a single statement is that it becomes difficult to predict the order of execution in extremely complex expressions. In the following statements, the order in which things happen is clear:

```
++i;      // Increment i.
++i;      // Increment i again.
j = i++;  // Assign i to j, then inc. i.
```

Introducing Boolean (Short-Circuit) Logic

Sometimes you need words like *and*, *or*, and *not* to express a decision. This is just common sense. For example, here (in pseudocode) is a decision that uses "and":

```
If age > 12 and age < 20
    The subject is a teenager
```

Programmers use *Boolean algebra*, named for the nineteenth-century mathematician George Boole, to express such conditions. For example, the subexpressions "age > 12" and "age < 20" are each evaluated, and if both are true, the whole expression is true:

age > 12 and age < 20

The following table summarizes the three logical (Boolean) operators in C++:

SYMBOL	OPERATION	C++ SYNTAX	ACTION
&&	AND	`expr1 && expr2`	Evaluate *expr1*. If it's true, evaluate *expr2*. Then, if both are true, return true; otherwise, return false.
||	OR	`expr1 || expr2`	Evaluate *expr1*. If false, evaluate *expr2*. Then, return true unless both are false.
!	NOT	`! expr1`	Evaluate *expr1*. Reverse the true/false value.

So, the earlier example expresses "and" this way in C++:

```
if (age > 12 && age < 20) {    // if age>12 AND age<20
     cout << "The subject is a teenager.";
}
```

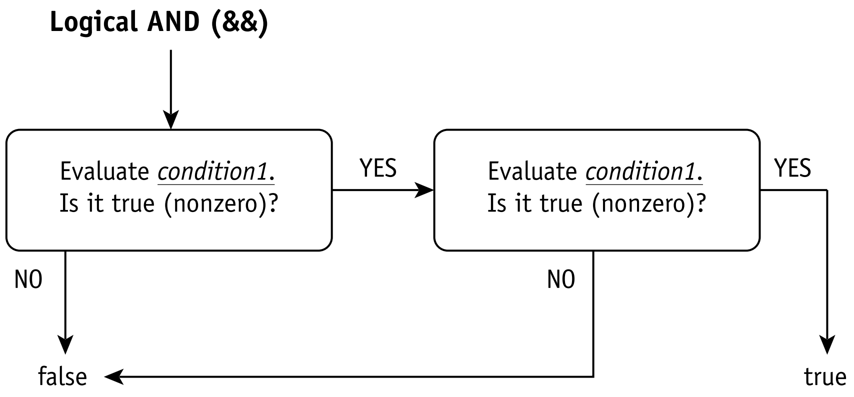
Logical operators have lower precedence than the relational operators (<, >, >=, <=, !, and ==), which in turn have lower precedence than arithmetic operators (such as + and *). Consequently, the following statement does what you'd probably expect:

```
if (x + 2 > y && a == b) {
    cout << "The data passes the test";
}
```

This means "If x + 2 is greater than y, and a is equal to b, then print the message." You can, if you choose, make the program clearer for humans who read it by using parentheses—even though this doesn't change what it does.

```
if (((x + 2) > y) && (a == b)) {
    cout << "The data passes the test";
}
```

In C++, the "and" and "or" operators (&& and ||) employ short-circuit logic: this means the second operand is evaluated only if it needs to be. This can make a difference if the second condition has side effects.


Logical AND (&&)
Evaluate condition1. Is it true (nonzero)?
YES
Evaluate condition1. Is it true (nonzero)?
YES
NO
NO
false
true

Note ▶ Don't confuse the logical operators with the C++ bitwise operators (&, |, ^, and ~). The bitwise operators compare each bit in one operand to the corresponding bit in the other. Bitwise operators do not use short-circuit logic, while logical operators do.

Interlude

What Is "true"?

The logical operators—&&, ||, and !—can take any expressions as input as long as they are convertible to **bool**, which is a subtype of integer. Any non-zero expression is considered "true." Some programmers take advantage of this behavior to write shortcuts:

```
if (n && b > 2) {
   cout << "n is nonzero and b is > than 2.";
}
```

Many programmers dislike the use of any conditions other than those that have obvious true/false values (such as "x > 0" or "x == 0"). However, the following condition, which says "Do as long as n is nonzero," is a short-cut favored by some:

```
if (n--)   {  // If n doesn't equal 0.
   cout << n << endl;
}
```

This fragment is a succinct way of counting down from n to zero. The problem is that if n somehow gets set to a negative value, you're in trouble because the decrement operator (−−) will just keep subtracting the value of n. So, even in this case, it's safer to use this:

```
if (n-- > 0) {
    cout << n << endl;
}
```

Example 2.3. *Testing a Person's Age*

This section demonstrates a simple use of the "and" operator (&&). The program here determines whether a number is in a particular range—in this case, the range of teen numbers, 13 through 19 inclusive.

range.cpp

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

int main() {
    int  n;
```

▼ continued on next page

range.cpp, cont.

```
    cout << "Enter an age and press ENTER: ";
    cin >> n;

    if (n > 12 && n < 20) {
        cout << "Subject is a teenager." << endl;
    } else {
        cout << "Subject is not a teenager." << endl;
    }
    return 0;
}
```

How It Works

This brief program uses a condition made up of two relational tests:

```
n > 12 && n < 20
```

Because logical "and" (&&) has lower precedence than relational operations (> and <), the "and" operation is performed last. The test performs as if written this way:

```
(n > 12) && (n < 20)
```

Consequently, if the number input is greater than 12 and less than 20, the condition evaluates to true and the program prints the message "Subject is a teenager."

In expressions like these, it's common for programmers to omit these extra parentheses because the precedence is assumed to be obvious. But it might not be so obvious if you're new to C++. In Appendix A, Table A.1 (pages 476-477) lists all operators, indicating precedence between them.

However, in addition to consulting that table, you can remember a few basic principles. In general, arithmetic operators have high precedence, relational operators (such as <, >, and ==) have lower precedence, and assignment (==) has nearly the lowest precedence of all.

Increment and decrement operators (++ and −−) have some of the highest precedence, despite the fact they have side effects.

EXERCISE

Exercise 2.3.1. Write a program to test a number for being in the range 0 to 100, inclusive.

Introducing the Math Library

Up to this point, I've used the C++ standard library for the support of input-output streams. That enabled the code to use **cout** and **cin**, which is why the programs had to include the following line:

```
#include <iostream>
```

Now, I'm going to introduce one of the math functions. You can use any of the C++ operators (such as +, *, –, /, and %) without library support because operators are intrinsic to the language itself. But to use math functions, you need to include this line:

```
#include <cmath>
```

This **#include** directive brings in declarations for all the math functions, so you don't have to prototype them yourself. (I'll have a lot more to say about function prototypes in Chapter 5, "Functions: Many Are Called.") C++ supports many math functions, such as the trig and exponential functions, but this chapter uses just one: **sqrt**, which returns a square root.

```
#include <cmath>
//...

double x = sqrt(2.0);  // Assign square root of 2 to x
```

Programmers affectionately refer to this as the "squirt" function. As with most math functions, this function accepts and returns a floating-point result. If you assign the result to an integer, C++ drops the fractional portion (and also issues a warning message).

```
int  n = sqrt(2.0);   // Place the value 1 into n,
                      //    after truncating 1.41421.

double x = sqrt(2);    // Ok: int converted to double.
```

The second statement in this example works because you can freely assign an integer to an argument that expects a floating-point value. Going in the reverse direction is more problematic and can cause loss of data. (In particular, having a result such as 1.41421 is a nearly catastrophic loss of data.)

Example 2.4. Prime-Number Test

We now have enough tools to do something interesting: determine whether a number is prime. A prime number is a number divisible only by itself and 1. It's obvious that 12,000 is not prime, since it's a multiple of 10, but it's less obvious whether 12,001 is a prime number. This is a classic math problem to give to a computer program. Here's the code:

prime1.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    int  n = 0;  // Number to test for prime-ness
    int  i = 2;  // Loop counter
    bool is_prime = true;   // Boolean flag...
                            // Assume true for now.

    // Get a number from the keyboard.

    cout << "Enter a number and press ENTER: ";
    cin >> n;

    // Test for prime by checking for divisibility
    //  by all whole numbers from 2 to sqrt(n).

    while (i <= sqrt(n)) {
        if (n % i == 0) {      //  If i divides n,
            is_prime = false;  //    n is not prime.
        }
        ++i;                   //  Add 1 to i.
    }

    // Print results

    if (is_prime) {
        cout << "Number is prime." << endl;
    } else {
        cout << "Number is not prime." << endl;
    }
    return 0;
}
```

When the program is run, if the user enters 12000, the program will print the following:

```
Number is not prime.
```

To discover what happens with 12001, I'll leave you to run the program for yourself.

Note ▶ When running the program, enter 12000 rather than 12,000. A C++ program doesn't normally expect or permit commas inside numerals. In Chapter 10, I show a way around this limitation.

How It Works

The core of the program is the following loop:

```
while (i <= sqrt(n)) {
    if (n % i == 0) {      //  If i divides n,
        is_prime = false;  //    n is not prime.
    }
    ++i;                   //    Add 1 to i.
}
```

Let's look at this a little more closely. Here's a pseudocode version of this loop:

```
Set i to 2.
While i is less than or equal to the square root of i,
    If n is divisible by the loop counter (i),
        n is not prime.
    Add 1 to i
```

The loop checks for divisors starting with 2, stopping at the square root of n, because if it was going to find divisors, it would have found them by that point.

The divisibility test uses the modulus operator (%) introduced earlier. This operator performs division and returns the remainder. If the second number, i, perfectly divides the first, the remainder is 0; in that case, n is not prime.

```
if (n % i == 0) {
    is_prime = false;
}
```

The beginning of the program assumes the number is prime (is_prime = true), so if no divisors are found, the result is true. Remember that the values **true** and **false** are predefined in all but the most ancient versions of C++.

Optimizing the Program

There are several ways this program can be improved, but the most important change is to exit the loop after a divisor is found. There is no reason to continue, since that would waste CPU time. The C++ **break** keyword exits the nearest enclosing loop. Here is the revised code:

```
while (i <= sqrt(n)) {
    if (n % i == 0) {
        is_prime = false;
        break;     // BREAK OUT OF LOOP NOW!
    }
    ++i;
}
```

EXERCISE

Exercise 2.4.1. Optimize the program by calculating the square root of n just once, rather than repeatedly. You'll need to declare another variable and set it to the square root of n. The type should be **double**. You can then use this variable in the **while** condition.

Example 2.5. *The Subtraction Game (NIM)*

In this final example, we'll use the tools described in the chapter to create a simple game and give the computer a strategy that wins every time—unless the human player adopts the absolute optimal strategy for his or her own side.

Welcome to the game of NIM. The simplest version is the Subtraction Game, in which two players take turns subtracting a number from a common total. Each player may subtract either 1 or 2. Whoever first reduces the total to zero or less wins. For example:

1. We agree to start with the number 7, and you go first.
2. You subtract 2 from the total, making it 5.
3. I also subtract 2 from the total, making it 3.
4. You subtract 1 from the total, making it 2.
5. I subtract 2 from the total, making it 0. I win!

This is a simple game with a simple winning strategy. Consider what happens if the total is 3. Then, regardless of whether you subtract 1 or 2, I can always cause the next subtraction to get to zero and I win. Therefore, if I make the total 3, I can force a win.

Similarly, if I make the total 6, then whether you subtract 1 or 2, I can make the total 3, which indirectly forces a win for me. I win on the following turn.

The winning strategy, therefore, is for me to always make the total a multiple of 3 if I can. Programmatically, how do I do this? I use our old friend, the remainder-division operator:

SITUATION	BEST RESPONSE
total % 3 produces 2.	Subtract 2; the new total will be an exact multiple of 3.
total % 3 produces 1.	Subtract 1; the new total will be an exact multiple of 3.
total % 3 produces 0.	The total is already a multiple of 3. All I can do is subtract 1 and hope for the best.

The pseudocode version of the program is as follows:

```
Print invitation to play and ask for a starting total.
While true   // This is an "infinite loop"
    If total % 3 is 2
        Subtract 2 from total and announce move
    Else
        Subtract 1 from total and announce move
    If total is 0 or less
        Announce "I win!" and exit
    Prompt opponent for move
    While input is not 1 or 2
        Re-prompt for input
    Adjust total by input amount and announce result
    If total is 0 or less,
        Announce "You win!" and exit
```

This is the most complex program yet. It creates a complete game, with an optimal computer strategy. The user can win only by selecting each move perfectly.

nim.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int total = 0, n = 0;

    cout << "Welcome to NIM. Pick a starting total: ";
    cin >> total;
    while (true) {

         // Pick best response and print results.

         if ((total % 3) == 2) {
              total = total - 2;
              cout << "I am subtracting 2." << endl;
         } else {
              total--;
              cout << "I am subtracting 1." << endl;
         }
         cout << "New total is " << total << endl;
         if (total <= 0) {
              cout << "I win!" << endl;
              break;
         }

         // Get user's response; must be 1 or 2.

         cout << "Enter num to subtract (1 or 2): ";
         cin >> n;
         while (n < 1 || n > 2) {
             cout << "Input must be 1 or 2." << endl;
             cout << "Re-enter: ";
             cin >> n;
         }
         total = total - n;
         cout << "New total is " << total << endl;
```

nim.cpp, cont.

```
            if (total <= 0) {
                 cout << "You win!" << endl;
                 break;
            }
       }
       return 0;
   }
```

How It Works

The program makes use of the "or" operator (||) introduced earlier. If either condition—n is less than 1 *or* n is greater than 2—is true, the user is prompted for a new value.

```
while (n < 1 || n > 2) {
    cout << "Input must be 1 or 2." << endl;
    cout << "Re-enter: ";
    cin >> n;
}
```

Because of short-circuit logic, if the first condition (n < 1) is true, the second condition is not evaluated because it's not necessary.

This program example also uses the **break** keyword, which exits out of the loop.

```
break;
```

EXERCISES

Exercise 2.5.1. One problem is that if the initial starting number is less than 1, the program will deal in negative numbers and never end. Revise the program so that it only accepts an initial total greater than 0.

Exercise 2.5.2. For the more ambitious: Write a version that permits subtracting *any* number from 1 to n, where n is stipulated at the beginning of the game. For example, the user when prompted might say that each player can subtract any number from 1 to 7. Can you create an optimal computer strategy for this general case?

Exercise 2.5.3. Revise the program so that it keeps playing the game until the user wants to quit. (Hint: You'll need to add yet another loop around the existing main loop.)

Chapter 2 *Summary*

Here are the main points of Chapter 2:

- Use the right data type for the job. A variable that can have no fractional portion should be given type **int**, unless it exceeds the range of **int** type—more than 2 billion (two thousand million).
- You can declare integer variables by using the data type's name followed by a variable name and semicolon. You can also declare multiple variables, separating adjacent variable names with a comma.

```
int variable;
int variable1, variable2, ...;
```

- Constants have **int** or **double** type as appropriate. Any value with a decimal point is automatically considered a floating-point value: 3 is stored as an **int**, but 3.0 is stored as a **double**.
- The simplest decision-making structure in C++ is the **if** statement.

```
if (condition)
   statement
```

- The **if** statement has an optional **else** clause, so you can use this form:

```
if (condition)
    statement
else
    statement
```

- Anywhere you can use a statement, you can use a compound statement (or *block*), consisting of zero or more statements enclosed in braces ({}). This syntax is preferred by many C++ programmers, even where not strictly necessary.

```
if (condition) {
    statements
}
```

- Don't confuse assignment (=) with test for equality (==). Here's a correct use of the two operators:

```
if (x == y) {
    is_equal = true;
}
```

- The **while** keyword executes a statement as long as the condition is true. As with other C++ statements, the compound-statement (or block) syntax is preferred even when not strictly necessary.

```
while (condition) {
    statement
}
```

- The modulus operator performs division and then returns the remainder. For example, the result of the following expression is 3:

```
13 % 5
```

- An expression is a value formed by a variable, literal, or smaller expression combined with C++ operators (this can include assignment, =).
- Any expression can be turned into a statement by adding a semicolon.

```
num++;
```

- The increment operator is convenient shorthand for adding 1 to a number. This creates an expression with a side effect:

```
cout << n++;    // Print n and then add 1 to n.
```

- You can use the C++ logical (Boolean) operators "and" (&&), "or" (||), and "not" (!) to create complex conditions. The "and" and "or" operators use short-circuit logic.
- The easiest way to get out of a loop is often to use the **break** keyword.

```
break;
```

This page intentionally left blank

3 And Even More Decisions!

Building on simple decision-making structures, Chapter 2, “Decisions, Decisions,” showed how computers can do complex and interesting things, such as play a game, repeat an operation a set number of times, and perform mathematical operations.

Critical to these operations are something called *control structures*—a set of keywords (**if**, **else**, **while**) and associated grammar—that control what the program does next. These control structures, nearly all inherited from C, include just a few simple but extremely versatile keywords. This chapter introduces two more: **do-while** and **switch**.

Remember that by controlling what the program does next, you control the program! It’s that simple.

The do-while Loop

If you’ve worked through Chapter 2, you should have seen that the **while** loop is powerful and versatile. In fact, if you had to rely solely on the **while** loop, you could still write sophisticated programs.

But **while** has its limitations. Take another look at the syntax:

```
while (condition) {
     statements
}
```

Testing the condition before executing the loop typically makes sense. But what if you need to execute the action *before* performing a test? Such cases are more common than you’d think.

Consider the task of flipping a coin until you get heads. In such a procedure, you always flip a coin before deciding whether to stop. You could try to use ordinary “while” logic, resulting in this pseudocode:

```
While you haven't flipped any heads
    Flip a coin and note the result
```

This works, but it's not optimal. The first time you evaluate the condition—Have you flipped any heads?—the answer ought to be, "Of course I haven't flipped any heads, I haven't flipped the coin yet." So the first time you apply the test, you're doing unnecessary work.

A more optimal plan would look like this:

```
Do
    Flip a coin and note the result
While you've flipped zero heads.
```

It should be noted that this pseudocode is equivalent to the following:

```
Flip a coin and note the result
While you haven't flipped any heads
    Flip a coin and note the result
```

There's no huge gain in efficiency here, no ability to write something that couldn't be done before. But this approach is more efficient and in the C/C++ family of languages, efficiency is considered important.

We can obtain this efficiency with **do-while** keywords, which have the following syntax:

```
do statement
while (condition);
```

As always, the *statement* can be a block (or "compound statement") even when there is just one statement inside the loop:

```
do {
     statements
} while (condition);
```

Remember, in this context *statements* must be filled in with zero or more C++ statements. You can always use just one statement in the body of the loop if you choose.

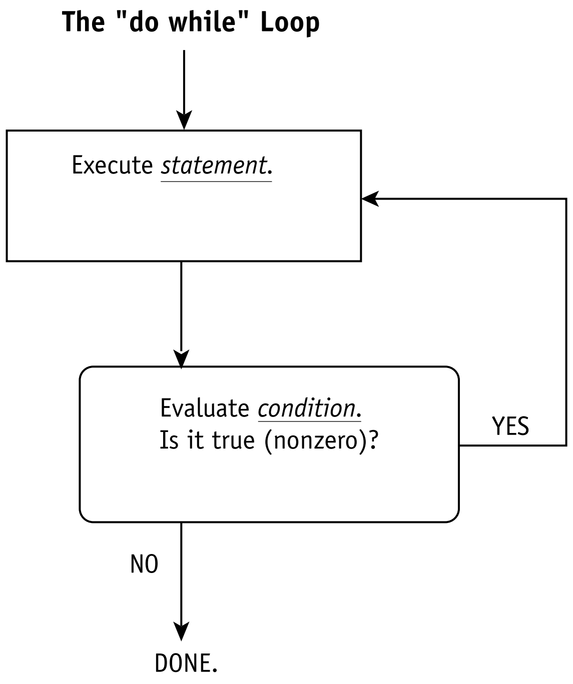
As a simple example, you can use a **do-while** loop to count down from a specified number:

```
do {
     cout << n << endl;
```

```
        --n;
    } while (n > 0);
```

This looks a lot like the **while** loops introduced in Chapter 2. The one difference is that the body of the loop, which prints the value of n, is executed at least once, no matter what.

Here's a flow-chart representation of **do-while**.



Example 3.1. *Adding Machine*

There's not really anything you can do with **do-while** that you can't do with a standard **while** loop, but sometimes **do-while** fits the needs of the program better.

Consider a virtual adding machine, in which the user enters a series of numbers and then enters a special code in order to exit. (We can use zero.) The program prompts the user repeatedly for input but—and this is the important part—the user will always be prompted at least once when running the program. This is why it's a good **do-while** example.

adding.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int  sum = 0;
    int  n = 0;

    do {
        cout << "Enter a number (0 for exit): ";
        cin >> n;
        n += sum;
    } while (n > 0);
    cout << "The sum is " << sum << endl;
    return 0;
}
```

This program, simple though it is, is actually useful. If all you need to do is add a column of figures, this program, when run, provides a faster, more convenient tool than a spreadsheet. Here's what a sample session looks like for adding four numbers—11, 124, 89, and 477:

```
Enter a number (0 for exit): 11
Enter a number (0 for exit): 124
Enter a number (0 for exit): 89
Enter a number (0 for exit): 477
Enter a number (0 for exit): 0
The sum is 701
```

How It Works

When interpreting the statement "sum + = n," remember that this statement is equivalent to:

```
sum = sum + n;
```

The one difference—which has no effect in this case—is that in this second version (sum = sum + n) evaluates the variable "sum" twice. I could have used the longer version of this statement, but I chose to use the more compact version. Here's the pseudocode for the program:

```
Initialize Sum to 0
Do
    Prompt the user for a number
    Input that number into N
    Add N to Sum
While N > 0
Print Sum
```

In simple terms, this amounts to "Prompt for a number N and add it to sum. Repeat this action until N > 0 is no longer true. Then print sum."

You could write this as a standard **while** loop if you chose. The easiest way to do that would be to create an infinite loop—by using **while**(**true**)—and using **break** to exit.

```
while (true) {
    cout << "Enter a number (0 for exit): ";
    cin >> n;
    n += sum;
    if (n <= 0) {  // If n is NOT greater than 0,
        break;     //  exit.
    }
}
```

EXERCISES

Exercise 3.1.1. Revise the program so that it accepts floating-point input and prints a floating-point result. Make sure that you notate the constants correctly.

Exercise 3.1.2. Can you revise the example so that it uses a **while** loop, but does not use either the **do** or **break** keyword? Note that the result should produce exactly the same behavior under all conditions that Example 3.1 does.

Introducing Random Numbers

In the next section, we're going to use a **do-while** loop in another game program. To keep it interesting, we need a way to generate random numbers. That's often true of games and simulations.

It may seem strange, but the production of random numbers is not so trivial. Ask yourself: Just where do you get such numbers? What do you use for "virtual dice"?

Humans turn out to be poor at generating random numbers. If asked to write down 10 random numbers, you'd probably never write sequences that contained 1-2-3 or 9-9-9. Randomness, we reason, should not produce patterns. And yet, given enough random numbers, these sequences do show up and eventually must show up!

Fortunately, C++ programs can generate random numbers through a two-step process. This is what provides "virtual dice":

1 Set a random-number "seed."

2 Produce the next number in the series by using a complex mathematical transformation (the details of which you don't need to know).

I'll consider these in reverse order. The complex math transformation (step 2) takes a number and produces another using an algorithm so complex the user can't predict what the next number will be. That's enough to simulate randomness—to give us numbers that for all practical purposes are random. There's just one problem: Although the sequence is virtually impossible to predict, it is nonetheless deterministic, just as everything inside a digital computer is deterministic.

Consequently, if you don't set a random "seed," a program will get the same series of supposedly random numbers every time you run it. So, the second time you run the program, the numbers being generated won't really be random at all.

To prevent that from happening, you need to set a seed and make sure it's guaranteed to be different every time you set it. Now we're almost back to the original problem: where do you get such a number?

Fortunately, there's an easy answer: use the system time.

From the C++ programmer's point of view, getting a random number is really a three-step process. First, you need to make use of some new **include** statements to bring in support for two parts of the C++ library:

```
#include <cstdlib>  // Supports srand and rand functs.
#include <ctime>    // Supports ctime function.
```

The first line, including <cstdlib>, brings in the declarations for randomization functions. The second line, including <ctime>, brings in declarations for time functions.

The next step is to set the seed. Remember that no matter how many times the program gets a random number, you need set this seed only once.

```
srand(time(nullptr));
```

Note The **nullptr** keyword has been supported since C++11 as a way of representing null pointer values. But if you have a compiler that's more than a few years old, you may need to use NULL instead.

After setting the randomization "seed," the program can produce all the random numbers needed, by calling the **rand** function. But again, it shouldn't need to set the randomization seed more than once. For example:

```
cout << rand() << endl;   // Print a random number.
cout << rand() << endl;   // Print another.
```

What kind of random number do you get? What range is it in?

The answer is you get a number that may be anywhere in the unsigned integer range. The largest possible value is defined in <cstdlib> as RAND_MAX. This is probably not much help. But if you apply the remainder-division operator, %, also called the *modulus* operator, you'll always get a number in the range 0 to n – 1. Then, if you want to get a number in the range 1 to n, add one.

For example, to simulate the rolling of 10 dice, you could use this mini-program:

```
#include <iostream>
#include <cstdlib> // Include support for randomizing.
#include <ctime>   // Include support for ctime.

using namespace std;

int main() {
     srand(time(nullptr));
     for (int i = 0; i < 10; ++i) {
          cout << (rand() % 6) + 1 << endl;
     }
     return 0;
}
```

Note that the expression "(rand() % 6) + 1" can be written without the outer set of parentheses because the precedence of % is higher than plus (+):

```
          cout << rand() % 6 + 1;
```

Now that you know how to generate random numbers, you're prepared for the next section, which creates an interesting (and fun!) game.

Example 3.2. *Guess-the-Number Game*

The idea is simple enough. First, the program "thinks up a random number"—it does this by using the randomization procedures in the last section—and then it asks you, the end user, to guess the number.

At that point, the program has to make a decision, but it's fairly simple:

- If your guess is higher than the target number, it reports that fact and prompts for another guess.
- If your guess is lower than the target number, it reports that fact and prompts for another guess.
- If your guess is correct, it reports that fact and causes the program to break out of the loop. The game then ends.

This game is going to require a loop, because until you guess the right number, the program must continue. A **do-while** is a natural choice because it executes the body of the loop at least once, which is what this program needs to do.

There's another consideration to keep in mind. Despite how fascinating this game is, the user might want a way to exit the program early. Let's use 0 to indicate a desire for early exit. The pseudocode for the program is:

```
Set Boolean variable do_more to true.
Do
    Randomly choose a target number between 1 and 50.
    Prompt the user for a number and store in N.
    If N equals 0
        Set do_more to false
    Else if N > target number
        Print "Guess is too high"
    Else if N < target number
        Print "Guess is too low"
    Else
        Print "You win!"
        Set do_more to false
While do_more is true
```

You should be able to see what's going on here. A Boolean variable, do_more, controls the action of the loop. Therefore, the program can easily terminate the loop by setting do_more to false. If not terminated, this loop just reports whether the user's guess was too high, too low, or just right, and then (if appropriate) it continues to prompt the user.

guessing.cpp

```
#include <iostream>
#include <cstdlib>    // Supports rand and srand.
#include <ctime>      // Supports time function.

using namespace std;

int main()
{
    int  n = 0;
    bool do_more = true;

    srand(time(nullptr));         // Set random seed.
    int target = rand() % 50 + 1; // Get random 1-50.

    do {
        cout << "Enter your guess: ";
        cin >> n;
        if (n == 0) {
            cout << "Bye! ";
            do_more = false;
        } else if (n > target) {
            cout << "Guess is too high. ") << endl;
        } else if (n < target) {
            cout << "Guess is too low. ") << endl;
        } else {
            cout << "You win! ";
            cout << "Answer is " << n << endl;
            do_more = false;
        }
    } while (do_more);

    return 0;
}
```

Suppose the program is running and it selects the number 35 as the secret target. Sample input/output for this program might be:

```
Enter your guess: 25
Guess is too low.
Enter your guess: 40
Guess is too high.
Enter your guess: 32
Guess is too low.
Enter your guess: 36
Guess is too high.
Enter your guess: 35
You win! Answer is 35
```

If you have a sense for what the right strategy is, you should be able to get the right number in far fewer than 50 attempts. You should even be able to do it in considerably fewer than 25 steps. Can you figure out what is the maximum number of guesses you will ever, ever need? In other words, what is the minimum of the maximum? How many guesses do you need to guarantee that you can get the answer—assuming you have an optimal strategy that makes this number as small as possible? There is, in fact, a precise answer.

How It Works

This program is fairly simple. The basic idea is easy enough to grasp: "Generate a random number. Then, prompt the user until he or she either guesses this number or enters 0." In addition, the program tells the user whether the guess is too high or low, as appropriate.

First, however, the program needs to choose a new random number, held in secret, each and every time it is run; otherwise, the game would be more boring than watching paint dry. But trying to guess a random, and unknown, number should keep it interesting. Generally speaking, the following lines of code should be included in every program that needs random numbers:

```
#include <cstdlib>    // Supports rand and srand.
#include <ctime>      // Supports time function.
...
srand(time(nullptr)); // Set random seed.
```

The first two lines, the **#include** directives, bring in the appropriate declarations for functions you'll be calling. The call to **srand**, setting a random seed, must be made from within a function, such as **main**, but it need only be done once in any program that is going to generate one or more random numbers.

The next line of code, executed from within **main**, actually generates the random number:

```
target = rand() % 50 + 1;  // Get random 1 to 50.
```

Let's break this down. The call to **rand** generates a number that could be anywhere in the unsigned **int** range, which is pretty large! The remainder, or modulus, operator (%) has the effect of dividing by 50 and then returning the remainder. The result must be in the range 0 to 49 inclusive.

Adding 1 to this number then produces a number in the range 1 to 50, which is what we wanted.

C++14 ▶ The C++14 specification includes new-and-improved randomization functions in the Standard Template Library. These functions don't save you from having to do the work of setting a seed, but they do provide a more flexible choice of ranges, random-number generators ("engines"), and probability distributions. However, for the simplest applications, the standard old-style randomization functions featured here work just fine.

The main loop itself prompts for a choice by the user. It then reacts to this choice by reporting whether the guess is too high, too low, or just right, and finally continues or exits depending on the value of a Boolean variable, do_more:

```
do {
    cout << "Enter your guess: ";
    cin >> n;

    // Respond to this choice...

} while (do_more);
```

There's a subtlety to this code that sometimes even experienced programmers sometimes fail to understand. When testing a Boolean variable, it isn't necessary to compare it explicitly to **true**.

Instead, you could write the following—and I know at least one programmer with many years of experience at Microsoft who might have done just that. The part that's different here is shown in bold.

```
do {
    cout << "Enter your guess: ";
    cin >> n;

    // Respond to this choice...

} while (do_more == true);
```

Why is this test, "do_more == true", unnecessary (legal, but unnecessary)? Think about it: The result of a comparison such as "do_more == true" is either **true** or **false**. But here we can assume that do_more *already* evaluates **true** or **false** precisely because it's a Boolean value! Therefore, this test has the effect of converting a Boolean value into the *same* Boolean value, which is pointless.

What if you want to reverse the true/false condition, that is, test for the condition that do_more is false? You can certainly do it this way:

```
do_more == false
```

This is true if do_more is false, and vice-versa. But a simpler way to do the same thing is to use the logical NOT operator (!).

```
! do_more
```

This means "NOT do_more," which is true if do_more is false, and vice-versa.

Note ▶ In the very oldest versions of the C language, and even in some of the oldest C++ compilers, the **bool** type was not supported, and programmers had to use integer variables instead, using 1 for true and 0 for false. However, unless you're living in the Stone Age, this is no longer the case and your compiler should support the **bool** type.

Optimizing the Code

If you're working with Microsoft Visual Studio, you may have noticed that the call to **srand** causes the environment to generate a warning error. This error (which really poses no possible danger in this situation) is about possible "loss of information" because data of type **time_t** (a long integer) is assigned to a function that takes data of type **unsigned int**.

Because both fields are integers and because the positive or negative sign makes no difference here, you can actually ignore this warning. Many programmers, however, like to write their code in such a way that all warning errors are prevented.

If you really want to get rid of that warning message, the solution is to use a cast. The simplest cast is the old-style C cast.

```
srand((unsigned int) time(nullptr));
```

The recommended and preferred style, however, is to use one of the new casting operators, **static_cast**. This syntax is verbose and ugly; however, there are reasons why **static_cast** is preferred. See Appendix A, "Operators," for more information on casts.

```
srand(static_cast<unsigned int>(time(nullptr)));
```

The general syntax for **static_cast** is the following:

```
static_cast<type>(expression)
```

The action takes the specific *expression* and converts it to a numerically equivalent expression having the specified *type*. Even if the compiler already knows how to do this (for example, in assigning a signed integer type to an unsigned integer type), the effect is to suppress warning errors.

EXERCISES

Exercise 3.2.1. Revise the program so that it uses a while loop. You can either continue to use the do_more variable, or you can instead do away with this variable and use the **break** keyword to exit as appropriate.

Exercise 3.2.2. After you've played the game a few times (or even if you haven't), you should have some idea of the optimum player's strategy. Can you summarize this strategy in pseudocode?

Exercise 3.2.3. Assuming that you've completed Exercise 3.2.2, you should be able to write a program that implements the optimum strategy. Assume that you, the user, have picked a number in secret. The program should go into a loop in which it asks YOU (the user) whether the guess is too high, too low, or just right. The program should then use the high/low answer to refine its guess and make another. Note that you are on your honor to answer honestly. Tip: To make the program easy to write, use an arbitrary system of numbers to signal the answer: 1=too high, 2=too low, 3=success. This should be printed in the prompt:

```
Tell me how I did. 1=too high, 2=too low, 3=success.
```

Exercise 3.2.4. This exercise is more of a math question than a programming question, but it's still interesting. For any game in which there are n possible choices, what is the minimum number of guesses needed to ensure success in the game? (For example, if 7 is this number, it would mean that by following optimum strategy, you'll never need more than 7 guesses.)

The switch-case Statement

As with the **do-while** loop, the **switch-case** control structure isn't strictly necessary. You can write any number of useful programs in C++ by relying on **if**, **else**, and **while**.

The **switch-case** statement is useful because so many sections of so many programs do nothing more than test for a sequence of values. Although **if** and **else** are sufficient for handling all such situations, the **switch-case** syntax can be used to write somewhat cleaner, easier-to-maintain code. (In addition, the compiler may be able to implement **switch-case** more efficiently than **if-else**, making it ever-so-slightly faster at run time.) Here's a common example:

```
if (n == 1) {
     cout << "one" << endl;
} else if (n == 2) {
     cout << "two" << endl;
} else if (n == 3) {
     cout << "three" << endl;
}
```

You can see what this is doing, because such programming code is pretty simple. It tests a number, n, and depending on the numeric value (1, 2, or 3), it prints a different word.

The following **switch-case** statement does the same thing, but I would suggest that it is somewhat easier for a human programmer to read. (As for the computer itself, it doesn't care.)

```
switch (n) {
     case 1:
          cout << "one" << endl;
          break;
     case 2:
          cout << "two" << endl;
          break;
     case 3:
          cout << "three" << endl;
          break;
}
```

Although this version of the code is cleaner and easier to read (or at least I'd contend it is), it does require more lines and it also requires the use of **break** statements—except in the case where you want control to "fall through" to a case below it, which is rare.

The **switch** statement has this general syntax:

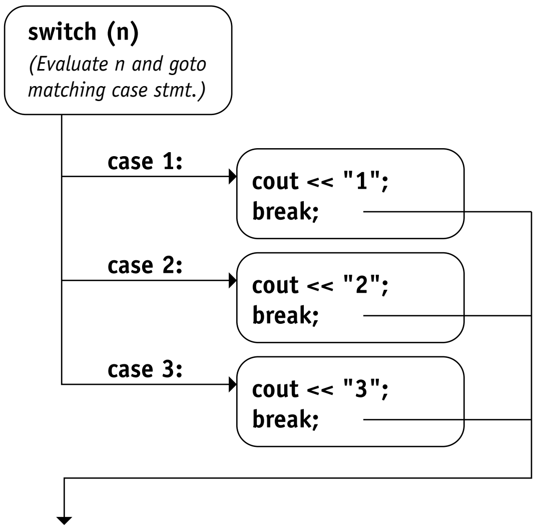
Keyword

```
switch(value) {
     statements
}
```

Here's how the **switch-case** statement works:

1. It evaluates the expression inside the parentheses. This value should have some integer or single-character type.
2. It then jumps to the line with the matching **case** label, which must be a constant.
3. Program execution then continues normally, but will break out of the enclosing **switch** block if the **break** keyword is encountered.

This creates a flow of control that looks like this:



This looks pretty simple. But the **switch-case** statement has some subtle features. First, you can optionally have a statement labeled **default**. This is the statement that control will jump to if none of the cases match the target value. This, in effect, is a "None of the Above" case. For example:

```
default:
     cout << "none of the above" << endl;
     break;
```

The **break** statement in this particular case is largely unnecessary, but some programmers put it in anyway as a matter of style.

What happens if you don't include the **break** statement? The answer is that one case would just "fall through" into the case below it. There are some unusual situations in which you might want to do that, but usually you will want to break.

Another subtlety is that a labeled statement has the following form:

label: statement

The **case** and **default** labels are just special cases of label names. Because a labeled statement is *itself* a statement, it's legal to have a statement that has many labels. For example:

```
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
     cout << "Char. Is a vowel.";
     break;
```

Example 3.3. *Print a Number*

Although computers deal in simple numbers, they need to format those numbers for presentation to humans. The most sophisticated example is that of computerized phone systems, which change numerical amounts into spoken words.

We're not going to do anything quite that advanced, but we can do the written equivalent: printing out numbers in a natural language (English). The basic logic is the same as that used for phone systems.

What the following application does is take a numeric amount from 20 to 99 and print it out in English—for example printing out the digits 53 as "fifty three."

printnum.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int  n = 0;

    cout << "Enter a number from 20 to 99";
    cin >> n;
    int tens_digits = n / 10;
    int units_digits = n % 10;
```

printnum.cpp, cont.

```
      switch(tens_digits) {
        case 2: cout << "twenty "; break;
        case 3: cout << "thirty "; break;
        case 4: cout << "forty "; break;
        case 5: cout << "fifty "; break;
        case 6: cout << "sixty "; break;
        case 7: cout << "seventy "; break;
        case 8: cout << "eighty "; break;
        case 9: cout << "ninety "; break;
      }
      switch(units_digits) {
        case 1: cout << "one" << endl; break;
        case 2: cout << "two" << endl; break;
        case 3: cout << "three" << endl; break;
        case 4: cout << "four" << endl; break;
        case 5: cout << "five" << endl; break;
        case 6: cout << "six" << endl; break;
        case 7: cout << "seven" << endl; break;
        case 8: cout << "eight" << endl; break;
        case 9: cout << "nine" << endl; break;
      }
  }
```

How It Works

If you've programmed before in another computer language, you might object that this example could be made much more efficient with something called an "array." That's absolutely true, and I'll get to that efficiency improvement in Chapter 6, "Arrays: All in a Row...." But assume for now that **switch-case** is the best we have.

To understand how this program works, it's necessary to review the division (/) and remainder (%) operators. The division operator, when applied to two integers, produces an integer result, rounded down. So, for example, let's suppose the user enters the number 49. The first thing the program does is extract the tens digit as follows:

```
49 /10
```

If the program were, instead, working with floating-point data (as in the expression 49.0 / 10.0), the answer would be 4.9, which, if rounded, becomes 5.0. But only integers are involved here, so the result is rounded down to 4.

The remainder operator, %, works only on integers. Remember that it produces the *amount left over* from division. Therefore, the following expression produces the value 9:

```
49 % 10
```

In summary, then, the following lines of code provide an efficient way to break a two-digit number down into its individual digits, 4 and 9:

```
int tens_digits = n / 10;
int units_digits = n % 10;
```

After extracting these two numbers, the program then uses them in a **switch-case** block and, as a result, prints the following:

```
forty nine
```

Is this an impressive result? Maybe not so much. But consider that a computerized phone system would use this same logic to select the sounds to pronounce "forty-nine" aloud.

EXERCISES

Exercise 3.3.1. The program in Example 3.3 (the number-printing program) is more useful if it allows repeated use rather than requiring you to restart it every time. (Actually that's true of most programs.) Therefore, place the bulk of the program in a do-while loop that repeats until the user enters 0, at which point it exits.

Exercise 3.3.2. Revise the program so that it can handle numbers in the range 0 to 9. This should be easy.

Exercise 3.3.3. Revise the program so that it can handle numbers in the range 11 to 19. This will require a good deal more work than Exercise 3.3.2 does, because you have to account for all the "teen" words.

Exercise 3.3.4. Extend the program so that it handles values as high as 999. (Hint: you'll need to complete Exercise 3.3.3 first, or the range of acceptable values will have many "holes" in it.)

Chapter 3 *Summary*

Here are the main points of Chapter 3:

- The **do-while** loop is similar to the **while** loop except for one important detail: with the **do-while** loop, the body of the loop is guaranteed to be executed at least once.

```
do statement
while (condition);
```

- As with other control structures, you can replace the statement placeholder with a compound statement, also called a block. The effect is to place the body of the line inside braces.

```
do {
     statements
} while (condition);
```

- A common way to control a loop is by using a loop-control variable declared Boolean (**bool** type), which is then set to **true** or **false** as appropriate. Remember that such a value does not need to be compared to **true** but can be tested directly.
- You can, in effect, test such a Boolean value to false by using the logical NOT operator (!) to reverse its true/false meaning.
- To generate random numbers in a C++ program, include both <cstdlib>, which supports the srand and rand functions, and <ctime>, which supports the time function.

```
#include <cstdlib>
#include <ctime>
```

- The next step in generating random numbers is to set a random-number seed. This is necessary so that each time you run the program, you get a different series of "random" (actually pseudorandom) numbers.

```
srand(ctime(nullptr));
```

- Thereafter, you can generate the next random number in the sequence by calling **rand**. This produces a number in the unsigned int range. By applying the remainder operator (%), you can get a random number from 0 to n – 1:

```
cout << rand() % n; // Print random 0 to n - 1
```

- The switch-case statement is an alternative to repeated if and else if clauses, in situations where a single value is tested over and over. Such an example might be:

```
if (n == 1) {
     cout << "1";
} else if (n == 2) {
     cout << "2";
} else if (n == 3) {
     cout << "3";
}
```

- Such code could be replaced with a nearly equivalent switch-case statement.

```
switch(n) {
case 1: cout << "1"; break;
case 2: cout << "2"; break;
case 3: cout << "3"; break;
}
```

- Syntactically, a **switch-case** statement (or rather, a **switch-case** block) has the following syntax:

```
switch (value) {
    statements
}
```

- This block does the following: First, it evaluates the value. It then transfers control to the statement whose **case** label, if any, matches this value. If no such label matches the value, then control is transferred to a statement labeled **default**, if there is such a statement.
- Control then proceeds in a normal linear fashion until a **break** statement is encountered, transferring control to the end of the **switch-case** block.
- The addition-assignment operator (+=) provides a concise way to add a value to a variable. There is also a subtraction-assignment operator. Here are some examples:

```
n += 50;                  // n = n + 50
n -= 25;                  // n = n - 25
```

4 The Handy, All-Purpose "for" Statement

Some tasks are so common that C++ provides special syntax just to represent them with fewer keystrokes: the increment operator (++) is one example. Because adding 1 is so common, C++ provides the increment operator to save space, although many other languages make do without it.

```
++n;    // Add 1 to n.
```

Subtraction by one is common as well, so C++ also provides the decrement operator, used in the expressions −−n and n−−.

Another example is the C++ **for** statement. Its only purpose is to make certain kinds of loops more concise. However, this turns out to be so useful that programmers rely on it heavily and I use it throughout the rest of this book.

You'll find that once you use it a few times, the **for** statement will become second nature. Its most common use is "repeat a series of actions n times," but it's more versatile than that and you can use it in many different situations.

Loops Used for Counting

When you worked with **while** loops in Chapter 2, "Decisions, Decisions," you may have noticed the typical use of a loop is to count to a number, performing some action a specific number of times. For example:

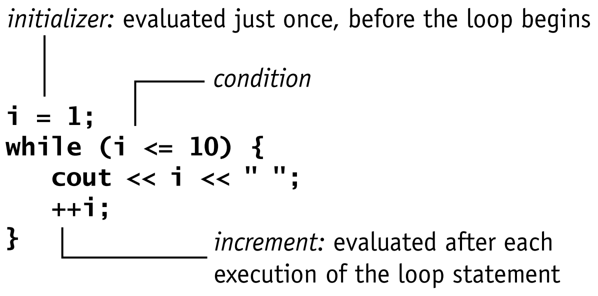
```
i = 1;
while(i <= 10) {
    cout << i << " ";
    ++i;
}
```

This code basically just prints numbers from 1 to 10. The loop variable gets an initial value of 1 and then is incremented each time through the loop. You can summarize what happens this way:

1. Set i to 1.
2. Perform the loop action.
3. Set i to 2.
4. Perform the loop action.
5. Set i to 3.
6. Perform the loop action.
7. Continue in this manner up to, and including, setting i to 10.

In other words, perform the loop 10 times, each time giving a successively higher value to i. This produces the sequence 1, 2, 3, … 10. The body of the loop might do something like "print the number."

We can identify three such actions: *initialize* the loop counter; test the loop *condition*; and, if true, execute the statement and the *increment*. Then go back to step 2 until the condition is false.

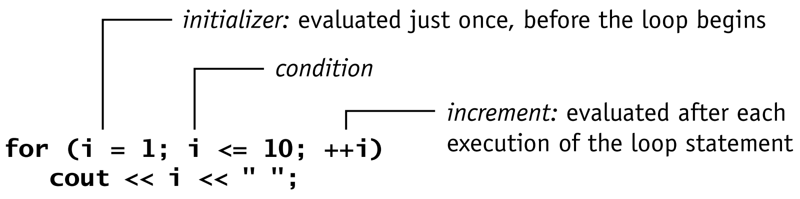

initializer: evaluated just once, before the loop begins
condition
increment: evaluated after each execution of the loop statement

```
i = 1;
while (i <= 10) {
   cout << i << " ";
   ++i;
}
```

It would be helpful to have a way to express these actions in one succinct statement. Then it would be easy to write a loop that counts to 10.

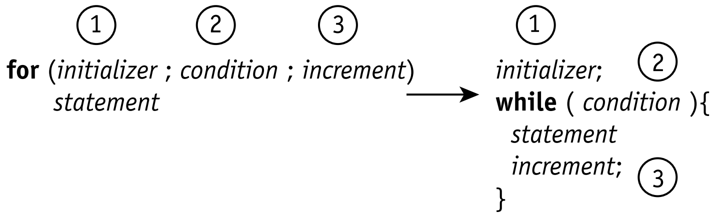
Introducing the "for" Loop

The **for** statement provides a mechanism that lets you specify the *initializer*, *condition*, and *increment* in one compact line.


initializer: evaluated just once, before the loop begins
condition
increment: evaluated after each execution of the loop statement

```
for (i = 1; i <= 10; ++i)
   cout << i << " ";
```

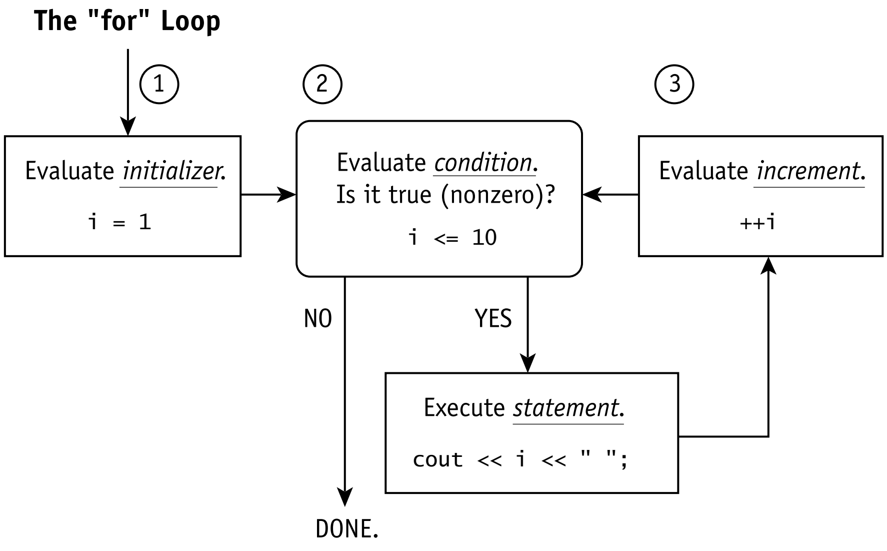
This is obviously more concise. All the settings that control the loop’s operation are placed between parentheses. More formally, here is the syntax of the **for** statement, along with the equivalent **while** loop:



```
for (initializer ; condition ; increment)
   statement
```

```
initializer;
while ( condition ){
  statement
  increment;
}
```

Syntax diagrams are all well and good, but sometimes it’s more helpful to look at a flowchart. It’s easy to see from the following chart that the *initializer* is evaluated one time, after which the **for** statement sets up an enhanced version of the **while** loop by evaluating *increment* before cycling. Remember, this prints all numbers from 1 to 10:



As with the other control structures, the loop statement can be a compound statement (or *block*). This syntax follows from the rule that wherever you can use a statement in C++, you can also use a block.

I’ve found that a structure like **for** can still be fuzzy until you look at a lot of examples. That’s the purpose of the next section.

C++14 ▶ With C++11 and later, C++ provides a new version of the **for** keyword that automatically works on all members of a collection, similar to “for each” in some other languages. To use this version, however, you need to understand arrays and containers. I present this new version of **for** in Chapter 17, “New Features of C++14.”

A Wealth of Examples

Let’s start with a slight variation of the example you’ve already seen. The loop variable, i, is initialized to 1 (i = 1), and the loop continues while the condition (i <= 5) is true. This is the same as the earlier example, except that the loop counts only to 5.

```
for(i = 1; i <= 5; ++i) {
    cout << i << " ";
}
```

This produces the following output:

```
1 2 3 4 5
```

The next example runs from 10 to 20 rather than 1 to 5:

```
for(i = 10; i <= 20; ++i) {
    cout << i << " ";
}
```

This produces the following output:

```
10 11 12 13 14 15 16 17 18 19 20
```

Here, the *initializer* is i = 10, and the *condition* is i <= 20. These expressions determine the initial and terminal settings of the loop. (The condition terminates the loop when it is no longer true; therefore, the highest value of i will be 20 in this case.)

These settings do not have to be constants. In this next example, they are determined by variables. The loop counts from n1 to n2.

```
n1 = 32;
n2 = 38;
for (i = n1; i <= n2; ++i) {
    cout << i << " ";
}
```

This produces the following output:

```
32 33 34 35 36 37 38
```

The *increment* expression can be any expression at all; it does not have to be ++i. You can just as easily use −−i, which causes the **for** loop to count downward. Note the use of greater than or equal to (>=) in the condition in this example.

```
for(i = 10; i >= 1; --i) {
    cout << i << " ";
}
```

This produces the following output:

```
10 9 8 7 6 5 4 3 2 1
```

The **for** statement is highly flexible. By changing the *increment* expression, you can count by 2 rather than by 1.

```
for(i = 1; i <= 11; i = i + 2) {
    cout << i << " ";
}
```

This produces the following output:

```
1 3 5 7 9 11
```

As a final example, you don't have to use i as the loop variable. Here's an example that uses a loop variable named j:

```
for(j = 1; j <= 5; ++j) {
    cout << j * 2 << " ";
}
```

This produces the following output:

```
2 4 6 8 10
```

Note that in this case, the loop statement prints j * 2, which is why this loop prints even numbers.

Interlude

Does "for" Always Behave Like "while"?

I said that a **for** loop is a special case of **while** and performs exactly as the corresponding **while** loop would. That's *almost* true. There is one minor exception, which—for the purposes of this entire book and 99 percent of all the code you will ever write—will likely make no difference. The exception involves the **continue** keyword. You can use this keyword in a loop, placing it in its own statement, to say "Advance immediately to the next cycle of the loop."

```
continue;
```

This is a kind of "Advance directly to Go" statement. It doesn't break out of the loop (which the **break** keyword does); it just speeds things up.

The difference in behavior is this: In a **while** loop, the **continue** statement neglects to execute the increment (++i) before advancing to the next cycle of the loop. In a **for** loop, the **continue** statement does execute the increment before advancing. This second behavior would usually be the behavior you'd want, and that provides one more reason why **for** is useful.

Example 4.1. *Printing 1 to N with "for"*

Now we'll apply the **for** statement in a complete program. This example does the same thing as Example 2.2 on page 43: It prints all the numbers from 1 to n. But this version is more compact.

count2.cpp

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

int main()
{
    int  n = 0;
    int  i = 0;   // Loop counter in "for" statement.

    // Get num from the keyboard and initialize i.

    cout << "Enter a number and press ENTER: ";
    cin >> n;
```

count2.cpp, cont.

```
    for (i = 1; i <= n; ++i){   // For i = 1 to n
        cout << i << " ";       //    Print i.
    }
    return 0;
}
```

When you run the program, it counts to the specified number. For example, if the user inputs 9, the program prints the following:

```
1 2 3 4 5 6 7 8 9
```

How It Works

This example features a simple **for** loop. The loop condition in this example uses n, a number that the program gets from the user.

```
cout << "Enter a number and press ENTER: ";
cin >> n;
```

The loop prints numbers from 1 to n, where n is the number entered.

```
for (i = 1; i <= n; ++i){   // For i = 1 to n
    cout << i << " ";       //  Print i.
}
```

To review:

- The expression "i = 1" is the *initializer* expression; it's evaluated just once, before the loop is executed. This initial value of i is therefore 1.
- The expression "i <= n" is the *condition*. This is checked before each loop cycle to see whether the loop should continue. If, for example, n is 9, the loop terminates when i reaches 10, so the loop is not executed for the case of i equal to 10.
- The expression "++i" is the *increment* expression, which is evaluated after each execution of the loop statement. This drives the loop by adding 1 to i each time.

The program logic is therefore equivalent to:

```
Set i to 1.
While i is less than or equal to n,
    Print i,
    Add 1 to i.
```

EXERCISES

Exercise 4.1.1. Use the **for** statement in a program that prints all the numbers from n1 to n2, where n1 and n2 are two numbers specified by the user. (Hint: You'll need to prompt for the two values; then, inside the **for** statement, initialize i to n1, and use n2 in the loop condition.)

Exercise 4.1.2. Rewrite the example so that it prints all the numbers from n to 1 in reverse order. For example, the user enters 5, and the program prints 5 4 3 2 1. (Hint: In the **for** loop, initialize i to n, use the condition i >= 1, and subtract 1 from i in the increment step.)

Exercise 4.1.3. Write a program that prints all numbers from 1 to n, but prints only even numbers or only odd numbers. Each number printed will be 2 higher than the last.

Declaring Loop Variables “On the Fly”

One of the benefits of the **for** statement is that you can efficiently use it to declare a variable that has scope local to the loop itself. The variable is declared “on the fly,” for the explicit use of the **for** loop. For example:

```
for (int i = 1; i <= n; ++i){ // For i = 1 to n
    cout << i << " ";         //  Print i.
}
```

Here, i is declared inside the *initializer* expression of the **for** statement. The variable i becomes local—not just to the function, but to the loop itself—so changes made to i within the loop do not affect any copies of i declared outside the loop.

If you use this technique, you don't need to declare i separately from the loop, of course. You can rewrite Example 4.1 this way:

count3.cpp

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

int main()
{
    int  n = 0;
```

count3.cpp, cont.

```
    // Get a number from the keyboard.

        cout << "Enter a number and press ENTER: ";
        cin >> n;

        for (int i = 1; i <= n; ++i){ // For i = 1 to n
            cout << i << " ";         //    Print i.
        }
        return 0;
    }
```

Example 4.2. *Prime-Number Test with "for"*

Now let's return to the prime-number example of Example 2.3 (page 53), but write that program using a **for** loop rather than **while**. This example determines whether a number input is a prime number. (Remember, a number is prime if it is evenly divisible only by itself and 1.)

The same basic logic is involved as in Example 2.3. Here is pseudocode for a prime-number test:

```
Set i to 2.
While i is less than or equal to the square root of n,
    If i divides evenly into n,
        n is not prime.
Add 1 to i.
```

The **for**-loop version uses exactly the same approach; when compiled, it carries out the same instructions as the **while** loop. However, because the essential nature of a **for** loop is to perform counting—in this case counting from 2 to the square root of n—we can think of it a little differently. The same things happen, but conceptually this approach is a little simpler:

```
For all whole numbers from 2 up to the square root of n,
    If i divides evenly into n,
        n is not prime.
```

Here's the complete program for testing whether a number is a prime number. Again, this is a version of the program described in Example 4.2, so most of it should look familiar.

prime2.cpp

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int  n = 0;   // Number to test for prime-ness
    bool is_prime = true; // Boolean flag; assume true
                          //   until proven otherwise

    // Get a number from the keyboard.

    cout << "Enter a number and press ENTER: ";
    cin >> n;

    // Test for prime by checking for divisibility
    //  by all whole numbers from 2 to sqrt(n).

    for (int i = 2; i <= sqrt(n); ++i) {
        if (n % i == 0) {
            is_prime = false;
        }
    }

    // Print results

    if (is_prime) {
        cout << "Number is prime." << endl;
    } else {
        cout << "Number is not prime." << endl;
    }
    return 0;
}
```

When the program is run, if the user enters 23, the program prints the following:

```
Number is prime.
```

How It Works

The beginning of the program uses **#include** directives to enable needed support from the C++ library. The C++ math library is used here because the program needs to call the **sqrt** function to get the square root of a number.

```
#include <iostream>
#include <cmath>
```

The rest of the program defines the main (and so far, only) function. The first thing that **main** does is declare the variables the program will use. (Note: The loop variable i is declared in the **for** loop itself.)

```
int  n = 0;   // Number to test for prime-ness
bool is_prime = true; // Boolean flag; assume
                      //  true until proven
                      //  otherwise
```

The purpose of is_prime is to store a value of **true** or **false**.

If the program can't find a divisor for n, it should conclude that the number is prime. Therefore, is_prime gets a default setting of **true**. In other words, we'll consider a number prime until that condition is proven false.

The heart of the program is the **for** loop that performs the prime-number test. As I described in Chapter 2, it's necessary to test for divisors only up to the square root of n. If a divisor is not found by then, the number is prime.

Remember that the expression n%i divides n by i and returns the remainder. A remainder of 0 means that i divides into n perfectly, which would mean that n is not prime.

```
for (int i = 2; i <= sqrt(n); ++i) {
    if (n % i == 0) {
        is_prime = false;
    }
}
```

Remember how a **for** loop works: the first expression in parentheses, i = 2, is the *initializer*; the second expression, i <= sqrt(n), is the *condition*; and the last is the *increment*. Let's review:

- The *initializer*, i = 2, is executed only once, before any cycles of the loop.
- The *condition*, i <= sqrt(n), is evaluated at the top of the loop. A false value (i not less than or equal to sqrt(n)) causes the loop to exit immediately. A true value causes the loop to keep going.

- The *increment*, ++i, is executed after the body of the loop.

This **for** loop is therefore equivalent to the following:

```
int i = 2;                  // Initializer: Do once.
while (i <= sqrt(n)) {     // Is i less than or
                           //   equal to sqrt(n)?
    if (n % i == 0) {       // If so, do all this...
        is_prime = false;
    }
    ++i;                   // Inc. at bottom of loop.
}
```

EXERCISE

Example 4.2.1. Revise Example 4.2 to make it more optimal. When you consider the speed of today's microprocessors, it's unlikely you'll see a difference in execution, although if you attempt to test an extremely large number, say, more than a billion, you might see a difference. (By the way, good luck in finding a prime number in that range, if you're just looking for one by chance. Prime numbers become rarer as you get into larger values.) In any case, the following changes to code make the program more efficient for large numbers:

- Calculate the square root of n only once by declaring a variable square_root_of_n and determining its value before entering the **for** loop. This variable should be a **double** variable.
- Once a divisor of n is found, you don't need to look for any more divisors. Therefore, in the **if** statement inside the loop, add a **break** statement (breaking out of the loop) after setting is_prime to false.

Comparative Languages 101: The Basic "For" Statement

If you've programmed in Basic or FORTRAN, you've seen statements much like the C++ **for** statement, whose purpose is to count from one number to another. For example, this Basic loop prints numbers from 1 to 10:

```
For i = 1 To 10
    Print i
Next i
```

The Basic "For" statement has the advantage of clarity and ease of use. It admittedly takes fewer keystrokes to use than C++'s **for**. But despite that, the advantage of the C++ **for** statement is its flexibility.

One way in which the C++ **for** statement is so much more flexible is that you can use it with any three valid C++ expressions. The condition (the middle expression) doesn't even have to be a logical expression such as i < n, although it ought to be. For the purposes of evaluating a condition in an **if**, **while**, or **for** statement, any nonzero value is considered "true."

The **for** statement does not require you to use all three expressions (*initializer*, *condition*, and *increment*). If *initializer* or *increment* is missing, it's ignored. If the *condition* is omitted, it's considered "true" by default, setting up an infinite loop.

```
for(;;) {
  // Infinite loop!
}
```

An infinite loop can be a bad thing—unless you have some way to break out of it, for example, by using the **break** statement. In the following example, the user can break out of the loop by entering the value 0:

```
for (;;) {
    // Do some stuff...

    cout << "Enter a number and press ENTER: ";
    cin >> n;
    if (n == 0) {
        break;
    }

    // Do some more stuff...
}
```

Chapter 4 *Summary*

Here are the main points of Chapter 4:

- The purpose of a **for** statement is usually to repeat an action while counting to a particular value. The statement has the following syntax:

```
for (initializer; condition; increment)
    statement
```

This is equivalent to the following **while** loop:

```
initializer;
while (condition) {
    statement
    increment;
```

- A **for** loop behaves exactly like its **while**-loop counterpart, with one exception: In a **for** loop, the **continue** statement increments the loop variable before advancing to the top of the next loop cycle.
- As with other kinds of control structures, by using opening and closing braces ({}) you can always use a compound statement, or block, with **for**. There are many programmers who recommend this style because it's easier to go back and add statements later, and because in complex programs, the braces can help clarify program structure.

```
for (initializer; condition; increment) {
    statement
}
```

- A variable such as i in the following example is called a *loop variable*:

```
for (i = 1; i <= 10; ++i) {
    cout << i << " ";
}
```

- In the *initializer* expression, you can declare a variable "on the fly." This declaration gives the variable scope local to the **for** loop itself, meaning that changes to the variable don't affect variables of the same name outside the loop.

```
for (int i = 1; i <= 10; ++i) {
    cout << i << " ";
}
```

- As with **if** and **while**, the loop condition of a **for** statement can be any valid C++ expression that evaluates to a true/false value or a numeric value; any nonzero value is considered "true."
- You can omit any and all of the three expressions inside the parentheses of the **for** statement (*initializer*, *condition*, *increment*). If the condition is omitted, the loop is executed unconditionally. (In other words, the loop is infinite.) Remember to use a **break** statement to get out of it.

```
for (;;) {

    // Infinite loop!

}
```

5 Functions: Many Are Called

Programmers and computer scientists talk about creating reusable code, the Holy Grail of software development. There are many tools designed with this goal in mind. But the function (called a *procedure* or *subroutine* in other languages) is the most basic tool of all.

The function—which in C++ may or may not return a value—is based on this simple idea: once someone figures out how to accomplish a specific task, such as calculating a square root, you shouldn't have to figure it out again.

So, instead of writing the same lines of programming code over and over, you write the function *just once*, and then execute it whenever you want to perform the desired task. This is known as "calling" a function.

In short, functions make your life easier.

The Concept of Function

If you've followed the book up until this point, you've already seen the use of a function—the sqrt function—that takes a single number as input and returns a result.

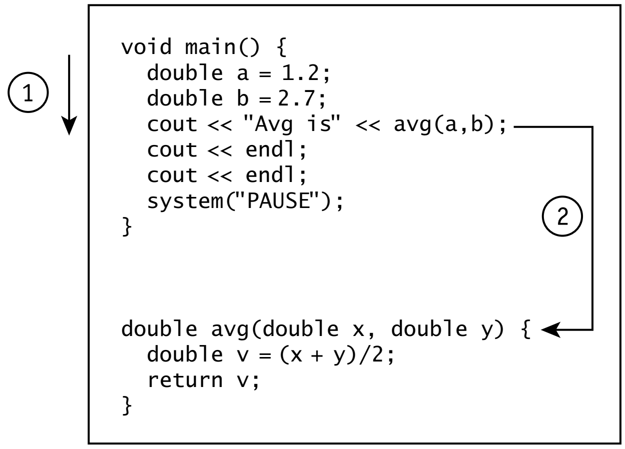
```
double sqrt_of_n = sqrt(n);
```

This is not far removed from the mathematical concept of function. A function takes zero or more inputs, called *arguments*, and returns an output, called a *return value*. Here's another example—this function takes two inputs and returns their average:

```
cout << avg(1.0, 4.0);
```

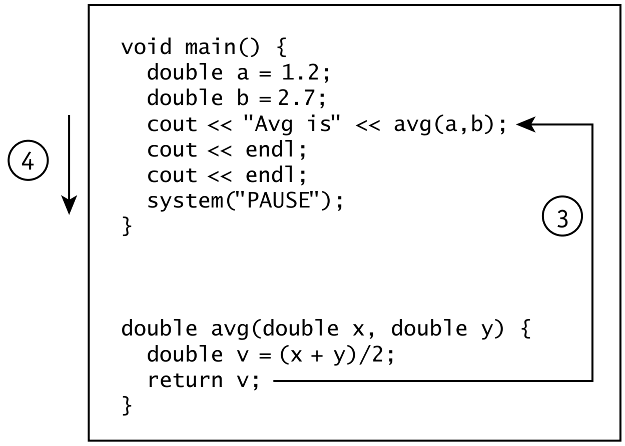
Once a function is written, you can call it any number of times. By *calling* a function, you transfer execution of the program to the function-definition code, which runs until it is finished or until it encounters a **return** statement; execution is then transferred back to the caller.

This may sound strange if you're not used to it. It's easy to see in a diagram. In the following example, the program 1) runs normally until it calls the function avg, passing the arguments a and b, and 2) as a result, the program transfers execution to avg. (The values of a and b are passed to x and y, respectively.)



```
void main() {
  double a = 1.2;
  double b = 2.7;
  cout << "Avg is" << avg(a,b);
  cout << endl;
  cout << endl;
  system("PAUSE");
}


double avg(double x, double y) {
  double v = (x + y)/2;
  return v;
}
```

The function runs until 3) it encounters **return**, which causes execution to return to the caller of the function. The function then prints the value that was returned. Finally, 4) execution resumes normally inside **main**, and the program continues until it ends.



```
void main() {
  double a = 1.2;
  double b = 2.7;
  cout << "Avg is" << avg(a,b);
  cout << endl;
  cout << endl;
  system("PAUSE");
}


double avg(double x, double y) {
  double v = (x + y)/2;
  return v;
}
```

Note that in a program, only **main** is guaranteed to be executed. Other functions run only when called. But there are many ways a function can be called. For example, **main** can call a function A, which in turn calls functions B and C, which in turn call D.

The Basics of Using Functions

I recommend the following approach for creating and calling user-defined functions:

1 At the beginning of your program, declare the function.

2 Somewhere in your program, *define* the function.

3 Other functions can then call the function.

Step 1: Declare (Prototype) the Function

Although not strictly necessary in all cases, you should usually prototype your functions—except for **main**—at the beginning of a program. C++ requires that a function be declared before being used: such a declaration can be either a prototype or a definition (which is Step 2).

If you're lazy and want to avoid work, you can define your functions in the reverse order in which they were called, and you might be able to get by without prototypes. But this strategy fails as soon as you have two functions that call each other, which is more common than you'd think.

The use of prototypes frees you from the worry, "Did I define this function yet?" before you call the function.

A function *prototype* provides type information only. It has this syntax:

Key Syntax

```
return_type   function_name (argument_list);
```

The *return_type* is a data type indicating what kind of value the function returns (what it passes back). If the function does not return a value, use **void**.

The *argument_list* is a list of zero or more argument names—separated by commas if there are more than one—each preceded by the corresponding type. (Technically, you don't need the argument names in a prototype, but it is a good programming practice.) For example, the following statement declares a function named avg, which takes two arguments of type **double** and returns a **double** value:

```
double avg(double x, double y);
```

The *argument_list* may be empty, which indicates that it takes no arguments.

Step 2: Define the Function

Key Syntax

The function definition tells exactly what the function does. It uses this syntax:

```
return_type   function_name (argument_list) {
    statements
}
```

Much of this looks like a prototype. The only thing that's different is that the semicolon is replaced by zero or more statements between two braces ({}). The braces are required no matter how few statements you have. For example:

```
double avg(double x, double y) {
    return (x + y) / 2;
}
```

The **return** statement causes immediate exit and it specifies that the function returns the amount (x + y) / 2. Functions with no return value can still use the **return** statement, but only to exit early.

```
return;
```

Step 3: Call the Function

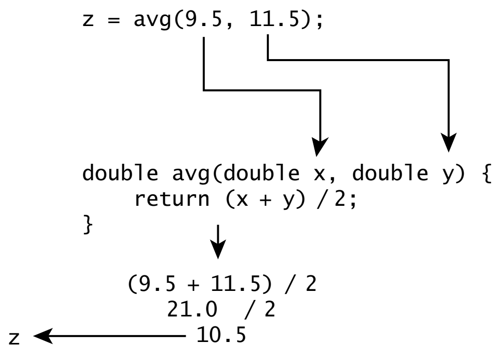
Once a function is defined, it can be used—or rather, *called*—any number of times, from any function. For example:

```
n = avg(9.5, 11.5);
n = avg(5, 25);
n = avg(27, 154.3);
```

A function call is an expression: as long as it returns a value other than **void**, it can be used inside a larger expression. For example:

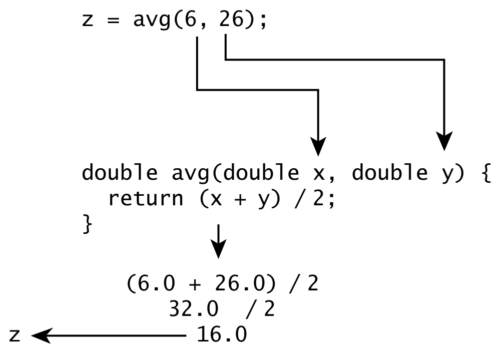
```
z = x + y + avg(a, b) + 25.3;
```

When the function is called, the values specified in the function call are passed to the function arguments. Here's how a call to the avg function works, with sample values 9.5 and 11.5 as input. These are *passed* to the function, as arguments. When the function returns, the value in this case is assigned to z.



```
z = avg(9.5, 11.5);

double avg(double x, double y) {
    return (x + y) / 2;
}

    (9.5 + 11.5) / 2
       21.0   / 2
z <-------- 10.5
```

Another call to the function might pass different values—in this case, 6 and 26. (Because these are integer values, they are implicitly converted, or *promoted*, to type **double**.)



```
z = avg(6, 26);

double avg(double x, double y) {
  return (x + y) / 2;
}

    (6.0 + 26.0) / 2
       32.0   / 2
z <-------- 16.0
```

Example 5.1. *The avg() Function*

This section shows a simple function call in the context of a complete program. It demonstrates all three steps: declare a function, define it, and call it.

avg.cpp

```
#include <iostream>
using namespace std;
// Function must be declared before being used.
double avg(double x, double y);
```

▼ *continued on next page*

avg.cpp, cont.

```
int main()
{
    double a = 0.0;
    double b = 0.0;
    cout << "Enter first number and press ENTER: ";
    cin >> a;
    cout << "Enter second number and press ENTER: ";
    cin >> b;

    // Call the function avg().
    cout << "Average is: " << avg(a, b) << endl;
    return 0;
}
// Average-number function definition
//
double avg(double x, double y) {
    return (x + y)/2;
}
```

How It Works

This code is a very simple program, but it demonstrates the three steps I outlined earlier:

1 *Declare* (that is, prototype) the function at the beginning of the program.

2 *Define* the function somewhere in the program.

3 *Call* the function from within another function (in this case, **main**).

Although function declarations (prototypes) can be placed anywhere in a program, you should almost always place them at the beginning. The general rule is that functions must be declared before being called. (They do not, however, have to be defined before being called, which makes it possible for two functions to call each other.)

```
double avg(double x, double y);
```

The function definition for the avg function is extremely simple, containing only one statement. In general, though, function definitions can contain as many statements as you want.

```
double avg(double x, double y) {
    return (x + y)/2;
}
```

The **main** function calls avg as part of a larger expression. The computed value (in this case, the average of the two inputs, a and b) is returned to this statement in **main**, which then prints the result.

```
cout << "Average is: " << avg(a, b) << endl;
```

Variation

Function, Call a Function!

A program can have any number of functions. For example, you could have two functions in addition to **main**, as in the following version of the program. Lines that are new or changed are in bold.

avg2.cpp

```
#include <iostream>
using namespace std;
// Functions must be declared before being used.
void print_results(double a, double b);
double avg(double x, double y);

int main()
{
    double a = 0.0;
    double b = 0.0;
    cout << "Enter first number and press ENTER: ";
    cin >> a;
    cout << "Enter second number and press ENTER: ";
    cin >> b;

    // Call the function pr_results().
    print_results(a, b);
    return 0;
}
// print_results function definition
//
void print_results(double a, double b) {
```

▼ continued on next page

avg2.cpp, cont.

```
    cout << "Average is: " << avg(a, b) << endl;
    }
    // Average-number function definition
    //
    double avg(double x, double y) {
        return (x + y)/2;
    }
```

This version is a little less efficient, but it illustrates an important principle: You are not limited to only one or two functions; you can have as many as you want. This program creates a flow of control as follows:

```
main() → print_results() → avg()
```

EXERCISES

Exercise 5.1.1. Write a program that defines and tests a factorial function. The factorial of a number is the product of all whole numbers from 1 to N. For example, the factorial of 5 is 1 * 2 * 3 * 4 * 5 = 120. (Hint: Use a **for** loop as described in Chapter 3.)

Exercise 5.1.2. Write a function named print_out that prints all the whole numbers from 1 to N. Test the function by placing it in a program that passes a number n to print_out, where this number is entered from the keyboard. The print_out function should have type **void**; it does not return a value. The function can be called with a simple statement:

```
print_out(n);
```

Example 5.2. *Prime-Number Function*

Chapter 2 included an example that was actually useful: determining whether a specified number was a prime number. We can also write the prime-number test as a function and call it repeatedly.

The following program uses the prime-number example from Chapters 2 and 3 but places the relevant C++ statements into their own function, prime.

prime2.cpp

```
#include <iostream>
#include <cmath>      // Include because sqrt is called.

using namespace std;

// Function must be declared before being used.
bool prime(int n);

int main()
{
   int n = 0;

   // Set up infinite loop; break if user enters 0.
   // Otherwise, evaluate n for prime-ness.

   while (true) {
      cout <<"Enter num (0 = exit) and press ENTER: ";
      cin >> n;
      if (n == 0) {        // If user entered 0, EXIT
         break;
      }
      if (prime(n)) {      // Call prime(i)
         cout << n << " is prime" << endl;
      } else {
         cout << n << " is not prime" << endl;
      }
   }
   return 0;
}

// Prime-number function. Test divisors from
//  2 to sqrt of n. Return false if a divisor
//  found; otherwise, return true.
bool prime(int n) {
    for (int i = 2; i <= sqrt(n); ++i) {
       if (n % i == 0) {   // If i divides n evenly,
          return false;    //  n is not prime.
       }
    }
    return true;   // If no divisor found, n is prime.
}
```

How It Works

As always, the program adheres to the pattern of 1) declaring function type information at the beginning of the program (*prototyping* the function), 2) defining the function somewhere in the program, and 3) calling the function.

The prototype says that the prime function takes an integer argument and returns a **bool** value, which will be either **true** or **false**. (Note: If you have a really old compiler, you may have to use the **int** type instead of **bool**.)

```
bool prime(int n);
```

The function definition is a variation on the prime-number code from Chapter 3, which used a **for** loop. If you compare the code here to Example 4.2 on page 93, you'll see only a few differences.

```
bool prime(int n) {
    for (int i = 2; i <= sqrt(n); ++i) {
       if (n % i == 0) {    // If i divides n evenly,
          return false;     //  n is not prime.
       }
    }
    return true;    // If no divisor found, n is prime.
}
```

Another difference is that instead of setting a Boolean variable, is_prime, this version returns a Boolean result. The logic here is as follows:

```
For all whole numbers from 2 to the square root of n,
    If n is evenly divisible by the loop variable (i),
        Return the value false immediately.
```

Remember that the modulus operator (%) carries out division and returns the remainder. If this remainder is 0, that means the second number divides the second evenly; in other words, it is a *divisor* or *factor* of the second number.

The action of the **return** statement here is key. This statement returns immediately, causing program execution to exit from the function and passing control back to **main**. There's no need to use **break** to get out of the loop.

The loop in the main function calls the prime function. The use of a **break** statement here provides an exit mechanism, so the loop isn't really infinite. As soon as the user enters 0, the loop terminates and the program ends. Here, I've put the exit lines in bold:

```
while (true) {
   cout <<"Enter num (0 = exit) and press ENTER: ";
```

```
            cin >> n;
            if (n == 0) {         // If user entered 0, EXIT
                break;
            } if (prime(n)) {     // Call prime(i)
                cout << n << " is prime" << endl;
            } else {
                cout << n << " is not prime" << endl;
            }
        }
```

The rest of the loop calls the prime function and prints the result of the prime-number test. Note that this function returns a true/false value, and so the call to prime(i) can be used as an if/else condition.

EXERCISES

Exercise 5.2.1. Optimize the prime-number function by calculating the square root of n only once during each function call. Declare a local variable sqrt_of_n of type **double**. (Hint: A variable is local if it is declared inside the function.) Then use this variable in the loop condition.

Exercise 5.2.2. Rewrite **main** so that it tests all the numbers from 2 to 20 and prints out the results, each on a separate line. (Hint: Use a **for** loop, with i running from 2 to 20.)

Exercise 5.2.3. Write a program that finds the first prime number greater than 1 billion (1,000,000,000).

Exercise 5.2.4. Write a program that lets the user enter any number n and then finds the first prime number larger than n.

Local and Global Variables

Nearly every programming language has a concept of local variable. As long as two functions mind their own data, as it were, they won't interfere with each other.

That's definitely a factor in the previous example (Example 5.2). Both **main** and prime have a local variable named i. If i were not local—that is, if it was shared between functions—then consider what could happen.

First, the **main** function executes prime as part of evaluating the **if** condition. Let's say that i has the value 24.

```
if (prime(i)) {
    cout << i << " is prime" << endl;
} else {
    cout << i << " is not prime" << endl;
}
```

The value 24 is passed to the prime function.

```
// Assume i is not declared here, but is global.
int prime(int n) {
    for (i = 2; i <= sqrt((double) n); ++i)
        if (n % i == 0) {
            return false;
        }
    }
    return true;  // If no divisor found, n is prime.
}
```

Look what this function does. It sets i to 2 and then tests it for divisibility against the number passed, 24. This test passes because 2 does divide into 24 evenly and the function returns. But i is now equal to 2 instead of 24.

Upon returning, the program executes

```
cout << i << " is not prime" << endl;
```

which prints the following:

```
2 is not prime
```

This is not what we wanted, since we were testing the number 24! So, to avoid this problem, declare variables local unless there is a good reason not to do so.

Is there ever a good reason to not make a variable local? Yes, although if you have a choice, it's better to go local because you want to avoid functions interfering with each other as much as possible.

You can declare global—that is, nonlocal—variables by declaring them outside of any function definition. It's usually best to put all global declarations near the beginning of the program, before the first function. A variable is recognized only from the point it is declared, to the end of the file.

For example, you could declare a global variable named status:

```
#include <iostream>
#include <cmath>
using namespace std;
int status = 0;
```

```
void main()
{
      //
}
```

Now, the variable named status may be accessed by any function. Because this variable is global, there is only one copy of it; if one function changes the value of status, this reflects the value of status that other functions see.

Interlude

Why Global Variables at All?

For the reasons shown in the previous section, global variables can be dangerous. Habitual use of global variables can cause shocks to a program because changes performed by one function cause unexpected effects in another.

But if they are so dangerous, why use them at all?

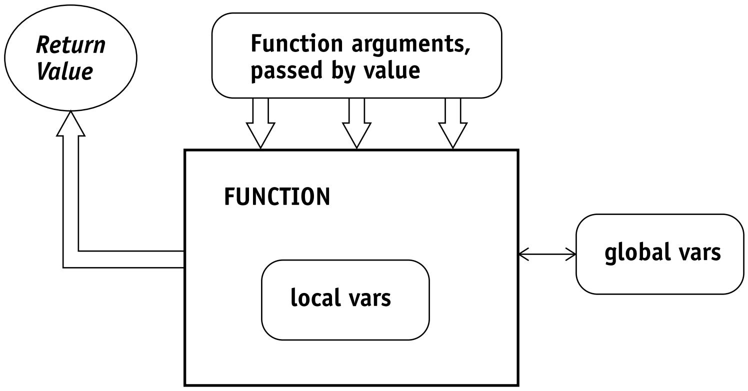
Well, they are often necessary, or nearly so. Global variables are sometimes the best way to communicate information among multiple functions; otherwise, you might need a long series of argument lists that transfer all the program information back and forth.

Beginning with Chapter 11, we'll work with classes, which provide an alternative, and generally superior, way for closely related functions to share data with each other: functions of the same class have access to private data that no one else does.

By default, C++ passes arguments by value, which means that a function gets its own copies of the data passed to it. Consequently, changes to the arguments have no effect outside the function; they are input-only data. The return value, in contrast, is output by the function.

This seems to create a situation in which the function can provide output to its caller only through its return value, or by modifying the value of global variables, which is sometimes acceptable but also has a downside (namely, it's undesirable to have too many global variables). Chapter 7, "Pointers: Data by Location," explains how to get around these limitations and use arguments to pass more than one value back.

For now, you can conceive of the data flow to and from functions this way, in which data flows to the function through its arguments and back through its return value.



Note that modification of global variables is another way for a function to change data that can affect the rest of the program. But in general, it's best to have as few global variables as possible.

Recursive Functions

So far, I've only shown the use of **main** calling other functions defined in the program, but in fact, any function can call any function. But can a function call itself?

Yes, it can. And as you'll see, it's not as crazy as it sounds. The technique of a function calling itself is called *recursion*. The obvious problem is the same as that for infinite loops: if a function calls itself, when does it ever stop? The problem is easily solved, however, by putting in some mechanism for stopping.

Remember the factorial function from Exercise 5.1.1 (page 106)? We can rewrite this as a recursive function:

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);  // RECURSION!
    }
}
```

For any number greater than 1, the factorial function issues a call to itself but with a lower number. Eventually, the function factorial(1) is called, and the cycle stops.

There is a literal *stack* of calls made to the function, each with a different argument for n, and now they start returning. The *stack* is a special area of memory maintained by the computer: It is a last-in-first-out (LIFO) mechanism that keeps track of information for all pending function calls. This includes arguments and local variables, if any.

You can picture how to call a factorial(4) this way:

$$\begin{tikzcd}
\texttt{factorial(4)} \arrow[d] & & & \\
\texttt{4 *} & \texttt{factorial(3)} \arrow[d] & & \\
& \texttt{3 *} & \texttt{factorial(2)} \arrow[d] & \\
& & \texttt{2 *} & \texttt{factorial(1)} \arrow[d] \\
& & & \texttt{1}
\end{tikzcd}$$

Many functions that use a **for** statement can be rewritten so they use recursion instead. But does it always make sense to use that approach?

No. The example here is not an ideal one, because it causes the program to store all the values 1 through n on the stack, rather than totaling them up directly in a loop. This approach is not efficient. The next section makes a better use of recursion.

However, this section does illustrate the two most important aspects of recursion. A recursive function, that is, a function that recalls itself, must do the following:

- Solve a general problem at level n by assuming the problem has already been solved for level n − 1.
- Specify at least one terminal condition, such as n = 1 or n = 0.

Example 5.3. *Prime Factorization*

The prime-number examples we've looked at so far are fine, but they have a limitation. They tell you, for example, that a number such as 12,001 is not prime, but they don't tell you anything more. Wouldn't it be more useful to know what numbers divide into 12,001?

It'd be more useful to generate the *prime factorization* for any requested number. This would show us exactly what prime numbers divide into that number. For example, if the number 36 is input, the output will be:

```
2, 2, 3, 3
```

If 99 is input, the output will be:

```
3, 3, 11
```

And, if a prime number is input, the result will be the number itself. For example, if 17 is input, the output will be 17.

We have almost all the programming code to do this already. Only a few changes need to be made to the prime-number code. To get prime factorization, first get the lowest divisor and then factor the remaining quotient. To get all the divisors for a number n, do this:

```
For all whole numbers from 2 to the square root of n,
    If n is evenly divisible by the loop variable (i),
        Print i followed by a comma, and
        Rerun the function on n / i, and
        Exit the current function
If no divisors found, print n itself
```

This logic is a recursive solution, which we can implement in C++ by having the function get_divisors call itself.

prime3.cpp

```
#include <iostream>
#include <cmath>

using namespace std;

void get_divisors(int n);
int main()
{
    int n = 0;
    cout << "Enter a number and press ENTER: ";
    cin >> n;
    get_divisors(n);
    cout << endl;
    return 0;
}
// Get divisors function
//  This function prints all the divisors of n,
```

prime3.cpp, cont.

```
//  by finding the lowest divisor, i, and then
//  rerunning itself on n/i, the remaining quotient.
void get_divisors(int n) {
double sqrt_of_n = sqrt(n);
     for (int i = 2; i <= sqrt_of_n; ++i) {
         if (n % i == 0) {   // If i divides n evenly,
            cout << i << ", ";    //   Print i,
            get_divisors(n / i);  //   Factor n/i,
            return;               //   and exit.
         }
     }
     // If no divisor is found, then n is prime;
     //  Print n and make no further calls.
     cout << n;
}
```

How It Works

As always, the program begins by declaring functions; in this case, there is one function other than **main**. The new function is get_divisors.

Also, the beginning of the program includes iostream and cmath because the program uses **cout**, **cin**, and **sqrt**. You don't need to declare **sqrt** directly, by the way, because this is done for you in cmath.

```
#include <iostream>
#include <cmath>

void get_divisors(int n);
```

The **main** function just gets a number from the user and calls get_divisors.

```
int main()
{
    int n = 0;
    cout << "Enter a number and press ENTER: ";
    get_divisors(n);
    cin >> n;
    cout << endl;
    return 0;
}
```

The get_divisors function is the interesting part of this program. It has a **void** return value, which means it doesn't pass back a value. But it still uses the **return** statement to exit early.

```
void get_divisors(int n) {
     double sqrt_of_n = sqrt(n);
     for (int i = 2; i <= sqrt_of_n; ++i)
         if (n % i == 0) {  // If i divides n evenly,
            cout << i << ", ";    //   Print i,
            get_divisors(n / i);  //   Factor n/i,
            return;               //   and exit.
         }
    // If no divisor is found, then n is prime;
    //  Print n and make no further calls.
     cout << n;
}
```

The heart of this function is a loop that tests numbers from 2 to the square root of n (which has been calculated and placed in the variable sqrt_of_n).

```
for (int i = 2; i <= sqrt_of_n; ++i) {
    if (n % i == 0) {  // If i divides n evenly,
       cout << i << ", ";    //   Print i,
       get_divisors(n / i);  //   Factor n/i,
       return;               //   and exit.
    }
}
```

If the expression n % i == 0 is true, that means the loop variable i divides evenly into n. In that case, the function does several things: it prints out the loop variable, which is a divisor, calls itself recursively, and exits.

The function calls itself with the value n/i. Because the factor i is already accounted for, the function needs to get the prime-number divisors for *the remaining factors* of n, and these are contained in n/i.

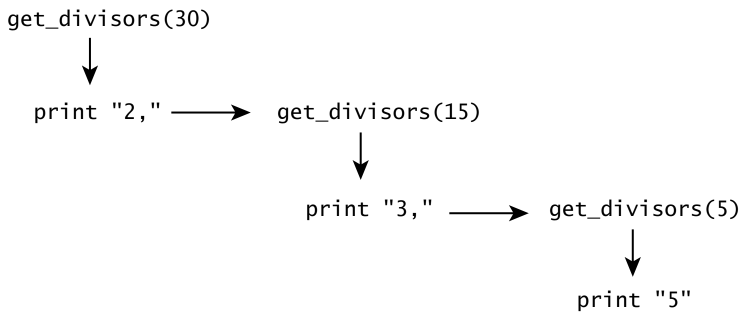
If no divisors are found, that means the number being tested is prime. The correct response is to print this number and stop.

```
cout << n;
```

For example, suppose that you input 30. The function tests to see what the lowest divisor of 30 is. The function prints the number 2 and then reruns itself on the remaining quotient, 15 (because 30 divided by 2 is 15).

During the next call, the function finds the lowest divisor of 15. This is 3, so it prints 3 and then reruns itself on the remaining quotient, 5 (because 15 divided by 3 is 5).

Here's a visual summary: each call to get_divisors gets the lowest divisor and then makes another call unless the number being tested is prime.



Interlude for Math Junkies

A little reflection shows why the lowest divisor is always a prime number. Suppose we test a positive whole number and that A is the lowest divisor *but is not a prime*. Since A is not prime, it must have at least one divisor of its own, B, that is not equal to either 1 or A.

But if B divides evenly into A and A is a divisor of the target number, then B must also be a divisor of the target number. Furthermore, B is less than A. Therefore, the hypothesis that the lowest divisor is not prime results in a contradiction.

This is easy to see by example. Any number divisible by 4 (a nonprime) is also divisible by 2 (a prime). The prime factors will always be found first, as long as you keep looking for the lowest divisor.

EXERCISES

Exercise 5.3.1. Rewrite the **main** function for Example 5.3 so that it prints the prompt message “Enter a number (0 = exit) and press ENTER.” The program should call get_divisors to show the prime factorization and then prompt the user again, until he or she enters 0. (Hint: If you need to, look at the code for Example 5.2 on page 106.)

Exercise 5.3.2. Write a program that calculates triangle numbers by using a recursive function. A triangle number is the sum of all whole numbers from 1 to n, in which n is the number specified. For example, triangle(5) = 5 + 4 + 3 + 2 + 1.

Exercise 5.3.3. Modify Example 5.3 so that it uses a *nonrecursive* solution. You will end up having to write more code. (Hint: To make the job easier, write two functions:

get_all_divisors and get_lowest_divisor. The **main** function should call get_all_divisors, which in turn has a loop: get_all_divisors calls get_lowest_divisor repeatedly, each time replacing n with n/i, where i is the divisor that was found. If n itself is returned, then the number is prime and the loop should stop.)

Example 5.4. *Euclid's Algorithm for GCF*

In the early grades of school, we're asked to figure out greatest common factors (GCFs). For example, the greatest common factor of 15 and 25 is 5. Your teacher probably lectured you about GCF until you didn't want to hear about it anymore.

Wouldn't it be nice to have a computer figure this out for you? We'll focus just on GCF because, as I'll show in Chapter 10, if you can figure out the CGF of two numbers, you can easily compute the lowest common multiple (LCM).

The technique was worked out almost 2,500 years ago by a Greek mathematician named Euclid, and it's one of the most famous in mathematics.

Pseudocode

```
To get CGF: For whole two numbers A and B:
If B equals 0,
    The answer is A.
Else
    The answer is GCF(B, A%B)
```

You may remember remainder division (%) from earlier chapters. A%B means this:

Divide A by B and produce the remainder.

For example, 5%2 equals 1 and 4%2 equals 0. A result of 0 means that B divides A evenly.

If B does not equal 0, the algorithm replaces the arguments A, B with the arguments B, A%B and calls itself recursively. This solution works for two reasons:

- The terminal case (B equals 0) is valid. The answer is A. You can easily see that the largest number that divides evenly into both A and 0 is A.
- The general case is valid: GCF(A, B) equals CGF(B, A%B), so the function calls itself with new arguments B and A%B.

The general case is valid if the following is true: the greatest common factor of the pair (B, A%B) is also the greatest common factor of the pair (A, B).

It turns out this *is* true and, because it is, the GCF problem is passed along from the pair (A, B) to the pair (B, A%B). This is the general idea of recursion: Pass the problem along to a simpler case involving smaller numbers.

It can be shown that the pair (B, A%B) involves numbers less than or equal to the pair (A, B). Therefore, during each recursive call, the algorithm uses successively smaller numbers until B is zero.

I save the rest of the proof for an interlude at the end of this section. Here is a complete program for computing greatest common factors:

gcf.cpp

```
#include <cstdlib>
#include <iostream>
using namespace std;
int gcf(int a, int b);

int main()
{
    int a = 0, b = 0; // Inputs to GCF.

    cout << "Enter a: ";
    cin >> a;
    cout << "Enter b: ";
    cin >> b;
    cout << "GCF = " << gcf(a, b) << endl;
    return 0;
}

int gcf(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcf(b, a%b);
    }
}
```

How It Works

All that **main** does in this case is to prompt for two input variables a and b, call the greatest-common-factor function (gcf), and print results:

```
cout << "GCF = " << gcf(a, b) << endl;
```

As for the gcf function, it implements the algorithm discussed earlier:

```
int gcf(int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcf(b, a%b);
    }
}
```

The algorithm keeps assigning the old value of B to A and the value A%B to B. The new arguments are equal or less to the old. They get smaller until B equals 0.

For example, if we start with A = 300 and B = 500, the first recursive call switches their order. (This always happens if B is larger.) From that point onward, each call to gcf involves smaller arguments until the terminal case is reached.

VALUE OF A	VALUE OF B	VALUE OF A%B (DIVIDE AND GET REMAINDER)
300	500	300
500	300	200
300	200	100
200	100	0
100	0	Terminal case: answer is 100

When B is 0, the gcf function no longer computes A%B, but instead produces the answer.

If the initial value of A is larger than B, the algorithm produces an answer even sooner. For example, suppose A = 35 and B = 25.

VALUE OF A	VALUE OF B	VALUE OF A%B (DIVIDE AND GET REMAINDER)
35	25	10
25	10	5
10	5	0
5	0	Terminal case: answer is 5

Interlude

Who Was Euclid?

Who was this Euclid guy? Wasn't he the Greek who wrote about geometry (something like "The shortest distance between two points is a straight line")?

Indeed he was. Euclid's *Elements* is one of the most famous books in Western civilization. For almost 2,500 years it was used as a standard textbook in schools. In this work, he demonstrated for the first time a *tour de force* of deductive logic, proving all that was then known about geometry. In fact, he invented the whole *idea* of proof. It is a great work that has had a profound influence on mathematicians and philosophers ever since.

It was Euclid who (according to legend) said to King Ptolemy of Alexandria, "Sire, there is no royal road to geometry." In other words, you gotta work for it.

Although its focus is on geometry, Euclid's book has results in number theory as well. The algorithm here is the most famous of these results. Euclid expressed the problem geometrically, finding the biggest length commensurable with two sides of a rectangle. He conceived the problem in terms of rectangles, but we can use any two integers.

EXERCISES

Exercise 5.4.1. Revise the program so that it prints out all the steps involved in the algorithm. Here is a sample output:

```
GCF(500, 300) =>
GCF(300, 200) =>
GCF(200, 100) =>
GCF(100, 0) =>
100
```

Exercise 5.4.2. For experts: Revise the gcf function so that it uses an iterative (loop-based) approach. Each cycle through the loop should stop if B is zero; otherwise, it should set new values for A and B and then continue. You'll need a temporary variable—temp—to hold the old value of B for a couple of lines: temp = b, b = a%b, and a = temp.

Interlude

Interlude for Math Junkies: Rest of the Proof

Earlier, I worked out some of a proof of Euclid's algorithm. What remains is to show that the greatest common factor of the pair (B, A%B) is also the greatest common factor of the pair (A, B). This is true if we can show the following:

- If a number is a factor of both A and B, it is also a factor of A%B.
- If a number is a factor of both B and A%B, it is also a factor of A.

If these statements are true, then all the common factors of one pair are common factors of the other pair. In other words, the set of common factors (A, B) is identical to the set of common factors (B, A%B). Since the two sets are identical, they have the *greatest member* and therefore they share the greatest common factor.

Consider the remainder-division operator (%). It implies the following, where m is a whole number:

```
A = mB + A%B
```

A%B is equal or less than A, so the general tendency of the algorithm is to get progressively smaller numbers. Assume that n, a whole number, is a factor of both A and B (meaning it divides both evenly). In that case:

```
A = cn
B = dn
```

where c and d are whole numbers. Therefore:

```
cn = m(dn) + A%B
A%B = cn - mdn = n(c - md)
```

This demonstrates that if n is a factor of both A and B, it is also a factor of A%B. By similar reasoning, we can show that if n is a factor of both B and A%B, it is also a factor of A.

Because the common factors for the pair (A, B) are identical to the common factors for the pair (B, A%B), it follows that they share the greatest common factor. Therefore, GCF(A, B) equals GCF(B, A%B). QED.

Example 5.5. *Beautiful Recursion: Tower of Hanoi*

Strictly speaking, the earlier examples don't require recursion. With some effort, they can be revised as iterative (loop-based) functions. But there is a mathematical puzzle that illustrates recursion beautifully, solving a problem that would otherwise be very difficult to figure out.

This is the Tower of Hanoi puzzle: You have three stacks of rings. Each ring is smaller than the one it sits on. The challenge is to move all the rings from the first stack to the third, subject to these constraints:

- You can move only one ring at a time.
- You can place a ring only on top of a larger ring, never a smaller.

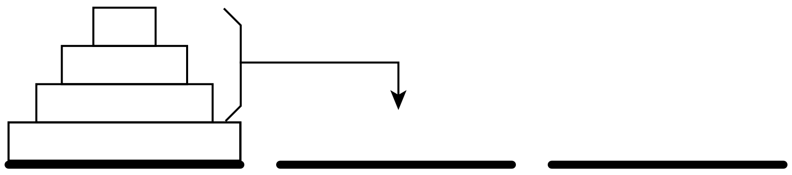
It sounds easy, until you try it! Consider a stack four rings high: You start by moving the top ring from the first stack, but where do you move it and what do you do after that?

To solve the problem, assume we already know how to move a group of n – 1 rings. Then, to move n rings from a source stack to a destination stack, do the following:

1. Move n – 1 rings from the source stack to the (currently) unused, or "other," stack.
2. Move a single ring from the source stack to the destination stack.
3. Move n – 1 rings from the "other" stack to the destination stack.

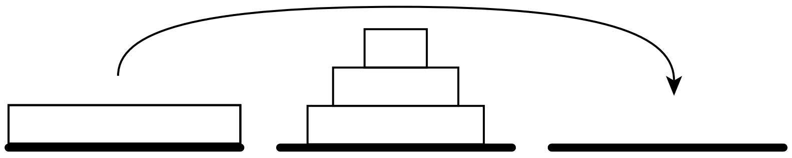
This is easier to envision graphically. First, the algorithm moves n – 1 rings from the source stack to the "other" stack ("other" being the stack that is neither source nor destination for the current move). In this case, n is 4 and n – 1 is 3, but these numbers will vary.

1. Move n – 1 rings from source to "other."



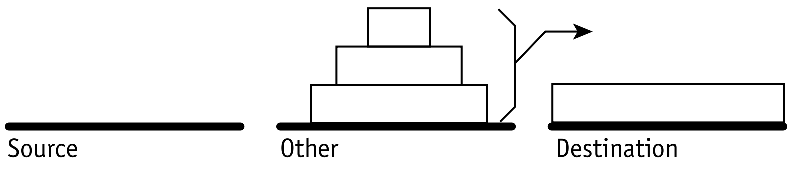
After this recursive move, at least one ring is left at the top of the source stack. This top ring is then moved: this is a simple action, moving one ring from source to destination.

2 Move one ring directly from source to destination.



Finally, we perform another recursive move, moving n – 1 rings from "other" (the stack that is currently neither source nor destination) to the destination.

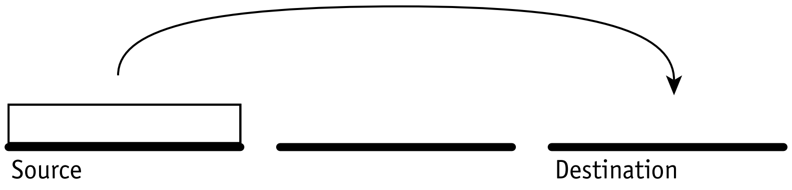
3 Move n – 1 rings from "other" to destination.



What permits us to move n – 1 rings in steps 1 and 3, when the constraints tell us that we can move only one?

Remember the basic idea of recursion. Assume the problem *has already been solved* for the case n – 1, although this may require many steps. All we have to do is tell the program how to solve the nth case in terms of the n – 1 case. The program magically does the rest. Of course, we don't know how to move n rings yet, but we will. The recursion technique enables us to move n rings by assuming that the problem has been solved for n – 1.

It's also important to solve the terminal case, n = 1. But that's trivial because where one ring is involved, we simply move the ring as desired.



The following program shows the C++ code that implements this algorithm:

tower.cpp

```
#include <iostream>
using namespace std;
void move_rings(int n, int src, int dest, int other);
void move_a_ring(int src, int dest);

int main()
{
    int n = 3;  // Stack is 3 rings high

    move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
    return 0;
}

void move_rings(int n, int src, int dest, int other) {
    if (n == 1) {
        move_a_ring(src, dest);
    } else {
        move_rings(n - 1, src, other, dest);
        move_a_ring(src, dest);
        move_rings(n - 1, other, dest, src);
    }
}

void move_a_ring(int src, int dest) {
    cout << "Move from " << src << " to "
         << dest << endl;
}
```

How It Works

The program is fairly short considering what it does. In this example, I've set the stack size to just three rings, although it can be any positive integer:

```
int n = 3;  // Stack is 3 rings high
```

The call to the move_rings function says that three rings should be moved from stack 1 to stack 3; these are determined by the second and third arguments, respectively. The "other" stack, stack 2, will be used in intermediate steps.

```
move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
```

This small example—moving only three rings—produces the following output. You can verify the accuracy of this solution by using three different coins, all of different sizes.

```
Move from 1 to 3
Move from 1 to 2
Move from 3 to 2
Move from 1 to 3
Move from 2 to 1
Move from 2 to 3
Move from 1 to 3
```

Try setting n to 4 and you'll get a list of moves more than twice as long.

The core of the move_ring function is the following code, which implements the general solution described earlier. Remember, this recursive approach assumes the n − 1 case has already been solved. The function therefore passes along most of the problem to the n − 1 case.

```
move_rings(n - 1, src, other, dest);
move_a_ring(src, dest);
move_rings(n - 1, other, dest, src);
```

Notice how the functional role of the three stacks is continually switched between *source* (where to move a group of rings from), *destination* (where the group is going), and *other* (the intermediate stack, to which some of the rings will go before they end up at the destination).

EXERCISES

Exercise 5.5.1. Revise the program so that the user can enter any positive integer value for n. Ideally, you should test the input to see whether it is greater than 0.

Exericse 5.5.2. Instead of printing the "Move" message directly on the screen, have the move_ring function call yet another function, to which you give the name exec_move. The exec_move function should take a source and destination stack number as its two arguments. Because this is a separate function, you can use as many lines of code as you need to print a message. You can print a more informative message:

```
Move the top ring from stack 1 to stack 3.
```

Example 5.6. *Random-Number Generator*

OK, we've had enough fun with recursion. It's time to move on to another, highly practical example. This one generates random numbers—a function at the heart of many game programs.

The test program here simulates any number of dice rolls. It does this by calling a function, rand_0toN1, which takes an argument, n, and randomly returns a number from 0 to n − 1. For example, if the user inputs the number 6, this program simulates dice rolls:

```
3 4 6 2 5 3 1 1 6
```

Here is the program code:

dice.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int rand_0toN1(int n);

int main()
{
    int n = 0;
    int r = 0;

    srand(time(nullptr)); // Set seed for randomizing.
    cout << "Enter number of dice to roll: ";
    cin >> n;
    for (int i = 1; i <= n; ++i) {
        r = rand_0toN1(6) + 1; // Get a number 1 to 6
        cout << r << " ";      // Print it
    }
    return 0;
}
// Random 0-to-N1 Function.
// Generate a random integer from 0 to N-1, with each
//  integer an equal probability.
//
int rand_0toN1(int n) {
    return rand() % n;
}
```

How It Works

Example 3.2 (page 72) in Chapter 3 laid out the basic principles of random-number generation in a C++ program. Here, I do a quick review.

The beginning of the program has to include certain files to support the functions needed for random-number generation:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
```

Next, the program then has to set a random-number seed to start off the sequence of numbers (actually *pseudo-random* numbers) that the program is going to generate. Using the system time this way guarantees a different set of random numbers every time the program is run.

```
srand(time(nullptr));
```

Note ▶ Remember that since C++11, compilers have been required to support the **nullptr** keyword, which has a zero value but has pointer type. If you have a compiler that's more than a few years old, you might need to use **NULL** or 0 instead. Also remember that you may need to apply the **static_cast** operator to get rid of all warning messages. See Chapter 3 for more information.

The rest of **main** prompts for a number and then prints the quantity of random numbers requested. A **for** loop makes repeated calls to rand_0toN1, a function that returns a random number from 0 to n – 1:

```
for (int i = 1; i <= n; ++i) {
    r = rand_0toN1(6) + 1;  // Get num from 1 to 6
    cout << r << " ";       // Print it out
}
```

Here is the function definition for the rand_0toN1 function:

```
int rand_0toN1(int n) {
    return rand() % n;
}
```

The range of numbers produced by **rand** is large; typically, it's all of the unsigned integer range (its highest value defined by MAX_RAND). But the beauty of the remainder-division ("mod") operator is that it is guaranteed to produce a result in the range 0 to n – 1, no matter how large the range of the values is, as long as the random-number generator produces numbers at least as high as N – 1.

In this case, the function is called with the argument 6, so it returns a value from 0 to 5. Adding 1 to the number gives a random value in the range 1 to 6, which is what we want.

EXERCISES

Exercise 5.6.1. Write a random-number generator that returns a number from 1 to N (rather than 0 to N – 1), where n is the integer argument passed to it.

Exercise 5.6.2. Write a random-number generator that returns a random floating-point number between 0.0 and 1.0. (Hint: Call **rand**, cast the result r to type **double** by using static_cast<double>(r), and then divide by the highest value in the **int** range, **RAND_MAX**.) Make sure you declare the function with the **double** return type.

Games and More Games

Now that we know how to write functions and generate random numbers, it's possible to enhance some game programs.

We can improve the Subtraction Game example at the end of Chapter 2. Right now, when the user plays optimal strategy, the computer responds by choosing 1, which is arbitrary and predictable. We can make this more interesting by randomizing the computer's response in these situations—situations in which there is no winning play. The following program makes the necessary changes, with the altered lines of code in bold:

nim2.cpp

```
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;
int rand_0toN1(int n);

int main()
{
    int total, n;

    srand(time(nullptr)); // Set seed for randomizing.
    cout << "Welcome to NIM. Pick a starting total: ";
    cin >> total;
```

▼ *continued on next page*

nim2.cpp, cont.

```
    while (true) {
        // Pick best response and print results.

        if ((total % 3) == 2) {
            total = total - 2;
            cout << "I am subtracting 2." << endl;
        } else if ((total % 3) == 1) {
            --total;
            cout << "I am subtracting 1." << endl;
        } else {
            n = 1 + rand_0toN1(2); // n = 1 or 2.
            total = total - n;
            cout << "I am subtracting ";
            cout << n << "." << endl;
        }
        cout << "New total is " << total << endl;
        if (total == 0) {
            cout << "I win!" << endl;
            break;
        }

        // Get user's response; must be 1 or 2.

        cout << "Enter num to subtract (1 or 2): ";
        cin >> n;
        while (n < 1 || n > 2) {
            cout << "Input must be 1 or 2." << endl;
            cout << "Re-enter: ";
            cin >> n;
        }
        total = total - n;
        cout << "New total is " << total << endl;
        if (total == 0) {
            cout << "You win!" << endl;
            break;
        }
    }
    return 0;
}

int rand_0toN1(int n) {
    return rand() % n;
}
```

Chapter 5 *Summary*

Here are the main points of Chapter 5:

- In C++, you can use functions to define a specific task, just as you might use a subroutine or procedure in another language. C++ uses the name function for all such routines, whether they return a value or not.
- You need to declare all your functions (other than **main**) at the beginning of the program so that C++ has the type information required. Function declarations, also called *prototypes*, use this syntax:

```
type  function_name (argument_list
```

- You also need to define the function somewhere in the program, to tell what the function does. Function definitions use this syntax:

```
type  function_name (argument_list) {
    statements
}
```

- A function runs until it ends or until the **return** statement is executed. A **return** statement that passes a value back to the caller has this form:

```
return expression;
```

- A return statement can also be used in a **void** function (a function with no return value) just to exit early, in which case it has a simpler form.

```
return;
```

- Local variables are declared inside a function definition; global variables are declared outside all function definitions, preferably before **main**. If a variable is local, it is not shared with other functions; two functions can each have a variable named i (for example) without interfering with each other.
- Global variables enable functions to share common data, but such sharing provides the possibility of one function interfering with another. It's a good policy not to make a variable global unless there's a clear need to do so.
- C++ functions can use recursion—meaning they call themselves. (A variation on this is when two or more functions call each other.) This technique is valid as long as there is a case that terminates the calls. For example:

```
int factorial(int n) {
    if (n <= 1) {
            return 1;
        } else {
            return n * factorial(n - 1);  // RECURSION!
        }
    }
```

6 Arrays: All in a Row…

One of the themes in this book so far has been that computers can only carry out instructions that are precise and clear. How, then, can a computer operate on thousands, millions, even billions of bytes of data?

The answer is that programming languages permit you to define something called an array. An *array* is a data structure that has similar items of data—called *elements*—and it can have as many of these items as you want.

The beauty of this mechanism is that as long as you can control and define the general case, it's as easy for a program to operate on an exceptionally large array (even billions of items, if memory will allow) as it is to operate on a small one.

This is a major clue as to why computers and programming are useful. Computers have no problem doing a repetitive task over and over… even if it's performed a million times on a million different items.

A First Look at C++ Arrays

Suppose you're writing a program to analyze scores given by five judges in an Olympic kite-flying contest. You need to store all five values for a while so you can measure statistical properties: range, average, median, and so on. Also, suppose the judges are known by number.

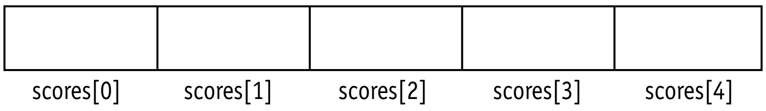
One way to store the information is to declare five separate variables. Since the scores have a fractional portion (0.1 being the lowest and 9.9 being about the highest), use type **double**.

```
double  scores1, scores2, scores3, scores4, scores5;
```

That's a lot to enter. Wouldn't it be nice to just enter the word *scores* and tell C++ to declare five variables for you? That's exactly what happens when you declare an array.

```
double  scores[5];
```

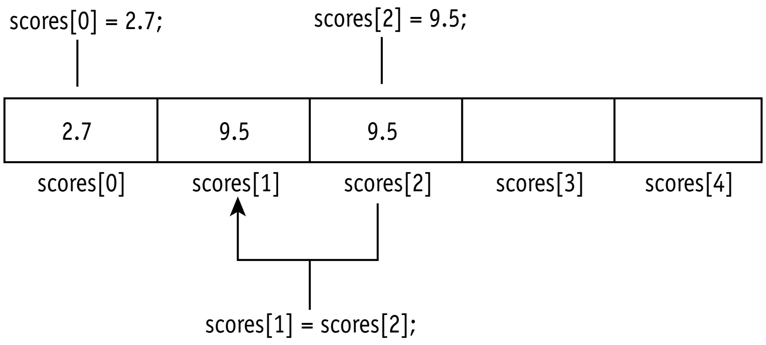
This declaration creates five data items of type **double** and places them next to each other in memory. In C++ programs, these items are referred to as scores[0], scores[1], scores[2], scores[3], and scores[4]. The numbers between brackets are *indexes*.



In the rest of the program, you can perform operations on each of these items as if it were an individual variable.

```
scores[0] = 2.7;        // Judge 0 gives a low score.
scores[2] = 9.5;        // Judge 2 gives a high score.
scores[1] = scores[2]; // Judge 1 copies Judge #2.
```

Each of these array elements (scores[0], scores[1], and so on) acts like a variable of type **double**—the difference being that it is referred to, in part, by a number. After these operations are performed, the array looks like this:



With only five elements, an array can be helpful. But that's nothing compared to the convenience of larger arrays. Look how much labor you save if you have an array with 1,000 elements:

```
int votes[1000];   // Declare array with 1000 elements
```

This declaration creates an array with 1,000 elements, running from votes[0] to votes[999]. Imagine if, instead, you had to type a thousand declarations yourself!

To summarize, you can declare variables of int or double (or any other supported data type) by using this syntax:

Key Syntax

```
type  array_name[size];
```

As a result, *array_name* is created as an array of the specified *size*. Each element of the array has the indicated *type*. The elements of the array range from *array_name*[0] to *array_name*[size-1].

Initializing Arrays

Referring to a variable you've forgotten to initialize can end up producing garbage (*garbage* being a technical term for a meaningless value). Remember, you can initialize a variable when it's declared, even when you declare more than one on the same line.

```
int  sum = 0, fingers = 10;
```

You can initialize an array with the use of a comma-separated list of initializers. This approach uses a simple notation involving brackets and commas:

```
double  scores[5] = {0.0, 0.0, 0.0, 0.0, 0.0};
int  ordinals[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Each of these lines is terminated with a closing brace followed by a semicolon (;). Data declarations and function prototypes always end with a semicolon.

Note ▶ If a variable or an array is global, then by default C++ initializes it to zero. (In the case of arrays, C++ initializes every element to zero.) But local variables not initialized contain random meaningless values, also known as "garbage."

Zero-Based Indexing

C++ arrays work a little differently from the way you might expect. If you have N items, they are not numbered 1 to N, but from 0 to N − 1. Again, for an array declared this way,

```
double  scores[5];
```

the elements are as follows:

```
scores[0]
scores[1]
scores[2]
scores[3]
scores[4]
```

No matter how you declare an array, the highest index number (in this case, 4) will always be *one less than* the size of the array (in this case, 5). This may seem counterintuitive.

But seen from another angle, it makes perfect sense. The index number in a C or C++ array is not an ordinal number (that is, a position) as much as it is an *offset*. That is, the index number of an element is a measure of the distance from the beginning of the array.

And the first element, of course, is zero positions away from the beginning. The index of the first element is therefore 0. This is worth restating as another cardinal rule:

In a C++ array of size N elements, the indexes run from 0 to N – 1.

Interlude

Why Use Zero-Based Indexes?

Many other languages use 1-based indexing. The declaration ARRAY(5) in FORTRAN creates an array with indexes running from 1 to 5. But all programs, regardless of what language they are written in, must ultimately be translated into machine language, which is what the CPU actually executes.

At the machine level, array indexing is handled through offsets. One register (a memory location on the CPU itself) contains the address of an array—actually, the address of the first element. Another register contains an offset: the distance to the desired element.

What is the offset of the first element? Zero, just as in C++. With a language such as FORTRAN, the 1-based index must first be translated into a 0-based index by subtracting by 1. It is then multiplied by the size of each element. To get the element with index I, do this:

```
address of element I = base address + ((I - 1) * size
of each element)
```

In a 0-based language such as C++, the subtraction no longer has to be done. This results in a more efficient calculation at runtime:

```
address of element I = base address + (I * size of
each element)
```

Even though it results in only a slight saving of CPU cycles, it's very much in the spirit of C-based languages to use this approach because it is closer to what the CPU does.

Example 6.1. *Print Out Elements*

Let's start by looking at one of the simplest programs possible that uses an array. The rest of the chapter gets into more interesting programming challenges.

print_arr.cpp

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

int main()
{
    double scores[5] = {0.5, 1.5, 2.5, 3.5, 4.5};

    for(int i = 0; i < 5; ++i) {
        cout << scores[i] << "  ";
    }
    return 0;
}
```

The program, when run, prints this:

```
0.5  1.5  2.5  3.5  4.5
```

How It Works

The program uses a **for** loop that sets the loop variable, i, to a series of values—0, 1, 2, 3, 4—corresponding to the range of indexes in the array, the scores.

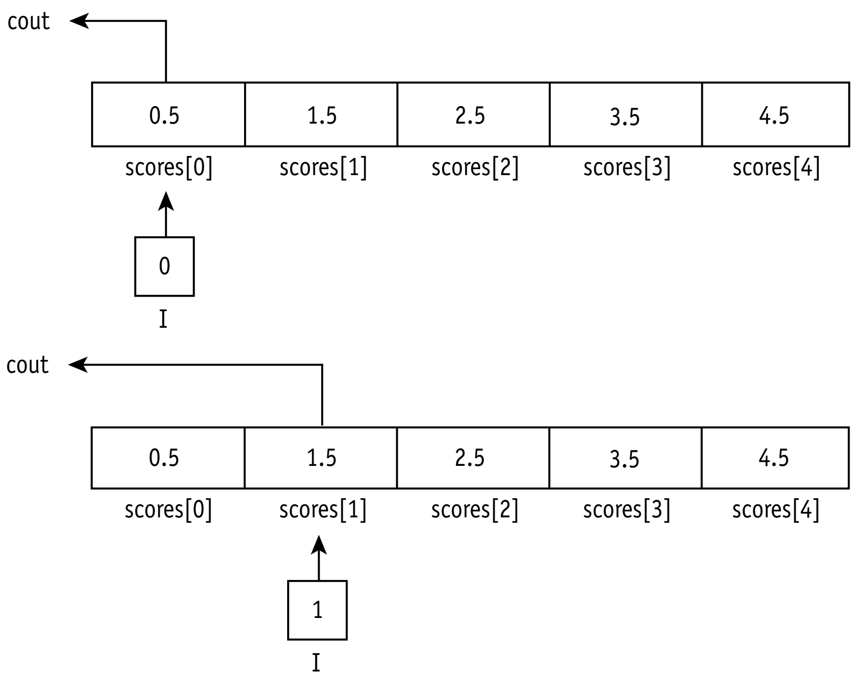
```
for(int i = 0; i < 5; ++i) {
    cout << scores[i] << "  ";
}
```

This kind of loop is extremely common in C++ code, so you often see these expressions used with **for**: i = 0, i < SIZE_OF_ARRAY, and ++i.

The loop cycles five times, each time with a different value for i.

VALUE OF I	ACTION OF THE LOOP	VALUE PRINTED
0	Print scores[0]	0.5
1	Print scores[1]	1.5
2	Print scores[2]	2.5
3	Print scores[3]	3.5
4	Print scores[4]	4.5

You can also understand the action of this loop visually. The following figure demonstrates the action of the first two cycles of the loop.



EXERCISES

Exercise 6.1.1. Write a program that initializes an array of eight integers with the values 5, 15, 25, 35, 45, 55, 65, and 75, and then prints each of these out. (Hint: Instead of using the loop condition i < 5, use i < 8, because in this case there are eight elements.)

Exercise 6.1.2. Write a program that initializes an array of six integers with the values 10, 22, 13, 99, 4, and 5. Print each of these out and then print their sum. (Hint: You'll need to keep a running total.)

Exercise 6.1.3. Write a program that prompts the user for each of seven values, stores these in an array, and then prints out each of them, followed by the total. You will need to write two **for** loops for this program: one for collecting data and another for calculating the sum and printing out values.

Example 6.2. *How Random Is Random?*

Chapter 3, "And Even More Decisions!" introduced the use of so-called random numbers. But randomness—the deliberate *lack* of predictability—is a philosophical paradox. The essence of a computer algorithm is predictability. True randomness may not be a theoretical possibility.

But is it a *practical* possibility? If we ask a program to output a series of these numbers, do they behave in a way that has all the qualities we'd expect of a true random sequence?

The rand_0toN1 function outputs an integer from 0 to N − 1, where N is the argument to the function. We can use this function to get a series of numbers from 0 to 9, and count how many we get of each digit. What you'd expect to happen is this:

- Each of the 10 digits should be produced about one-tenth of the time.
- But the digits shouldn't be produced with absolutely equal frequency. Especially with a small number of trials, you should see variation. However, as the number of trials increase, the ratio of actual hits to expected hits for each digit (one-tenth of the total number) ought to get closer and closer to 1.0.

If these conditions can be met, we have a good example of practical randomness, which is probably good enough for the great majority of game programs.

We can test these conditions by using an array of 10 integers to register the results. When the program is run, it will prompt for a number of trials to run. It will then report the total number of hits for each of the numbers 0 to 9. Here's what sample output for 20,000 trials should look like:

```
Enter number of cases to do: 20000

0: 1950  Accuracy: 0.975
1: 2026  Accuracy: 1.013
2: 1897  Accuracy: 0.9485
3: 2102  Accuracy: 1.051
4: 2019  Accuracy: 1.0095
5: 1997  Accuracy: 0.9985
6: 1999  Accuracy: 0.9995
```

```
7: 1969  Accuracy: 0.9845
8: 2033  Accuracy: 1.0165
9: 2008  Accuracy: 1.004
```

With 20,000 trials, you should get a fast response. Depending on your computer, it may take millions of trials before you see a noticeable delay. I have run this program with as many as 2 billion trials (input: 2000000000, or 2'000'000'000 in C++14, which accepts the apostrophe as a digit-group separator). My desktop computer, which is a few years old, takes 28 minutes to respond in that case. But your computer may be faster.

It's interesting to run this program with different values for N. You should find, as I have, that as the number of trials increase, the accuracy (the ratio of expected hits to actual hits) does in fact get closer to 1.0, consistent with math's Law of Large Numbers.

Here's the code for this program:

stats.cpp

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

using namespace std;

int rand_0toN1(int n);
int hits[10];

int main()
{
    int n = 0;  // Number of trials; prompt from user
    int r = 0;  // Holds a random value

    srand(time(nullptr));   // Set seed for randomizing.

    cout << "Enter how many trials and press ENTER: ";
    cin >> n;

    // Run n trials. For each trial, get a num 0 to 9
    //  and then increment the corresponding element
    //  in the hits array.

    for (int i = 0; i < n; ++i) {
        r = rand_0toN1(10);
```

stats.cpp, cont.

```
            ++hits[r];
        }

        // Print all elements in the hits array, along
        //  with ratio of hits to EXPECTED hits (n / 10).

        for (int i = 0; i < 10; ++i) {
            cout << i << ": " << hits[i] << " Accuracy: ";
            double results = hits[i];
            cout << results / (n / 10.0) << endl;
        }
        return 0;
    }

    // Random 0-to-N1 Function.
    // Generate a random integer from 0 to N-1.
    //
    int rand_0toN1(int n) {
        return rand() % n;
    }
```

How It Works

The program begins with a couple of declarations:

```
int rand_0toN1(int n);

int hits[10];
```

The rand_0toN1 function is declared here because it is going to be called by **main**. The declaration of hits creates an array of 10 integers, ranging in index from 0 to 9. Because this array is global (declared outside of any function), all its elements are initialized to 0.

Note Technically, the array is initialized to all-zero values because it has a static storage class. Local variables can also be declared static, causing them to retain values between calls even though they are not visible outside the function.

The main function begins by defining two integer variables, n and r, and by setting the seed for the sequence of random numbers. This needs to be done in every program that uses random-number generation. (Remember to use NULL if your computer is really old and doesn't support **nullptr**.)

```
srand(time(nullptr));  // Set seed for randomizing.
```

The program then prompts for the value of n. This should look familiar by now:

```
    cout << "Enter how many trials and press ENTER: ";
    cin >> n;
```

The next part of the program is a **for** loop that carries out the requested number of trials (namely, n) and stores results in the hits array.

```
// Run n trials. For each trial, get a num 0 to 9
//  and then increment the corresponding element
//  in the hits array.

for (int i = 0; i < n; ++i) {
    r = rand_0toN1(10);
    ++hits[r];
}
```

Note that r could actually be defined locally to the loop, which would make sense. That is left as an exercise.

Each time through, the loop gets a random number r between 0 and 9 and then records this as a "hit" for the number chosen by adding 1 to the appropriate array element. At the end of the process, the element hits[0] contains the number of 0s generated, hits[1] contains the number of 1s generated, and so on.

The expression hits[r]++ saves a lot of programming effort. If you weren't using an array, you'd have to write a series of if/else statements, or an equivalent **switch** statement, like this:

```
if (r == 0)
    ++hits0;
else if (r == 1)
    ++hits1;
else if (r == 2)
    ++hits2;
else if (r == 3)
    ++hits3;
// etc.
```

Because we're working with arrays, what would otherwise take 20 lines of code takes only one! This single statement adds 1 to whatever element is selected by r:

```
++hits[r];
```

The rest of **main** consists of a loop that prints all the elements of the array. This action reports the results and is run after all the trials have been performed. As before, this code is much more concise than would be the case if we weren't using an array:

```
// Print all the elements in the hits array, along
//  with ratio of hits to EXPECTED hits (n / 10).

for (int i = 0; i < 10; ++i) {
    cout << i << ": " << hits[i] << " Accuracy: ";
    double results = hits[i];
    cout << results / (n / 10.0) << endl;
}
```

The middle line of this compound statement may seem odd, but it is necessary. The results are put in a temporary variable of type **double**. Because **double** has a larger range than **int**, the compiler does not complain of information loss.

```
double results = hits[i];
```

This assignment needs to be done to force floating-point division in the statement that follows. Otherwise—as happens when you divide one integer by another—C++ would perform integer division, throwing fractional results away! An alternative would have been to use static_cast<double>(hits[i]) to cast the data.

The rand_0toN1 function is the same function I introduced at the end of Chapter 2, "Decisions, Decisions."

```
// Random 0-to-N1 Function.
// Generate a random integer from 0 to N-1.
//
int rand_0toN1(int n) {
    x = rand() % n;
}
```

EXERCISES

Exercise 6.2.1. Instead of declaring and defining the variable r at the beginning of the function, declare it inside the loop that uses it. With this approach, r does not need to be initialized to 0 because you can assign a meaningful value directly. (This saves only a small amount of code, but it does make sense.)

Exercise 6.2.2. Alter Example 6.2 so that it generates not 10 different values but 5: In other words, use the rand_0toN1 function to get a 0, 1, 2, 3, or 4. Then perform the requested number of trials, in which you'd expect each value of the five values to be produced one-fifth of the time.

Exercise 6.2.3. Alter the example so that it can work with any number of values, simply by changing one setting in the program. You can do this with a **#define** directive near the beginning of the code. This directive instructs the compiler to replace all occurrences of a symbolic name (in this case, VALUES) with the specified text.

For example, to have each random trial generate one of five different values (0 through 4), first put the following directive at the beginning of the code:

```
#define VALUES 5
```

Then use the symbolic name VALUES throughout the program, wherever the program refers to the number of possible values. For example, you'd declare the hits array as follows:

```
int hits[VALUES];
```

From then on, you can control the number of different values by going back and changing one line—the **#define** directive—with a different number and then recompiling. The beauty of this approach is that the behavior of the program can be so easily modified by that one line of code.

Exercise 6.2.4. Rewrite the code in **main** so that it uses a loop similar to the one in Example 5.2 (page 106), allowing the user to keep rerunning sessions any number of times until he or she enters 0 to terminate the program. Before each session, you need to reinitialize all the elements of the hits array to 0. You can do that either by including a **for** loop that sets each element to 0 or by calling a function that contains such a loop.

Strings and Arrays of Strings

To do the examples in the remainder of this chapter, I'm going to have to get a little ahead of the story to show how to declare arrays of strings. In Chapter 8, "Strings: Analyzing the Text," we'll return to the subject of strings.

Up until now, I've shown the use of string literals. For example, to print a message, you'd use a line of code like this:

```
cout << "What a good C++ am I.";
```

You can also have string variables, just as you can have integer and floating-point variables. There are actually two kinds of strings, as I'll explain in Chapter 8: traditional C-strings, which have type **char***, and the C++ **string** class, which has been supported by the standard C++ library for a number of years now.

For example, the following code first stores the string in the variable named message and then prints it. This approach uses the **string** class, and <string> must be included to support this class.

```
include <string>
using namespace std;
...
string message = "What a good C++ am I";
cout << message;
```

The rest of this chapter uses arrays of strings. You declare them just as you'd declare any type of array. For example:

```
string members[] = {"John", "Paul", "George", "Ringo"};
```

As with any other array, you can access an individual element by using an index. For example:

```
cout << "The leader of the band is " << members[0];
```

This prints out the following:

```
The leader of the band is John.
```

Because the names of the members are all stored in the array, we can use a loop to print all of them out efficiently. For example, this code

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

prints out this list of names:

```
John
Paul
George
Ringo
```

Example 6.3. *Print a Number (from Arrays)*

In this section, we'll look at another example which, although it can be written without arrays, is much more compact and efficient when written with them. Example 3.3, on page 80, translated a numeric value into English words. 49, for example, would produce the text "forty nine."

The **switch-case** statement is probably the cleanest, most compact way to do that without arrays. But the use of arrays makes the example much shorter still.

Instead of conditionally executing different lines of code, this example selects elements from two arrays of strings. The result is a classic "Choose one from column A, one from column B" approach, written elegantly and compactly.

print_n_arr.cpp

```
#include <iostream>
#include <string>  // REMEMBER TO INCLUDE THIS!

using namespace std;

string tens_names[ ] = {"", "", "twenty", "thirty",
   "forty", "fifty", "sixty", "seventy", "eighty",
   "ninety" };

string units_names[ ] = {"", "one", "two", "three",
   "four", "five", "six", "seven", "eight", "nine" };

int main()
{
    int  n = 0;

    cout << "Enter a number from 20 to 99: ";
    cin >> n;
    int tens_digits = n / 10;
    int units_digits = n % 10;
    cout << "The number you entered was ";
    cout << tens_names[tens_digits] << " ";
    cout << units_names[units_digits] << " ";
    return 0;
}
```

If you compare this version of the program, line by line, with the earlier version (Example 3.3 on page XX), you should be amazed at how much more compact this version is.

A sample session might look like the following:

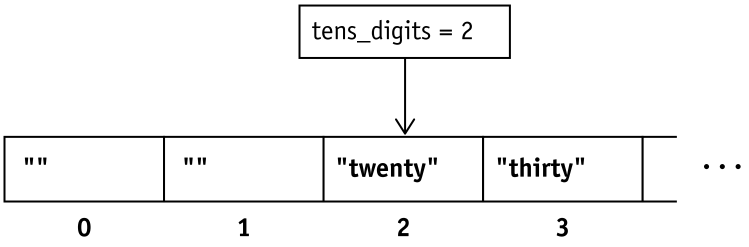
```
Enter a number from 20 to 99: 23
The number you entered was twenty three.
```

How It Works

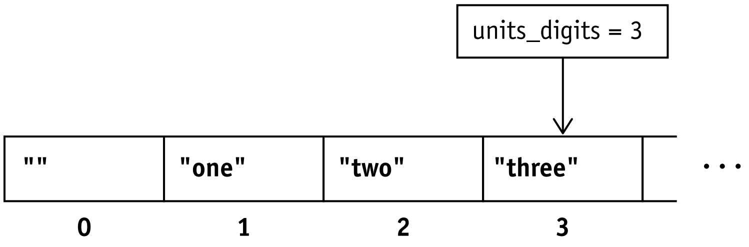
A comparison between this example and Example 3.3 demonstrates the power of arrays. The use of a value chosen from an array causes differing effects—printing a different word depending on the value—and this eliminates the need for a long series of **if-else** statements or even a **switch-case** statement.

This doesn't mean that you'll always be able to get rid of an **if-else** or **switch-case** statement and replace it by using arrays, but in this case you certainly can.

We can imagine the action of the program this way: First, it uses the value tens_digits as an index, selecting a value from the array tens_names:



Next, the program uses value units_digits as an index, selecting a value from the array units_names:



Keep in mind that C++ arrays (as with arrays in most of the C-programming language family) are zero based, so that the first element in every array is indexed by a 0 value. This is true for all kinds of arrays.

EXERCISES

Exercise 6.3.1. What happens if the end user enters a number outside the range 20 to 99? Values 1 to 19 will result in harmless, though erroneous, results, but high values (above 99) are potentially catastrophic because they cause out-of-range indexing errors, which you want to avoid at all costs. (Note: Managed environments such as Microsoft Studio limit the damage by throwing an exception, immediately halting the program.) Prevent this risk by writing code that only accepts an input value between 20 and 99. Ideally, you should use a loop that keeps querying the end user until a valid number is entered.

Exercise 6.3.2. The remaining exercises expand the extent of acceptable range. When applied to input from 1 to 9, the output will be correct, except that it has an extra space in front of it. Solve this problem.

Exercise 6.3.3. Introduce support for "teen" values 10 through 19. You'll need to add another array and another conditional test to the program.

Exercise 6.3.4. Finally, extend support for the hundreds digits, so that values from 1 to 999 can be handled. If you're really ambitious, you can even revise the program so it handles numbers all the way up to 999,999!

Example 6.4. *Simple Card Dealer*

For the final example in this chapter, I'll introduce a simple application that I'll return to in Chapter 15, "Object-Oriented Poker."

How do we simulate the action of a Poker dealer? To keep things easy, I'll make two simplifying assumptions:

1. We only care about ranks in this case, not suits.
2. We won't, for now, worry about the problem of reshuffling.

Sample output, therefore might look like this:

```
A 5 3 K K
```

or this:

```
Q 7 10 6 7
```

Also, let's test the deal-a-card function by dealing exactly five cards, as you'd get in a simple game of poker (although again, ignore suits for now).

Cards behave differently from dice, even though both are randomization devices. Each roll of a die is an independent event. Dice have no memory, but a deck of cards does. If, for example, four aces have been dealt but half the deck remains, the probability of getting another ace goes to zero.

The way we simulate this "deck memory" is with another array! The technique is to produce an array of integers and give them values equal to their position: 0, 1, 2, 3, 4, all the way up to 51. Then randomize the array—that is, shuffle it—and deal off the top.

Remember that because this application uses randomization, it needs to include the following:

```
#include <cstdlib>
#include <ctime>
```

We'll also need an array of card names, so <string> must be included as well.

dealer.cpp

```
#include <iostream>
#include <string>     // Needed for string class.
#include <cstdlib>    // Needed for randomization.
#include <ctime>

using namespace std;

int deck[52];

string card_names[ ] = {"A", "2", "3", "4", "5", "6",
   "7", "8", "9", "10", "J", "Q", "K" };

void swap_cards(int i, int j);
int rand0_to_N(int n);

int main()
{
    srand(ctime(NULL));  // Set random seed.

    // Initialize deck 0, 1, 2, 3... 51

    for (int i = 0; i < 52; ++i) {
        deck[i] = i;
    }

    // Shuffle deck.

    for (int i = 51; i > 0; --i) {
        int j = rand0_to_N(i);
```

▼ *continued on next page*

dealer.cpp, cont.

```
            swap_cards(i, j);
        }

        // Deal 5 cards.

        for (int i = 0; i < 5; ++i) {
            int j = deck[i] % 13;
            cout << card_names[j] << " ";
        }
        cout << endl;
        return 0;
    }

    //
    void swap_cards(int i, int j) {
        int temp = deck[i];
        deck[i] = deck[j];
        deck[j] = temp;
    }

    //
    int rand0_to_N(int n) {
        return rand() % (n + 1);
    }
```

How It Works

Although this program is longer than some others in this book, it's still simple and easy to understand. The central data structure in this program is deck[], an array of 52 integers.

```
int deck[52];
```

This array is not explicitly initialized. As a global variable, it's therefore initialized to zero values by default. But it's dangerous to rely on such behavior, because if deck[] is made local, its values are initialized to garbage—which is a humorous way of saying it could contain anything.

The real initialization of the deck array is done in the main function, which uses a simple loop to set values to 0, 1, 2, 3, and so on, in that order. This is achieved by setting each element to its index value.

```
for (int i = 0; i < 52; ++i) {
    deck[i] = i;
}
```

Next, shuffle the deck! That turns out to be fairly easy.

```
for (int i = 51; i > 0; --i) {
    int j = rand0_to_N(i);
    swap_cards(i, j);
}
```

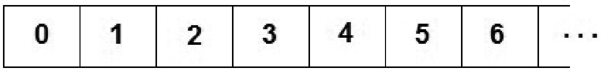
The effect here is to fill each position of the array with a randomly selected value, picked with equal probability from the numbers 0 to 51. Here's the pseudocode equivalent:

```
For I equal 51 counting down to 1,
    Set J equal to a random number from 0 to I
    Swap elements at positions I and J
```

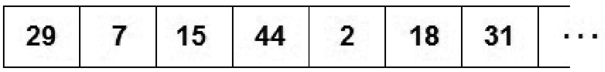
Think of it this way: Start with 52 cards. Take the bottom card and randomly swap it with any card in the deck, which (and this is significant) may include the bottom card itself. If the replacement card is the bottom card, then the swap is a no-op, which is fine. The final result is that the bottom card ends up being any of the 52 cards with equal probability.

Then, set that card aside and focus on the remaining 51 cards. Repeat the operation on those cards, resulting in the selection of one of those 51. Set that card aside, placing it on top of the previously selected card, and repeat again with the remaining 50 cards. After this has been done down to the last two cards, the deck is fully randomized. Moreover, if the randomization itself is good, then every card can be in any position, with equal probability for any case.

This next diagram, before and after shuffling.



Before shuffling.



After shuffling.

Given a fully shuffled deck, we now deal cards off the top, one at a time, converting a number to its corresponding card. This is done by using the remainder-division operator (%), which takes a number from 0 to 51 and produces a number in the range 0 to 12 (one of 13 different values) with equal distribution.

```
int j = deck[i] % 13;
```

After this calculation, j will be a number from 0 to 12 with (initially) equal probability. The last thing to do is to look this number up in the card_names array to convert a number from 0 to 12 into a short string such as "A", "K", "Q", "J", "10", and so on.

EXERCISES

Exercise 6.4.1. Alter the program so that it prints out card ranks as full names: "ace," "two," "three," and so on. You may want to print two and three as "deuce" and "trey," respectively.

Exercise 6.4.2. Print out suits as well as ranks, so that the program prints a full card name such as "ace of spades." There are just four suits: clubs, diamonds, hearts, and spades. This information can also be attached to the numbers 0 to 51. You can think of the first 13 numbers as clubs, the next 13 as diamonds, and so on. (Hint: you can divide by 4 to get a number from 0 to 3; in other words, you can use a combination of remainder division (%) and integer division (/) to associate a number with a unique combination of rank and suit.)

Exercise 6.4.3. Precisely what changes do you need to make to simulate dealing from a six-deck "shoe"? This involves six complete 52-card decks shuffled together. Revise the code for the six-deck shoe, using card numbers 0 through 51 only—thus preserving the suit-assigning ability from the previous exercise. (Hint: you can use remainder division to convert a larger set of numbers into repetitions of 0 through 51.) How does dealing from this shoe affect the probability of poker hands? Is it more or less likely to be dealt four aces?

2-D Arrays: Into the Matrix

Most computer languages provide the ability not only to create ordinary, one-dimensional arrays, but to create multidimensional arrays as well. C++ is no exception.

Two-dimensional arrays in C++ have this form:

```
type  array_name[size1][size2];
```

The number of elements is size1 * size2, and the indexes in each dimension are 0-based just as in one-dimensional arrays. For example, consider this declaration:

```
int  matrix[10][10];
```

This creates a 10-by-10 array, having 100 elements. Each dimension has index numbers running from 0 to 9. The first element is therefore matrix[0][0], and the last element is matrix[9][9].

To process such an array programmatically, you need to use a nested loop with two loop variables. For example, this code initializes all the members of the array to 0:

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        matrix[i][j] = 0;
    }
}
```

Here is how this code works:

1. The variable i is set to 0, and a complete set of cycles of the inner loop—with j ranging from 0 to 9—is done first.
2. One cycle of the outer loop is then complete and i is incremented to the next higher value, which is 1. Then, all the cycles of the inner loop run again, with j (as always) ranging from 0 to 9.
3. The process is repeated until i is incremented past its terminal value, 9.

Consequently, the values of i and j will be (0, 0), (0, 1), (0, 2), ... (0, 9), at which point the inner loop is complete, i is incremented, and the inner loop begins again: (1, 0), (1, 1), (1, 2) and so on. In all, 100 operations will be performed, because each cycle of the outer loop, which runs 10 times, performs 10 cycles of the inner loop.

In C++ arrays, the index on the right changes the fastest. This means the elements matrix[5][0] and matrix[5][1] are next to each other in memory.

Chapter 6 *Summary*

Here are the main points of Chapter 6:

- Use bracket notation to declare an array in C++. Declarations have this form:

```
type   array_name[number_of_elements];
```

- For an array of size n, the elements have indexes ranging from 0 to n − 1.
- You can use loops to process arrays of any size efficiently. For example, assume an array was declared with SIZE_OF ARRAY elements. The following loop initializes every element to 0:

  ```
  for(int i = 0; i < SIZE_OF_ARRAY; ++i)
       my_array[i] = 0;
  ```

- You can use a list of values between set braces to initialize arrays:

  ```
  double  scores[5] = {6.8, 9.0, 9.0, 8.3, 7.1 };
  ```

- You can use the **string** class to declare a string variable. (I'll explain more about this type as well as traditional C-strings in Chapter 8.) For example:

  ```
  #include <string>
  using namespace std;
  ...
  string name = "Joe Bloe";
  ```

- You can then declare arrays of strings just as you can declare other kinds of arrays. For example:

  ```
  string band[ ] = {"John", "Paul", "George", "Ringo"};
  ```

- You can index arrays of strings just as you can other kinds of arrays:

  ```
  cout << "The leader of the group was " << band[0];
  ```

- C++ does not check array bounds for you at runtime (except in managed environments such as Visual Studio). Therefore, show care that you don't write array-access code that overwrites other areas of memory.
- Two-dimensional arrays are declared this way:

  ```
  type array_name[size1][size2];
  ```

7 Pointers: Data by Location

C and C++ programmers are sometimes thought to be a special breed, in part because they understand pointers. This also gives C++ a reputation for being difficult. "What's a pointer, anyway?" But the idea is fairly simple.

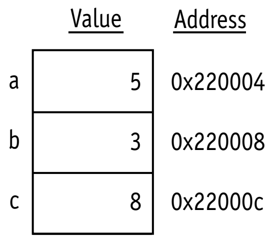
A pointer is just a variable that stores the location of another piece of data. Think of it this way: Sometimes it's easier to write down a location to a cabinet full of data rather than to copy all the contents.

Which would you rather do: spend all night copying the contents of a file cabinet, or just tell someone (assuming it's someone you trust) where the data is located? And consider what happens if you need to give this person the ability to change the data. Then you *must* give them the location of the original data, not copies of it.

If you can understand that, you can understand pointers.

What the Heck Is a Pointer, Anyway?

The CPU doesn't understand names or letters: It refers to locations in memory by number, or *address*. You usually don't know what these numbers are, although you can print them out if you want. For example, the computer might store variables a, b, and c at numeric addresses 0x220004, 0x220008, and 0x22000c. These are numbers in hexadecimal notation (that's base 16).

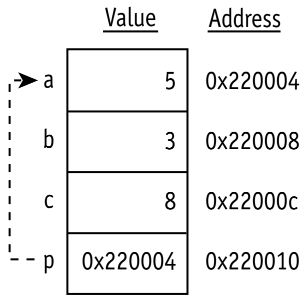


There's nothing magic about these particular addresses; they are just numbers I picked at random. In practice, many things will affect what addresses are used at runtime, and the physical addresses of your data will probably be different every time you run the program. You can't know in advance what addresses will be assigned to your variables, but you can use those addresses during run time, as you'll shortly see.

Now you're ready to understand what a pointer is.

The Concept of Pointer

A pointer is a variable that contains a numeric address. While most variables contain useful information (such as 5, 3, and 8 in this example), a pointer contains *the location of another variable*. So, a pointer is useful only as a way to get to something else. But—as with the file cabinet of data you'd rather not have to make copies of—sometimes it's much more efficient to use pointers, that is, to pass the location of the data, not copies of it.



Sometimes a function needs to send another function a large amount of data. One way to do this is to copy all that information and pass it along. But another, more efficient way is just to give the address of the data to work on.

By default, arguments in C++ are passed *by value*. When an argument is passed to a function, it gets its own copy of that value, which it can do anything with: manipulate the value, print it, double it, divide it—anything. But those changes only affect the temporary copies.

How, then, does a function change the value of a variable passed to it? One way to do that is to pass the location of the data. As with a file cabinet, if you tell people the location, they can modify the data. But if you give *copies* of the data, changes to that data will have no permanent effect.

There are still other reasons for using pointers. As you'll see in Chapter 12, "Two Complete OOP Examples," pointers enable you to create data structures with links to other data structures, to any level of complexity. So, you can have linked lists and internal networks in memory. Later in this book, we'll look at data structures like that.

Interlude

What Do Addresses Look Like?

In the previous section, I assumed the variables a, b, and c had the physical addresses 0x220004, 0x220008, and 0x22000c. These are hexadecimal numbers, meaning they use base 16.

There's a good reason for using hexadecimal notation. Because 16 is an exact power of 2 (2 * 2 * 2 * 2 = 16), each hexadecimal digit corresponds to a pattern of exactly four binary digits—no more, no less. Here's how hexadecimal digits work:

HEXADECIMAL DIGIT	EQUIVALENT DECIMAL	EQUIVALENT BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
a	10	1010
b	11	1011
c	12	1100
e	13	1101
e	14	1110
f	15	1111

The advantage of hexadecimal notation is its close relation to binary. For example, the hex numeral 8 is 1000 in binary, and hex numeral f is 1111. Therefore, 88ff is equivalent to 1000 1000 1111 1111.

▼ *continued on next page*

Interlude

▼ continued

For computer architects, who need to translate numbers into bit patterns quickly, this is essential. Also, because every hex digit corresponds to four binary digits—no more, no less—you can tell at a glance how wide an address is: 0x8000 has four digits, therefore it corresponds to precisely 16 binary digits.

Every computer architecture uses addresses of some fixed width. It's important to know at a glance whether an address is too large to fit on a certain computer.

At the time of this writing, 32-bit architecture is the universal norm on personal computers, although it will someday give way to 64-bit architecture. With 32-bit addresses, no address can have more than 32 binary digits (eight hexadecimal digits). Technically, all addresses should be expressed in eight hex digits: for example, 0x000080ff. For simplicity's sake, this chapter uses smaller addresses and neglects the leading zeroes, just to make the figures manageable on the page.

Remember that 32-bit addressing supports more than 4 billion locations in memory, although this is even now being exceeded by memory hardware, requiring clever work-arounds on the part of computer and operating-system designers. Someday, 64-bit architecture will support a virtually unlimited number of addresses.

Declaring and Using Pointers

Key Syntax

A pointer declaration uses the following syntax:

```
type  *name;
```

For example, you can declare a pointer p, which can point to variables of type **int**:

```
int  *p;
```

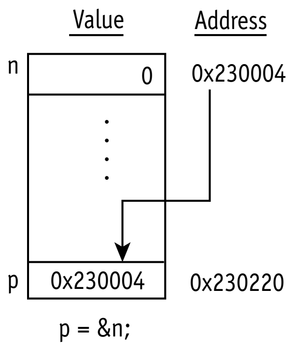
At the moment, the pointer is not initialized. All we know is that it can point to data objects of type **int**. But type matters. The base type of a pointer determines how the data it points to is interpreted: p has type **int***, so it should point only to **int** variables.

The next statements declare an integer n, initialize it to 0, and assign its address to pointer p:

```
int n = 0;
p = &n;             // p now points to n!
```

The ampersand (&) gets the address of its operand. You generally don't care what the address is. All that matters is that p contains the address of n; that is, p *points to* n and now you can use p to manipulate n.

After **p = &n** is executed, p contains the address of n. A possible memory layout for the program is shown here.

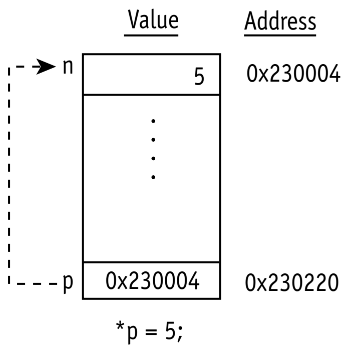


In all these examples, the addresses shown are arbitrary. A program will likely use different addresses every time it is run. The important thing about pointers is the relationships they create.

Here comes the interesting part: Applying the indirection operator (*) says "the thing pointed to." Assigning a value to *p has the same effect as assigning that value to n, because n is what p points to.

```
*p = 5;    // Assign 5 to the int pointed to by p.
```

So, because of the asterisk (*), this operation changes the thing *that p points to*, not the value of p itself. Now the memory layout looks like this:

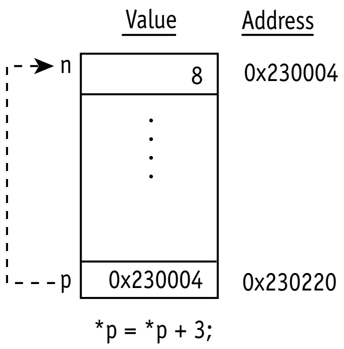


The effect of the statement, in this case, is the same as n = 5. The computer finds the memory location pointed to by p and puts the value 5 at that location.

You can use a value pointed to by a pointer both to get and to assign data. Here's another example of pointer use:

```
*p = *p + 3;    // Add 3 to the int pointed to by p.
```

The value of n changes yet again—this time from 5 to 8. The effect of this statement is the same as n = n + 3. The computer finds the memory location pointed to by p and adds 3 to the value at that location.



To summarize, when p points to n, referring to *p has the same effect as referring to n. Here are more examples:

WHEN P POINTS TO N, THIS STATEMENT	HAS THE SAME EFFECT AS THIS STATEMENT
`*p = 33;`	`n = 33;`
`*p = *p + 2;`	`n = n + 2;`
`cout << *p;`	`cout << n;`
`cin >> *p;`	`cin >> n;`

But if using *p is the same as using n, why bother with *p in the first place? One reason, remember, is that pointers enable a function to change the value of an argument passed to it. Here's how it works in C and C++:

1. The caller of a function passes the address of a variable to be changed. For example, the caller passes &n (the address of n).
2. The function has a pointer argument, such as p, that receives this address value. The function can then use *p to manipulate the value of n.

Example 7.1. *Print Out Addresses*

Before making practical use of pointers, let's print some data and compare pointer values to the standard **int** variables. What's essential here is to understand the difference between a variable's *content* and its *address*.

pr_addr.cpp

```
#include <iostream>
#include <stdlib.h>

using namespace std;

int main()
{
    int a = 2, b = 3, c = 4;
    int *pa = &a;
    int *pb = &b;
    int *pc = &c;
    cout << "Value of pointer pa is: " << pa << endl;
    cout << "Value of pointer pb is: " << pb << endl;
    cout << "Value of pointer pc is: " << pc << endl;
    cout << "The values of a, b, and c are: ";
    cout << a << ", " << b << ", " << c << endl;
    return 0;
}
```

When run, this program should output results similar (but not necessarily identical to) the following, which is what I got:

```
The value of pointer pa is: 0x22ff74
The value of pointer pb is: 0x22ff70
The value of pointer pc is: 0x22ff6b
The values of a, b, and c are: 2, 3, 4
```

This tells us that the values of a, b, and c are 2, 3, and 4. The addresses are expressed as hexadecimal numbers, 0x22ff74, 0x22ff70, and 0x22ff6b, but your results will vary. Physical addresses depend on many things you don't control.

What matters is that once you get a pointer—a variable containing another variable's address—you can use it to manipulate the thing to which it's pointing.

Although a, b, and c were declared in that order, my C++ compiler assigned their addresses in reverse order: c got a lower address than a. There's a lesson here: except in the case of array elements (which we'll get to in this chapter), and classes (which we'll get to later), never make assumptions about the ordering of variables in memory.

After a, b, and c are declared, the program declares pointers and initializes them to the addresses of a, b, and c. Remember that the ampersand (&) means "Get the address of."

```
int *pa = &a;
int *pb = &b;
int *pc = &c;
```

Example 7.2. *The double_it Function*

Now let's put pointers to use. This program uses a function named double_it, which doubles a variable passed to it or, rather, doubles a variable whose address is passed to it.

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

void double_it(int *p);

int main()
{
   int a = 5, b = 6;

   cout << "Val. of a before doubling: " << a << endl;
   cout << "Val. of b before doubling: " << b << endl;

   double_it(&a);      // Pass address of a.
   double_it(&b);      // Pass address of b.

   cout << "Val. of a after doubling: " << a << endl;
   cout << "Val. of b after doubling: " << b << endl;
   return 0;
}

void double_it(int *p) {
   *p = *p * 2;
}
```

How It Works

This is a straightforward program. The main function does just the following:

1 Print the values of a and b.

2 Call the double_it function to double the value of a by passing the address of a (&a).

3 Call the double_it function to double the value of b by passing the address of b (&b).

4 Print the values of a and b again.

This example needs pointers to work. You could write a version of double_it that took a simple **int** argument, but such a function would do nothing.

```
void double_it(int n) {    // THIS DOESN'T WORK!
    n = n * 2;
}
```

The problem is that when an argument is passed to a function, the function gets its own copy of the argument. As soon as the function returns, that copy is thrown away.

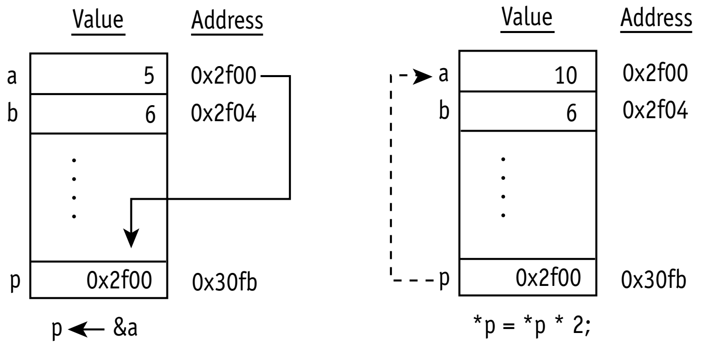
Getting a variable passed to you is like getting photocopies of a secret document. You can view the information, but you have no access to the originals. But getting a pointer is like getting the location of the original documents. You not only get to look at them, but you can also make changes. So to enable a function to change the value of a variable, use pointers.

```
void double_it(int *p);
```

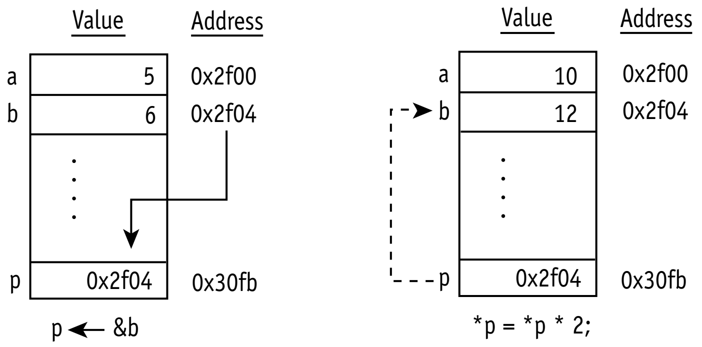
This declaration says that "the thing pointed to by p" has **int** type. Therefore, p itself is a pointer to an **int**. The caller must pass an address, which it does by using the address operator (&).

```
double_it(&a);

void double_it(int *p) {
   *p = *p * 2
}
```

Visually, here's the effect in terms of a hypothetical memory layout. The address of a is passed to the function, which then uses it to change the value of a.



The program then calls the function again, this time passing the address of b. The function now uses this address to change the value of b.



EXERCISES

Exercise 7.2.1. Write a program to call a function triple_it that takes the address of an **int** and triples the value pointed to. Test it by passing an argument n, which is initialized to 15. Print out the value of n before and after the function is called. (Hint: The function should look similar to double_it in Example 7.1. Remember to pass &n.)

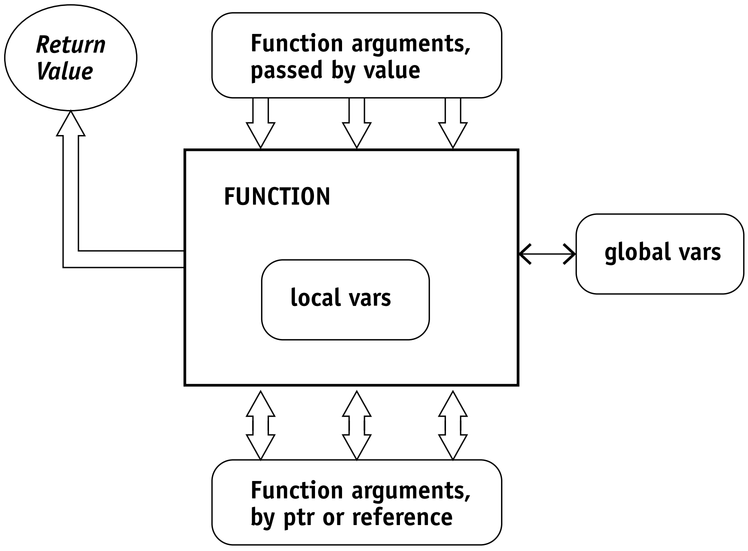
Exercise 7.2.2. Write a program with a function named convert_temp: The function takes the address of a variable of type **double** and applies Centigrade-to-Fahrenheit conversion. A variable that contains a Centigrade temperature should, after the function is called, contain the equivalent Fahrenheit temperature. Test the function. (Hint: The relevant formula is F = (C * 1.8) + 32.)

Data Flow in Functions

Passing a pointer is a way of achieving (or simulating) *pass by reference* of ordinary variables. In other words, by receiving a pointer value, a function gains the ability to manipulate not just *copies* of these values, but the values of the actual variables passed to it.

This has the obvious advantage of enabling a function to, in effect, pass back more than one value. Sometimes a function needs to set a whole series of values as part of its "data-output" effect—and simply having one return value is insufficient. One way to pass back information (outputting data to the rest of the program) is to have the function manipulate global variables, but it's best to limit the number of global variables if you can.

Now you know how to pass arguments directly (by value) or by reference through a pointer, and this enables you to write functions with a more sophisticated input/output data flow:


Return Value
Function arguments, passed by value
FUNCTION
local vars
global vars
Function arguments, by ptr or reference

Swap: Another Function Using Pointers

Suppose you have two **int** variables and you want to swap their values. It's easy to do this with a third variable, temp, whose function is to hold a temporary value.

```
temp = a;
a = b;
b = temp;
```

Now, wouldn't this be a useful bit of functionality to put into a function, which you could then call whenever you needed? So, for example, if you had two variables A and B, and you wanted to exchange their values, you could call a swap function.

This looks good, but remember: unless you pass pointers to the variables (that is, pass their addresses), changes to the variables are ignored.

Here's a solution that works, using pointers to enable the function to alter the variables:

```
// Swap function.
// Swap the values pointed to by p1 and p2.
//
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

The expressions *p1 and *p2 are integers, and you can use them as you would any integer variables. But remember that p1 and p2 are also addresses, and the addresses themselves do not change. The data that's altered is the data *pointed to* by p1 and p2. This is easy to see with an example.

Assume that big and little are initialized to 100 and 1, respectively.

```
int big = 100;
int little = 1;
```

The following statement calls the swap function, passing the addresses of these two variables. Note the use of the address operator (&) here:

```
swap(&big, &little);
```

Now if you print these variables, you'll see that the values have been exchanged.

```
cout << "The value of big is now " << big << endl;
cout << "The value of little is now " << little;
```

The values at the addresses in p1 and p2 change, not p1 and p2 themselves. This is why the indirection operator (*) is often called the *at* operator.

Example 7.3. Array Sorter

Now it's time to show the power of this swap function. Pointers are not limited to pointing to simple variables, although I used that terminology to keep things simple. An **int** pointer, for example, can point to any memory location that stores an **int** value. This means it can point to elements of an array as well as point to a variable.

Here, for example, the swap function is used to swap the values of two elements of an array named arr:

```
int arr[5] = {0, 10, 30, 25, 50};

swap(&values[2], &values[3]);
```

Given the right procedure, you can use the swap function to sort the values of an array. Take a look at arr again—this time with the data jumbled around.

30	25	0	50	10
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]

Here's a straightforward solution—the well-known (but not sophisticated) "selection sort" algorithm:

1. Find the lowest value and put that value in arr[0].
2. Find the *next* lowest value and put that value in arr[1].
3. Continue in this manner until you get to the end.

Here is the solution written out in pseudocode:

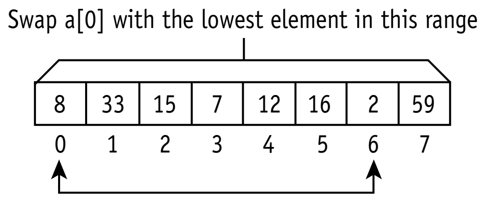
```
For i = 0 to n – 2,
    Find the lowest value in the range a[i] to a[n–1]
    If i is not equal to the index of the lowest value found,
        Swap a[i] and a[index_of_lowest]
```

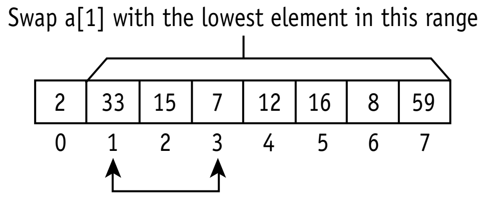
That's the plan. The effect will be to put the lowest value in a[0], the next lowest value in a[1], and so on. Note that by

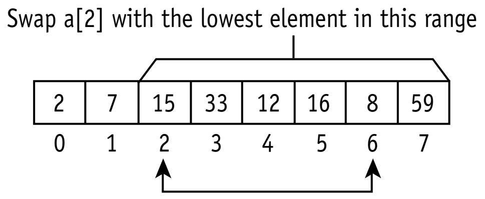
```
For i = 0 to n - 2,
```

I mean a **for** loop in which i is set to 0 during the first cycle of the loop, 1 during the next cycle of the loop, and so on, until i is set to n – 2, at which point it completes the last cycle. Each cycle of the loop places the correct element in a[i] and then increments i.

Inside the loop, a[i] is compared to all the *remaining elements* (the range a[i] to a[n – 1], which includes all elements on the *right*). By the time every value of i has been processed, the whole array will have been sorted. Here's an example of the first three cycles of the loop, illustrated:







But how do we find the lowest value in the range a[i] to a[n − 1]? We need another algorithm.

What the following algorithm does is 1) start by assuming that i is the lowest element and so initialize "low" to i; and 2) whenever a lower element is found, this lower element becomes the new "low" element.

To find the lowest value in the range a[i] to a[n − 1], do the following:

```
Set low to i
For j = i + 1 to n − 1,
    If a[j] is less than a[low]
        Set low to j
```

We then combine the two algorithms. After this, it's an easy matter to write the C++ code.

```
For i = 0 to n − 2,
    Set low to i
```

```
For j = i + 1 to n – 1,
    If a[j] is less than a[low]
        Set low to j
If i is not equal to low,
    Swap a[i] and a[low]
```

Here's the complete program that uses this algorithm to sort an array:

sort.cpp

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

void sort(int n);
void swap(int *p1, int *p2);

int a[10];

int main ()
{
    for (int i = 0; i < 10; ++i) {
        cout << "Enter array element #" << i << ": ";
        cin >> a[i];
    }
    sort(10);

    cout << "Here is the array, sorted:" << endl;
    for (int i = 0; i < 10; ++i) {
        cout << a[i] << "  ";
    }
    return 0;
}

// Sort function: sort array named a with n elements.
//
void sort (int n) {
    int lowest = 0;

    for(int i = 0; i < n - 1; ++i) {
```

▼ continued on next page

sort.cpp, cont.

```
            // This part of the loop finds the lowest
            //  element in the range i to n-1; the index
            //  is set to the variable named low.

            low = i;
            for (int j = i + 1; j < n; ++j) {
                if (a[j] < a[low]) {
                    low = j;
                }
            }

            // This part of the loop performs a swap if
            //  needed.

            if (i != low) {
                swap(&a[i], &a[low]);
            }
        }
    }

    // Swap function.
    // Swap the values pointed to by p1 and p2.
    //
    void swap(int *p1, int *p2) {
        int temp = *p1;
        *p1 = *p2;
        *p2 = temp;
    }
```

How It Works

Only two parts of this example are directly relevant to understanding pointers. The first is the call to the swap function, which passes the addresses of a[i] and a[low]:

```
swap(&a[i], &a[low]);
```

An important point here is that you can use the address operator (&) to take the address of array elements, just as you can use it with variables.

The other part of the example that's relevant to pointer use is the function definition for swap, which I described in the previous section.

```
// Swap function.
// Swap the values pointed to by p1 and p2.
//
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

As for the sort function, the key to understanding it is to note what each part of the main loop does. The main **for** loop successively sets i to 0, 1, 2, …, up to and including n − 2. Why n − 2? It's because by the time it gets to the last element (n − 1), all the sorting will have been done. (There is no need to compare the last element to itself.)

```
for(int i = 0; i < n - 1; ++i) {
    //...
}
```

The first part of the loop finds the lowest element in the range that includes a[i] and all the elements *to its right*. An inner loop conducts this search using a variable, j, initialized to start at i + 1 (one position to the right of i).

```
low = i;
for (int j = i + 1; j < n; ++j) {
    if (a[j] < a[low]) {
        low = j;
    }
}
```

This, by the way, is an example of a *nested loop*, and it's completely legal. A **for** statement is just another kind of statement; therefore, it can be put inside another **if, while,** or **for** statement, and so on, to any degree of complexity.

The other part of the loop has an easy job. All it has to do is ask whether i differs from the index of the lowest element (stored in the variable "low"). Remember that the != operator means "not equal." There's no reason to do the swap if a[i] is already the lowest element in the range; that's the reason for the **if** condition here.

```
if (i != low) {
    swap(&a[i], &a[low]);
}
```

EXERCISES

Exercise 7.3.1. Rewrite the example so that instead of ordering the array from low to high, it sorts the array in reverse order: high to low. This is easier than it may look. It's helpful, for the sake of clarity, if you rename the variable low as "high." Otherwise, you need to change only one statement; this statement does the comparison.

Exercise 7.3.2. Rewrite the example so that it sorts an array that has elements of type **double**. It's essential that you rewrite the swap function to work on data of the right type for the example to work correctly. But note that you should not change the type of any variables that serve as loop counters or array indexes—such variables should always have type **int**, regardless of the remaining data's type.

Exercise 7.3.3. Revise the example so it implements the bubble-sort algorithm, which is potentially faster than the selection-sort algorithm. Bubble sort compares each element to its neighbor and swaps if they're not in order. After this is done for the whole array, the highest value in the array "bubbles up" to the highest array position. After the whole array is processed, then the first n − 1 elements are processed, then the first n − 2 elements, and so on, each time putting the highest element remaining into the rightmost position. The advantage of the algorithm is that if the array is sorted at any point, it can quit early.

Here is the pseudocode for the bubble sort of Exercise 7.3.3:

```
For I equals N – 1 down to but not including 0:
    For J equals 0 up to but not including I:
        Set in_order flag to true
        If arr[J + 1] < arr[J]
            Swap arr[J + 1], arr[J]
            Set in_order flag to false
    If in_order, break out of loop
```

If you choose, you can implement this algorithm with no reference to the in_order flag, but then you won't be able to take advantage of early exit. This means possibly longer execution time, but less code to write.

Reference Arguments (&)

The previous section implemented something called "pass by reference," although technically, what it did was to pass pointers.

In classic C, that was as close as you could get to passing by reference, and therefore, use of pointers was generally mandatory. It was difficult for any serious program to avoid them. But in C++, you have the option of using a reference argument.

To declare a reference argument in C++, apply the ampersand (&) to the declaration. In other contexts, this is the "address" operator, but in the context of a declaration, it turns a symbol into a reference—which is an alias for another variable. For example:

```
void swap(int &a, int &b);
```

The function definition then just manipulates the arguments normally, not treating them as pointers. Because these are reference arguments, declared with &, changes to the arguments are permanent and will affect the caller. Notice that no pointer syntax is involved when you use this approach:

```
void swap(int &a, int &b) {
    int temp = p1;
    p1 = p2;
    p2 = temp;
}
```

The advantage of this approach is that once they are declared as reference arguments, you can then pass arguments without having to take their address or use pointer syntax. This is an easier way to implement pass by reference.

```
swap(a[i], a[low]); // Swap a[i] and a[low]
```

Under the covers, references are frequently implemented through the use of pointers, even though this under-the-cover stuff is kept invisible to you. It's worth understanding how pointers work, however, because they have so many other applications.

Pointer Arithmetic

One of the important uses of pointers is to process arrays efficiently. Suppose you declare this array:

```
int arr[5] = {5, 15, 25, 35, 45};
```

Of course, the elements arr[0] through arr[4] can all be used like individual integer variables. You can, for example, write statements such as "arr[1] = 10;". But what is the expression "arr" itself? Can "arr" ever appear by itself?

Yes, it can because "arr" is a constant that translates into an address—specifically, the address of the first element. Because it's a constant, you cannot change the value of "arr" itself. You can, however, use it to assign a value to a pointer variable:

```
int *p;
p = arr;
```

The statement "p = arr" is equivalent to this:

```
p = &arr[0];
```

The former expression (p = arr) is a more concise, cleaner way to initialize a pointer to the address of the first element, arr[0]. Is there a similar technique for the other elements? You betcha. For example, to assign p the address of arr[2], you use this:

```
p = arr + 2;                    // p = &arr[2];
```

C++ interprets all array names as address expressions. arr[2] translates into the following:

```
*(arr + 2)
```

If you've been paying attention, you may think this looks wrong. We add 2 to the address of the start of the array, arr. But the element arr[2] is not 2, but 8 bytes away (4 for each integer—assuming you are using a 32-bit system)! Yet this still works. Why?

It's because of *pointer arithmetic*. Only certain arithmetic operations are allowed on pointers and other address expressions (such as arr). These are as follows:

Key Syntax

```
address_expression + integer
integer + address_expression
address_expression - integer
address_expression - address_expression
```

When integer and address expressions are added together, the result is another address expression. Before the calculation is completed, however, the integer is automatically *scaled* by the size of the base type. The C++ compiler performs this scaling for you.

```
new_addr = old_addr + (integer * size_of_base_type)
```

So, for example, if p has base type **int**, adding 2 to p has the effect of increasing it by 8 because 2 times the size of the base type (4 bytes) yields 8.

Scaling is an extremely convenient feature of C++, because it means that when a pointer p points to an element of an array and it is incremented by 1, this always has the effect of making p point to the next element:

```
++p;     // Point to next element in the array.
```

This is one of the most important things to remember when using pointers:

When an integer value is added or subtracted from an address expression, the compiler automatically multiplies that integer by the size of the base type.

Another way of saying this is to say that adding N to a pointer produces an address N elements away from the original pointer value.

Address expressions can also be compared to each other. You should not make assumptions about a memory layout except where array elements are involved. The following expression is always true:

```
&arr[2] < &arr[3]
```

This is another way of saying that the following is always true, just as you'd expect:

```
arr + 2 < arr + 3
```

Pointers and Array Processing

Because pointer arithmetic works the way it does, functions can access elements through pointer references rather than array indexing. The result is the same, but the pointer version (as I'll show) executes slightly faster.

In these days of incredibly fast CPUs, such minor speed increases make little difference for most programs. CPU efficiency was far more important in the 1970s and 1980s, with their slow processors. CPU time was often at a premium.

But for a certain class of programs, the superior efficiency gained from C and C++ can still be useful. C and C++ are the languages of choice for people who write operating systems, and subroutines in an operating system or device driver may be called upon to execute thousands or even millions of times a second. In such cases, the small efficiencies due to pointer usage can be significant.

Here's a function that uses a pointer reference to zero out an array with n elements:

```
void zero_out_array(int *p, int n) {
    while (n-- > 0) {  // Do n times:
        *p = 0;        //  Assign 0 to element pointed
                       //    to by p.
```

```
            ++p;           //  Point to next element.
        }
    }
```

This is a remarkably compact function that would appear more compact still without the comments (but remember that comments have no effect on a program at runtime). Here's another version of the function, using code that may look more familiar:

```
    void zero_out_array2(int *arr, int n) {
        for (int i = 0; i < n; i++) {
            arr[i] = 0;
        }
    }
```

But this version, while nearly as compact, may run a bit slower (depending on the ability of the compiler to optimize the runtime code). The value of i must be scaled and added to arr each and every time through the loop to get the location of the array element arr[i].

```
    arr[i] = 0;
```

This, in turn, is equivalent to the following:

```
    *(arr + i) = 0;
```

It's actually worse than that, because the scaling effect has to be done at runtime; so at the level of machine code, the calculation is as follows:

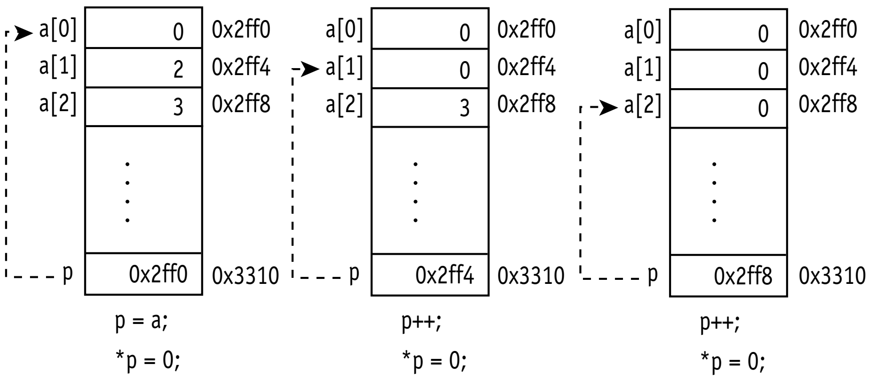
```
    *(arr + (i * 4)) = 0;
```

The problem is that the address has to be recalculated over and over again. In the pointer version, the address arr is figured in only once. The loop statement does less work.

```
    *p = 0;
```

Of course, p has to be incremented each time through the loop, but both versions have a loop variable to update. Incrementing p is no more work than incrementing i.

Here's how the pointer version works. Each time through the loop, *p is set to 0, and then p itself is incremented to the next element. (Because of scaling, p is actually increased by 4 each time through the loop, but that's an easy operation.)



Example 7.4. *Zero Out an Array*

Here's the zero_out_array function in the context of a complete example. All this program does is initialize an array, call the function, and then print the elements so that you can see how it worked.

zero_out.cpp

```
#include <iostream>

using namespace std;

void zero_out_array(int *arr, int n);

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int main() {

    zero_out_array(a, 10);

    // Print out all the elements of the array.

    for (int i = 0; i < 10; ++i) {
        cout << a[i] << "  ";
    }
    return 0;
}
```

▼ continued on next page

zero_out.cpp, cont.

```
// Zero-out-array function.
// Assign 0 to all elements of an int array of size n.
//
void zero_out_array(int *p, int n) {
    while (n-- > 0) {  // Do n times:
      *p = 0;          //  Assign 0 to element pointed
                       //    to by p.
      ++p;             //  Point to next element.
    }
}
```

How It Works

The key to understanding the zero_out function is to remember that adding 1 to a pointer makes it point to the next element of an array.

```
++p;
```

This example demonstrates how to pass an array in C++. The first argument shown here, a, translates into the address of the first element:

```
zero_out_array(a, 10);
```

Therefore, to pass an array, just use the array name. The function gets the address of the first element and should treat this as a pointer value.

Writing More Compact Code

In Example 7.3, the **while** loop in the zero_out_array function does two things: it zeros out an element and then increments the pointer so it points to the next element.

```
while (n-- > 0) {
    *p = 0;
    ++p;
}
```

If you recall from past chapters, p++ is just an expression, and expressions can always be used within larger expressions. That means we can combine the pointer-access and increment operations to produce this:

```
while (n-- > 0) {
    *p++ = 0;
}
```

To properly interpret *p++, I have to introduce two aspects of expression evaluation: precedence and associativity. Operators such as assignment (=) and test for equality (==) have low precedence, meaning that they are applied after other operations are resolved.

The pointer-indirection (*) and increment (++) operators both have the same level of precedence, but (unlike most operators) they associate right to left. Therefore, the statement *p++ = 0; is evaluated as if it were written this way:

```
*(p++) = 0;
```

This means increment pointer p, but only after using its value in this operation:

```
*p = 0;
```

Incidentally, using parentheses differently would produce an expression that is legal, but not, in this case, useful.

```
(*p)++ = 0;  // Assign 0 to *p and then increment *p.
```

The effect of this statement would be to set the first array element to 0 and then to 1, over and over; p itself would never be incremented, and you'd fail to process most of the array. The expression (*p)++ says, "Increment the thing p points to," not p itself.

Whew! That's a lot of analysis required to understand a tiny piece of code. You're to be forgiven if you swear never to write such cryptic statements yourself. But statements such as "*p++ = 0" are actually pretty common.

Note ▶ Appendix A, "Operators," summarizes precedence and association for all C++ operators.

EXERCISES

Exercise 7.4.1. Rewrite the program to use a direct-pointer reference for the loop that prints out the values of the array. Declare a pointer p and initialize it to start the array. The loop condition should be p < a + 10.

Exercise 7.4.2. Write and test a copy_array function that copies the contents of one **int** array to another array of the same size. The function should take two pointer arguments. The operation inside the loop should be as follows:

```
*p1 = *p2;
p1++;
p2++;
```

If you want to write more compact but cryptic code, you can use this statement:

```
*(p1++) = *(p2++);
```

Or, you can even use the following, which means the same thing:

```
*p1++ = *p2++;
```

Chapter 7 *Summary*

Here are the main points of Chapter 7:

- A pointer is a variable that can contain a numeric memory address. You can declare a pointer by using the following syntax:

  ```
  type *p;
  ```

- You can initialize a pointer by using the address operator (&).

  ```
  p = &n;            // Assign address of n to p.
  ```

- Once a pointer is initialized, you can use the indirection operator (*) to manipulate data pointed to by the pointer.

  ```
  p = &n;
  *p = 5;            // Assign 5 to n.
  ```

- To enable a function to manipulate data (pass by reference), pass an address.

  ```
  double_it(&n);
  ```

- To receive an address, declare an argument that has a pointer type.

  ```
  void double_it(int *p);
  ```

- An array name is a constant that translates into the address of its first element. A reference to an array element a[n] is translated into the pointer reference, *(a + n).

- When an integer value is added to an address expression, C++ performs scaling, multiplying the integer by the size of the expression's base type.

  ```
  new_addr = old_addr + (integer * size_of_base_type)
  ```

- The unary operators * and ++ operators associate right-to-left. Consequently, this expression

  ```
  *p++ = 0;
  ```

  does the same thing as the following expression, which sets *p to 0 *and then* increments the pointer p to point to the next element:

  ```
  *(p++) = 0;
  ```

8 Strings: Analyzing the Text

Most computer programs, at some point in their lives, have to communicate with a human being. The standard way to do this is to use text strings. The word *string* conjures up an image of a series of characters, closely strung together.

However, there seems to be a paradox: Computer processors only understand numbers. How, then, can they communicate with humans? The answer is: through a special kind of code that associates each letter with a number. That's the basis for understanding text strings, so this chapter starts by examining that subject.

For years now, C++ compilers have supported a sophisticated **string** class that makes working with text strings easier. For example, the following line of code concatenates strings without paying any attention to string lengths or capacities—it just magically works.

```
string  titled_name = "Sir " + beatle_name;
```

This chapter begins by presenting the "old school" C-string type, but if you want to work with the sophisticated, and easier-to-use, **string** class, you can go directly to the section of this chapter titled "The C++ String Class" (page 201).

Text Storage on the Computer

In Chapter 1, "Start Using C++," I stated the computer stores text numerically, just like any other kind of data. But with text data, each byte uses a special code that corresponds to a particular character: This is called ASCII code. Suppose I declared the following string:

```
char str[] = "Hello!";
```

C++ allocates exactly seven bytes—one byte for each character and one for a terminating null byte. This is the standard "C-string," as opposed to the sophisticated (and easier-to-use) **string** class. A C-string is a simple array of **char**. Here's what the string data looks like in memory:

Actual data:	72	101	108	108	111	33	0
ASCII code for:	'H'	'e'	'l'	'l'	'o'	'!'	(null)

You can turn to Appendix D and see the ASCII code for every character. In reality, the computer doesn't actually store alphanumeric characters; it stores only numbers. When and how, then, are the numeric values translated into text characters?

This translation happens at least two times: when data is typed at the keyboard and when it's displayed on the monitor. When you press H on the keyboard, a series of actions happen at a low level that result in the ASCII code for H (72) being read into your program, which then stores that value as data.

The rest of the time, a text string is just a series of numbers—or more specifically, a series of bytes ranging in value from 0 to 255. But as programmers, we can think of C++ storing text characters in memory, one byte per character. (Exception: The international standard, Unicode, uses more than one byte per character.)

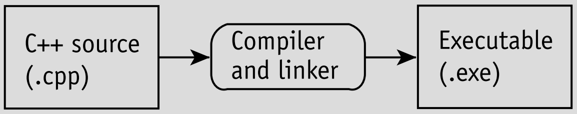
Interlude

How Does the Computer Translate Programs?

Programming books sometimes point out that the CPU doesn't understand the C++ language. All the C++ statements must be translated into machine code before they can be executed. But who or what does the translation?

Oh, that's no mystery, they say; the translation is done by the compiler, which itself is a computer program. But in that case, the computer is doing the translation.

When I was first learning how to program, this seemed to me an insolvable paradox. The CPU (the "brain" at the heart of the computer) doesn't understand a word of C++, yet it performs the translation between C++ and its own internal language. Isn't that a contradiction?



A large part of the answer is this: C++ source code is stored in a text file, just as you might store an essay or a memo. But text characters, as I've

Interlude

▼ continued

pointed out, are stored in numeric form. Therefore, when the compiler works on this data, it's doing another form of number crunching, evaluating data and making decisions according to precise rules.

In case that doesn't clear things up, imagine this: You have the task of reading letters from a person who knows Japanese but no English. You, meanwhile, know English but not one word of Japanese. (My apologies to philosopher John Searle, who originated this idea with his "Chinese Room" thought experiment.)

But suppose you have an instruction book that tells you how to translate Japanese characters into their English-language equivalent. The instruction book itself is written in English, so you have no problem using it.

So, even though you don't understand Japanese, you're able to translate all the Japanese you want by carefully following instructions.

That's really what a computer program is—an instruction book read by the CPU. A computer program is an inert thing—a sequence of instructions and data—yet the "knowledge" inside a computer arises from its programs. Programs enable a computer to do all kinds of clever things, including translating a text file containing C++.

A compiler, of course, is a very special program, but what it does is not at all strange or impossible. As a computer program, it's an "instruction book" as described. What it tells the computer is how to read a text file containing C++ source code and output *another* instruction book: this output is your C++ program in executable form.

The very first compilers had to be written in machine code. Later, old compilers could be used to write new compilers; so, through a bootstrap process, even skilled programmers were writing machine code with less frequency.

It Don't Mean a Thing if It Ain't Got that String

If you read Chapter 6, "Arrays: All in a Row...," you may have already guessed what a string is: an array. More specifically, a string is an array of base type **char**.

Technically, **char** is an integer type, one byte wide, large enough to store 256 different values (ranging from 0 to 255). This is more than enough space to contain all the different ASCII codes for the standard set of characters, including uppercase and lowercase letters, as well as numerals and punctuation marks. (However, note that some languages, such as Japanese and Chinese, have far more than 256 characters and therefore require a wider character type.)

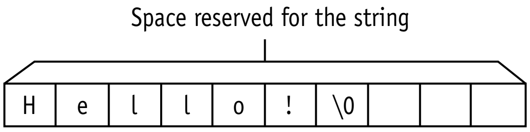
You can, if you want, create a **char** array of a definite size but no initial values:

```
char str[10];
```

This creates a string that can hold up to 10 bytes but has yet to be initialized. More often, programmers give initial values to a string when they declare it; for example:

```
char str[10] = "Hello!";
```

This declaration creates the array of **char** shown and equates the name str to the starting address of this array. (Remember that the name of an array always translates into its starting address.) This figure shows only the characters represented, not the ASCII codes—but underneath, it's all numbers.

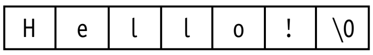


The character \0 is C++ notation for a null character: it means the actual value 0 is stored in this byte (as opposed to value 48, the ASCII code for the digit "0"). A C++ string terminates with a null byte, which indicates where the string data ends.

If you don't specify a definite size but initialize the string anyway, C++ allocates just enough space necessary for the string (including its null-terminator byte).

```
char s[] = "Hello!";
char *p  = "Hello!";
```

The effect of these two statements is roughly the same, but there are some differences: s is considered to name an array; therefore s itself is a constant that cannot change. But technically speaking, p is a pointer rather than an array, and it can be reassigned to point to other locations. In either case, C++ allocates just enough space in the data segment and assigns the starting address to the name s (which can't change) or, alternatively, to the initial value of p (which can).



String-Manipulation Functions

Just as it provides math functions to crunch numbers, C++ provides functions to manipulate strings. These functions take pointer arguments; that is, they get the addresses of the strings, but they work on the string data pointed to.

Here are some of the most commonly used string functions:

FUNCTION	DESCRIPTION
strcpy(*s1*, *s2*)	Copy contents of s2 to destination string s1
strcat(*s1*, *s2*)	Concatenate (join) contents of s2 onto the end of s1
strlen(*s*)	Return length of string s (not counting terminating null)
strncpy(*s1*, *s2*, *n*)	Copy s2 to s1, but copy no more than n characters
strncat(*s1*, *s2*, *n*)	Concatenate contents of s2 onto the end of s1, copying no more than n characters

Possibly the most common are **strcpy** ("string copy") and **strcat**, which stands for "string concatenation." Here's an example of their use:

```
char s[80];
strcpy(s, "One");
strcat(s, "Two");
strcat(s, "Three ");
cout << s;
```

This produces the following output:

```
OneTwoThree
```

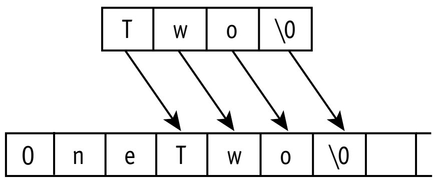
This example illustrates some important points:

- The string variable, s, must be declared with enough space to hold all the characters in the resulting string. C++ does nothing to ensure that there is space enough to hold all the string data necessary; this is your responsibility.
- Although the string is not initialized, 80 bytes are reserved for it. This example assumes that storing 80 characters (including the null) will be sufficient.
- The string literals "One" and "Two" and "Three" are arguments. When a string literal appears in code, C++ allocates space for the string and returns the address of the data: That is, in the C++ code, a string name is treated as equivalent to an address. Therefore, "Two" and "Three" are interpreted as address arguments.

The action of the statement

```
strcat(s, "Two");
```

looks like this:



You incur a risk with these string functions, as you can see: How do you guarantee that the first string is large enough for the existing string data along with the new? One approach is to make the target string so large you don't think its capacity will ever be exceeded.

A more secure technique is to use the **strncat** and **strncpy** functions. Each of these functions avoids copying more than n characters, including the terminating null. For example, the following operation cannot exceed the memory allocated for s1:

```
char s1[20];
// . . .
strncpy(s1, s2, 20);
strncat(s1, s3, 20 - strlen(s1));
```

Example 8.1. *Building Strings*

Let's start with a simple string operation: building a long string out of smaller strings. The following program gets a couple of strings from the user (by calling the **getline** function, described later), builds a larger string, and then prints the results:

buildstr.cpp

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char str[600];
    char name[100];
    char addr[200];
    char work[200];
```

buildstr., cont.

```
    // Get three strings from the user.

    cout << "Enter name and press ENTER: ";
    cin.getline(name, 100);
    cout << "Enter address and press ENTER: ";
    cin.getline(addr, 200);
    cout << "Enter workplace and press ENTER: ",
    cin.getline(work, 200);

    // Build the output string, and then print it.

    strcpy(str, "\nMy name is ");
    strcat(str, name);
    strcat(str, ", I live at ");
    strcat(str, addr);
    strcat(str, ",\nand I work at ");
    strcat(str, work);
    strcat(str, ".");

    cout << str << endl;
    return 0;
}
```

Here's a sample session using this program:

```
Enter name and press ENTER: Niles Cavendish
Enter address and press ENTER: 123 May Street
Enter work and press ENTER: Bozo's Carnival of Fun

My name is Niles Cavendish, I live at 123 May Street,
and I work at Bozo's Carnival of Fun.
```

How It Works

This example starts with a new include-file directive:

```
#include <cstring>
```

This is needed because it brings in declarations for the **strcpy** and **strcat** functions. As a general rule, using any standard-library function that begins with the three letters *str* requires you to include <cstring>.

The first thing that the main function does is to declare a series of strings to hold data. The program assumes these strings are sufficiently large that they won't be exceeded:

```
        char str[600];
        char name[100];
        char addr[200];
        char work[200];
```

It seems absurd that you'd ever want to enter a name longer than 100 characters, so these limits are probably sufficient, especially if you're writing the program for your own use.

But of course any such limits *can* be exceeded, and if you write programs for large numbers of other people, it's wise to assume that users are going to test every limit they can at some point. (This problem is addressed in Exercise 8.1.1.)

The other part of the example that's new is the use of the **getline** member function (or *method*):

```
    cin.getline(name, 100);
```

The **getline** method gets an entire line of input—all the characters input before the user pressed ENTER. The first argument (in this case, name) specifies the destination string. The second argument specifies the maximum number of characters to copy; this should never be more than n − 1, where n is the number of bytes allocated for the string.

After entering input into the three strings—name, addr, and work—the program builds the string. The first call is to **strcpy**, which copies string data to the beginning of str. (Calling **strcat** wouldn't produce correct results in this case, unless you knew that the first byte of str was a null—not a safe assumption here.)

```
    strcpy(str, "\nMy name is ");
```

The characters \n are a C++ *escape sequence*: They are not intended literally but instead represent a special character. In this case, \n denotes a newline character.

The program builds the rest of the string by calling **strcat** repeatedly.

```
        strcat(str, name);
        strcat(str, ", I live at");
        strcat(str, addr);
        strcat(str, ",\nand I work at ");
        strcat(str, work);
        strcat(str, ".");
```

EXERCISES

Exercise 8.1.1. Rewrite the example so that it cannot exceed the limits of str. For example, you'd replace the statement

```
strcat(str, addr);
```

with the following statement:

```
strncat(str, addr, 600 - strlen(str));
```

Exercise 8.1.2. After completing **Exercise 8.1.1,** test it by experimenting with different limitations for the str string. It helps if you replace the number 600 with the symbolic constant STRMAX, putting the following **#define** directive at the beginning of the program. During preprocessing, this directive causes the compiler to replace occurrences of STRMAX in the source code with the indicated text (600).

```
#define STRMAX 600
```

You can then use STRMAX to declare the length of str

```
char str[STRMAX];
```

and then use STRMAX to determine how many bytes to copy:

```
    strncpy(str, "\nMy name is ", STRMAX);
    strncat(str, name, STRMAX - strlen(str));
```

The beauty of this approach is that if you need to change the maximum string size, you need to change only one line of code (the line containing the **#define** directive) and then recompile.

Interlude

What about Escape Sequences?

Escape sequences can create some odd-looking code, if you're not used to them. Consider this statement:

```
cout << "\nand I live at";
```

This has the same effect as the following:

```
cout << endl << "and I live at";
```

The key to understanding an odd-looking string such as \nand is to remember this rule:

▼ *continued on next page*

Interlude

▼ continued

✱ In C++ source code, when the compiler reads a backslash (\\), the very next character is interpreted as having a special meaning.

In addition to \n, which represents a newline, other escape sequences include \t (tab) and \b (backspace).

Now, if you have an inquiring mind, you may be asking this: How do I print an actual backslash? The answer is simple. Two backslashes in a row (\\\\) represent a single backslash. For example, consider this statement:

```
cout << "\\nand I live at";
```

This prints the following text:

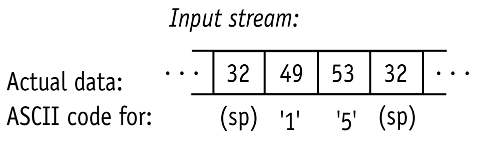
```
\nand I live at
```

Note that Chapter 17, "New Features of C++14," explains how to create "raw string literals," which give no special meaning to the backslash (\\).

Reading String Input

So far, I've been treating data input in a simplistic way. In previous examples, I assumed that the user types a number—for example, 15—and that this value is entered directly into the program. Actually, there's more to it than that.

All the data entered with a keyboard is initially text data: This means ASCII codes. So, when you're a user and you press 1 and 5 on the keyboard, the first thing that happens is that these characters are entered into the input stream.



The **cin** object, which has been told to get a number, analyzes this text input and produces a single integer value, in this case the value 15. That number gets assigned to an integer variable in a statement such as this one:

```
cin >> n;
```

If the type of n were different (say, if it had type **double**), a different conversion would be called for. Floating-point format requires a different kind of value to be produced. Normally, the stream-input operator (>>), as interpreted by the **cin** object, handles all this for you.

The previous section introduced the **getline** method, which has some strange-looking syntax:

```
cin.getline(name, 100);
```

The dot (.) is necessary to show that **getline** is a member of the object **cin**. Admittedly, there's some new terminology here.

I'll explain a lot more about objects starting in Chapter 10, "Classes and Objects." For now, think of an object as a data structure that comes with built-in knowledge of how to do certain things. The way you call upon an object's abilities is to call a member function:

Key Syntax

```
object.function(arguments)
```

The *object* is what the function applies to—in this case, **cin**. The *function* in this case is **getline**. (Also, file-input objects, introduced in Chapter 9, "Files: Electronic Storage," support this function.) Calling **cin.getline** is an alternative to getting input by using the stream operator (>>):

```
cin >> var;
```

We've seen this kind of statement used to get **int** and **double** data. Can you use it with strings? Yes.

```
cin >> name;
```

The problem with this statement is that it doesn't do what you might expect. Instead of getting an entire line of input—that is, all the data that the user types before pressing ENTER—it gets data up to the first white space ("white space" being programmer-ese for a blank space, tab, or newline). So, given this line of input,

```
Niles Cavendish
```

the effect of "cin >> name" would be to copy the letters "Niles" into the string variable, name, while "Cavendish" would remain in the input stream to be picked up by the next input operation.

So, assume the user types in the following and then presses ENTER:

```
50 3.141592 Joe Bloe
```

This works fine if you're expecting two numbers and two strings separated by a blank space. Here's the statement that would successfully read the input:

```
cin >> n >> pi >> first_name >> last_name;
```

But in general, the use of the stream input operator decreases your control. I avoid it myself, except for simple test programs. One of the limitations of this operator is that it doesn't allow you to set a default value. Suppose, for example, you prompt for a number:

```
cout << "Enter number: ";
cin >> n;
```

If the user presses ENTER without typing anything, nothing happens. The computer just sits there, waiting for the user to type a number and press ENTER again. If the user keeps pressing ENTER, the program will wait forever, like a stubborn child.

Personally, I think it's much better to have the program support the behavior implied by the following prompt:

```
Enter a number (or press ENTER to specify 0):
```

Wouldn't you find it convenient to have 0 (or whatever number you choose) as a default value? But how do you implement this behavior? This next example demonstrates how.

Note ▶ If you use the **getline** function at all, you may find that further operations using the stream input operator (>>) do not work correctly. This is because the **getline** function and the stream-input operator make different assumptions about when a newline character is "consumed." It's a good idea to stick to one approach or the other.

Example 8.2. *Get a Number*

The following program gets numbers and prints their square roots, until the user either presses 0 or presses ENTER directly after the prompt:

get_num.cpp

```
#include <iostream>
#include <cstring>
#include <cmath>
#include <cstdlib>
using namespace std;

double get_number();
```

get_num.cpp, cont.

```
int main()
{
    double x = 0.0;

    while(true) {
        cout << "Enter a num (press ENTER to exit): ";
        x = get_number();
        if (x == 0.0) {
            break;
        }
        cout << "Square root of x is: " << sqrt(x);
        cout << endl;
    }
    return 0;
}

// Get-number function.
// Get number input by the user, taking only the first
//  numeric input entered. If user presses ENTER with
//  no input, then return a default value of 0.0.
//
double get_number() {
    char s[100];

    cin.getline(s, 100);
    if (strlen(s) == 0) {
        return 0.0;
    }
    return atof(s);
}
```

You can use this same function (get_number) in all your programs as a better way of getting numeric input.

How It Works

The program begins by including <cstring> and <cmath>, which bring in type information for string functions and math functions; also, <cstdlib> brings in the declaration of the **atof** function used in this example. In addition, the program declares the get_number function up front.

```
#include <iostream>
#include <cstring>
#include <cmath>
#include <cstdlib>

using namespace std;

double get_number();
```

What the main function does should be familiar by now. It performs an infinite loop, which is terminated when 0 is returned by the get_number function. When any value other than zero is entered, the program calculates a square root and prints the results.

```
    while (true) {
        cout << "Enter a num (press ENTER to exit): ";
        x = get_number();
        if (x == 0.0) {
            break;
        }
        cout << "Square root of x is: " << sqrt(x);
        cout << endl;
    }
```

What's new here is the get_number function itself. When this function calls **getline**, it returns an entire line of input up to n − 1 characters. Since the n argument in this case is 100, it will read, at most, 99 characters, leaving one byte for the terminating null. If the user presses ENTER directly after the prompt, **getline** returns an empty string.

```
double get_number() {
    char s[100];

    cin.getline(s, 100);
    if (strlen(s) == 0) {
        return 0.0;
    }
    return atof(s);
}
```

Once the input line is stored in the local string, s, it's a trivial matter to return 0 if the string is empty.

```
if (strlen(s) == 0) {
    return 0.0;
}
```

The literal 0.0 is equal to 0 but is stored in **double** format. Remember that every literal containing a decimal point is considered a floating-point number by C++.

If the length of string s is not 0, data in the string must be converted. Because we're not relying on the stream operator (>>), the get_number function must take responsibility for interpreting data itself. Therefore, it needs to examine the characters read—the ASCII codes sent from the keyboard—and produce a **double** value.

Fortunately, the C++ standard library supplies a handy function—**atof**—for doing just that and we can make use of it here. The **atof** function takes string input and produces a floating-point (**double**) value, just as its cousin **atoi** produces an **int** value.

```
return atof(s);
```

This function has a sibling—**atoi**—that does the same thing for integers:

```
return atoi(s);       // Return an int value.
```

EXERCISE

Exercise 8.2.1. Rewrite Example 8.2 so that it accepts only integer input. (Hint: You'll want to change all types directly affected, from **double** to **int** format, including constants.)

Example 8.3. *Convert to Uppercase*

In this example, I'll show a simple program that accesses individual characters. Remember that although you can think of a string as a single entity, it's actually made up of a series of characters, which are typically (but not always) uppercase and lowercase letters.

upper.cpp

```
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
```

▼ *continued on next page*

upper.cpp, cont.

```
void convert_to_upper(char *s);

int main()
{
    char s[100];

    cout << "Enter string to convert & press ENTER: ";
    cin.getline(s, 100);

    convert_to_upper(s);
    cout << "The converted string is:" << endl;
    cout << s << endl;
    return 0;
}

void convert_to_upper(char *s) {
    int length = strlen(s);

    for (int i = 0; i < length; i++) {
        s[i] = toupper(s[i]);
    }
}
```

How It Works

The main purpose of this example is to show that you can manipulate individual characters of a string. To pass a string to a function, pass its address. To do that, of course, you just give the name of the string. (This is the standard procedure to pass any kind of array.)

```
        convert_to_upper(s);
```

The function uses the argument passed—which, after all, is an address—to index into the string data.

```
    void convert_to_upper(char *s) {
        int length = strlen(s);

        for (int i = 0; i < length; i++) {
            s[i] = toupper(s[i]);
        }
    }
```

This example introduces a new function, **toupper**. The two functions, **toupper** and **tolower**, operate on individual characters:

FUNCTION	DESCRIPTION
toupper(*c*)	If c is a lowercase letter, return the uppercase equivalent; otherwise, return c as is.
tolower(*c*)	If c is an uppercase letter, return the lowercase equivalent; otherwise, return c as is.

The following statement therefore converts a character to uppercase (if it is a lowercase letter) and replaces the original character with the result:

```
s[i] = toupper(s[i]);
```

Again, note that to use these functions, you must include <cctype>:

```
#include <cctype>
```

EXERCISES

Exercise 8.3.1. Write a program that is similar to Example 8.3 but converts the string input to all lowercase. (Hint: Use the **tolower** function from the C++ library.)

Exercise 8.3.2. Rewrite Example 8.3 so that it uses direct pointer reference, described at the end of Chapter 6, rather than array indexing. If you have reached the end of the string, the value of the current character is a null-terminator, so you can test for the end-of-string condition by using *p == '\0'. You can also use *p itself as the condition, because it is nonzero if it's not pointing to a zero (or null) value.

```
while (*p++) {
    // Do some stuff...
}
```

Individual Characters versus Strings

C++ makes a distinction between individual characters and strings. A lot depends on whether you use single or double quotation marks.

The expression 'A' represents a single character. During compilation, C++ replaces this expression with the ASCII value for a letter 'A', which happens to be 65 decimal.

On the other hand, expression "A" represents a string of length 1. When C++ sees this expression, it places two bytes in the data area:

- The ASCII code for the letter 'A', as shown earlier.
- A null-terminating byte.

The C++ compiler then replaces the expression "A" *with the address* of this two-byte array. 'A' and "A" are different because one is converted to an integer value, whereas the other represents a string and so is converted to an address.

This may seem like a lot to digest, but just remember to pay close attention to the quotation marks. The following code provides an example of how they can be intermixed correctly:

```
char s[] = "A";
if (s[0] == 'A') {
    cout << "The first letter of the string is 'A'. ";
}
```

This produces a correct result. But this next comparison is an error, because it tries to compare a character to an address:

```
if (s[0] == "A") {             // WRONG!
    //...
```

This fragment attempts to compare an element of the string array s with an *address expression*, "A". The cardinal rules are as follows:

* **Expressions in single quotation marks (such as 'A') are treated as numeric values after translation into ASCII codes. They are not arrays.**

* **Expressions in double quotation marks (such as "A") are arrays of char and, as such, are translated into addresses.**

Example 8.4. *Breaking Up Input with strtok*

When you read in a line of text (for example, with the **getline** function), you'll often find you need to break it into smaller strings. For example, consider this text input:

```
Me, myself, and I.
```

Suppose you want to break this into the individual substrings separated by commas and spaces (*delimiters*). As a test, you might print each of the substrings on its own line.

```
Me
Myself
and
I
```

You can do this yourself through a combination of searching for delimiter characters and then indexing the string to select the substrings you find. But it's easier to use the **strtok** function (string token) from the C++ standard library.

In this context, the word *token* means a substring containing a single word. There are two ways to use this function.

FUNCTION USAGE	DESCRIPTION
strtok(*source_string*, *delims*)	Return the first token from source string, using the delimiters found in *delims*.
strtok(**nullptr**, *delims*)	Using the source string already specified (during an earlier call to **strtok**), get the next token. Use the delimiters found in *delims*.

The first time you use **strtok**, specify both the source string and the delimiter characters; **strtok** returns a pointer to the first substring (that is, token) it finds. For example:

```
p = strtok(the_string, ", ");
```

Thereafter, call **strtok**, specifying a null value for the first argument; **strtok** returns the next token from this same source string. The function remembers what source string it was working on and where it was in that string.

```
p = strtok(nullptr, ", ");
```

If, instead, you specify *source_string* again, **strtok** starts over and returns the first token.

The return value from the function is usually a pointer to a token; but if there are no further tokens (substrings) left to read, **strtok** returns a null value, which can be tested for equality to zero or to **false**.

Note If you have a compiler more than a few years old, you may need to use NULL in place of **nullptr**. This keyword has been a part of the specification since C++11.

Here is a simple program that interprets spaces and commas as delimiters (separator characters) and prints each substring (each token) on its own line:

tokenize.cpp

```
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    char the_string[81], *p;

    cout << "Input a string to parse: ";
    cin.getline(the_string, 81);
    p = strtok(the_string, ", ");
    while (p != nullptr) {
          cout << p << endl;
          p = strtok(nullptr, ", ");
    }
    return 0;
}
```

How It Works

This program is a simple demonstration of **strtok**. It begins with **#include** directives.

```
#include <iostream>
#include <cstring>
```

Before going into the **while** loop, the program calls **strtok** and specifies the input string. It finds the first token (substring), if any, and returns a pointer to it.

```
p = strtok(the_string, ", ");
```

This is the only place that the_string is specified. After that, the loop calls **strtok** with a nullptr first argument, which means “keep working on the same input string, and return the next token (substring) within it.”

```
while (p != nullptr) {
      cout << p << endl;
```

```
            p = strtok(nullptr, ", ");
    }
```

A nullptr return value means "there are no tokens left to read."

EXERCISES

Exercise 8.4.1. Revise the example so that in addition to printing out tokens (substrings) one to a line, the program also prints a statement telling how many tokens it found.

Exercise 8.4.2. Put all the tokens back together but separated by ampersands (&). Print the result.

Exercise 8.4.3. Use an ampersand (&) as a delimiter.

The C++ String Class

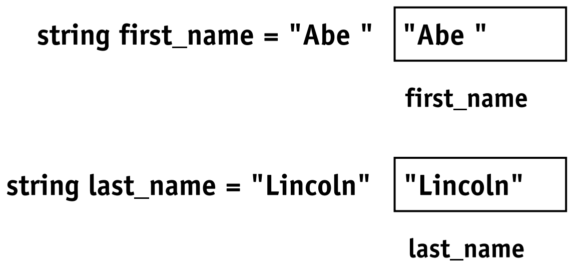
Null-terminated strings used in C and C++ are referred to as *C-strings*. They're not as convenient as the built-in strings that Visual Basic provides, which hide nearly all the details from you.

For a number of years now, almost all C++ compilers have provided a similar type, called by the highly original name, **string**. (Technically, the name of this C++ type is **std::string**, but if you include the **using namespace** statement, you don't have to use the **std::** prefix.)

The **string** type is an example of a class, and individual strings are objects, just as **cin** and **cout** are. For example, suppose you have two strings labeled first_name and last_name:

```
#include <string>
using namespace std;
...
string first_name("Abe ");
string last_name("Lincoln");
```

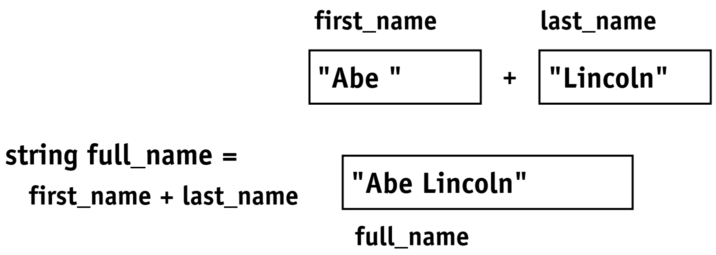
Instead of worrying about arrays or about indexing characters, there are many operations you can perform almost as if these objects were primitive pieces of data.



For example, you can use the addition sign (+) to concatenate these strings without worrying about issues such as length or capacity.

```
string full_name = first_name + last_name;
```

This statement joins the two strings and forms a new string named full_name, which is automatically the right length. You don't need to worry about whether the new string has enough space to store the combined names because the **string** class itself manages all storage issues for you.



And, if you need to index individual characters, you still can, indexing them just as you would a C-string. Similar considerations apply: an individual character has type **char**, just as it would if you were indexing a C-string.

```
string s = "I am what I am.";
cout << s[3];  // Print fourth char (m).
```

The **string** class has nearly all the advantages of the C-string type but is often easier and more convenient to use. About the only disadvantage of the **string** class is that it is not compatible with the **strtok** function, which requires C-strings to work with.

Include String-Class Support

To use the new **string** type, the first thing you need to do is turn on support for it by using an **#include** <**string**> directive. This is not the same directive that enables C-string support in the function library. These directives look similar.

```
#include <string>     // Support new string class
```

But remember, C-string support uses “cstring” rather than “string”:

```
#include <cstring>    // Support old-style string
                      //   functions
```

What a difference the “c” makes! By the way, you can turn on support for both, but including “cstring” is only necessary if you are going to be calling old-style functions such as **strcpy**.

As with **cin** and **cout**, the name **string** must be qualified with the **std::** prefix unless you include the using statement at the beginning of your programs:

```
using namespace std;
```

If you don’t include this statement, you can always refer to the **string** class with its **std** prefix, as **std::string**. However, the “using namespace” statement makes the **std::** prefix unnecessary for anything from the C++ library.

Declare and Initialize Variables of Class string

Once you have turned on support for the **string** type, it’s easy to use it to declare variables. (Again, if you omit the namespace statement, you’d refer to the class as **std::string** instead of **string**.)

```
string a, b, c;
```

This creates three variables having C++ standard-library class **string**. Notice how easy this is: You don’t have to worry about how much space might be needed for each string. You can initialize the strings in a number of ways. For example:

```
string a("Here is a string."), b("Here's another.");
```

You can also use the assignment operator (=) for this purpose.

```
string a, b;
a = "Here is a string.";
b = "Here's another.";
```

You can also combine declaration and initialization by using the equal sign (=):

```
string a = "Here is a string.";
```

Working with Variables of Class string

The standard-library **string** class works as you’d probably expect. Unlike C-strings, **string** objects can be copied and compared without calling library functions.

For example, suppose you have the following strings:

```
string cat = "Persian";
string dog = "Dane";
```

You can assign new data without worrying about capacity. In this case, dog, which had four characters, "automagically" grows to store seven characters:

```
dog = "Persian";
```

You can compare these strings by using the test-for-equality operator (==). This does what you'd expect: it returns **true** if the contents are identical. (To perform this comparison with C-strings, you'd need to call **strcmp**.)

```
if (cat == dog) {
     cout << "cat and dog have the same name";
}
```

To copy from one string variable to another, just use the assignment operator (=). Again, this does what you'd expect: it copies string contents, not a pointer value.

```
string country = dog;
```

Remember, you can concatenate (join) strings by using a plus sign (+).

```
string new_str = a + b;
```

You can even embed string literals in this operation:

```
string str = a + " " + b;
```

However, the following statement does not compile:

```
string str = "The dog" + " is my friend"; // ERROR!
```

The problem is that although the plus sign (+) is supported as a concatenation operator between two **string** variables or between a **string** variable and a C-string, it is not supported between two C-strings and string literals are still C-strings.

Note ▶ You can solve this problem—the inability to concatenate two string literals—by using an "s" suffix to make them true instances of the C++ **string** class. Chapter 17, "New Features of C++14," describes this feature.

Another way to solve the problem is to place two C-string literals next to each other, separated only by a space or a newline. The compiler will automatically concatenate C-string literals placed next to each other.

Input and Output

Variables of type **string** work with **cin** and **cout** just as you'd expect.

```
string prompt = "Enter your name: ";
string name;
cout << prompt;
cin >> name;
```

Use of the stream-input operator (>>) has the same drawback that it does with C-strings: characters are returned from the keyboard up until the first white-space character.

But you can use the **getline** function to put an entire line of input into a string variable. This version doesn't require you to enter a maximum number of characters to read because the string variable will store data of any size.

```
getline(cin, name);
```

Example 8.5. Building Strings with the string Class

This example performs the same action as Example 7.1, except that it uses string variables:

buildstr2.cpp

```
#include <iostream>
#include <string>   // Include support for string class.
using namespace std;

int main()
{
    string str, name, addr, work;

    // Get three strings from the user.

    cout << "Enter name and press ENTER: ";
    getline(cin, name);
    cout << "Enter address and press ENTER: ";
    getline(cin, addr);
    cout << "Enter workplace and press ENTER: ";
    getline(cin, work);
```

▼ continued on next page

buildstr2.cpp, cont.

```
    // Build the output string, and then print it.

    str = "\nMy name is " + name + ", " +
          "I live at " + addr +
          ",\nand I work at " + work + ".\n";

    cout << str << endl;
    return 0;
}
```

How It Works

If anything, this version of the program is easier to write than the version in Exercise 8.1. The first difference is the **include** directive, which must refer to <string>, not <cstring>.

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

The using namespace statement, as usual, enables you to refer to **std** symbols (such as **cin**, **cout**, and also **string**) without a **std** prefix.

Then, things get easier. This version of the program declares four string variables without worrying about how much space to reserve for each.

```
    string str, name, addr, work;
```

The program then calls the **getline** function without needing to specify the maximum number of characters to read.

```
    cout << "Enter name and press ENTER: ";
    getline(cin, name);
    cout << "Enter address and press ENTER: ";
    getline(cin, addr);
    cout << "Enter workplace and press ENTER: ";
    getline(cin, work);
```

Finally, the program builds the string. The addition operator (+) provides a concise way to represent string concatenation.

```
    str = "\nMy name is " + name + ", " +
          "I live at " + addr +
          ",\nand I work at " + work + ".\n";
```

Then, it's an easy matter to print the resulting string.

```
cout << str;
```

EXERCISES

Exercise 8.5.1. Get three pieces of information from the user: a dog's name, its breed, and its age. Then print a sentence combining all this information.

Exercise 8.5.2. Print a complex paragraph that uses all this data multiple times across several sentences.

Example 8.6. *Adding Machine #2*

Working with strings—whether C-strings or the **string** class—provides a way of getting one line of input at a time and making intelligent decisions about what to do with it. Using a combination of the **getline** function and pointer usage, you can build a much better version of the Adding Machine program in Chapter 2, "Decisions, Decisions."

That program had to use an arbitrary code (such as 0) to terminate the series of numbers to add, which creates obvious problems. What this improved version does is to continue to accept numbers until the end user presses ENTER after entering no input.

You could write this program with either kind of string: a traditional C-string (a null-terminated array of **char**) or an instance of the STL **string** class. After you've used both, however, you'll probably agree that working with the latter is easier.

adding2.cpp

```
#include <iostream>
#include <string>   // Include support for string class.
using namespace std;

bool get_next_num(int *p);

int main()
{
     int sum = 0;
     int n = 0;
```

▼ continued on next page

adding2.cpp, cont.

```
    while (get_next_num(&n)) {
        sum += n;
    }
    cout << "The total is: " << sum << endl;
    return 0;
}

bool get_next_num(int *p) {
    string input_line;
    cout << "Enter num (press ENTER to quit): ";
    getline(cin, input_line);
    if (input_line.size() == 0) {
        return false;
    }
    *p = stoi(input_line);
    return true;
}
```

How It Works

This is a simple program. All it does is prompt for another number until the user presses ENTER after entering a zero-length string.

One subtlety of this program is that it uses a different version of the **getline** function than that described earlier in this chapter. Remember: for C-strings, use the **getline** method. For objects of string type, use the **getline** function. This is admittedly a little counterintuitive.

```
char my_cstr[10];   // C-string decl.
string my_str;      // string object.

cin.getline(my_cstr, 10); // Use this on C-strings
getline(cin, my_str);     // Use this on string objects.
```

The return value in this example is used to signal the "terminate now" condition. Therefore, the numeric value entered has to be returned some other way. This is done by using a pointer argument to simulate pass by reference. The numeric value is "returned," in effect, through this pointer:

```
*p = stoi(input_line);
```

The **stoi** function has been provided since C++11 to convert from the **string** type to an integer. For converting to floating-point numbers, C++11 also provides the **stof** function. If your compiler is too old to support **stoi** and **stof**, you can still use the old standbys, **atoi** and **atof**. The only problem is that if you use one of these functions, you need to use the **c_str** function to convert to C-style format first.

```
*p = atoi(input_line.c_str());
```

This looks ugly, but it works with earlier versions of C++.

EXERCISES

Exercise 8.6.1. Revise the get_next_num function so that a default value is specified as one of the arguments. If the end user presses ENTER without entering any text, the function returns this default value.

Exercise 8.6.2. Rewrite the example so that it accepts floating-point numbers and prints a floating-point result. Remember that C++ supports the **stof** and **atof** functions just described.

Other Operations on the string Type

Again, you can access individual characters in a **string** object with the same syntax used to access characters inside C-strings.

```
string[index]
```

For example, the following code prints out a string's individual characters, one to a line:

```
#include <string>
using namespace std;
//...
string dog = "Mac";
for (int i = 0; i < dog.size(); ++i) {
    cout << dog[i] << endl;
}
```

When run, this code prints the following:

```
M
a
c
```

As with C-strings, or any array in C, **string** variables use zero-based indexing. This is why the initial setting for i is 0.

The loop condition depends on the length of the string. To find this length with a C-string, you'd use the **strlen** function. With a string object, use the **size** member function.

```
int length = dog.size();
```

Chapter 8 *Summary*

Here are the main points of Chapter 8:

- Text characters are stored in the computer according to their ASCII codes. For example, the string "Hello!" is represented by the byte values 72, 101, 108, 108, 111, 33, and 0 (for the terminating null).
- The traditional "C-string" type uses a terminating null—a 0 byte value. This enables string-handling functions to determine where the string ends. When you declare a string literal such as "Hello!", C++ automatically allocates space for this terminating null along with the other characters.
- The current length of a string (determined by searching for the terminating null) is not the same as the total amount of storage reserved for the string. The following declaration reserves 10 bytes of storage for str but initializes it so that its current length is only 6. The string will have 3 unused bytes as a result, enabling it to grow later if needed.

  ```
  char str[10] = "Hello!";
  ```
- Library functions such as **strcpy** (string copy) and **strcat** (string concatenation) can alter the length of an existing string. When you perform these operations, it's important that the string have enough space reserved to accommodate the new string length.
- The **strlen** function gets the current length of the string.
- Include the string.h file to provide type information for string-handling functions.

  ```
  #include <cstring>
  ```
- If you try to increase the length of a string without having the necessary space reserved, you'll overwrite another variable's data area, creating hard-to-find bugs.

  ```
  char str[] = "Hello!";
  strcat(str, " So happy to see you.");   // ERROR!!!!
  ```

- To ensure that you don't copy too many characters to a string, you can use the **strncat** and **strncpy** functions.

  ```
  char str[100];
  strncpy(str, s2, 100);
  strncat(str, s2, 100 - strlen(str));
  ```

- The stream operator (>>), used with the **cin** object, provides only limited control over input. When you use it to send data to a string address, it only gets the characters up to the first white space (blank, tab, or newline).
- To get a full line of input, use the **cin.getline** member function (method). The second argument specifies the maximum number of characters to copy to the string (not counting the terminating null).

  ```
  cin.getline(input_string, max);
  ```

- An expression such as 'A' represents a single integer value (after translation into ASCII code); an expression such as "A" represents an array of **char** and is therefore translated into an address.
- The STL **string** class lets you create, copy contents (=), test for equality of contents (==), and concatenate (+) strings without having to worry about size issues.
- To use the string class, include <string>. Also, remember that the full name is std::string, although the std prefix is unnecessary if you use a "using namespace std" statement.

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

- You can index a **string** object to get an individual character (a **char** value), just as you can with C-strings.

  ```
  char c = str_obj[2];    // c = third character.
  ```

- To get a line of input with a **string** object (which is a flexible operation because you don't have to specify a maximum number of characters), use the **getline** function, which is a global function, not a member.

  ```
  getline(cin, str_obj);  // str_obj gets input line.
  ```

- The C++ library provides the **stoi** and **stof** functions for converting a string object to a numeric value. The library also provides **atoi** and **atof** functions for conversion of a **char*** (C-string) to a numeric value, integer and floating-point (**double**) respectively.
- To use a **string** object as a C-string, convert from one to the other by calling the **string** object's **c_str** method.

This page intentionally left blank

9 Files: Electronic Storage

At some point in your programming career, you're going to have to deal with disk files. Most real-world applications store and retrieve persistent information, for example, applications such as payroll programs, spreadsheets, and text editors, to name just a few. Even simpler applications often need long-term data storage.

When the program ends, you don't want this information to vanish into the ether. For example, what if this data was your payroll information or the Kentucky Colonel's secret fried-chicken recipe? In some cases, you may want it to stick around—for years and years.

Disk files, unlike main memory (or RAM), continue to maintain their state even when the computer is turned off. So, when you need a place to put data for later use, put it in a disk file.

Introducing File—Stream Objects

In using **cin** and **cout** (console input and output), you've already made use of "objects"—self-contained entities that know how to respond to requests. Now it's time to introduce some new objects. C++ provides file streams that support the same set of function calls and operators that **cin** and **cout** do.

C++ programmers often talk about *streams*. A stream is something to which you can read or write data. It's that simple. The term evokes the image of data flowing like water in a river, that is, flowing from some source (for example, the console) or toward some destination (for example, a file). Although data streams are not always as endless as a river, it's a useful image.

Writing to a text file involves a few simple steps. The first step is to enable support for file-stream operations by using an #include <fstream> directive. This implicitly brings in declarations for file-stream operations.

```
#include <fstream>
```

The second step is to create a file-stream object and associate it with a disk file. (I've chosen the name fout, but you could choose any name you want: MyGoofyFile, RoundFile, Trash, or whatever.) Also, I've specified a file named output.txt.

```
ofstream fout("output.txt");  // Open file output.txt
```

Now the object fout is associated with the file output.txt and has type **ofstream**. When you open a file stream, you can use any of the following types:

- **ofstream**, for file-output streams
- **ifstream**, for file-input streams
- **fstream**, a generic file stream (to which you have to specify input, output, or both when you open it; more on that later)

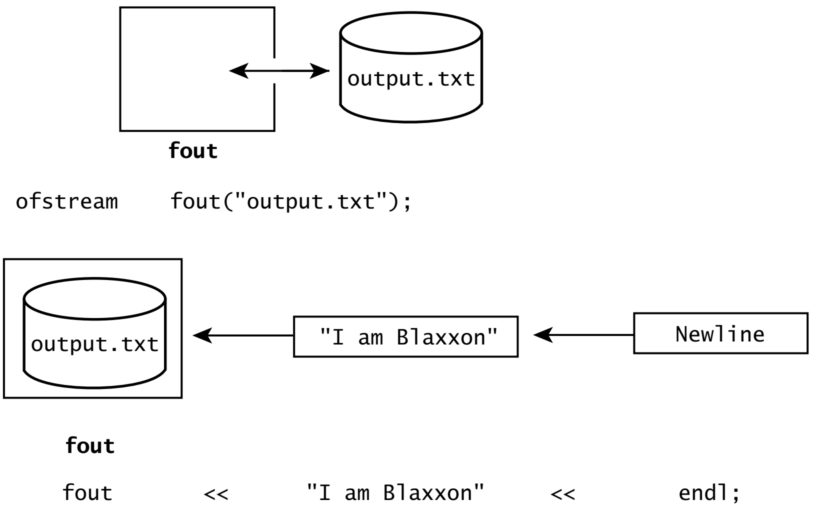
The third step is as follows: After you successfully create the object, you can write to it just as you would write to **cout**. This sends data to the associated file, in this case, output.txt.

```
fout << "This is a line of text.";
```

As a variation on an example from Chapter 1, "Start Using C++," you could use the following code to write to the disk file output.txt:

```
#include <fstream>
// ...
ofstream fout("output.txt");  // Open file output.txt

fout << "I am Blaxxon," << endl;
fout << "the cosmic computer.";
```

The object fout provides access to the disk file. In object-oriented terms, we can say that fout *encapsulates* the file, in terms of its ability to receive output. First, fout is declared, associating it with a particular disk file (output .txt in this example). Afterward, writing to fout results in sending data to this same file.



You can have multiple file-stream objects open at the same time—one for each file you want to interact with.

```
ofstream  out_file_1("memo.txt");
ofstream  out_file_2("messages.txt");
```

When you're done reading or writing to a file, you should call the **close** function. This causes the program to give up ownership of the file so that some other process can access it. C++ closes the file for you when the program exits successfully, but it's a good idea to close files as soon as you no longer need them.

```
out_file_1.close();
out_file_2.close();
```

How to Refer to Disk Files

In the previous section, I showed how you can create a file object by specifying the file's name. If successful, this declaration opens the file for output, giving you exclusive access.

```
ofstream fout("output.txt");
```

By default, the file referred to is in the current directory—the directory from which you run the program. (Or, to use Windows or Macintosh lingo, this is the current *folder.*) But you can, if you want, specify a complete path name, optionally including a drive letter. This is all part of the complete filename or, more precisely, the file specification.

For example, you can open a file in a particular directory on the C: drive.

```
ofstream fout("c:\\Users\\Briano\\output.txt");
```

This works on my computer, by the way, because I have a directory with this name. But you probably don't (unless you're a *really* big fan of mine!).

The string literal here uses the C++ backslash notation. The backslash character has a special meaning in C++ programs: for example, \n represents a newline, and \t represents a tab. To represent the backslash itself, use two in a row. So, "c:\\Users\\Briano\\output.txt" in a C++ program code names this file:

```
c:\Users\Briano\output.txt
```

Yet there is a better, more portable approach. Although Windows (and other systems) use the backslash to navigate the file system, C++ accepts a forward slash (/) as a file-path divider and then translates it as appropriate for the local platform. To put it simply, you can use a string like this instead:

```
string path_name = "c:/Users/Briano/output.txt"
```

So, for example, you might create a file object this way:

```
ofstream fout("c:/Users/Briano/output.txt");
```

This statement, executed on a Windows or DOS system, would open the file named earlier for writing, assuming that the path is valid.

Example 9.1. *Write Text to a File*

The example in this section does about the simplest thing you can do with a text file: open it, write a couple of lines of text, close it, and exit.

The program prompts the user for the name of a file to which you want to write. As a user, you enter the exact filename, including drive letter and complete path if desired. Do *not* use two backslashes to represent one: That is a notational convention within C++ program code only and has no effect on the user or on how strings are stored generally.

For example, as the user you might enter this

```
c:\documents\output.txt
```

and if the file is successfully opened, text will be written to this file.

Note ▶ This program will replace whatever file you specify, destroying its old contents. Therefore, when you run it, be careful not to enter the name of an existing file unless you don't mind losing that file's contents.

writetxt.cpp

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    char filename[MAX_PATH + 1];

    cout << "Enter a file name and press ENTER: ";
    cin.getline(filename, MAX_PATH);
    ofstream file_out(filename);
    if (! file_out) {
        cout << filename << " could not be opened.";
        cout << endl;
        return -1;
    }
    cout << filename << " was opened." << endl;
    file_out << "I read the" << endl;
    file_out << "news today," << endl;
    file_out << "ooh boy.";
    file_out.close();
    return 0;
}
```

After running this program, you'll probably want to view its contents to verify that the program wrote the text successfully. You can use any text editor or word processor to do that. (Or, if you are inside an MS-DOS command shell, you can use the TYPE command.)

How It Works

The program starts by enabling support for the iostream and fstream portions of the C++ library.

```
#include <iostream>
#include <fstream>

using namespace std;
```

There's only one function, **main**. The first thing it does is to prompt for a filename:

```
        char filename[FILENAME_MAX];

        cout << "Enter a file name and press ENTER: ";
        cin.getline(filename, FILENAME_MAX);
```

This last line refers to FILENAME_MAX, a predefined constant that specifies the maximum length for filenames (including the path name) supported on the system. (This is portable; alternatively, you can use MAX_PATH, but that is supported by Windows systems only.) Allocating FILENAME_MAX characters guarantees the string named "filename" will be big enough to hold any valid filename.

The next thing that the main function does is create a file object, file_out.

```
        ofstream file_out(filename);
```

This statement attempts to open the named file. If the attempt to open the file is unsuccessful, a null value is placed in file_out. This value can then be tested in an **if** statement: A null value equates to false in this context.

If the file was not successfully opened, the program prints an error message and exits. The logical negation operator (!) reverses the true/false value, so in effect this is a test to see whether file_out is null, indicating failure to open the file.

```
    if (! file_out) {
        cout << filename << " could not be opened.";
        cout << endl;
        return -1;
    }
```

There are a couple of reasons the file open could fail. The user may have entered an invalid file specification. Or, the user attempted to open a file that has been given read-only privileges by the operating system and cannot be overwritten.

If the file was successfully opened, the program writes confirmation on the console, writes text to the file, and then closes the file stream.

```
        cout << filename << " was opened." << endl;
        file_out << "I read the" << endl;
        file_out << "news today," << endl;
        file_out << "ooh boy.";
        file_out.close();
        return 0;
```

EXERCISES

Exercise 9.1.1. Rewrite Example 9.1 so it prompts for directory location and filename separately. (Hint: Use two strings and use the **strcat** function to join them.)

Exercise 9.1.2. Write a program that lets the user enter any number of lines of text, one at a time. In effect, this creates a primitive editor that permits text entry but no editing of a line of text after it's been entered. Set up a loop that doesn't terminate until the user presses ENTER without typing any text (a zero-length string).

Alternatively, you can recognize a special code (for example, @@@) to terminate the session. You can then use the **strcmp** ("string compare") function to detect this string. You may recall that what this function does is to compare two C-strings and return 0 if they have the same contents.

```
if (strcmp(input_line, "@@@") == 0) {
    break;
}
```

Remember to print a short prompt before each line of text, such as the following:

```
Enter (@@@ to exit) >>
```

Example 9.2. *Display a Text File*

After you write to a file, you'll want to view it. Writing a complete text editor is beyond the scope of this book, but the examples in this chapter cover some of the basic elements. The main thing a word processor or text editor does is open a file, read lines of text, let the user manipulate those lines of text, and then write out the changes.

This example displays 24 lines of text at a time, asking the user whether to continue. The user can print another 24 lines or quit. I picked this number as typical of the number of vertical lines in a window less one to interact with the user.

This example opens a stream as an **ifstream**, which assumes text and input modes: The file will not open successfully unless you have named a file that already exists.

readtxt.cpp

```
#include <iostream>
#include <fstream>
using namespace std;
#define COL_WIDTH  80

int main() {
    int c;   // input character
    char filename[FILENAME_MAX];
    char input_line[COL_WIDTH + 1];

    cout << "Enter a file name and press ENTER: ";
    cin.getline(filename, FILENAME_MAX);

    ifstream file_in(filename);

    if (! file_in) {
        cout << filename << " could not be opened.";
        cout << endl;
        return -1;
    }

    while (true) {
        for(int i = 1; i <= 24 && !file_in.eof(); ++i) {
            file_in.getline(input_line, COL_WIDTH);
            cout << input_line << endl;
        }
        if (file_in.eof()) {
            break;
        }
        cout << "More? (Press 'Q' and ENTER to quit)";
        cin.getline(input_line, COL_WIDTH);
        c = input_line[0];
        if (c == 'Q' || c == 'q') {
            break;
        }
    }
    return 0;
}
```

How It Works

This example is similar to Example 8.1 but it checks a couple of different conditions to determine whether it should keep reading more lines.

After determining whether the file stream was successfully opened, the program sets up an infinite loop that exits when either of the following conditions is true:

- The end of the file is reached.
- The user indicates that he or she does not want to continue.

Here's the main loop in skeletal form:

```
while (true) {
    // ...
}
```

Within the loop, the program reads up to 24 lines—less, if the end of file is reached first. The easy way to implement this is to use a **for** loop with a complex condition:

```
for(int i = 1; i <= 24 && ! file_in.eof(); ++i) {
    file_in.getline(input_line, FILENAME_MAX);
    cout << input_line << endl;
}
```

The loop continues only as long as i is less than or equal to 24 and the end-of-file condition is not detected. The expression

```
file_in.eof()
```

returns true if the end of the file has been reached. Logical "not" (!) reverses this condition, so that "! file_in.eof()" returns true only as long as there is more data to read.

The rest of the main loop checks to see whether it should continue; if not, it breaks out of the loop and the program ends.

```
if (file_in.eof()) {
    break;
}
cout << "More? (Press 'Q' and ENTER to quit)";
cin.getline(input_line, 1);
c = input_line[0];
if (c == 'Q' || c == 'q') {
    break;
}
```

EXERCISES

Exercise 9.2.1. Alter Example 9.2 so the user can optionally enter a number in response to the "More?" prompt. The number determines how many lines to print at a time instead of 24. (Hint: Use the **atoi** library function to convert string input to integer; if the value entered is greater than 0, modify the numbers of lines to read.)

Exercise 9.2.2. Alter the example again so that it prints the contents of the file in all-uppercase letters. You may find it helpful to copy some of the code from Exercise 8.3 on page 195.

Text Files versus "Binary" Files

So far, we've used text files; a text file can be read or written to just like the console. As with the console, text files contain data in character form.

If you view the file with a text editor, or print the file on the console, you'll see the contents in human-readable form. For example, when you write the number 255 to a text file, the program writes the ASCII character codes for 2, 5, and 5.

```
file_out << 255;
```

But there's another way to store data. Instead of writing the ASCII character codes for 255, you could write the value 255 directly. If you then tried to view the file with a text editor, you wouldn't see the numerals 255. Instead, the text editor would try to show you ASCII code 255, which is not a regular printable character.

Programming manuals talk about two kinds of files:

- Text files, which you read and write to as you would the console. Usually, every byte written to a text file is the ASCII code for a printable character.
- So-called binary files, which you read and write to, use the actual numeric values of the data. With this approach, ASCII translation is not involved.

This second technique may sound simpler, but it's not. To view such a file in a meaningful way, you need an application that understands what the fields of the file are supposed to be and how to interpret them. Are a group of bytes to be interpreted as integer, floating-point, or string data? And where does one group of bytes start and another begin?

When you create a file-stream object, you can specify text mode (the default) or binary mode. The mode setting itself changes one important detail:

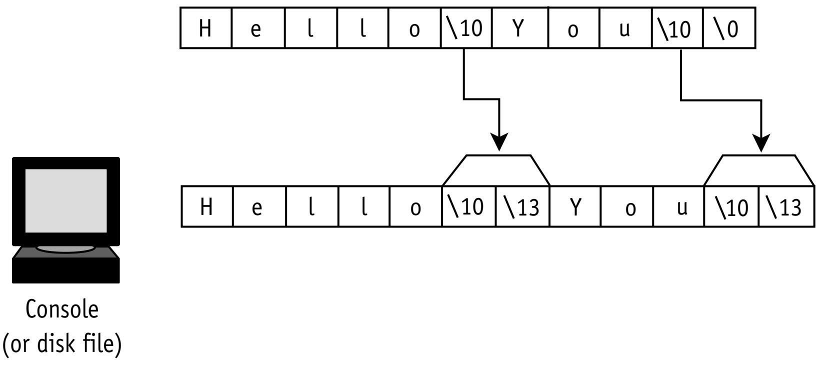
In text mode, each newline character (ASCII 10) is translated into a carriage return–linefeed pair during a write operation; during a read operation, a carriage return–linefeed pair is translated back into a newline.

Let’s consider why the translation is necessary for text mode. Early in the book, the examples used newline characters. These can be printed separately or embedded in the strings themselves.

```
char *msg_string = "Hello\nYou\n";
```

Strings embed a single byte (ASCII code 10) to indicate a newline. But printing to the console requires two actions: printing a carriage return (ASCII code 13), which moves the cursor to the beginning of the line, and printing a linefeed (ASCII code 10).

When a string is written to the console, each newline in memory is translated into a carriage return–linefeed pair. For example, here’s what the string “Hello\nYou\n” looks like when stored in main memory, and what it looks like when written to the console:



OK, you say. So, this translation must be done when printing strings on the console. But is it necessary for text files as well?

Yes, it is. Data sent to a text file must have the same format as data sent to the console. This allows C++ to treat all streams of text (whether console or on disk) the same way.

But with a binary file, no such translation should ever be performed. The value 10 may occur in the middle of a numeric field and it must not be interpreted as a newline. If you translated this value, you’d likely create a great many errors.

There's another difference—probably the most important—between text-mode and binary-mode operations. It concerns the choices you make as a programmer.

- If you open a file in text mode, you should use the same operations you use for communicating with the console; these involve the stream operators (<<, >>) and the **getline** function.
- If you open a file in binary mode, you should transfer data only by using the **read** and **write** member functions. These are direct read/write operations.

In the next section, I discuss these two functions.

Interlude

Are "Binary Files" Really More Binary?

The reason people use the term *binary file* is that with such a file, if you write the byte value 255, you're actually writing the binary value of 255 directly.

```
11111111
```

The use of the term *binary* is in some ways misleading. If you write 255 as text, you're *still* writing binary data, except that now each of these binary values is an ASCII character code. Conceptually, programmers tend to think of this as "text" format as opposed to "binary," because a text editor displays a file as text.

Incidentally, here's how 255 is actually written in text mode:

```
00110010  00110101  00110101
```

This binary sequence represents the numbers 50, 53, and 53, which in turn are the ASCII codes for the numerals 2, 5, and 5. When this data is sent to the console, you see the string of digits 255.

But the important point here is this: This is text mode, as opposed to binary mode, because while working in text mode, you don't care about the underlying binary representation. All you care about is that the file is seen as a stream of text characters. So, even though ultimately *everything* is binary, you should just think of this mode as "text mode."

Throughout this chapter, I adopt the standard term *binary file* to mean a file in which data isn't necessarily interpreted as ASCII character codes. With a text file, everything is assumed to be readable as text, given the right text-file reader. Files in which this is not true are called *binary*.

Introducing Binary Operations

When working with binary files, you read and write data directly to the file rather than translating data into text representations. Suppose you have the following data declarations. These variables occupy 4, 8, and 16 bytes, respectively.

```
int n = 1;
double x = 215.3
char *str[16] = "It's C++!"
```

The following statements write the values of the three variables (n, amount, and str) directly to the file. Assume that binfil is a file-stream object successfully opened in binary mode.

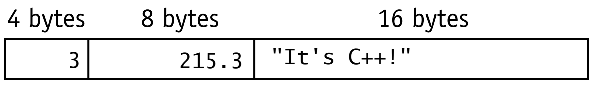
```
binfil.write((char*)(&n), sizeof(n));
binfil.write((char*)(&x), sizeof(x));
binfil.write(str, sizeof(str));
```

By the way, in this chapter (and Chapter 10, "Classes and Objects") I am using the old-fashioned C-language type cast:

```
(type) data_item
```

Normally, use of the **reinterpret_cast** operator (which recasts pointers) would be preferred here. But frankly, I just didn't have space to get all that in and, while the C++ specification committee doesn't like old-style casts, it has decided to live with them. You will not get an error message using the shorter—and frankly more convenient—older style. (For information on all the newer, preferred casts, see Appendix A.)

Here's what the data looks like after being written. (The actual binary representations use strings of 1s and 0s, but I've translated these to make them more readable.)



To read this file, you need to know how to interpret these three fields. In reality, the lines between different fields are invisible; in fact, they don't even exist, except in the mind of the programmer. (Remember, data on a computer, including disk files, is nothing but a series of bytes containing binary numbers.)

There is nothing in the file itself that tells you where one field begins and another starts. With a text file, you can always read a field by reading to the next newline or white space, but you can't do that with a binary file.

Therefore, when you read a binary file, you have to know what kind of data to expect. In the example just shown, data had this structure: an **int**, a **double**, and a 16-byte array of **char**, in that order. Thus, you could read the data by following this procedure:

1. Read 4 bytes directly into an integer variable.
2. Read 8 bytes directly into a **double** variable.
3. Read 16 bytes into a string.

That's exactly what the following lines of code do.

```
binfil.read((char*)(&n), sizeof(n));
binfil.read((char*)(&x), sizeof(x));
binfil.read(str, 20);
```

The order in which these reads are done is critical. If, for example, you tried to read the **double** (floating-point) field first, the results would be garbage because integer data and floating-point have incompatible formats.

Binary reads require a lot more precision than reading streams of text. With text input, a string of digits such as "12000" can be read as either integer or floating-point, because the text-to-numeric conversion functions know exactly how to interpret such a string. But a direct binary read performs no conversions of any kind. Copying an 8-byte double directly to a 4-byte integer would create a really unfortunate situation.

The moral: Know your data formats precisely before proceeding with binary I/O.

You perform input/output to a binary file by using the **read** and **write** member functions. These functions each take two arguments: a data address and a number of bytes, represented here as *size*.

```
fstream.read(addr, size);   // Read data into addr

fstream.write(addr, size);  // Write data from addr
```

The first argument is a data address in memory: In the case of the **read** function, this is a destination to read the file data into. In the case of the **write** function, this is a source address telling where to get the data to write to the file.

In either case, this first argument must have the type **char***, so you need to pass an address expression (a pointer, an array name, or an address obtained with &). You also need to change the type by using a **char*** type cast, unless the type is already **char***.

```
binfil.write((char*)(&n), sizeof(n));
```

In the case of string data, you don't need to use the **char*** cast because strings already have that type.

```
binfil.write(str, sizeof(str));
```

The **sizeof** operator is helpful here for specifying the second argument. It returns the size of the specified type, the variable, or the array.

Example 9.3. *Random-Access Write*

This next example writes binary data to a file. Again, observing a strict format is critical. Data fields are differentiated not by a newline or whitespace (as in a text file) but by program behavior.

The programs in this section and the next view a file as a series of fixed-length records, in which each record stores two pieces of data:

- A string field 20 bytes in length (19 characters maximum, plus one byte for a terminating null)
- An integer

This next example supports *random access.* The user can go directly to any record, specified by number. Data does not have to read data sequentially, starting at the beginning of the file and reading or writing each record in sequence.

If the user writes to an existing record number, that record is overwritten. If the user writes to a record number beyond the current length of the file, the file is automatically extended in length as needed.

writebin.cpp

```
#include <iostream>
#include <fstream>

using namespace std;

int get_int(int default_value);

int main() {
    char filename[FILENAME_MAX];
    int n = 0;
    char name[20];
```

▼ *continued on next page*

writebin.cpp, cont.

```
    int age = 0;
    int recsize = sizeof(name) + sizeof(int);


    cout << "Enter file name: ";
    cin.getline(filename, FILENAME_MAX);

    // Open file for binary write.

    fstream  fbin(filename, ios::binary | ios::out);
    if (!fbin) {
        cout << "Could not open " << filename << endl;
        return -1;
    }

    //  Get record number to write to.

    cout << "Enter file record number: ";
    n = get_int(0);

    // Get data from end user.

    cout << "Enter name: ";
    cin.getline(name, sizeof(name) - 1);
    cout << "Enter age: ";
    age = get_int(0);

    // Write data to the file.

    fbin.seekp(n * recsize);
    fbin.write(name, sizeof(name) - 1);
    fbin.write((char*)(&age), sizeof(int));
    fbin.close();
    return 0;
}

#define COL_WIDTH 80  // 80 is typical column width

// Get integer function
// Get an integer from keyboard; return default
//  value if user enters 0-length string.
//
```

writebin.cpp, cont.

```
int get_int(int default_value) {
    char s[COL_WIDTH+1];

    cin.getline(s, COL_WIDTH);
    if (strlen(s) == 0) {
         return default_value;
    }
    return atoi(s);
}
```

How It Works

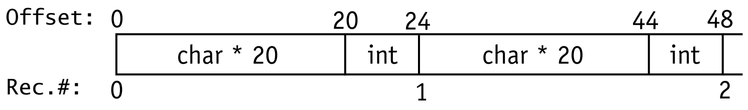
The concept of *record* is at the heart of this example. A record is a data format repeated throughout a file, giving uniformity to the file's structure. No matter how long the file grows, it's always easy to find a record by using its record number.

Note Whenever you use records in an array or a binary file, the more natural way to implement them is to use a C structure or C++ class. I spend a lot of time on classes starting in Chapter 10, "Classes and Objects."

One of the first things the program does is to calculate this record length:

```
int recsize = sizeof(name) + sizeof(int);
```

You can use this length information to go to any record. For example, record number 0 is at offset 0 in the file, record number 1 is at offset 24, record number 2 is at offset 48, record number 3 is at offset 72, and so on.



The program opens the file by specifying two flags: **ios::binary** and **ios::out**. Opening in **ios::out** mode enables a file to be opened for writing; however, be careful, because this will destroy old contents of existing files. It also lets you open new files.

```
fstream  fbin(filename, ios::binary | ios::out);
```

If the file was opened successfully, the program gets a record number from the user.

```
        cout << "Enter file record number: ";
        n = get_int(0);
```

The get_int function uses a technique for getting an integer, described in the previous chapter. The program then gets new data from the user.

```
        cout << "Enter name: ";
        cin.getline(name, sizeof(name) - 1);
        cout << "Enter age: ";
        age = get_int(0);
```

Moving to the location of the specified record is just a matter of multiplying the number by the record size (recsize, equal to 24) and then moving to that offset. The **seekp** member function performs this move.

```
        fbin.seekp(n * recsize);
```

The program then writes the data and closes the file.

```
        fbin.write(name, sizeof(name) - 1);
        fbin.write((char*)(&age), sizeof(int));
        fbin.close();
```

EXERCISES

Exercise 9.3.1. Write a program similar to Example 8.3 that writes records to a file, in which each record contains the following information: model, a 20-byte string; make, another 20-byte string; year, a five-byte string; and mileage, an integer.

Exercise 9.3.2. Revise Example 9.3 so that it prompts the user for a record number and then prompts the user for the rest of the data and repeats. To exit, the user enters –1.

Example 9.4. *Random-Access Read*

Of course, the program in Example 8.3 isn't very useful unless you have a way of reading the data placed there. The program here reads data using the same record format used in the previous section: a 20-byte string followed by a four-byte integer. The code is similar to that of Example 8.3, except for a few key statements.

In this case, the file is opened with flags **ios::bin** and **ios::in**; the latter requires the file to already exist to be successfully opened.

readbin.cpp

```
#include <iostream>
#include <fstream>

using namespace std;

int get_int(int default_value);

int main() {
    char filename[FILENAME_MAX];
    int n = 0;
    char name[20];
    int age = 0;
    int recsize =  sizeof(name) + sizeof(int);


    cout << "Enter file name: ";
    cin.getline(filename, FILENAME_MAX);

    // Open file for binary read-write access.

    fstream  fbin(filename, ios::binary | ios::in);
    if (!fbin) {
        cout << "Could not open " << filename << endl;
        return -1;
    }

    // Get record number and go to record.

    cout << "Enter file record number: ";
    n = get_int(0);
    fbin.seekp(n * recsize);

    // Read data from the file.

    fbin.read(name, sizeof(name) - 1);
    fbin.read((char*)(&age), sizeof(int));

    // Display the data and close.
```

▼ *continued on next page*

readbin.cpp, cont.

```
        cout << "The name is: " << name << endl;
        cout << "The age is: " << age << endl;
        fbin.close();
        return 0;
    }

    // Get integer function
    // Get an integer from keyboard; return default
    //  value if user enters 0-length string.
    //
    int get_int(int default_value) {
        char s[81];

        cin.getline(s, 80);
        if (strlen(s) == 0) {
             return default_value;
        }
        return atoi(s);
    }
```

How It Works

Most of this program does the same thing as Example 9.3, but because this program reads input from the file, it must be opened in **ios::in** mode (and that requires that it already exists). As before, the program gets a record number and moves to the appropriate offset after multiplying by recsize.

```
fbin.seekp(n * recsize);
```

The statements that differ from those in Example 9.3 read data from the file into the variables name and age. These are nearly the same as the corresponding **write** statements in the other example; in fact, the arguments are the same.

```
fbin.read(name, sizeof(name) - 1);
fbin.read(((char*)(&age), sizeof(int));
```

Once data is read into the two variables—name and age—the program prints the data, closes the file, and it's done.

```
cout << "The name is: " << name << endl;
cout << "The age is: " << age << endl;
fbin.close();
```

EXERCISES

Exercise 9.4.1. Write a program similar to Example 9.4 that reads records from a file, in which each record contains the following information: model, a 20-byte string; make, another 20-byte string; year, a five-byte string; and mileage, an integer.

Exercise 9.4.2. Revise Example 9.4 so that it prompts the user for a record number and then prints the data at that record and repeats. To exit, the user enters –1.

Exercise 9.4.3. Revise the example further so that it performs both random-access read *and* write. Once this is completed, you'll have one program that can handle all input/output operations for files observing this format. The file should open with the flags **ios:binary | ios::out | ios::in**. The latter requires the file to exist before being opened.

You'll need to present a command to the user by printing a menu of options:

1 Write a record.

2 Read a record.

3 Exit.

The general loop of the program should do the following: print the menu, carry out a command, and exit if option 3 is chosen. Then repeat.

Chapter 9 *Summary*

Here are the main points of Chapter 9:

- To switch on file-stream support from the C++ standard library, use this **#include** statement, which brings in prototypes and declarations as needed.

```
#include <fstream>
```

- File-stream objects provide a way to communicate with files. To create a file-output stream, use an **ofstream** type declaration. For example:

```
ofstream fout(filename);
```

- You can then write to the stream as you'd write to **cout**.

```
fout << "Hello, human." << endl;
```

- To create a file-input stream, use an **ifstream** declaration. A file-input stream supports the same operations that **cin** does, including the **getline** function.

```
ifstream fin(filename);

char input_string[MAX_PATH + 1];
fin.getline(input_string, MAX_PATH);
```

- If the file can't be opened, the file-stream object is set to a null (zero). You can test the object in a condition: if the value is zero, there was an error and the program should react as appropriate.

```
if (! file_in) {
    cout << "File " << filename;
    cout << " could not be opened.";
    return -1;
}
```

- After you're done working with a file-stream operator (regardless of mode), it is good programming practice to close it. This frees up the file so that it can be accessed by other programs.

```
fout.close();
```

- Files can be opened in either text mode or binary mode. In text mode, you read and write to a file just as you would the console. In binary mode, you use member functions to read and write data directly. To open a file stream in binary, random-access mode, use the flags **ios::out** and **ios::binary** or **ios::in** and **ios::binary**.
- Random-access mode enables you to go directly to any position in the file. You can read any portion of the file and overwrite any existing portions without affecting the rest. If the file pointer is moved beyond the file's current length, the file is automatically extended as needed.
- Use the **seekp** member function to move the file pointer. The function takes an argument giving an offset (in bytes) from the beginning of the file.

```
fbin.seekp(offset);
```

- The **read** and **write** functions each take two arguments: a data address and the number of bytes to copy.

```
fstream.read(addr, size);
fstream.write(addr, size);
```

- With the **read** function, the address argument specifies a destination; the function reads data from the file into this location. With the **write** function, the address argument specifies a source; the function reads data from that source into the file.
- Because the type of the address argument is **char***, you need to apply a cast if it is not a string. Use the **sizeof** operator to determine the number of bytes to read or write.

```
binfil.write((char*)(&n), sizeof(n));
binfil.write((char*)(&x), sizeof(x));
binfil.write(str, sizeof(str));
```

This page intentionally left blank

10 Classes and Objects

One of the most fascinating topics in C++ is object orientation. Once you understand it and have written a few programs using the object-oriented-programming (OOP) techniques, it becomes a natural way to program. However, the concepts are subtle and challenging at first.

Object orientation is, above all, a way of approaching analysis and design. There are some helpful tools in C++, but they work only as long as you understand what OOP design is all about.

As I'm going to show in the next six chapters, there are many projects that would be much harder to do without the object-oriented approach.

OOP, My Code Is Showing

Object-oriented programming (OOP) is a modular approach to programming: it creates groups of closely related code and data that work together. The major rule is this:

* **In OOP design you begin by asking: what are the principal data structures you're working on, and what actions need to be performed on each?**

In upcoming chapters, such as Chapter 15, "Object-Oriented Poker," I'll show how a potentially messy and difficult project—writing a Video Poker game—becomes easier to design if you use an object-oriented approach.

I'll go into this in much greater detail in Chapter 15, but here's a quick overview. The classes in the poker game will be:

- The Deck class. This class takes care of all the randomization, shuffling, and reshuffling of a deck of cards, freeing the rest of the program from worrying about these details.

- The Card class. This class contains the information needed to track a specific card: both rank (deuce through ace) and suits (because in Poker, flushes are possible). We'll give each Card object the ability to display itself.

With these two classes written, writing a main program that plays the game is straightforward. Remember that each class is a combination of closely related functions and data structures. Many books talk about "encapsulation" and "data abstraction," but these are just words for: *Hide the details!*

After you write a class, the next step is to use that class to create objects. But just what is an object?

What's an Object, Anyway?

A class is a data type—although potentially an intelligent one. There's a one-to-many relationship between a data type and instances of that type. For example, there's only one **int** type (along with a few related types, such as **unsigned**), but you can have any number of integers—even millions of them.

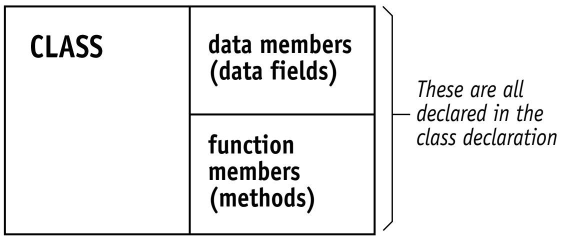
The term *object* refers to an instance in C++, especially an instance of a class. In the poker game, there will be one instance of the Deck class and at least five instances of the Card class.

Simply put, an object is an *intelligent data structure*, the structure of which is determined by its class. An object is like a data record but it can potentially do so much more. It can respond to requests in the form of function calls. If you're new to this idea, you may find it exciting. I hope you do.

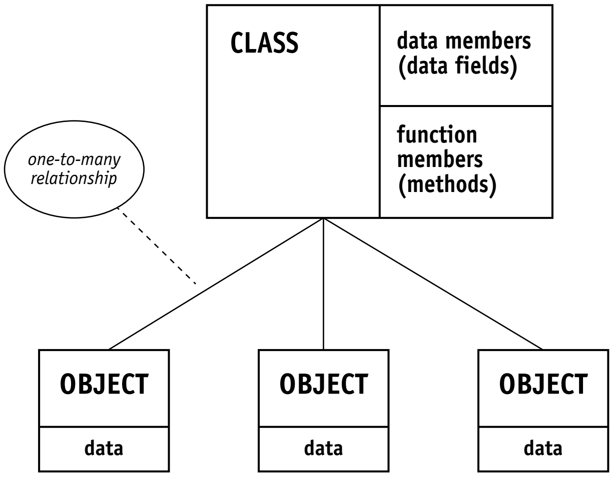
Here are the general steps in OOP. They are performed more or less in this order, although in practice you'll probably move back and forth:

1. Declare a class, or acquire one through a library.
2. Create one or more instances (called *objects*) of this class.
3. Manipulate the objects to accomplish your goals.

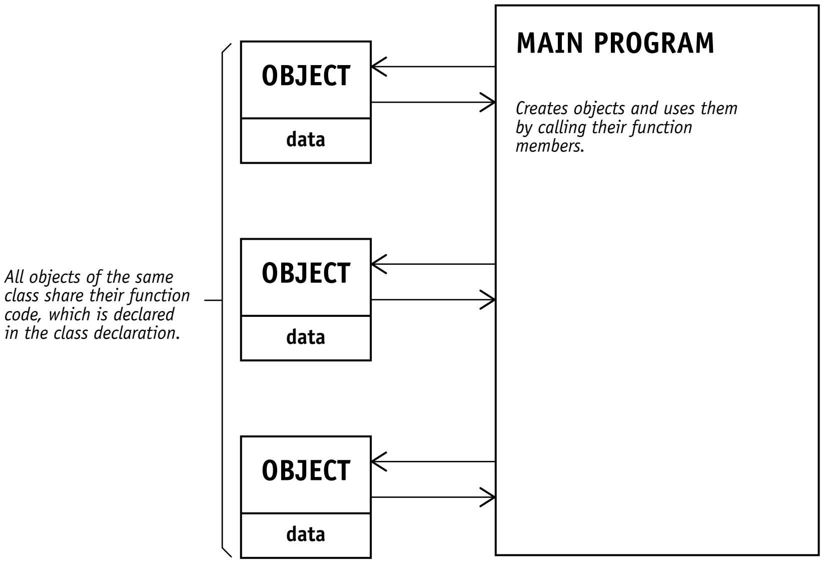
Consider each of these in turn. First, design and write the class. A class is an extended data structure, one that defines behavior for its instances (in the form of function members, or *methods*) as well as data fields.


CLASS
data members (data fields)
function members (methods)
These are all declared in the class declaration

Once a class is declared and its members defined, the program can create any numbers of instances of that class—that is, the objects. This is a one-to-many relationship.



Finally, the program uses the objects to store data. Moreover, the program can make requests of these objects, asking them to perform tasks. While each object contains its own data, its function code is shared with all other objects of the same class.


MAIN PROGRAM
Creates objects and uses them by calling their function members.
OBJECT
data
OBJECT
data
OBJECT
data
All objects of the same class share their function code, which is declared in the class declaration.

This is not yet a complete picture, however, because there are other possibilities, such as objects containing other objects. Still, this description provides a general picture of the relationship between classes, objects, and the rest of the program.

To illustrate these mechanics, I'll spend the rest of the chapter focusing on two simple classes: Point and Fraction.

Interlude

OOP...Is It Worth It?

Object orientation goes at least as far back as the 1960s with the Simula language, along with other attempts to make programming more data-centric. It got a boost in the 1970s when the Xerox PARC group (the same people who developed the graphical user interface) invented Smalltalk, a language built on the idea of independent objects sending messages to each other. By the 1980s, the concepts began to be widely evangelized.

In the early 1990s, OOP became the standard it is today. Bjarne Stroustrup married OOP to the popular C language, creating C++. Pascal and Basic also got object-oriented extensions. Thereafter, new languages followed, such as C# and Java. Today, you can't get away from it.

But do OOP concepts actually help you program more efficiently? There has been some backlash to the great push for everyone to become object oriented. Detractors argue that you end up writing the same amount of code and data anyway.

Yet a couple of points are undeniable:

- Graphical-user-interface (GUI) systems have come to dominate the world. Although you don't *have* to use an OOP language to write for such systems, they are well matched. Conceptually, they are highly compatible ideas, both developed at PARC.
- More and more, code and data are packaged into OOP form. If you want to take advantage of libraries such as Microsoft Foundation Classes (for Windows) or the C++ Standard Template Library (STL), you have no choice but to master the basics of object-oriented syntax.

Clearly, then, OOP is here to stay. And when you use the Structured Template Library, as I'll show in Chapter 13, "Easy Programming with STL," you'll reap big benefits.

Point: A Simple Class

Keyword

Here's the general syntax of the C++ class keyword:

```
class class_name {
    declarations
};
```

Except when you write a subclass, the syntax is no more complicated than this. The declarations can include data declarations, function declarations, or both. Here's a simple example that involves only data declarations:

```
class Point {
    int x, y;       // private -- may not be accessed
};
```

But members of a data structure declared with the **class** keyword are private by default, which means they cannot be accessed from outside the class. This first attempt at declaring a Point class therefore produces a class that is not useful. To be of any use, the class needs to have at least one public member.

```
class Point {
public:
    int x, y;
};
```

This is better. Now the class can actually be used. Given a class declaration for Point, you can go on to the second major step: declaring objects. In this case, the objects are pt1, pt2, and pt3.

```
Point pt1, pt2, pt3;
```

After creating these objects, you can assign values to individual data fields (called *data members*):

```
pt1.x = 1;      // Set pt1 to 1, -2.
pt1.y = -2;
pt2.x = 0;      // Set pt2 to 0, 100.
pt2.y = 100;
pt3.x = 5;      // Set pt3 to 5, 5.
pt3.y = 5;
```

What the Point class declaration does is to say that each Point object contains two data fields, x and y, which are also called *members*. You can use these members just as you would any integer variable.

```
cout << pt1.y + 4;     // Print sum of two integers.
```

In general, to refer to a data field of an object, use the following syntax:

Key Syntax

object.member_name

In this case, *object* refers to an instance of the Point class, and *member_name* can be either x or y.

Before we leave this simple version of the Point class, there's an aspect of syntax worth commenting on: a class declaration ends with a semicolon.

```
class Point {
public:
    int x, y;
};
```

When you're starting to write C++ code, it's easy to get tripped up on the semicolon. A class declaration requires a semicolon after the closing brace (}), whereas a function definition should not be followed by a semicolon. (At best, you'd be producing a null statement.)

Keep in mind this cardinal rule:

✱ **A class or data declaration always ends with a semicolon.**

So, class declarations place a semicolon after the closing brace, whereas function definitions do not.

Interlude

Interlude for C Programmers: Structures and Classes

In C++, the **struct** and **class** keywords are equivalent, except that members of a **struct** are public by default. Both keywords create classes in C++. This means that the general term *class* and the keyword **class** are not precisely co-extensive; in other words, it's possible to have a class that is not created with the **class** keyword.

In C, when you declare a structure, you have to reuse the **struct** keyword wherever the new type name appears—for example, when creating individual data items.

```
struct Point pt1, pt2, pt3;
```

This is not necessary in C++. Once you declare a class (with either the **struct** or **class** keyword), you can use the name in all contexts involving a

Interlude

▼ continued

type. So after you port C-language code to C++, you can replace the previous data declaration with this:

```
Point pt1, pt2, pt3;
```

The support of **struct** in C++ arises from the need for backward compatibility. C code often uses the **struct** keyword.

```
struct Point {
    int x, y;
};
```

The C language has no **public** or **private** keyword, and the user of a **struct** type must be able to access all members. For backward compatibility with C, therefore, types declared with **struct** had to have members that were public by default.

Does C++ really even need a **class** keyword? Technically, no, but the **class** keyword performs a self-documenting function because the purpose of a class is usually to add function members. Moreover, **class** members are private by design. In object orientation, making a member public ought to happen only as a deliberate choice.

Private: Members Only (Protecting the Data)

In the previous section, the Point class permitted direct access to its data members because they were declared public. But what if you want to control access to data members? You might, for example, want to ensure that the data is in a particular range. The way to do that is to make the members private and provide access through public functions.

The following version of Point prevents direct access to x and y from outside the class:

```
class Point {
private:             // Data members (private)
    int x, y;
public:              // Member functions
    void set(int new_x, int new_y);
    int get_x();
    int get_y();
};
```

This class declaration declares three public *member functions*—set, get_x, and get_y—as well as two private data members. Now, after declaring Point objects, the object's user can manipulate values only by calling one of the functions:

```
Point point1;
point1.set(10, 20);
cout << point1.get_x() << ", " << point1.get_y();
```

This prints the following:

```
10, 20
```

This syntax is not really new. I've used it in past chapters with objects, such as strings and **cin**. The dot (.) syntax says that a certain function (in this case, get_x) applies to a particular object.

```
point1.get_x()
```

Of course, the member functions don't work by magic; like other functions, they have to be defined somewhere. But you can place the function definitions anywhere you like, as long as the class has been declared.

The "Point::" prefix clarifies the scope of these definitions so that the compiler knows they apply to the Point class. The prefix is important, because other classes could have their own functions with these same names.

```
void Point::set(int new_x, int new_y) {
    x = new_x;
    y = new_y;
}

int Point::get_x() {
    return x;
}

int Point::get_y() {
    return y;
}
```

The "Point::" scope prefix is applied to the function name. The return type (**void** or **int**, as the case may be) still appears where it would be with a standard function definition—at the very beginning. So think of "Point::" as a function-name modifier.

The syntax for member-function definitions can be summarized as follows:

Key Syntax

```
type  class_name::function_name (argument_list) {
    statements
}
```

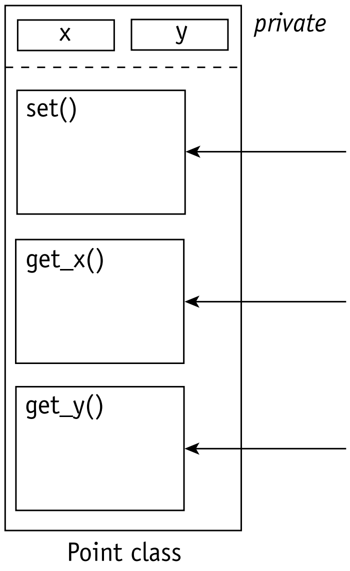
These function definitions give you, the author of the class, control over the data. You can, for example, rewrite the Point::set function so that negative input values are converted to positive.

```
void Point::set(int new_x, int new_y) {
    if (new_x < 0) {
        new_x *= -1;
    }
    if (new_y < 0) {
        new_y *= -1;
    }
    x = new_x;
    y = new_y;
}
```

Here, I'm using the multiplication-assignment operator (*=); "new_x *= –1" that has the same effect that "new_x = new_x * –1" does.

Although function code *outside the class* cannot refer to private data members x and y, function definitions *within* the class can refer to class members directly, whether private or not.

You can visualize the Point class this way: every Point object shares this same structure.



Remember that the class declaration describes the structure and behavior for the type (Point). But each Point object stores its own individual data values. For example, the following statement prints the x value stored in p1:

```
cout << pt1.get_x();  // Print value of x in pt1.
```

But this next statement prints the x value stored in p2:

```
cout << pt2.get_x();  // Print value of x in pt2.
```

Example 10.1. *Testing the Point Class*

The following program performs some simple tests on the Point class, using it to set and get some data. Code that's new is in bold; the rest is existing code previously used in this chapter.

Point.cpp

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

class Point {
private:             // Data members (private)
    int x, y;
public:               // Member functions
    void set(int new_x, int new_y);
    int get_x();
    int get_y();
};

int main() {
    Point pt1, pt2;   // Create two Point objects.

    pt1.set(10, 20);
    cout << "pt1 is " << pt1.get_x();
    cout << ", " << pt1.get_y() << endl;
    pt2.set(-5, -25);
    cout << "pt2 is " << pt2.get_x();
    cout << ", " << pt2.get_y() << endl;
    return 0;
}
```

Point.cpp, cont.

```
    void Point::set(int new_x, int new_y) {
        if (new_x < 0)
            new_x *= -1;
        if (new_y < 0)
            new_y *= -1;
        x = new_x;
        y = new_y;
    }

    int Point::get_x() {
        return x;
    }

    int Point::get_y() {
        return y;
    }
```

When run, the program should print out the following:

```
p1 is 10, 20
p2 is 5, 25
```

How It Works

This is a simple example. The Point class must be declared first so that it can be used by **main**. Then, **main** can directly use the name "Point" to create objects pt1 and pt2.

```
Point pt1, pt2;    // Create two Point objects.
```

The set, get_x, and get_y member functions can then be applied to any Point objects. For example, the following three statements call Point functions through the Point object p1, thereby accessing p1's data:

```
pt1.set(10, 20);
cout << "pt1 is " << pt1.get_x();
cout << ", " << pt1.get_y() << endl;
```

These next statements call Point functions through the p2, thereby accessing p2's data:

```
pt2.set(-5, -25);
cout << "pt2 is " << pt2.get_x();
cout << ", " << pt2.get_y() << endl;
```

You can create any number of Point objects and each stores its own copy of the data members. But all objects of the same class support the function members defined in that class. Therefore, all Point objects support the set, get_x, and get_y functions, but each will have its own data values for x and y.

EXERCISES

Exercise 10.1.1. Revise the set function so that it establishes an upper limit of 100 for values of x and y; if a value greater than 100 is entered, it is reduced to 100. Revise **main** to test this behavior.

Exercise 10.1.2. Write two new functions for the Point class, set_x and set_y, which set the individual values x and y. Remember to reverse the negative sign, if any, as is done in the set function.

Exercise 10.1.3. Revise the example so that it displays the x and y values of five Point objects.

Exercise 10.1.4. Revise the example so that it creates an array of seven Point objects. Set up a loop that prompts for values for each of the seven objects and another loop to print out all the values. (Hint: you can use a class name to declare an array, just as you can with any other type.)

```
    Point array_of_points[7];
```

Introducing the Fraction Class

One of the best ways to think about object orientation is to consider it a way to define useful new data types. In C++, a class becomes an extension to the language itself. A perfect example is a Fraction class (which could also be called a "rational number" class) that stores numbers representing a numerator and a denominator.

The Fraction class is useful if you ever need to store numbers such as 1/3 or 2/7 and you need to store them precisely. You can even use the class to store dollar-and-cents figures, such as $1.57.

In creating the Fraction class, it becomes important to restrict access to the data members for several reasons. For one thing, you should never allow a 0 denominator, because the ratio 1/0 is not a legal operation.

And, even with legal operations, it's important to simplify ratios so there's a unique expression of every rational number. For example, 3/3 and 1/1 specify the same quantity, as do 2/4 and 1/2.

In the next few sections, we'll develop functions that automatically handle this work of rejecting 0 denominators and reducing fractions. Users of the class will be able to create any number of Fraction objects, and operations such as the following will do the right thing "automagically."

```
Fraction a(1, 6);    // a = 1/6
Fraction b(1, 3);    // b = 1/3

if (a + b == Fraction(1, 2))
    cout << "1/6 + 1/3 equals 1/2" << endl;
```

Yes! You can even support the addition operator (+), as I'll show in Chapter 18, "Operator Functions: Doing It with Class"! But let's start with the simplest version of this class.

```
class Fraction {
private:
    int num, den;        // Numerator and denominator.
public:
    void set(n, d);
    int get_num();
    int get_den();
private:
    void normalize();    // Convert to standard form.
    int gcf();           // Greatest Common Factor.
    int lcm();           // Lowest Common Denominator.
};
```

This class declaration has three parts:

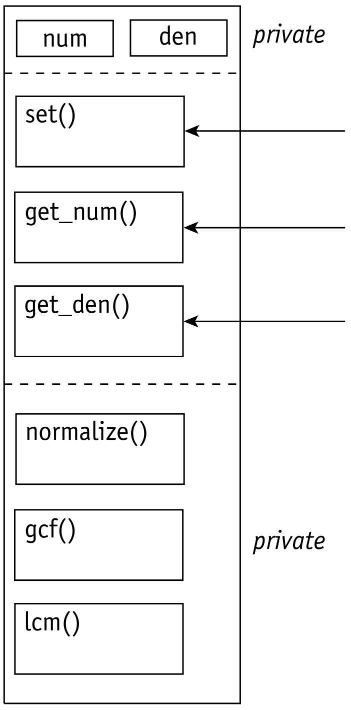
- Private data members, num and den, which store numerator and denominator. In the fraction 1/3, for example, 1 is the numerator and 3 is the denominator.
- Public function members; these provide access to class data.
- Private function members; these are support functions we'll make use of later in the chapter. For now, they just return zero values. As private members, they can't be accessed from outside, but they are useful for internal operations, as you'll see.

With these functions declared, you can use the class for simple operations such as these:

```
Fraction my_fract;
my_fract.set(1, 2);
```

```
cout << my_fract.get_num();
cout << "/"';
cout << my_fract.get_den();
```

So far, this isn't very interesting, but it's a place to start. You can visualize the Fraction class this way:



Fraction class

The member functions need to be defined somewhere. These definitions can be placed anywhere after the class declaration.

```
void Fraction::set(int n, int d) {
    num = n;
    den = d;
}

int Fraction::get_num(){
    return n;
}
```

```
int Fraction::get_den(){
    return d;
}

// TO BE DONE...
// The remaining functions are syntactically correct,
//  but don't do anything useful yet.
//  We'll fill them in later.

void Fraction::normalize(){
    return;
}

int Fraction::gcf(int a, int b){
    return 0;
}

int Fraction::lcm(int a, int b){
    return 0;
}
```

Inline Functions

Three of the functions in the Fraction class do simple things: set or get data. They are good candidates for *inlining*.

When a function is inlined, the program does not transfer control to a separate block of code. Instead, the compiler replaces the function call with the body of the function. For example, suppose that the set function is inlined as follows:

```
void set() {num = n; den = d;}
```

Now, whenever the following statement is encountered

```
fract.set(1, 2);
```

the compiler inserts the machine instructions for the "set" function. The result is the same as if the following code was inserted into the program. (This code is legal even if num and den are private because it's being performed by a member function.)

```
{fract.num = 1; fract.den = 2;}
```

You can make functions inline by placing their function definitions in the class declaration itself. These function definitions do not need to be followed by semicolons (;) even though they are member declarations.

The altered lines in the following example are in bold:

```
class Fraction {
private:
    int num, den;      // Numerator and denominator.
public:
    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num()  {return num;}
    int get_den()  {return den;}
private:
    void normalize();    // Convert to standard form.
    int gcf(int a, int b)  // Greatest Common Factor.
    int lcm(int a, int b)  // Lowest Common Denom.
};
```

Because the three private functions are not inlined, their function definitions still need to be included separately, using the "Fraction::" prefix to clarify that these are definitions of Fraction class functions.

```
void Fraction::normalize(){
    return;
}

int Fraction::gcf(int a, int b){
    return 0;
}

int Fraction::lcm(int a, int b){
    return 0;
}
```

If a function definition is short, you can improve efficiency by writing it as an inline function. Remember, this is done just by simply including the function's definition inside the class declaration itself, so it doesn't need to be defined anywhere else.

The following table compares class inline functions to other functions:

INLINE FUNCTIONS	OTHER CLASS FUNCTIONS
Defined (not just declared) inside the class declaration itself	Defined outside the class declaration but prototyped inside the class
No scope prefix (such as "Point::") is used	The scope prefix must be used in the definition
At run time, body of function is "inlined"—inserted into the code	At run time, a true function call is made, transferring execution to another code location
Appropriate for small functions	Appropriate for longer functions
Has some restrictions; cannot use recursive calls	No special restrictions

Find the Greatest Common Factor

Actions inside the Fraction class are based on two concepts in number theory: greatest common factor and lowest common multiple. Chapter 5, "Functions: Many Are Called," described Euclid's algorithm for greatest common factor and we can use that here.

NUMBERS	GREATEST COMMON FACTOR
12, 18	6
12, 10	2
25, 50	25
50, 75	25

Here is Euclid's algorithm from Chapter 5, written as a recursive C++ function:

```
int gcf(int a, int b) {
    if (b == 0) {
        return a;
    } else
        return gcf(b, a%b);
    }
}
```

Now, to rewrite this as a member function, we just add the Fraction:: prefix to give it Fraction-class scope:

```
int Fraction::gcf(int a, int b) {
    if (b == 0) {
            return a;
        } else
            return gcf(b, a%b);
        }
    }
```

Strange things happen if negative numbers are passed to the GCF function: it still produces correct results—gcf(35, –25) produces 5—but the resulting sign becomes difficult to predict. To remove this problem, we can use the **abs** (absolute value) function to ensure only positive values are returned. Here, the changes to the original version of gcf are in bold:

```
    int Fraction::gcf(int a, int b) {
        if (b == 0) {
            return abs(a);
        } else {
            return gcf(b, a%b);
        }
    }
```

Find the Lowest Common Denominator

Another useful support function gets the lowest common multiple (LCM). Because we already have the GCF function, LCM should be easy.

The LCM is the lowest number that is a multiple of both of two inputs. This is the converse of the greatest common factor (GCF). So, for example, the LCM of 200 and 300 is 600. The greatest common factor is 100.

The trick in finding the LCM is to isolate the greatest common factor and multiply by this factor only once. In multiplying A and B, you implicitly include the same factor twice. The common factor must therefore be removed from A and from B. The formula is

$$n = GCF(a, b)$$

$$LCM(A, B) = n * (a / n) * (b / n)$$

which simplifies to the following:

$$LCM(A, B) = a / n * b$$

The LCM function is now easy to write.

```
int Fraction::lcm(int a, int b) {
    int n = gcf(a, b);
    return a / n * b;
}
```

Example 10.2. *Fraction Support Functions*

The GCF and LCM functions can now be added to the Fraction class. Here's a first working version of the class. I've also added code for the normalize function, which simplifies fractions after each operation. Code that's new or altered from earlier versions is in bold.

Fract1.cpp

```
#include <cstdlib>

class Fraction {
private:
    int num, den;      // Numerator and denominator.
public:
    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num()  {return num;}
    int get_den()  {return den;}
private:
    void normalize();    // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);  // Lowest Common Denom.
};

// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
void Fraction::normalize(){

    // Handle cases involving 0

    if (den == 0 || num == 0) {
        num = 0;
```

▼ continued on next page

Fract1.cpp, cont.

```
            den = 1;
        }

        // Put neg. sign in numerator only.

        if (den < 0) {
            num *= -1;
            den *= -1;
        }

        // Factor out GCF from numerator and denominator.

        int n = gcf(num, den);
        num = num / n;
        den = den / n;
    }

    // Greatest Common Factor
    //
    int Fraction::gcf(int a, int b){
        if (b == 0)
            return abs(a);
        else
            return gcf(b, a%b);
    }

    // Lowest Common Multiple
    //
    int Fraction::lcm(int a, int b){
        int n = gcf(a, b);
        return a / n * b;
    }
```

How It Works

When the gcf function calls itself in the recursive function call,

```
gcf(a/i, b/i)
```

it's not necessary to use the Fraction:: prefix. That's because inside a class function, class scope is assumed. Similarly, when the Fraction::lcm function calls gcf, class scope is again assumed.

```
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}
```

In general, each time the C++ compiler comes across a variable or function name, it looks for the declaration of that name in this order:

- It looks within the same function (in the case of local variables).
- It looks within the same class.
- If no declaration is found at the function or class level, the compiler looks for a global declaration.

The normalize function is the only new code here. The first thing the function does is to handle cases involving zero. A denominator equal to 0 is invalid, so the fraction is changed to 0/1. In addition, all values with numerators equal to 0 are equivalent.

```
0/1    0/2    0/5    0/-1    0/25
```

These are all put in a standard form of 0/1.

One of the main goals of the Fraction class is to ensure that equal values are represented the same way. This will make it easy to implement the test-for-equality operator later on. Negative numbers pose an issue. For example, these two expressions represent the same value

```
-2/3    2/-3
```

as do these:

```
4/5    -4/-5
```

The easiest solution is to test the denominator: If it's less than 0, reverse the sign of both the numerator and denominator.

```
if (den < 0) {
    num *= -1;
    den *= -1;
}
```

The rest of the function is straightforward: Find the greatest common factor and then divide both the numerator and denominator by this amount:

```
int n = gcf(num, den);
num = num / n;
den = den / n;
```

For example, take the fraction 30/50. The greatest common factor is 10. The normalize function executes the necessary division and produces 3/5.

The normalize function is important because it ensures equivalent values are expressed the same way. Also, when we start crunching numbers with the Fraction class, large numbers can accumulate for the numerator and denominator. To avoid overflow errors at runtime, it's important to reduce Fraction expressions at every opportunity.

EXERCISES

Exercise 10.2.1. Rewrite the normalize function so that it uses the division-assignment operator (/=). Remember that this operation

```
a /= b
```

is equivalent to the following:

```
a = a / b
```

Exercise 10.2.2. Inline every class function in which it would be reasonable to do so. (Hint: gcf can't be inlined because it is recursive, and normalize is too long.)

Example 10.3. *Testing the Fraction Class*

Once you have completed a class declaration, you need to test it by creating and using objects. The following code prompts for input values and displays values after simplifying the fractions:

Fract2.cpp

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

class Fraction {
private:
    int num, den;      // Numerator and denominator.
public:
    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num()  {return num;}
    int get_den()  {return den;}
```

Fract2.cpp, cont.

```
private:
    void normalize();   // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);   // Lowest Common Denom.
};

int main()
{
    int a, b;
    string str;
    Fraction fract;
    while (true) {
        cout << "Enter numerator: ";
        cin >> a;
        cout << "Enter denominator: ";
        cin >> b;
        fract.set(a, b);
        cout << "Numerator is   " << fract.get_num()
             << endl;
        cout << "Denominator is " << fract.get_den()
             << endl;
        cout << "Do again? (Y or N) ";
        cin >> str;
        if (!(str[0] == 'Y' || str[0] == 'y'))
             break;
    }
    return 0;
}

// ---------------------------------------------------
// FRACTION CLASS FUNCTIONS

// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
void Fraction::normalize(){

    // Handle cases involving 0
```

▼ continued on next page

Fract2.cpp, cont.

```
    if (den == 0 || num == 0) {
        num = 0;
        den = 1;
    }

     // Put neg. sign in numerator only.

    if (den < 0) {
        num *= -1;
        den *= -1;
    }

    // Factor out GCF from numerator and denominator.

    int n = gcf(num, den);
    num = num / n;
    den = den / n;
}

// Greatest Common Factor
//
int Fraction::gcf(int a, int b) {
    if (b == 0) {
        return abs(a);
    } else {
        return gcf(b, a%b);
    }
}

// Lowest Common Multiple
//
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}
```

How It Works

A common practice is to put class declarations, along with any other needed declarations and directives, into a header file. Assuming that the name of this header file was Fraction.h, you'd need to add the following to a program that used the Fraction class:

```
#include "Fraction.h"
```

Function definitions that are not inlined must be placed somewhere in the program, or else they must be separately compiled and linked into the project.

The third line of **main** creates an uninitialized Fraction object.

```
Fraction fract;
```

Other statements in **main** then set the Fraction and print its value. Note that the call to the "set" function assigns values, but it also calls the normalize function, which automatically causes the fraction to be simplified as appropriate.

```
fract.set(a, b);
cout << "Numerator is   " << fract.get_num()
     << endl;
cout << "Denominator is " << fract.get_den()
     << endl;
```

Interlude

A New Kind of #include?

In the previous example, you may notice I introduced new syntax for the **#include** directive. Remember that to turn on support for an area of the C++ library, the preferred method is to use angle brackets.

```
#include <iostream>
```

But to include declarations from your own project files, you need to use quotation marks.

```
#include "Fraction.h"
```

The two forms of the **#include** directive do almost the same thing, but with the quote-mark syntax, the C++ compiler is directed to look first in the current directory and then, only after that, to look in the standard include-file directory (which is usually set by an environment variable or environment setting of the operating system).

Depending on what C++ compiler you have, you could probably get away with using the quote-mark syntax for both library files and project files. But the standard practice is to use angle brackets to turn on features of the standard library, which is the practice I follow in this book.

EXERCISES

Exercise 10.3.1. Write a program that uses the Fraction class to set a series of values: 2/2, 4/8, −9/−9, 10/50, 100/25. Have the program print out the results and verify that each fraction was correctly simplified. So, for example, 100/25 should be automatically simplified to 5/4.

Exercise 10.3.2. Create an array of five Fraction objects. Then, write a loop to input the numerator and denominator for each. Finally, write a loop to print each of the five objects, using "get" functions.

Exercise 10.3.3. If you're really ambitious, write and test another member function. This one should display both numerator and denominator. If you like, you can even have it display this data in fractional form, such as "1/2" or "2/5," for example.

Example 10.4. *Fraction Arithmetic: add and mult*

The next step in creating a Fraction class is to add some arithmetic functions, add and mult. Addition is the hardest, but you may recall the technique from school. Consider this addition of two fractions:

```
A/B + C/D
```

The trick is first to find the lowest common denominator, which is the same as the lowest common multiple (LCM) between B and D.

```
LCD = LCM(B, D).
```

We now have a convenient utility function, LCM, to do just that. Then A/B has to be converted to a fraction that uses this lowest common denominator (LCD):

```
A   *   LCD/B
--      -----
B   *   LCD/B
```

We then get a fraction in which the denominator is LCD. It's similar for C/D.

```
C   *   LCD/D
--      -----
D   *   LCD/D
```

After these multiplications are done, the two fractions will have a common denominator (LCD), and they can be added together. The resulting fraction is as follows:

```
(A * LCD/B)  + (C * LCD/D)
--------------------------
           LCD
```

Therefore, the algorithm is as follows:

```
Calculate LCD from LCM(B, D)
Set Quotient1 to LCD/B
Set Quotient2 to LCD/D
Set numerator for the new fraction to A * Quotient1 + C * Quotient2
Set denominator for the new fraction to LCD
```

The solution for multiplication of two fractions is quite a bit easier.

```
Set numerator for the new fraction to A * C
Set denominator for the new fraction to B * D
```

We can now write code that declares and implements the two new functions. As before, the lines that are bold represent new or altered lines; everything else is the same as in the previous example.

Fract3.cpp

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

class Fraction {
private:
    int num, den;      // Numerator and denominator.
public:
    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num()  {return num;}
    int get_den()  {return den;}
    Fraction add(Fraction other);
    Fraction mult(Fraction other);
```

▼ continued on next page

Fract3.cpp, cont.

```
private:
    void normalize();   // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);  // Lowest Common Denom.
};

int main()
{
    Fraction fract1, fract2, fract3;

    fract1.set(1, 2);
    fract2.set(1, 3);
    fract3 = fract1.add(fract2);
    cout << "1/2 plus 1/3 = ";
    cout << fract3.get_num() << "/" << fract3.get_den()
         << endl;
    return 0;
}

// ---------------------------------------------------
// FRACTION CLASS FUNCTIONS
// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
void Fraction::normalize(){

    // Handle cases involving 0

    if (den == 0 || num == 0) {
        num = 0;
        den = 1;
    }

    // Put neg. sign in numerator only.

    if (den < 0) {
        num *= -1;
        den *= -1;
    }
```

Fract3.cpp, cont.

```
    // Factor out GCF from numerator and denominator.

    int n = gcf(num, den);
    num = num / n;
    den = den / n;
}

// Greatest Common Factor
//
int Fraction::gcf(int a, int b) {
    if (b == 0) {
        return abs(a);
    } else {
        return gcf(b, a%b);
    }
}


// Lowest Common Denominator
//
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}

Fraction Fraction::add(Fraction other) {
    Fraction fract;
    int lcd = lcm(den, other.den);
    int quot1 = lcd/den;
    int quot2 = lcd/other.den;
    fract.set(num * quot1 + other.num * quot2, lcd);
    return fract;
}

Fraction Fraction::mult(Fraction other) {
    Fraction fract;
    fract.set(num * other.num, den * other.den);
    return fract;
}
```

How It Works

The add and mult functions apply the algorithms that I described earlier. They also use a new type signature: each of these functions takes an argument of type Fraction and also returns a value of type Fraction. Consider the type declaration of the add function.

$$\underset{\textcircled{1}}{\underline{\texttt{Fraction}}}\quad \underset{\textcircled{2}}{\underline{\texttt{Fraction::}}}\texttt{add}\quad (\underset{\textcircled{3}}{\underline{\texttt{Fraction}}}\ \texttt{other});$$

Each occurrence of "Fraction" in this declaration has a different purpose.

- The use of Fraction at the beginning of the declaration indicates that the function returns an object of type Fraction.
- The name prefix Fraction:: indicates that the add function is declared within the Fraction class.
- Within the parentheses, Fraction indicates that there is one argument, named other, which has type Fraction.

Each of these uses is distinct. For example, you could have a function that takes an argument of type **int** and returns a Fraction object, but is not declared within the Fraction class. The declaration would look like this:

```
Fraction my_func(int n);
```

Because the Fraction::add function returns an object of type Fraction, it must first create a new object.

```
Fraction fract;
```

The function then applies the algorithm I described earlier.

```
int lcd = lcm(den, other.den);
int quot1 = lcd/other.den;
int quot2 = lcd/den;
```

Finally, after setting the values for the new Fraction object (fract), the function returns this object:

```
return fact;
```

EXERCISES

Exercise 10.4.1. Rewrite **main** so that it adds any two fractions input and prints the results.

Exercise 10.4.2. Rewrite **main** so that it multiplies any two fractions input and prints the results.

Exercise 10.4.3. Write an add function for the Point class introduced earlier. The function should add the x values to get the new value of x, and it should add the y values to get the new value of y.

Exercise 10.4.4. Write sub and div functions for the Fraction class, along with code in **main** to test these functions. (The algorithm for sub is similar to that for add, although you can write an even simpler function by multiplying the numerator of the argument by –1 and then just calling the add function.)

Chapter 10 *Summary*

Here are the main points of Chapter 10:

- A class declaration has this form:

```
class class_name {
    declarations
};
```

- In C++, the **struct** keyword is equivalent to the **class** keyword, except that in classes declared with **struct**, members are public by default.
- Because members of a class declared with the **class** keyword are private by default, you need to declare at least one member public.

```
class Fraction {
private:
    int num, den;
public:
    void set(n, d);
    int get_num();
    int get_den();
private:
    void normalize();
```

```
        int gcf();
        int lcm();
    };
```

- Class and data declarations end with a semicolon; function definitions do not.
- Once a class is declared, you can use it as a type name, just as you would **int**, **float**, **double**, and so on. For example, you can declare a series of objects:

```
    Fraction a, b, c, my_fraction, fract1;
```

- Functions of a class can refer to other members within that same class (whether private or not) without use of the scope operator (::).
- To place a member-function definition outside its class's declaration, use this syntax:

    type class_name::*function_name* (*argument_list*)

      statements

    }

- If you place a member-function definition inside the class declaration, the function is inline. When the function is called, machine instructions that implement the function are placed into the body of the program.
- When you inline a function, no semicolon is needed after the closing brace.

```
    void set(n, d) {num = n; den = d;}
```

- The class declaration must precede all uses of the class. The function definitions can be placed anywhere in the program (or even in a separate module), but they must follow the class declaration.
- If a function has a class for its return type, it must return an object of that type. One way to do this is to first declare such an object as a local variable.

11 Constructors: If You Build It...

One of the themes in this book is that object orientation is a way to create fundamental new data types—types that, if useful enough, can be reused in multiple programs.

An important aspect of types is that you can initialize them. It's also reasonable to ask to do this with objects as well to be able to initialize them upon construction. In fact, that makes object-oriented syntax much more convenient and programmer-friendly.

A constructor is essentially an initialization function. Welcome to the craft of C++ construction.

Introducing Constructors

A *constructor* tells the compiler how to interpret declarations like this:

```
Fraction a(1, 2);    //a = 1/2
```

Given what you've seen of the Fraction class, you'd probably guess that what this declaration *ought* to do is have the same effect as the following statements:

```
Fraction a;
a.set(1, 2);
```

And in fact, in this chapter, we're going to make the class behave precisely that way. That's what constructors are for.

A constructor declaration has this syntax:

Pseudocode

class_name(*argument_list*)

This makes for an odd-looking function. There is no return type—not even **void**. The class name, in a sense, is the return type. Here's an example:

```
Fraction(int n, int d);
```

Within the context of the class, the declaration looks like this:

```
class Fraction {
public:
// ...
    Fraction(int n, int d);
// ...
};
```

This is only a declaration, of course. Like any function, the constructor needs to be defined somewhere. You can place the definition outside the class declaration, if you choose, in which case you have to clarify scope.

```
Fraction::Fraction(int n, int d) {
    set(n, d);
}
```

A constructor defined outside of the declaration has this syntax:

Key Syntax

class_name::*class_name*(*argument_list*) {
  statements
}

The first use of *class_name* is in the name prefix (*class_name*::), which states that this is a member function of the specified class—that is, it has class scope. The second use of *class_name* says that this function is a constructor.

You can also inline the constructor. Because most constructors are short, they are often good candidates for inlining.

```
class Fraction {
public:
// ...
    Fraction(int n, int d) {set(n, d);}
// ...
};
```

Now, with this constructor in place, you can initialize Fraction objects as you declare them.

```
Fraction frOne(1, 0), frTwo(2, 0), frHalf(1, 2);
```

Multiple Constructors (Overloading)

In C++, you can reuse a name to create different functions, relying on different argument lists to differentiate them. This extends to constructors.

For example, you can declare several different constructors for the Fraction class—one with no arguments, another with two, and a third with just one argument. At compile time, the compiler looks at the argument list to see which constructor to call.

```
class Fraction {
public:
// ...
    Fraction();
    Fraction(int n, int d);
    Fraction(int n);
// ...
};
```

C++11/C++14 Only: Initializing Members

C++14 ▶ **This section applies only to C++11 and later compilers.**

Beginning with C++11, the language provides a new way to specify default initial values for data members. This technique might sound like it conflicts with writing constructors or makes them unnecessary. Actually, it doesn't; the two techniques work together smoothly.

With the Point class, it's reasonable for an object to default to zero values. C++11 enables you to do this by initializing members within the class declaration itself.

```
class Point {
  public:
      int x = 0;
      int y = 0;
};
```

Now, an uninitialized Point object takes on zero values even if it is local.

```
int main() {
    Point silly_point;
    cout << silly_point.x;  //This prints 0.
```

For the Fraction class, you'd want to assign 1 to the denominator, not 0, because 0/0 is not a legitimate fraction!

```
class Fraction {
   private:
      int num = 0;
```

```
        int den = 1;
    ...
```

When you use C++ to initialize values this way, every constructor assigns the specified value (in this case, 0 and 1) for each data member you choose to initialize, except where a constructor overrides that setting with values of its own.

If you write no constructors at all, this approach provides an alternative way to initialize objects to reasonable default values. As usual, however, it's recommended that you always ought to write a default constructor, unless you want to prevent the class user from creating an object without initializing it (as described in the next section).

The Default Constructor—and a Warning

Every time you write a class, you should usually write a default constructor—that's the constructor with no arguments—unless you want to strictly require the user of the class to initialize every object as soon as he or she declares it. This is because if you write no constructors, the compiler supplies a default constructor for you, which does nothing. But if you write any constructors at all, the compiler does not supply a default constructor.

Suppose you declare a class with no constructors.

```
class Point {
private:
    int x, y;
public:
    set(int new_x, int new_y);
    int get_x();
    int get_y();
};
```

Because you wrote a class with no constructors, the compiler supplies one: a default constructor. That's the constructor with no arguments. Because this constructor is supplied for you, you can go ahead and use the class to declare objects.

```
Point a, b, c;
```

But look what happens as soon as you do define a constructor:

```
class Point {
private:
    int x, y;
public:
    Point(int new_x, int new_y) {set(new_x, new_y);}
```

```
        set(int new_x, int new_y);
        int get_x();
        int get_y();
    };
```

With this constructor in place, you can now declare objects this way:

```
    Point a(1, 2), b(10, -20);
```

But now, you get an error if you try to declare objects with no arguments!

```
    Point c;    //ERROR! No more default constructor
```

The compiler-supported default constructor that you had been relying on is rudely yanked away!

When you first start writing classes, this behavior can take you by surprise. You use a class without writing constructors, letting users of the class declare objects this way:

```
    Point a, b, c;
```

But this innocent-looking code breaks as soon as you write a constructor other than the default constructor.

In some cases, you might not want a default constructor at all; instead, you might want to force the user of the class to initialize objects to specific values. In that case, this behavior is perfectly acceptable.

Interlude

Is C++ Out to Trick You with the Default Constructor?

It may seem strange that C++ operates this way, lulling you into a false sense of security by supplying a default constructor (again, that's the constructor with no arguments) and then yanking it away from you as soon as you write any other constructor.

Admittedly, this is weird behavior. It's one of the quirks that C++ has because it has to be both an object-oriented language and a language designed to be relatively backwardly compatible with C. (Actually, it is not 100 percent backwardly compatible, but it comes close.)

The **struct** keyword, in particular, causes some issues. C++ treats a **struct** type as a class (as I've mentioned), but it also has to work so that C code, such as the following, still compiles successfully in C++:

```
    struct Point {
      int x, y;
    };
```

▼ *continued on next page*

Interlude

▼ continued

```
struct Point a;
a.x = 1;
```

The C language has no **public** or **private** keyword, so this code can compile only if **struct** classes have members that are public by default. Another problem is that the C language has no concept of a constructor; so if this code is to compile in C++, then the C++ compiler *must* supply a default constructor, enabling statements like this to compile:

```
struct Point a;
```

By the way, C++ supports this usage but also allows you to drop *struct* in this context:

```
Point a;    //"struct" not necessary
```

So, for backward compatibility, C++ had to supply an automatic default constructor. However, if you write any constructor at all, it's assumed that you are writing original code in C++ and, therefore, that you know all about member functions and constructors.

In that case, your excuse—that you don't know about constructors—is gone, and C++ assumes you ought to write everything you need, including the default constructor.

Remember, also, that C++ gives you the choice of not writing a default constructor in order to force the class user to initialize objects explicitly. This can be quite useful at times; for example, in Chapter 12, "Two Complete OOP Examples," the default constructor is deliberately omitted so that someone who creates a node must initialize it.

C++11/C++14 Only: Delegating Constructors

C++14 ▶ **This section applies only to compilers implementing the C++11 spec and later.**

Once you've written a constructor for a given class, it would be nice to be able to reuse it in other constructors. C++14 lets you do this—in fact, it has been part of the standard specification beginning in C++11 (but some compiler vendors have implemented it only recently). Suppose you have this simple declaration of the Point class:

```
Class Point {
private:
  int x, y;
public:
  Point(int new_x, int new_y) {x = new_x; y = new_y;}
};
```

It would be nice to be able to write a default constructor by reusing the existing constructor. Here's how to do that with C++11 and later compilers:

```
Class Point {
private:
  int x, y;
public:
  Point(int new_x, int new_y) {x = new_x; y = new_y;}
  Point():Point(0, 0) {}
};
```

Look again at that new line of code:

```
    Point():Point(0, 0) {}
```

This declares a default constructor, but it delegates the work to the other constructor: it calls the constructor with two integer arguments and passes the arguments 0, 0. With the C++11 specification (fully supported in C++14 of course), still another way to achieve this result is to initialize individual data members within the class.

```
Class Point {
private:
  int x = 0;
  int y = 0;
public:
  Point(int new_x, int new_y) {x = new_x; y = new_y;}
  Point(){}
};
```

Example 11.1. *Point Class Constructors*

This example revisits the Point class from the previous chapter and adds a couple of simple constructors: the default constructor and a constructor taking two arguments. It then tests these in a simple program.

Point2.cpp

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

class Point {
private:               //Data members (private)
    int x, y;
public:                 //Constructors
    Point() {x = 0; y = 0;}
    Point(int new_x, int new_y) {set(new_x, new_y);}

// Other member functions

    void set(int new_x, int new_y);
    int get_x();
    int get_y();
};

int main() {
    Point pt1, pt2;
    Point pt3(5, 10);

    cout << "The value of pt1 is ";
    cout << pt1.get_x() << ", ";
    cout << pt1.get_y() << endl;

    cout << "The value of pt3 is ";
    cout << pt3.get_x() << ", ";
    cout << pt3.get_y() << endl;
    return 0;
}

void Point::set(int new_x, int new_y) {
    if (new_x < 0)
        new_x *= -1;
    if (new_y < 0)
        new_y *= -1;
    x = new_x;
    y = new_y;
}
```

Point2.cpp, cont.

```
int Point::get_x() {
    return x;
}

int Point::get_y() {
    return y;
}
```

How It Works

Two declarations in the class declaration create the constructors.

```
public:                //Constructors
    Point() {x = 0; y = 0;}
    Point(int new_x, int new y) {set(new_x, new_y);}
```

Note that the constructors are declared in the public section of the class. If they were declared private, they wouldn't be accessible to users of the Point class, and so the whole point (as it were) would be lost.

The default constructor sets Point members to zero, which is useful behavior if the user of the class forgets to initialize them explicitly.

```
Point() {x = 0; y = 0;}
```

The code in **main** uses the default constructor twice (for pt1 and pt2), and it uses the second constructor once (for pt3).

```
Point pt1, pt2;
Point pt3(5, 10);
```

EXERCISES

Exercise 11.1.1. Add code to the two constructors of the Point class to report their use. The default constructor should print "Using default constructor," and the other should print "Using (int, int) constructor." (Tip: If you want to keep these functions inline, you can have the function definitions span multiple lines if you need to do so.)

Exercise 11.1.2. Add a third constructor that takes just one integer argument. This constructor should set x to the argument specified and set y to 0.

Exercise 11.1.3. If you have a C++11 or later compiler, use individual member initialization (described a few sections earlier) to create a default value of 0 for both x and y. Then, write a default constructor that does nothing and a constructor that assigns a value to x but not to y. Finally, test the combinations. You should find that 0 is an effective default for x and y, but that one or both can be overridden in the constructors.

Example 11.2. *Fraction Class Constructors*

This example features a default constructor that sets the fraction to 0/1. As always, lines that are new or altered are in bold. Everything else is unchanged from the previous version of the Fraction class in Chapter 10.

Fract4.cpp

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

class Fraction {
private:
    int num, den;      //Numerator and denominator.
public:
    Fraction() {set(0, 1);}
    Fraction(int n, int d) {set(n, d);}

    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num()  {return num;}
    int get_den()  {return den;}
    Fraction add(Fraction other);
    Fraction mult(Fraction other);
private:
    void normalize();  // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);  // Lowest Common Denomin.
};

int main() {
    Fraction f1, f2;
    Fraction f3(1, 2);

    cout << "The value of f1 is ";
```

Fract4.cpp, cont.

```
    cout << f1.get_num() << "/";
    cout << f1.get_den() << endl;

    cout << "The value of f3 is ";
    cout << f3.get_num() << "/";
    cout << f3.get_den() << endl;
    system("PAUSE");
    return 0;
}

// ---------------------------------------------------
// FRACTION CLASS FUNCTIONS

// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
void Fraction::normalize(){

    // Handle cases involving 0

    if (den == 0 || num == 0) {
        num = 0;
        den = 1;
    }

    // Put neg. sign in numerator only.

    if (den < 0) {
        num *= -1;
        den *= -1;
    }

    // Factor out GCF from numerator and denominator.

    int n = gcf(num, den);
    num = num / n;
    den = den / n;
}

// Greatest Common Factor
//
```

▼ *continued on next page*

Fract4.cpp, cont.

```
int Fraction::gcf(int a, int b){
    if (b == 0)
        return abs(a);
    else
        return gcf(b, a%b);
}

// Lowest Common Multiple
//
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}

Fraction Fraction::add(Fraction other) {
    Fraction fract;
    int lcd = lcm(den, other.den);
    int quot1 = lcd/den;
    int quot2 = lcd/other.den;
    fract.set(num * quot1 + other.num * quot2, lcd);
    return fract;
}

Fraction Fraction::mult(Fraction other) {
    Fraction fract;
    fract.set(num * other.num, den * other.den);
    return fract;
}
```

How It Works

If you followed Example 11.1, this example is straightforward. The twist is that the default constructor needs to set the denominator value to 1 rather than 0.

```
Fraction() {set(0, 1);}
```

The code in **main** uses constructors three times. The first two variable declarations (f1, f2) invoke the default constructor. The declaration of f3 invokes the other constructor.

EXERCISES

Exercise 11.2.1. Rewrite the default constructor so that instead of calling set(0, 1), it sets the data members, num and den, directly. Is this more or less efficient? Is it necessary to call the normalize function?

Exercise 11.2.2. Write a third constructor that takes just one *int* argument. Respond by setting num to this argument and by setting den to 1. Do you need to call normalize?

Reference Variables and Arguments (&)

Before you proceed to learn about the other special constructor (called the *copy constructor*), it's necessary to take a detour to learn about C++ references. These were briefly introduced in Chapter 6.

The simplest way to manipulate a variable, of course, is to do so directly.

```
int n;
n = 5;
```

The next way to manipulate a variable—as you should recall from Chapter 6—is to use a pointer.

```
int n, *p;
p = &n;      // Let p point to n.
*p = 5;      // Set the THING p POINTS TO, to 5.
```

Here, p points to n, so setting *p to 5 has the same effect as setting n to 5.

The important idea here is that there's one copy of n, but you can have any number of pointers to it. Getting a pointer to n does not create a new integer, just another way to manipulate n.

A reference does much the same thing, although it avoids the pointer syntax.

```
int n;
int &r = n;
```

The ampersand (&) is the same character used for the address operator. The difference is that here it's being used in a data declaration. In this context, it creates a reference variable that refers to the variable n. This means that changes to r (the reference variable) cause changes to n.

```
r = 5;       // This has the effect of setting n to 5.
```

The crucial thing that references and pointers have in common here is that they just create another way to refer to an existing data item; they do not allocate new data. For example, I can create many references to n.

```
int n;
int &r1 = n;
int &r2 = n;
int &r3 = n;

r1 = 5;   // n is now 5.
r2 = 25;  // n is now 25.
cout << "New value of n is " << r3;  // Print n.
```

Now, changes to *any* of the reference variables—r1, r2, and r3—all cause changes to n.

Reference variables like this are rarely used in C++. Much more useful are reference arguments. Remember the swap function from Chapter 6, which required pointers? You can create the same behavior by using reference arguments.

```
void swap_ref(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Remember that in this example, the swap function does not get copies of a and b but references to them. This enables swap to make permanent changes to the arguments.

Passing a reference has an effect much like passing a pointer, but the pointer syntax is eliminated. So, in this example, you pass integers rather than pointers to integers.

```
int big = 100;
int little = 1;
swap_ref(big, little);    // swap big and little
```

The Copy Constructor

Another special constructor is the *copy constructor*. It's notable for two reasons. First, this constructor gets called in a number of common situations, whether you're aware of its existence or not.

Second, if you don't write one, the compiler automatically supplies one for you, although it may not always do what you want. This compiler-supplied constructor just performs a simple member-by-member copy (although for most classes, that behavior is sufficient).

The copy constructor is automatically called in these circumstances:

- When the return value of a function has class type.
- When an argument has class type. A copy of the argument is made and then passed to the function.
- When you use one object to initialize another. For example:

```
Fraction a(1, 2);
Fraction b(a);
```

Remember that you can also use the equal sign (=) to perform initialization of one object by another.

```
Fraction b = a;
```

Finally, you should realize that in C++11 and later, braces can—and should—be used to specify multiple initialization arguments.

```
Fraction a {1, 2};
```

Now, how do you write your own customized copy constructor, should you need one? Remember, if you don't write one the compiler automatically supplies one for you.

The syntax for a copy constructor declaration is:

Keyword

class_name(*class_name* **const** *&source*)

The **const** keyword ensures that the argument cannot be altered by the function, which makes sense because making a copy of something should never corrupt the original.

This syntax also uses a reference argument. The function gets a reference to the *source* object, not a new copy of it.

Here's an example for the Point class. First, the copy constructor has to be declared within the class declaration.

```
class Point {
//...
public:              // Constructors
    Point(Point const &src);
//...
};
```

Because this function definition was not inlined, the definition has to be included separately. Outside of the class declaration, Point must occur three times, the first time to define the scope.

```
Point::Point(Point const &src) {
    x = src.x;
    y = src.y;
}
```

Why write a copy constructor at all, if the compiler supplies one? In this case—and also in the case of the Fraction class—it isn't necessary. The compiler-supplied copy constructor performs a simple member-by-member copy operation, which is sufficient.

Writing your own copy constructor is necessary only in cases where each object has resources allocated for it (such as memory), requiring not a member-by-member copying procedure but deep copying, where each new instance of a class is allocated its own resources. But none of the examples in this book rely on deep copying.

Interlude

The Copy Constructor and References

One of the main reasons C++ needs to support references is so that you can write copy constructors. Consider, for example, what would happen if you declared a copy constructor this way:

```
Point(Point const src)
```

The compiler doesn't allow this, and a little reflection shows why. When an argument is passed to a function, a copy of that object must be placed on the stack (the area of memory used to store function arguments and addresses). But this would mean that for the copy constructor to work, it would have to make a copy of that same kind of object first; therefore, it would have to call itself! This would be an infinite regress.

So why not declare it this way?

```
Point(Point const *src)
```

There's nothing syntactically wrong with such a declaration and, in fact, it's a valid constructor. But it's not a *copy* constructor. This syntax indicates that a pointer, not an object, is the argument, so it would only work with pointers to objects, not with Point objects themselves.

Happily, the difficulty is not insurmountable because it's basically a problem in syntax.

Using a reference enables the function to work as a copy constructor. Syntactically, the argument is an object, not a pointer. However, because the *implementation* of the call most likely involves pointers under the cover, no infinite regress occurs.

```
Point(Point const &src)
```

A Constructor from String to Fract

Wouldn't it be nice to initialize a Fraction object from a string? For example:

```
Fraction a = "1/2", b = "1/3";
```

This is possible if we write a constructor that takes a **char*** string as its one argument. With this constructor, it becomes easier to initialize arrays of Fraction objects.

```
Fraction arr_of_fract[4] = {"1/2", "1/3", "3/4"};
```

Of course, it would be nice to do away with the quotation marks. But that won't do. Without the use of a **char*** string (necessitating the quotation marks), C++ would actually carry out the integer division 1/3 and then round it down to 0.

```
Fraction a = 1/3;       // This won't do what we want!
```

So, let's accept the quotation marks. First, we're going to need to access C-string functions, so that mandates an **include** directive.

```
#include <cstring>
```

Next, the constructor has to be declared within the Fraction class declaration.

```
    Fraction(char *s);
```

OK, that was easy. Writing the function definition is more involved, but don't worry; we'll deconstruct the code and see how it works.

```
Fraction::Fraction(char *s) {
    int n = 0;
    int d = 1;
    char *p1 = strtok(s, "/, ");
    char *p2 = strtok(NULL, "/, ");
    if (p1) {
        n = atoi(p1);
    }
    if (p2) {
        d = atoi(p2);
    }
    set(n, d);
}
```

The first thing this function does is declare two integer variables and give them reasonable defaults.

```
int n = 0;
int d = 1;
```

The default value of d (which will be assigned to the denominator) is 1. This enables the user of the class to initialize Fraction objects this way:

```
Fraction a = "5";   // Initialize a to 5/1.
```

The next two statements extract two substrings separated by the divide sign (/) or, optionally, a comma (,).

```
char *p1 = strtok(s, "/, ");
char *p2 = strtok(NULL, "/, ");
```

If the **strtok** function (explained in Chapter 7) can't get another substring from the input string, it returns a null pointer. Therefore, the code must test to see whether p1 and/or p2 are null pointers before passing them to the **atoi** function and finally calling **set**.

```
if (p1) {
     n = atoi(p1);  // Convert to int.
}
if (p2) {
     d = atoi(p2);  // Convert to int.
}
set(n, d);
```

Putting this code into the Fraction class and testing it is left as an exercise for the reader.

Chapter 11 *Summary*

Here are the main points of Chapter 11:

- A constructor is an initialization function for a class. It has this form:

```
class_name(argument_list)
```

- If a constructor is not inlined, the constructor's function definition has this form:

```
class_name::class_name(argument_list) {
    statements
}
```

- You can have any number of different constructors. They have the same function name (which is the name of the class). But each constructor must be uniquely identified by number or by type of argument.
- The default constructor is the constructor with no arguments at all. It has this declaration:

  ```
  class_name()
  ```

- The default constructor is called when an object is declared with no argument list. For example:

  ```
  Point a;
  ```

- If you declare no constructors, the compiler automatically supplies a default constructor for you. This automatic constructor does nothing; it is a no-op. However, if you write any constructors at all, the compiler does not supply a default constructor for you.
- So, to program defensively, you will usually want to write a default constructor. It can include zero statements, if you want. For example:

  ```
  Point a() {};
  ```

- In C++, a reference is a variable or argument declared with the ampersand (&). The result is (almost always) that a pointer is passed under the covers, but no pointer syntax is involved. The program appears to be passing a value, even though it's probably passing a pointer.
- A class's copy constructor is called whenever an object needs to be copied. This includes situations in which an object is passed to a function or the function returns an object as its return value.
- The copy constructor uses a reference argument, as well as the **const** keyword, which prevents changes to an argument. The copy constructor has this syntax:

  ```
  class_name(class_name const &source)
  ```

- If you don't write a copy constructor, the compiler supplies one for you. It carries out a simple member-by-member copy.

This page intentionally left blank

12 Two Complete OOP Examples

In the last two chapters, I laid out the basic syntax of class and object declarations. Now we're ready to apply object-oriented principles to programs that do something fun and useful.

First, I'll explore a Binary Tree example, which is one of the more intriguing and challenging topics in programming, and then I'll return to the Tower of Hanoi puzzle from Chapter 5, "Function: Many Are Called." The new version of the program uses character-based animation to show the puzzle solution in action.

But first, a few more preliminaries....

Dynamic Object Creation

Pointers have yet another use: constructing networks of objects. This is called *dynamic memory allocation* because it requests memory at run time, letting the program decide when to allocate new objects rather than having memory needs fixed for all time, before the program is run.

In C++, the easiest way to allocate memory during run time is to use the **new** keyword.

Keyword

ptr = **new** *type***;**

In this syntax, *type* can be either a built-in type such as **int** or **double**, or a user-defined type such as a class; and *ptr* is a pointer of matching type.

For example, assuming Fraction is a class that's already been declared, the following statement creates a Fraction object and returns a pointer to it:

```
Fraction *p = new Fraction;
```

The object itself has no name—remember that "Fraction" is the name of the class, not the object—and you might think that makes referring to the object difficult. But you can access the object all you want through the pointer. Here are some statements that use a pointer to manipulate an object:

```
(*p).set(10, 20);        // Set values 10, 20.
(*p).set(2, 27);         // Set values 2, 27
cout << (*p).get_num();  // Print the num value.
cout << (*p).get_den();  // Print the dent value.
```

This example—which I'm about to rewrite—uses the following syntax:

(*ptr).*member_name*

You might think that such syntax is extremely common, and you'd be right. It's so common that there is an operator that exists merely to represent this operation more succinctly. It saves only two keystrokes, but it makes programs more readable:

ptr->*member_name*

This means: dereference the *ptr* to get an object and then access the specified member of that object.

In the next few sections, you're going to see this operator (–>) used a great deal. For example, you can rewrite the statements shown earlier as:

```
p->set(10, 20);        // Set values 10, 20.
p->set(2, 27);         // Set values 2, 27.
cout << p->get_num();  // Print the num value.
cout << p->get_den();  // Print the dent value.
```

The **new** keyword has some variations. You can give an object an initial value by specifying arguments. For example:

```
Fraction *p = new Fraction(2, 3);
```

What this statement does is to pass along the arguments 2 and 3 to the matching constructor. If there's no such constructor, this is a syntax error.

Other Uses of new and delete

This is a brief detour, so if you're eager to see a sample app, skip ahead to the next section. But there is more to the **new** keyword, as well as to the **delete** keyword, which is often used in conjunction with **new**.

You can use **new** to create a whole series of data items—this can be done with data items of any valid type that's been defined, either a primitive or user-defined class. The following statements allocate 19 integers (**int**) and 50 Point objects:

```
int pInt = new int[10];     // Allocate 10 ints.
Point pPt = new Point[50];  // Allocate 50 Point objs.
```

There is a size dimension in each case; this dimension can either be a constant or a value computed at run time, such as a variable.

Having allocated memory items, you can then access them through the address returned by **new** and stored in the pointer, just as if all the items were part of an array. For example, the following statements initialize this data:

```
for (int i = 0; i < 10; ++i) {
     pInt[i] = i;
}

for (int i = 0; i < 50; ++i) {
     pPt[i].set(i, 2);
}
```

To avoid "memory leaks," it's a good idea to explicitly delete memory you've requested. When a C++ program terminates, all memory requested is released back to the system. However, there are some programs that run in the background or, at any rate, run for a long time. When such programs neglect to release allocated memory, they can cause memory leaks that eventually can slow down and even crash your system.

The **delete** keyword has two forms. Use the second form if you allocated multiple items as shown earlier in this section. The effect in each case is not to destroy the pointer *ptr* but to release memory previously allocated to this pointer.

Keyword

delete *ptr***;**
delete [] *ptr***;**

For example:

```
delete pNode;    // Delete a node
delete [] pInt;  // Delete all 10 ints.
```

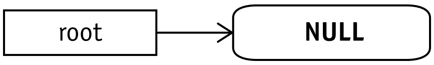
Blowin' in the Wind: A Binary Tree App

Okay, let's move onto a practical application: how about an app that takes a list of names and prints them out in alphabetical order? This is relevant to real-world situations in which you need to sort a list.

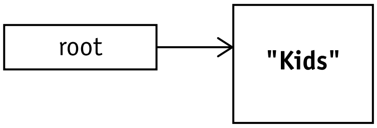
There are many ways to accomplish this goal, but for this chapter, I'm choosing something called an "ordered binary tree." Why it's called a tree, you'll see in a moment.

Note The C++ Standard Template Library (STL) implements its own forms of binary trees, in the <set> and <map> template classes. However, it's useful to program binary trees yourself to learn how they work.

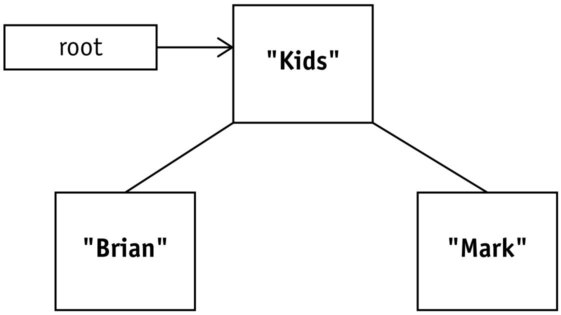
A binary tree starts with a pointer to a root node. If the tree is empty, the root pointer will have a null value.



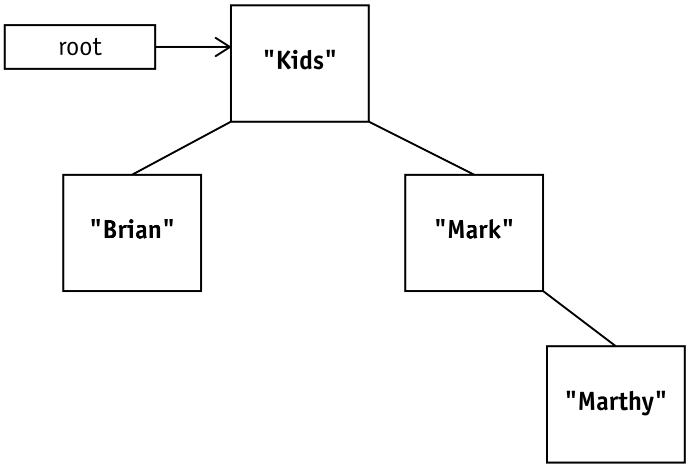
After we insert the first node, the tree looks like the diagram shown below. This creates a tree consisting of a single node, which of course is at the root. You can think of "Kids" as having two child nodes, each of which are NULL. But to keep the figure uncluttered, I've just assumed the NULLs are there.



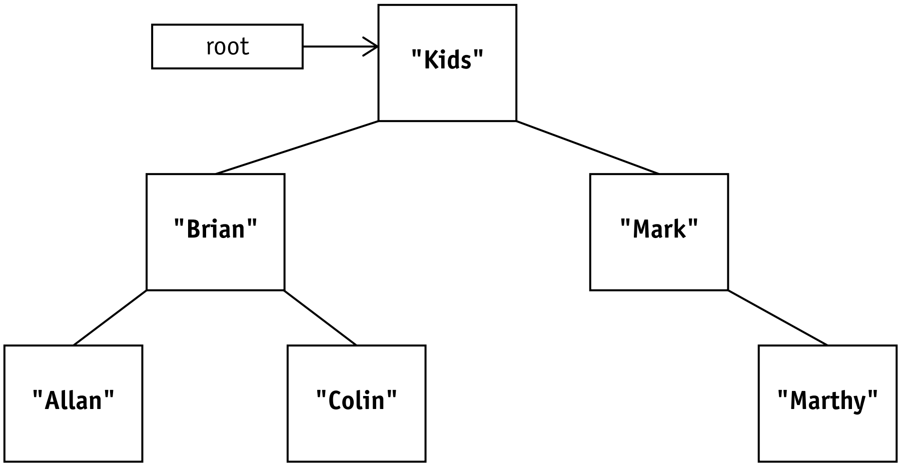
Let's add two more values: "Mark" and "Brian." Each new node is added into its proper place. If a new node has a value alphabetically earlier in the sorting order, it gets added as a left-side child of some existing node. If a new node is alphabetically later in the sorting order, it gets added as a right-side child. In the case of "Mark" and "Brian," one of these values is attached as a left node (lesser) and the other is attached as the right node. Here is the result:



Now we add a third fourth value: "Marthy." Where should this go in the tree? Both the "Brian" and "Mark" nodes have open spots to add children, but it makes sense to put "Marthy" on the far right because this string is later in alpha order than any of the nodes so far.



Finally, let's add nodes containing the strings "Allan" and "Colin." Can you see why they have to be added where they are, as child nodes of "Brian"? Remember the rules: 1) a child node can only be added in an open position, and 2) a "lesser" value, that is, a string earlier in alpha order than its parent, can be added as a left child, while a "greater" value, a string later in alpha order than its parent, can be added as a right child.



By now, you should be able to figure out for yourself where "Lisa" and "Zelda" would go. Every time a new node is inserted into the tree, there's an unambiguous place it should go. The tree is maintained so everything in a subtree to its left is "less" than everything in a subtree to its right.

You may also be able to guess how the tree can be printed out, producing all the names in alphabetical order. To me, this algorithm is amazing, because it does so much in just a few steps. This is a beautiful example of recursion.

```
To Print out a subtree whose root is pointed to by p:
If the node pointed to is NOT null,
    Print out the subtree on the left
    Print out the value of the current node
    Print out the subtree on the right
```

It may seem amazing this tiny algorithm can do so much. In addition, it works perfectly even if the tree grows to thousands or even millions of nodes in size. That's the power of recursion for you.

To create a name-sorting application, we need to design and code two classes: Bnode and Btree.

The Bnode Class

First, we need a class on which to model nodes. Nodes don't do much of anything. They are passive. But the constructor for this class turns out to be both convenient and an excellent way to prevent errors—and for that reason, it's useful to make a class for nodes.

Each node object needs three public members, containing its string value and two pointers to subtrees—one on the left and one on the right.

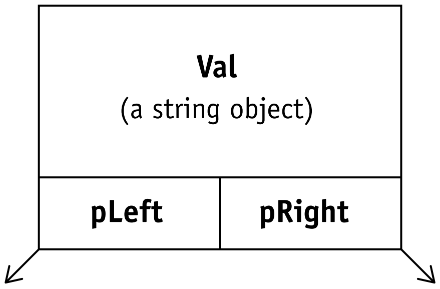
Here's the declaration of the Bnode class. As with all class declarations, its closing brace is followed by a semicolon.

```
class Bnode {
public:
     string  val;
     Bnode*  pLeft;
     Bnode*  pRight;
     Bnode(string s){val = s; pLeft = pRight = nullptr;}
};
```

A class cannot contain instances of itself. That would be like Bertrand Russell's paradox of the set that contains itself. Such a class would be infinitely large if it were legal (which it's not).

But pLeft and pRight are not really instances of Bnode. They are pointers to other objects of the same class. Such pointers make in-memory networks and trees possible. Think of it this way: a parent does not contain a child (except for nine months, of course!), but a parent can have a family tie to one or more children.

We can picture each instance of Bnode as having the following design: Each of the pointers, pLeft and pRight, can either have a null value or point to a child node. Either or both of the pointers may be null—a null value indicating there is currently no child on that side. If both pointers are null, the node object is a "leaf" or terminus that currently has no children.



The constructor is very helpful because of its convenience and error-prevention. There is no default constructor for this class, and therefore the user of this class cannot create a node without assigning a value:

```
Bnode my_node;   // ERROR! Not assigned a value!
```

Instead, the user of the class *must* initialize the node with some string value if he or she wants to create a node.

```
Bnode my_node("Emily");   // This is legal.
```

But the biggest advantage of this constructor is that it's impossible to create a node without initializing the two pointers to a null value (**nullptr**, or NULL if the **nullptr** keyword is not supported). The importance of this feature can't be overstated. If the pointers were permitted to be uninitialized, and therefore contain random "garbage" values, the consequences for the program could be catastrophic. That source of error has been eliminated.

Note ▶ The nullptr keyword has been supported beginning with the C++11 specification. If your **compiler** is more than a few years old, you may need to use NULL instead of **nullptr**.

C++14 ▶ Compilers that are C++11 and later provide support for in-class initialization. So, for example, when pLeft and pRight are declared in the private section of the Bnode declaration, they can be initialized with null pointers. Constructors, if any, can choose to override these settings, but otherwise, such initialization imposes default values. See page 271 in the previous chapter for more information.

The Btree Class

The Btree class (for "binary tree") is the other class needed in this program. Could the program be written as a series of separate functions and data structures, without being made a class? Yes, but writing Btree as a class is helpful in a number of ways.

First, the functions for this class are designed for use with class data and should not be used in any other context. The code and data are designed to work closely together, and OOP offers a good way to package them.

More importantly, access to data within the tree is controlled. It's impossible for users of the class to reach inside and tinker with private data. No direct access to any node is permitted. The user can't do something stupid like assign a bad value to a pointer. With a correctly written class, the class user has to do only two things: insert a name into the tree and print out the contents.

Here's an initial declaration of the Btree class. (We'll refine it in a moment.) Note how the location of the tree—its root—is kept private.

```
// INITIAL VERSION OF THE BTREE CLASS
class Btree {
public:
     Btree() {root = nullptr; }
     void insert(string s);
     void print();
private:
     Bnode* root;
};
```

Because the user of the class has no access to the root, he or she has no way to directly access any node at all, and this prevents him or her from doing something dangerous such as changing a pointer value. It's not that users want to cause the system failure (at least most do not!), but for many people it's too tempting to open up the internals of a data structure and go mucking about.

Here's the full declaration of the class, including helper functions; these functions, like the root variable, are private and not for outside use.

```
// FULL BTREE CLASS DECLARATION, W/ HELPER FUNCTIONS
class Btree {
public:
     Btree() {root = nullptr; }
     void insert(string s)
         {root = insert_at_sub(s, root);}
     void print() {print_sub(root);}
private:
     Bnode* root;
     Bnode* insert_at_sub(string s, Bnode* p);
     void   print_sub(Bnode* p);
};
```

To the class declaration, we add a couple of "helper" functions to carry out operations needed to support the public functions. These helper functions, insert_at_sub and print_sub, cannot be inlined because they are recursive. They have to be defined outside of the class declaration and therefore require the use of the "Btree::" prefix to clarify scope.

```
Bnode* Btree::insert_at_sub(string s, Bnode* p) {
    if (!p) {
        return new Bnode(s);
    } else if (s < p->val) {
        p->pLeft = insert_at_sub(s, p->pLeft);
    } else if (s > p->val) {
        p->pRight = insert_at_sub(s, p->pRight);
    }
    return p;
}

void Btree::print_sub(Bnode* p) {
    if (p) {
        print_sub(p->pLeft);
        cout << p->val << endl;
        print_sub(p->pRight);
    }
}
```

The second of the two function definitions, print_sub, is pure elegance. All it does is say "Print the tree on my left, then print my value, then print the tree on my right." Recursion makes this algorithm incredibly easy. Note the terminal condition: if a pointer to a child has a null value, the function just returns.

The other function, insert_at_sub, could have been written without recursion, but the function is easier to write this way. An *iterative* solution depends on loops instead of the function calling itself. Writing the iterative solution will be discussed in an upcoming exercise.

Like all recursive functions, insert_at_sub needs a terminating condition. It finds this condition when, after traversing the tree, it reaches a null pointer value. A new node should then be created.

```
return new Bnode(s);
```

The address of this new object is returned to the caller, where it's assigned, as appropriate, to pLeft, pRight, or the root pointer. If a null pointer has not been reached, the function just returns the pointer passed to it.

However, that means in the majority of cases—cases where no new node is created—nothing special is done with the return value, and therefore this approach is admittedly not optimal. It's also one reason why the iterative solution, while resulting in more lines of programming code, is more efficient.

Example 12.1. *Names in Alpha Order*

With the Bnode and Btree classes in place, it's easy to write a program that prompts for a series of strings and then prints them out in alphabetical order. In the following program listing, the three lines that refer to the binary-tree object are in bold.

alpha_tree.cpp

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

// INSERT BNODE AND BTREE DECLARATIONS
//  HERE, ALONG W/ BTREE FUNCTION CODE

int main()
{
    Btree my_tree;
    string sPrompt = "Enter a name (ENTER when done): ";
    string sInput = "";

    while (true ) {
        cout << sPrompt;
```

alpha_tree.cpp, cont.

```
            getline(cin, sInput);
            if (sInput.size() == 0) {
                break;
            }
            my_tree.insert(s);
        }
        cout << "Here are the names, in order." << endl;
        my_tree.print();
    }
```

Here's a sample session with this program. I've put user input in bold.

```
Enter a name (ENTER when done): John
Enter a name (ENTER when done): Paul
Enter a name (ENTER when done): George
Enter a name (ENTER when done): Ringo
Enter a name (ENTER when done): Brian
Enter a name (ENTER when done): Mick
Enter a name (ENTER when done): Elton
Enter a name (ENTER when done): Dylan
Enter a name (ENTER when done):
Here are the names, in order:
Brian
Dylan
Elton
George
John
Mick
Paul
Ringo
```

How It Works

This program creates one Btree object named my_tree. The Btree object, in turn, creates multiple Bnode objects, one for each name entered by the user, but these nodes are all hidden from everything but the Btree object itself.

There are many other ways to produce a sorted list of names. You could, for example, put all the names into an array and then sort the array, using techniques described in Chapter 6, "Arrays: All in a Row...," but a binary tree has special advantages.

For one thing, a binary tree can grow without limit, subject only to the amount of memory available. For another, in the case of extremely large data types, using a binary tree is potentially much faster than using an array. This is because access time in a tree grows *logarithmically*, meaning that it doesn't take much longer to find one element out of a million than it does one out of a thousand.

That assumes, however, that the tree remains relatively balanced, and there's no guarantee of that. Algorithms to keep a tree continually balanced exist, but they are relatively difficult and are outside the scope of this book, but you're welcome to research that subject on your own!

The heart of the program, in many ways, is the Btree::insert_at_sub function, which guarantees that strings are added to the tree in strict alphabetical order. The pseudocode version of this function can be summarized as follows:

```
To insert a string s into subtree pointed to by p:
If p is null,
    Create a new node and return a pointer to it
Else if s is "less than" string at this node
    Insert s into left subtree
Else if s is "greater than" string at this node
    Insert s into right subtree
Return p
```

There are a couple of subtleties to this procedure. If target string s is neither less than or greater (in alpha order) than the value of the current node, we've found a matching string and no further action should be taken. In that case, the function just returns without ever creating a new node.

Most of the time, the return value has little significance because the function usually just returns the pointer argument passed to it. However, when a new node is finally created, its address is passed back to its parent node, as appropriate, so that the new node is properly attached.

EXERCISES

Exercise 12.1.1. Write and test a get_size function for the Btree class. You should be able to do this by adding another private data member called nSize.

Exercise 12.1.2. Write and test a Btree function named size_of_subtree, which calculates the number of nodes at a subtree pointed to by p. When applied to the entire tree (by passing the root pointer), you should get the same answer as in Exercise 12.1.1. Test that theory to see if it's true.

Exercise 12.1.3. Write and test a find function for the Btree class. This function should take a string as input, and return a Boolean value (**true** or **false**) depending on whether or not that string is in the tree. Write a recursive solution.

Exercise 12.1.4. Write and test a find function as in Exercise 12.1.3, but this time use an iterative solution. Remember that an iterative solution is one that relies on loops and does not call itself.

Exercise 12.1.5. Write and test a get_first and a get_last function for Btree; these functions will return the alphabetically first or alphabetically last string in the tree. You can use either a recursive or iterative approach.

Exercise 12.1.6. Objects that persist for a long time but do not release unused memory can cause the problem of "memory leaks" in your computer, hogging resources and slowing down performance. So, when you're done with the binary tree, a good idea is to release it—including all the nodes in the tree. Write a function to do just that: remove every node. You will need to write a recursive function, to which you give the root address. (Hint: to release an object allocated with **new**, use the statement "delete p;" where p is the pointer to the object.)

Exercise 12.1.7. Write an iterative insert function, replacing the one shown in the example. (Hint: within the loop, first test to see if the target string is, in alpha order, less than or greater than the string at the current node. Then check to see if the relevant child node—pLeft or pRight—is or is not null.

Interlude

Recursion versus Iteration Compared

The recursive solution for deleting a list is tempting—the code for that approach is shorter. But is it more efficient? The truth is, as much as I pushed recursion in Chapter 4, "The Handy, All-Purpose 'for' Statement," if you have a choice between an iterative or a recursive solution, the iterative solution is usually more efficient. You may prefer the recursive solution because you have to write fewer lines of code, but it's useful to consider what happens with recursion.

A recursive solution causes a function call at each level. In the binary-tree example, there would be one additional function call for each node. And, if the tree were a million nodes deep, the system would have to execute a million function calls!

▼ *continued on next page*

Interlude

▼ continued

When we get into numbers like that, function-call overhead gets expensive. The program traverses the list and puts the addresses of each node on the special C++ stack segment—that's the area in memory reserved for holding arguments and local variables. The functions then return, popping addresses off the stack and deleting nodes, in reverse order. For example:

```
0x1000ff40
0x1000ff30
0x1000ff20
0x1000ff10
Etc.
```

In contrast, the iterative solution goes through the list and deletes the nodes in the order it finds them. In contrast, the recursive solution is a "breadcrumb" solution that leaves a trail of breadcrumbs as it traverses the list (metaphorically speaking) and then goes back and picks up the crumbs, deleting nodes in the process. But that's inefficient.

So, recursion is not only more elegant, but for a certain class of problem, it is the only practical solution. The Tower of Hanoi puzzle, which we'll return to shortly, would be vastly more difficult to solve without recursion. And there's an even bigger class of recursive problem: the C++ code that built the compiler you're using now involves a lot of recursive function calls, without which it would've been far more difficult to write.

Tower of Hanoi, Animated

Chapter 5 showed how to solve the Tower of Hanoi puzzle by printing instructions on how to move the rings. But wouldn't it be more fun to *watch* the rings move around?

To do this—to show the animated version of the solution—is going to require more programming. How do we break down the problem?

One way to start is to realize that the puzzle consists of three rings (or "stacks"). We can design a general class called Cstack and use it to create three objects. Each Cstack object will obey the following ideas:

- The highest level in each stack will be 0. The level below it is 1, the level below that is 2, and so on.
- For each Cstack object, the variable **tos** will represent **Top of Stack**, or rather, one level above the highest ring. So, tos will range in value from –1 (full stack) to n – 1 (empty stack).

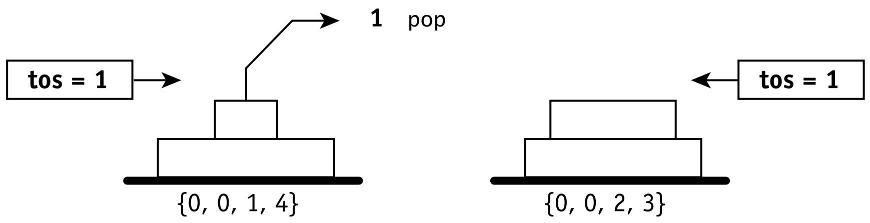
We need to track the state of each of the three stacks. Let's say that each of the rings has a number that indicates its relative size, 1 being the smallest and 0 indicating an empty space. So, for example, if there are four rings total, then the following is true:

- An empty stack has array values {0, 0, 0, 0}; tos = 3. This means that index number 3, corresponding to the fourth position, is one level "above" the top of the stack.
- A stack with just one ring, the third largest, has {0, 0, 0, 3}; tos = 2. This means that index number 2, corresponding to the third position, is one level "above" the top of the stack.
- If the next smallest ring is pushed onto that stack, it has {0, 0, 2, 3}; tos = 1, indicating that index number 1, which corresponds to the second position, is one level above the top of the stack.
- If the smallest ring is pushed onto that stack, it has {0, 1, 2, 3}; tos = 0, indicating the first position in the array—again, one level above the top of the stack.
- A completely full stack has array values {1, 2, 3, 4}; tos = –1.

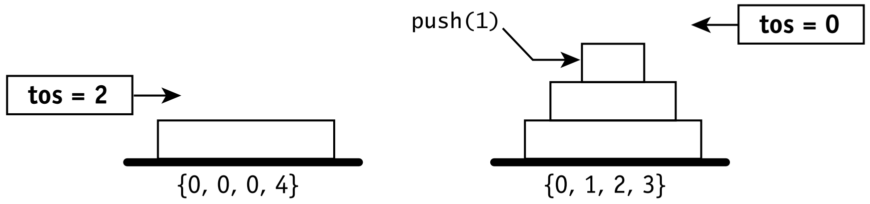
The top-of-stack value, tos, is an array index one position "higher" than the top ring. Before a ring is popped, tos for the stack increases, pointing to a "lower" position. After a ring is pushed onto a stack, its tos value decreases, indicating a "higher" position.

This may seem counterintuitive, but when the stacks need to be displayed, this approach is easiest. The first array position in each stack corresponds to the physically highest position. Gravity pulls the rings down, so the top position is usually empty space (0).

1. State of stacks just prior to `n = stacks[0].pop();`



2. State of stacks just after `stacks[1].push(n);`

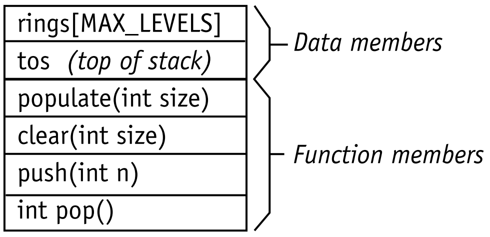


After Mystack Class Design

To store the ring positions for the three stacks (pole positions), we need to create a data structure called a stack. Earlier, I discussed a special area of memory called "*the* stack," which stores arguments and local variables. But that's not the stack I'm talking about here.

This example calls for a special, customized stack class. Unlike most stacks, this class has to allow for empty space at the top, letting the rings fall to the bottom. The first few places will often contain 0. When the program displays a picture of the three stacks, you'll see why this approach is necessary.

We need to design our own customized "stack" class—let's call it Cstack—from which three objects will be created. The design of the class is:



Cstack class

The rings array in each object contains most of the data. It stores rings of different sizes by using a series of integers, 1 indicating the smallest ring and 0 indicating empty space. (So, {1, 2, 3} indicates the three smallest rings, and {0, 0, 2} indicates the second smallest ring, with two empty spaces above it.) The tos member signifies the top-of-the-stack position.

The previous section illustrated the use of the push and pop functions. These are common operations on any kind of stack.

The populate and clear functions perform initialization that's useful if you want the program to be able to reset and start over.

Using the Cstack Class

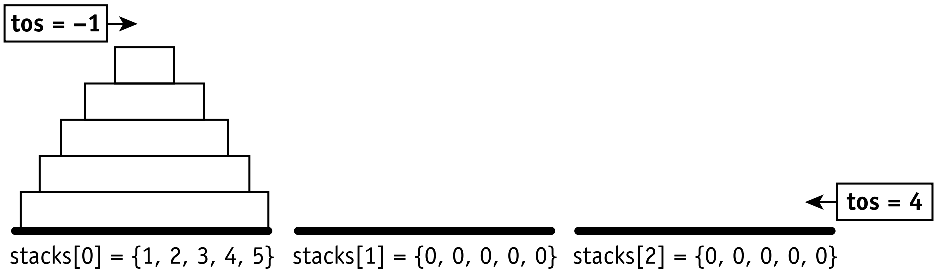
Once the Cstack class is declared, we use it by creating three of these customized stack objects and initializing them. The three objects are placed in an array of these objects:

```
    Cstack  stacks[3];
```

At the beginning of each animation cycle, the following happens:

- `stacks[0].populate( )` is called to fill the first stack.
- `stacks[1].clear( )` and `stacks[2].clear( )` are called to set the other stacks to the empty state.

This approach provides a lot of flexibility. As long as the size doesn't exceed MAX_LEVELS (a constant declared at the beginning of the program), the animation can be restarted with any size requested by the user. For example, if the stack size is 5, the populate and clear functions are called to fill the first stack with values 1 to 5 and to leave the other two stacks empty (0 value) in the first five positions.


tos = –1
tos = 4
stacks[0] = {1, 2, 3, 4, 5}
stacks[1] = {0, 0, 0, 0, 0}
stacks[2] = {0, 0, 0, 0, 0}

With this array of three objects in place (each of which contains its own array), we can now move rings between the three positions by calling pop and push as appropriate and then, after each move, printing a picture that represents the new state.

Example 12.2. *Animated Tower*

With the Cstack class design in place, we can now write the animated version of the Tower of Hanoi. This program builds on the Tower of Hanoi example in Chapter 5, using the recursive logic there to solve the problem of moving all the rings from pole 1 to pole 3.

Remember the two constraints: you can move only one ring at a time, and a larger ring can never be placed on a smaller ring.

This version solves the problem and displays the state of the three poles (that is, the three stacks of rings) after each and every move.

tower_visi.cpp

```
#include <iostream>

using namespace std;

#define MAX_LEVELS  10

// Declare three pole positions, or rather, stacks.
// Each stack is an object containing ring values.
// stacks[3] is an array three of these objects.

class Cstack {
   public:
      int rings[MAX_LEVELS];  // Array of ring values.
      int tos;                // Top-of-stack index.
      void populate(int size);  // Initialize stack.
      void clear(int size);     // Clear the stack.
      void push(int n);
      int pop(void);
} stacks[3];

void Cstack::populate(int size) {
     for (int i = 0; i < size; i++) {
          rings[i] = i + 1;
     }
     tos = -1;
}

void Cstack::clear(int size) {
     for (int i = 0; i < size; i++) {
          rings[i] = 0;
     }
     tos = size - 1;
}

void Cstack::push(int n) {
     rings[tos--] = n;
}

int Cstack::pop(void) {
     int n = rings[++tos];
```

tower_visi.cpp, cont.

```
        rings[tos] = 0;
        return n;
   }

   void move_stacks(int src, int dest, int other, int n);
   void move_a_ring(int source, int dest);
   void print_stacks(void);
   void pr_chars(int ch, int n);

   int stack_size = 7;

   int main() {
       stacks[0].populate(stack_size);
       stacks[1].clear(stack_size);
       stacks[2].clear(stack_size);
       print_stacks();
       move_stacks(stack_size, 0, 2, 1);
       return 0;
   }

   // Move stacks: solve problem recursively...
   // move N stacks by assuming problem solved for N-1.
   // src = source stack, dest = destination stack.
   //
   void move_stacks(int n, int src, int dest, int other){
        if (n == 1) {
             move_a_ring(src, dest);
        } else {
             move_stacks(n-1, src, other, dest);
             move_a_ring(src, dest);
             move_stacks(n-1, other, dest, src);
        }
   }

   // Move a Ring: Pop off a ring from source (src) stack,
   // place it on destination stack, and print new state.
   //
   void move_a_ring(int source, int dest) {
        int n = stacks[source].pop(); // Pop off source.
        stacks[dest].push(n);         // Push onto dest.
        print_stacks();               // Show new state.
   }
```

▼ *continued on next page*

tower_visi.cpp, cont.

```
// Print Stacks: For each physical level, print the
// ring for each of the three stacks.
//
void print_stacks(void) {
   int n = 0;
   for (int i = 0; i < stack_size; i++) {
      for (int j = 0; j < 3; j++) {
         n = stacks[j].rings[i];
         pr_chars(' ', 12 - n);
         pr_chars('*', 2 * n);
         pr_chars(' ', 12 - n);
      }
      cout << endl;
   }
   system("PAUSE"); // A pause is needed here; use
}                   //  another method if you need to.

void pr_chars(int ch, int n) {
     for (int i = 0; i < n; i++) {
          cout << (char) ch;
     }
}
```

How It Works

The core of this program is a recursive function that works just like the Tower of Hanoi solution in Chapter 5; see that chapter for an explanation of the logic. Much of the time this function, move_stacks, calls itself. Only some of its action involves actually moving a ring.

But the difference comes when it's time to actually move a ring: instead of just printing a message, this version calls a new function, `move_a_ring`, that takes care of the details of moving a single ring from one stack to another and displaying the result.

```
// Move stacks: solve problem recursively...
// move N stacks by assuming problem solved for N-1.
// src = source stack, dest = destination stack.
//
void move_stacks(int n, int src, int dest, int other){
     if (n == 1) {
          move_a_ring(src, dest);
```

```
        } else {
             move_stacks(n-1, src, other, dest);
             move_a_ring(source, dest);
             move_stacks(n-1, other, dest, src);
        }
   }
```

Now, how exactly do we move a single ring from one position to another?

There was no way to do this in the application in Chapter 5, which just printed a message. But now, since we have three stack objects that reflect the current state, we manipulate that state by doing the following:

1 Pop the top ring off the source stack and remember its size as n.

2 Push a ring of size n onto the destination stack.

3 Print the new state.

That's exactly what the move_a_ring function does in three simple statements:

```
// Move a Ring: Pop off a ring from source stack,
// place it on destination stack, and print new state.
//
void move_a_ring(int source, int dest) {
     int n = stacks[source].pop(); // Pop off source.
     stacks[dest].push(n);         // Push onto dest.
     print_stacks();               // Show new state.
}
```

The pop and push functions are member functions defined for the class. These are functions that use each object's top-of-stack indicator, tos, to get the top ring from the stack (pop) or to put a new ring onto the top of the stack (push).

Recall that member functions are defined inside a class (Cstack in this case) and are actually called through objects (stacks[ ]).

```
void Cstack::push(int n) {
     rings[tos--] = n;
}

int Cstack::pop(void) {
     int n = rings[++tos];
     rings[tos] = 0;
     return n;
}
```

The pop function gets the size number of the ring at the top of the stack; replaces that ring with empty space, 0, thus removing that ring from the stack; and finally returns n, the saved size number. That number, returned by this function, is then given as input to the push function so that it puts a ring of the correct size onto another stack.

Finally, the program uses print_stacks and a convenient support function, pr_chars, to print the current state.

You should now be able to see why the program uses 0 to indicate empty space and why 0s must be placed "above" the positive ring values. That is, if a stack is less than full, then there will be one or more 0s in the topmost (early) positions. Where there is a 0 at a particular level, meaning no ring is there, the program prints blank spaces. This arrangement reflects the physics of the simulation: rings fall to the bottom!

In programmatic terms, the function accesses the rings[] array within each object, getting the value for an entire physical level before moving on to the next level. If the ring value is 0 for a particular position, nothing is printed but blank spaces. (This reflects the empty space atop a less-than-full stack.) If the ring value is greater than 0, the program prints two asterisks (*) times the size of the ring and prints spaces around it. For example, if the ring size is 3, the program prints six asterisks.

```
// Print Stacks: For each physical level, print the
// ring for each of the three stacks.
//
void print_stacks(void) {
   int n = 0;
   for (int i = 0; i < stack_size; i++) {
      for (int j = 0; j < 3; j++) {
         n = stacks[j].rings[i];
         pr_chars(' ', 12 - n);
         pr_chars('*', 2 * n);
         pr_chars(' ', 12 - n);
      }
      cout << endl;
   }
}
```

The pr_chars function is just a handy way of printing repeated characters.

```
void pr_chars(int ch, int n) {
     for (int i = 0; i < n; i++)
          cout << (char) ch;
}
```

Finally, when this program runs, you probably don't want all the output to be produced without pause because it would then produce several screen's worth of characters. Therefore, you need a way to stop after each iteration and ask the end user to continue. On Windows-based systems, the system("PAUSE") command is ideal for this purpose.

```
system("PAUSE");
```

But if your system does not support this command—or you want to write something more portable—you can, instead, prompt the user to continue by using the techniques introduced in Chapter 8, "Strings: Analyzing the Text."

```
#include <string>  // Put at top of program
. . .
string dummy;
cout << "Press ENTER to continue.";
getline(cin, dummy);
```

Exercises

EXERCISES

Exercise 12.2.1. Prompt the user for how many rings to use in the starting position. This number should be no more than MAX_LEVELS (which we set to 10, largely because of screen-space considerations). Then repeat; if the user enters 0, then end the program; otherwise, continue with a new session. With this version, the use of the populate and clear member functions becomes evident. You need to be able to reset the initial state.

Exercise 12.2.2. Instead of implementing rings as an array inside each object, implement it as a pointer of type **int***. Then, use **new**, within the populate and clear member functions, to allocate a series of integers. Can you use the **delete** keyword to efficiently prevent memory leaks in this situation?

Chapter 12 *Summary*

Here are the main points of Chapter 12:

- Some C++ code deals heavily with pointers to objects. With such pointers, it's convenient to use the indirection operator –> to access a member of the object pointed to. For example:

```
// Get the num member of object pointed to.
int n = pFraction->num;
```

```
// Call the set function for object pointed to.
pFraction->set(0, 1);
```

- The use of dynamic memory allocation, along with pointers, to objects makes it possible to create complex structures in memory such as linked lists and binary trees. These can be as simple or complex as you choose.
- Use the **new** keyword to dynamically allocate an object at runtime.

```
Node *pNode = new Node;
```

- When you create your own lists and trees in memory, it becomes important to prevent memory leaks by deleting each individual object as soon as you know you no longer need it. Programs that fail to deal with memory leaks can cause your computer to prematurely run low on memory and need to be rebooted.
- Use the **delete** keyword to release objects and free up memory.

```
delete p;     // p points to an object
delete[] p;   // p points to an array of objects.
```

- You can create arrays of objects (instances of a class), just as you can create arrays of other kinds of data.

```
class Cstack {
   ...
} stacks[3];
```

- Sometimes an application has so much output that you need a way to pause the program and prompt the end user before continuing. The system("PAUSE"); statement is ideal for this purpose if your system supports it. But if your system does not, or if you want to write more portable code, you can instead prompt for a string object as described in Chapter 8.

```
string dummy;
cout << "Press ENTER to continue.";
getline(cin, dummy);
```

- Recursion is sometimes the only practical way to solve a problem, as in the Tower of Hanoi puzzle. But otherwise, if there is both an iterative (loop-based) and recursive solution, the iterative version is almost always more efficient.

13 Easy Programming with STL

One of the best things about C++ is the availability of the Standard Template Library (STL), which now comes with most compilers.

A template is a generalized data type you can use to create sophisticated containers. For example, the list template enables you to build linked lists of integers, floating-point numbers, or even your own kind of objects.

Don't worry if this sounds new or exotic. The STL is an amazing resource that solves many common programming problems. The general philosophy—as with functions, classes, and objects—is: once a programming problem is solved, why should anyone have to solve it again?

These days, the great majority of C++ compilers widely used now provide full support for STL. If your compiler is C++14 (or even C++11) compliant, it should definitely support it.

Introducing the List Template

STL—remember, that's the Standard Template Library—provides broad support for collections, or containers, built on top of other types.

You specify an underlying type and STL builds a sophisticated container around this type. For example:

```
list<int>        iList;        // List of integers
list<string>     strList;      // List of strings
list<Fraction>   bunchOFract;  // List of Fractions
```

There is no limit to the different kinds of lists you can create this way. The base type, as you see in this example, can be a primitive type or a class you've written (such as Fraction).

Using the **list** template creates a linked list that makes insertions and deletions particularly efficient. The STL supports other kinds of generalized data structures such as **vector**, an array that grows without limit, and **set** and **map**, which are built on top of binary trees.

Note All the STL names are part of the **std** namespace, which means you must either use the characters **std::** before each and every STL name, such as "stack" or "list," or just keep **using namespace std;** in your programs, as I have been suggesting in every example.

Interlude

Writing Templates in C++

The earliest versions of C++ contained no support for writing templates at all. But within a few years of C++'s first appearance, programmers (especially professional programmers) were clamoring for template support.

The template technology is still one of the most advanced realizations of the philosophy of reusable code, which says once a problem has been solved, it shouldn't have to be solved again.

Templates take this idea to the level of general algorithms. Once a mechanism has been created to contain integers, for example, it should be possible to reuse the same code with other data types: **double**, for example, or strings, or objects of any class. It's as though someone took a set of container classes and related functions and did a global search-and-replace operation, replacing all instances of **int** with some other type.

C++ provides strong support for writing your own template classes and template functions. For example, you can use the template keyword to declare a generalized container class called "pair":

```
template class<T>
class pair {
public:
    T first, last;

};
```

Given this declaration, you can then declare any number of "pair" container classes:

```
pair<int>     intPair;
pair<double>  floatPair;
pair<string>  full_name;

intPair.first = 12;
```

Interlude

▼ continued

However, for the most part, the subject of writing your own templates is outside the scope of this book, which is intended as a general introduction to how to write programs in C++ and how to think like a C++ programmer. Writing your own templates is a subject that, if covered in full detail, could easily add a few hundred pages.

Template writing is an advanced but fascinating topic. A number of excellent advanced texts are devoted exclusively to that subject.

But, although the subject of writing your own templates is outside the scope of this book, I encourage even relatively new C++ programmers to take advantage of the Standard Template Library as soon as they understand classes and pointers. STL classes are amazing time-savers and are easy to use. Someone else has done the work and you take advantage of that work.

Creating and Using a List Class

Before using the list template, you need to turn on support for it by using an **#include** directive.

```
#include <list>
using namespace std;
```

Then you can create your own linked-list classes. The syntax for declaring an STL list class is as follows:

Keyword

```
list<type>  list_name;
```

If you do not include the "using namespace" statement, items from the STL must be qualified with the **std::** prefix, as is true for many other objects and the templates in the standard library, but "using namespace std" takes care of that problem. To use the template without a "using namespace" statement, use this syntax:

```
std::list<type>  list_name;
```

Here are some more examples that are really easy to use:

```
#include <list>
using namespace std;
...
list<int>     list_of_ints;
list<int>     another_list;
list<double>  list_of_floatingpt;
list<Point>   list_of_pts;
list<string>  LS;
```

Once you've created a list, it starts out empty and you need to add elements to it. You can do this with the **push_back** member function. This adds elements to the end (that is, the back) of the list. For example:

```
list<string>  LS;

LS.push_back("Able");
LS.push_back("Baker");
LS.push_back("Charlie");
```

You can also use the **push_front** member function, which adds elements to the front of the list. If you think about it, the effect is the same except that it ends up adding the strings in reverse order.

```
LS.push_front("Able");
LS.push_front("Baker");
LS.push_front("Charlie");
```

For a numeric list, of course, you could add numeric elements.

```
list<int>  list_of_ints;

list_of_ints.push_back(100);
```

As you can see, we can create a linked list of any base type and add data to it. To do much more than this, you usually need *iterators*.

C++14 ▶ The following paragraph applies to C++14 compilers only. (Actually, the feature was introduced in C++11 but it took time for some vendors, including Microsoft, to support it.)

With C++14-compliant compilers, you can initialize most STL containers, including **list** containers, by using a comma-separated list, just as you can with arrays. For example:

```
list<int>  iList = {1, 2, 3, 4, 5];
```

Creating and Using Iterators

A number of templates in STL use iterators, which are devices for stepping through a list one element at a time—that is, *iterating* through it.

Iterators look and feel a great deal like pointers, especially in their use of ++, −−, and * operators, even though there are differences. You declare an iterator this way:

```
list<type>::iterator  iterator_name;
```

For example, to declare a list and an iterator for it, you could use these statements:

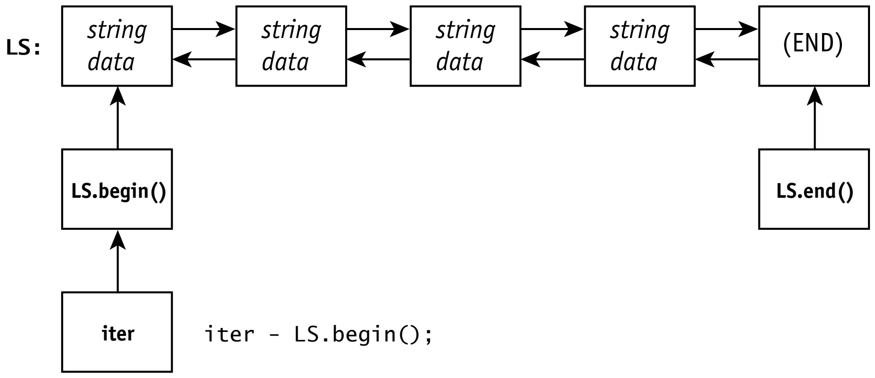
```
list<string>            LS;
list<string>::iterator  iter;
```

Now iter can be used to iterate through the list LS, since their underlying types (**string**) are compatible.

STL lists have **begin** and **end** functions that return an iterator to the beginning and the end of the list, respectively. The following statement uses **begin** to initialize an iterator:

```
list<string>::iterator iter = LS.begin();
```

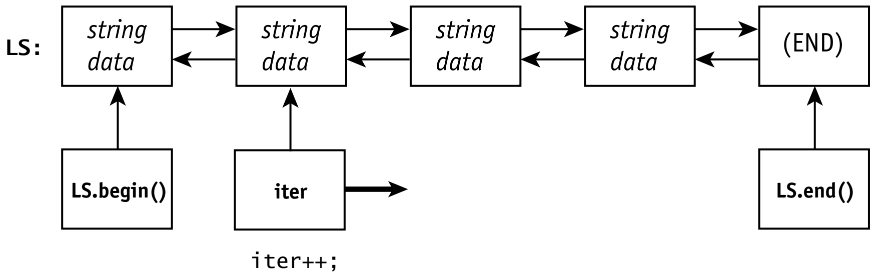
For a string list LS with four elements, you can visualize the operation this way:



Properly initialized, iter can now be used almost like a pointer. You can make iter point to the next item by using the increment operator.

```
++iter;   // Advance one element in the list.
```

When the iterator is incremented, it steps through the list, much as a pointer does within an array.



To access the data the iterator points to, apply the indirection operator (*) just as you would with a pointer.

```
cout << *iter << endl;    // Print string pointed to.
```

Combining these elements, you can write a loop that prints all the elements of the list, one to a line. This works because the **end** member function produces an iterator that points to *the position just after the last element*, not the last element itself. Therefore, if we are at the end position, we've processed all the elements. We're done!

Conversely, if we haven't reached `LS.end()`, we aren't done. That makes the loop condition easy to write.

```
iter = LS.begin();                // Start at beginning.
while (iter != LS.end()){         // While not done,
     cout << *iter << endl;       //   Print string
     ++iter;                      //   Advance to next.
}
```

This can be written more compactly with a **for** loop.

```
for (iter = LS.begin();  iter != LS.end(); ++iter) {
     cout << *iter << endl;
}
```

C++11/C++14 Only: For Each

With the ranged-based **for** syntax introduced in Chapter 10, it's even easier to print out all the items in a list. You don't even need to use an iterator.

C++14 ▶ The following paragraph describes a feature first introduced in C++11 that should be implemented by all C++14 compilers. However, if your compiler is more than a few years old, this feature might not be supported.

Here's how you print out a list by using range-based **for**. Chapter 17, "New Features of C++14," provides more details on how this works. The following code will work with *any* STL container (including any **list** container); only the name "LS" will change from one case to the next!

```
for (auto x : LS) {
    cout << x << endl;

}
```

Interlude

Pointers versus Iterators

By now, you can see why I stated that iterators look a lot like pointers. The designers of STL classes deliberately made iterators look and feel like pointers in order to work smoothly with the rest of the C++ language. Reusing prefix and postfix increment (++) is convenient—they do what you'd expect them to do—as is using the indirection operator (*). And all this is supported by C++'s operator-overloading syntax.

But don't confuse iterators with true pointers, or what we might call "raw" pointers. The latter offer no protection against invalid memory access, so you must be more careful with pointers.

Iterators are safe and designed to be that way. The program can attempt to increment an iterator off the end of a cliff, so to speak (by iterating past the end of a container or list), but if it does so, nothing drastic happens. The iterator no longer accesses data inside the container. An out-of-control pointer can overwrite and corrupt memory all over the system, but an iterator on the loose can't touch anything it shouldn't.

Example 13.1. *STL Ordered List*

We now have nearly all the knowledge of iterator and list syntax to write an ordered list program. When you see how short this is, you'll understand why programmers love STL.

We need to add just one piece to the puzzle, however. STL list classes come with a built-in sort function, among other things.

```
LS.sort();      // Sort the list alphabetically.
```

Note ▶ To support the **sort** function as well as other member functions of **list**, the underlying data type of the list must define reasonable behaviors for the less-than operator (<) as well as the assignment and test-for equality operators (= and ==). The string class, of course, already defines these behaviors. If these operators are not defined, then some **list** member functions may not be supported.

Here's the complete program:

alphalist2.cpp

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

int main()
{
    string s;
    list<string> LS;
    list<string>::iterator iter;

    while (true) {
          cout << "Enter string (ENTER to exit): ";
          getline(cin, s);
          if (s.size() == 0) {
               break;
          }
          LS.push_back(s);
    }
    LS.sort();      // Sort, and then print elements.

    for (iter = LS.begin(); iter != LS.end(); iter++) {
        cout << *iter << endl;

    }
    return 0;
}
```

How It Works

This is a short program considering all it does. It lets the user enter any number of strings of any size (subject only to the physical limits of the system itself). After the user is done, the program prints the strings all in alphabetical order. For example, if I enter the following strings

```
John
Paul
George
```

```
Ringo
Brian Epstein
```

the program prints the names in this order:

```
Brian Epstein
George
John
Paul
Ringo
```

Most of this program logic you've seen before. Half the statements in **main** do nothing more than prompt the user for input and then add a string to the end of the list by using `LS.push_back()`. As usual, if the user just presses ENTER, resulting in a string of zero length, that is taken as a convenient "I'm done" signal.

```
while (true) {
      cout << "Enter string (ENTER to exit): ";
      getline(cin, s);
      if (s.size() == 0) {
           break;
      }
      LS.push_back(s);
}
```

The real power of the class is revealed by one call to **sort**, a power member function.

```
LS.sort();
```

Finally, the iterator (iter) is used to print all the members.

STL iteration makes a "print all members" function easy to write. In particular, when the iteration reaches `LS.end()`, that means the iteration has moved *one past the last element in the list*, and so the work is done.

Conversely, a condition of `iter != LS.end()` means the list has not been entirely processed, and so work should continue.

```
for (iter = LS.begin(); iter != LS.end(); ++iter) {
    cout << *iter << endl;
}
```

Variation

A Continually Sorted List

The only problem with the approach in the previous section is that it doesn't sort until all the desired elements have been inserted in the list. That's fine for small applications, where you'll never notice the difference, but for exceptionally long lists (thousands or even millions of elements in length), the sorting time may be noticeable.

A better solution for large databases is to maintain data so that it is continually sorted. Each new element is added into its proper place in the ordering. This was true of the binary tree created in the last chapter.

Maintaining a list that's continually sorted is not difficult. Instead of using this statement to add a string

```
LS.push_back(s);
```

use the following statements every time you add an element; these statements first determine the alphabetically correct position. The **insert** then function inserts a new element (in this case, a string) just before the element pointed to by the iterator.

```
for(iter = LS.begin(); iter != LS.end() && s > *iter;) {
     ++iter;
}
SL.insert(iter, s);
```

How can it be this easy? One reason is that, once again, `iter != LS.end()` is such a convenient test condition. Because `LS.end()` corresponds to *one position past the last element*, it enables the loop to test every element, from beginning to end, inclusive. The last element doesn't have to be treated as a special case.

Another thing that makes this loop easy to write is that the STL **insert** function is robust; it behaves well in potentially bad situations, further removing the need to deal with special cases. Consider what happens if the list is empty. In that case, the **insert** function just adds s as the first element.

Or what if the iterator, iter, gets to the very end without finding an insertion point? In that case, the **insert** function does exactly what you'd want: It adds s to the end of the list, inserting it just before the "end"—meaning just after the last element.

But even such an ordered list may not be an ideal solution. For exceptionally large data sets, the delay experienced while the program searches through a ten-million-element list may be unacceptable. Let's say that it searches through, *on average*, five million elements to find a correct insertion point. That's a costly operation. In contrast, the binary-tree example in Chapter 12 theoretically enables near-instantaneous access times. The Btree class described in that chapter created a primitive binary tree. More sophisticated binary trees are provided in the STL through the **map** and **set** templates. (Still, I contend it's a lot of fun to try writing your own binary tree!)

EXERCISES

Exercise 13.1.1. Revise Example 13.1 so that it keeps a continually sorted list (as just shown).

Exercise 13.1.2. Revise Example 13.1 so that it keeps a continually sorted list, but keeps it in reverse order.

Exercise 13.1.3. Revise the example so that it reports the size of the list. You can write code to count the number of insertions, or you can simply call the template's size function, which has the syntax *list*.**size**().

Exercise 13.1.4. Using the list template, write a program that takes any number of floating-point amounts as inputs. Add each to the list and then report the following information by iterating through the list: 1) lowest number, 2) highest number, 3) total, and 4) average. Is it possible to produce this information without a list or array? Why might you want to use a list anyway?

Designing an RPN Calculator

No, this isn't a Registered Practicing Nurses calculator; it's the amazing Reverse Polish Notation (RPN) calculator, which can take an input line of any complexity, analyze it, and perform all the calculations specified.

This may sound daunting, and to be honest, it's a project usually reserved for serious computer-science majors at large colleges and universities. But the STL, along with the **strtok** function from the standard C++ library, does most of the work for you.

The beauty of RPN is that it specifies mathematical and logical expressions unambiguously, removing the need for parentheses. As a grammar, it has just two rules:

expression → *numeric-literal*

expression → *expression expression operator*

This notation indicates that every expression to be evaluated is either a simple number (that's easy enough) or two expressions followed by an operator. Do you see how recursive this is? Smaller expressions can be recombined inside larger ones, to any level of complexity.

If you don't get this yet, stay with me just a bit. Most obviously, RPN notation can evaluate something like this:

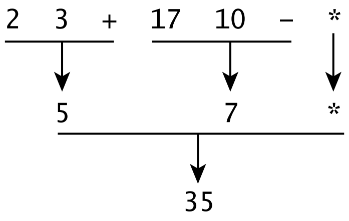
```
2 3 +
```

This means "Add 2 and 3." The result is 5. This works because *expression expression operator* is valid; 2 and 3, each being numeric literals, are valid expressions and they are followed by an operator (+). So far, so good. Now consider this more complex expression:

```
2 3 + 17 10 - *
```

When you understand RPN, this is clear. An expression can be made up of any two operands followed by an operator. The critical point here is that the operands are *themselves* expressions. Constructing expressions within expressions within expressions can go on for as long as you like.

In effect, the operator that is closest to the operands has precedence. 2 3 + is a valid expression, but so is 17 10 –. These two expressions are then followed by multiplication (*), forming the one large expression.



The result to be calculated here is 35. The RPN expression is equivalent to the following input line using standard notation (also called *infix* notation):

```
(2 + 3) * (17 - 10)
```

A disadvantage of standard arithmetic notation is that it is heavily dependent on parentheses, which are unnecessary with RPN.

By the way, to complete the syntax, we ought to list the operators supported.

operator → +

operator → *

operator → –

operator → /

This just means that the *operator* can be +, *, –, or /.

Or, we can express this syntax in one line with "OR." Note that OR is not in bold here, which means the "OR" is not intended literally.

operator → + OR * OR – OR /

Here are some more examples of Reverse Polish Notation and what each means in terms of standard arithmetic notation:

```
2 10 5 4 - / +        // ==>  2 + (10 / (5 - 4))
1 2 3 * * 10 9 - +    // ==>  (1 * (2 * 3)) + (10 - 9)
5 3 - 15 *            // ==>  (5 - 3) * 15
```

Interlude

A Brief History of Polish Notation

Polish Notation was invented by a distinguished philosopher, professor, and logician named Jan Łukasiewicz. Although little known to the general public in most countries, he made significant contributions to early twentieth-century systems of axiomatic logic.

In 1920, Professor Łukasiewicz created a scheme to remove the need for parentheses from logical expressions, thus making those expressions more succinct, but the scheme was equally applicable to math. He named it Polish Notation in honor of his nationality. In his version, which we might call *forward* Polish notation, operators are prefixes, as in:

```
+ 2 3
```

In the early 1960s, computer scientists F. L. Bauer and E. W. Dijkstra invented a similar scheme but used operator suffixes rather than prefixes. They named it Reverse Polish Notation, honoring Łukasiewicz for inventing it first, albeit with prefixes.

Reverse Polish Notation (or RPN) became widely used by the public in the 1970s and 1980s when it was used in many hand-held scientific calculators. RPN (as you'll see in this chapter) is particularly easy to implement on a stack-based computing system. RPN is still the basis of some existing programming languages such as PostFix.

Is it possible for computers to implement *forward* Polish notation? Yes, but it is much more difficult because when you read an operator, you don't know what to apply it to yet. Your best bet in writing a forward Polish interpreter is to read all the items (tokens) into a list and then reverse the list!

Using a Stack for RPN

If you've read the chapters of this book in sequence, you've come across the term *stack* before. First, there's "*the* stack," which is used to store local variables, argument values, and return addresses.

Then there's the customized stack described in Chapter 12 for the Tower of Hanoi puzzle. That was a special stack because it did something unusual: keeping track of empty spaces. For example, if a pole for a five-ring puzzle contains the two smallest rings and no other rings, this state of affairs is notated as {0, 0, 0, 1, 2}.

STL provides support for a generalized stack class that does a lot of the same things as these other stacks. An STL stack is a simple last-in-first-out mechanism (LIFO).

Here's how it's used to implement an RPN calculator. Again, consider this input line:

```
2 3 + 17 10 - *
```

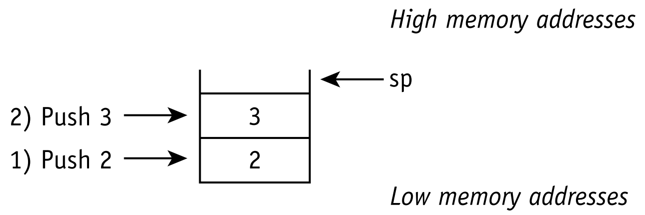
How can we attack this? Common sense tells us a couple of things: first, when the program reads a number, it must save it for later use; second, when the program reads an operator, it should perform an operation, perform "number crunching" on the two operands, and save the result.

The strategy is therefore:

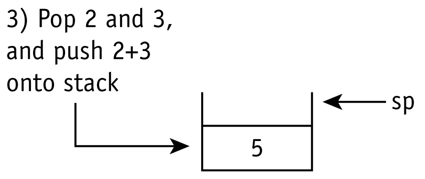
1. When the program reads a number, it pushes it on top of a stack.
2. When the program reads an operator (+, *, –, or /), it pops two values off the stack, computes the result, and pushes this result back onto the stack. Because a last-in-first-out mechanism is used, an operator will bind to the nearest two expressions that precede it, which is exactly what we want.

Let's see how this works for the line of input `2 3 + 17 10 - *`.

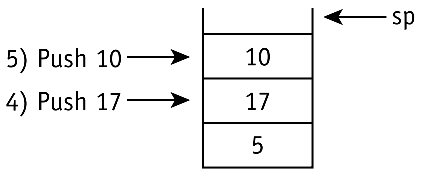
First, the algorithm reads the numbers 2 and 3, and pushes these numbers onto the stack. In the following diagram, sp is the stack pointer, indicating the top of the stack. STL provides no access to this pointer, but it's useful to understand that it's there.



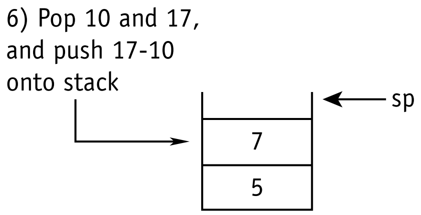
Next, the program reads an addition sign (+). It pops two numbers, adds them, and pushes the result back on the stack.



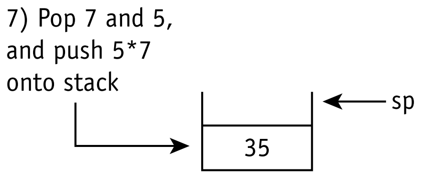
Then, the algorithm reads the next two numbers, 17 and 10, and pushes these on top of the stack.



Next (we're almost done), the algorithm reads another operator: subtraction (–). Again, it pops the top two numbers, performs a calculation, and pushes the result onto the stack.



Finally, the algorithm reads a multiplication operator (*). One final time, it pops two numbers. Then it multiplies them together and pushes the result onto the stack.



The final result is 35, which is correct.

Introducing the Generalized STL Stack Class

In the previous section, we saw how a simple stack mechanism, storing numbers, could be used to implement a Reverse Polish Notation calculator. Now, let's use an STL stack for the program.

Before using the list template, you need to turn on support for it by using an **#include** directive.

```
#include <stack>
```

You can then create a generalized stack mechanism with syntax similar to that used for STL lists.

```
stack<type>  stack_name;
```

Remember, as with the **list** template, that the **std::** prefix is necessary unless you include a "using namespace std;" statement. Otherwise, you must refer to the stack template as **std::stack**.

For example, you can create stacks with statements like these:

```
#include <stack>
using namespace std;
...
stack<int>       stack_of_ints;
stack<Fraction>  stack_of_Fraction_objects;
stack<double>    xStack;
```

Each of these statements creates an empty stack. To insert elements, use the **push** member function. The most useful of the stack member functions are shown in this table:

STACK CLASS FUNCTION	DESCRIPTION
stack.**push**(*data*)	Pushes data (of stack's underlying type) onto top of stack
stack.**top**()	Returns data (of stack's underlying type) from top of stack, but it does not remove data; for that, use **pop**
stack.**pop**()	Removes top item of stack (but does not return its value)
stack.**size**()	Returns the number of items currently stored in the stack
stack.**empty**()	Returns true if stack is empty, false otherwise

Using a stack class to push data onto the top of the stack is easy enough.

```
stack<int>     stack_of_ints;
...
stack_of_ints.push(-5);
```

However, the STL stack design splits the pop operation into two steps, so popping off an item requires a "top and pop" operation.

```
int n = stack_of_ints.top();  // Copy top of stack.
stack_of_ints.pop();          // Remove top of stack.
```

Example 13.2. *Reverse Polish Calculator*

The following program involves about a page of code, which is a remarkably small size for a program that can evaluate complex expressions. The **strtok** function does most of the work of interpreting input; the STL stack class, num_stack, does most of the work of pushing and popping numbers off a stack.

rpn.cpp

```
#include <iostream>
#include <cstring>  // Use old-style cstrings so that we
                    //  can use the strtok function.
#include <stack>

using namespace std;
#define MAX_CHARS 100

int main()
{
   char input_str[MAX_CHARS], *p;
   stack<double>  num_stack;
   int c;
   double a, b, n;

   cout << "Enter RPN string: ";
   cin.getline(input_str, MAX_CHARS);
   p = strtok(input_str, " ");
   while (p) {
      c = p[0];
      if(c == '+' || c == '*' || c == '/' || c == '-'){
          if (num_stack.size() < 2) {
              cout << "Error: too many ops."<< endl;
              return -1;
          }
          b = num_stack.top(); num_stack.pop();
          a = num_stack.top(); num_stack.pop();
          switch (c) {
              case '+': n = a + b; break;
              case '*': n = a * b; break;
```

▼ *continued on next page*

rpn.cpp, cont.

```
                case '/': n = a / b; break;
                case '-': n = a - b; break;
            }
            num_stack.push(n);
        } else {
            num_stack.push(atof(p));
        }
        p = strtok(nullptr, " ");
    }
    cout <<"The answer is: " << num_stack.top()<< endl;
    return 0;
}
```

How It Works

The program begins, as always, with **#include** directives. Note that <stack> is used to turn on support for the STL stack template.

```
#include <stack>
```

This enables a stack to be created and built on any base type. What type is needed here?

Clearly, this should be **double**, the floating-point type, because there's no good reason to limit the end user to integer data only; we want to be able to do things like adding 1.4 to 2.345.

The next statement creates a stack of type **double**.

```
stack<double>  num_stack;
```

Next, the program gets a line of string input from the user and starts to break it down. The **strtok** function, as explained in Chapter 8, finds the first "token" (that is, word or item) from the input string. The first argument to **strtok** specifies this string. The second argument specifies the character or characters (in this case, a blank space) to be recognized as token separators.

```
p = strtok(input_str, " ");
```

The function returns a pointer to a substring containing the first token. Note that for this to work properly, each and every item must be separated by one or more spaces, including the operators. So, the following will work as expected

```
2 3 +  17 10 -  *
```

but not this:

```
2 3+ 17 10-*
```

It might be reasonable to allow this input, but you'd need a more sophisticated lexical analyzer that you'd have to write yourself. (Note that the C++14 library includes support for regular expressions, which can be used to perform "tokenizing," but that's an advanced topic.)

After the initial call to **strtok**, you can call **strtok** again, specifying **nullptr** as the first argument. Calling the function this way says, "Give me the next token from the same input string used before." In other words, using a **nullptr** argument enables you to get the next token, and the next after that, and so on, without starting over.

The program carries out this function call near the bottom of the main loop.

```
p = strtok(nullptr, " ");
```

Note The **nullptr** keyword has been supported since C++11. If your compiler is more than a few years old, you may need to use NULL instead.

The program's main loop processes the next token as long as there is another to process. The first step in responding to a token is to determine whether it is an operator (+, *, –, or /). If it is an operator, the program does several things.

The program's first response to an operator is to ensure there are at least two items on the stack. This is important because if you attempt to pop an empty stack, the STL **pop** function goes off into the Twilight Zone and causes severe problems. To prevent that, the program has a short error-checking section that prints an error message and exits if needed.

```
if (num_stack.size() < 2) {
    cout << "Error: too many ops." << endl;
    return -1;
}
```

The second response to reading an operator is to pop two numbers off the stack. These are put into b and a, in reverse order. Remember that a stack is a last-in-first-out device, so reverse order has to be observed.

```
b = num_stack.top(); num_stack.pop();
a = num_stack.top(); num_stack.pop();
```

With STL stack classes, you have to use a "top and pop" technique, because the two member functions, **top** and **pop**, each do only part of a "pop" operation.

The third response to an operator is to perform the specified calculation and put the result back on the stack. The program uses **switch**-**case** logic explained in Chapter 3. Depending on whether the operator is +, *, /, or –, the program jumps to a different **case** statement, performs a calculation, and then breaks out of the **switch** block.

```
switch (c) {
    case '+': n = a + b; break;
    case '*': n = a * b; break;
    case '/': n = a / b; break;
    case '-': n = a - b; break;
}
```

After the calculation is performed, the result—stored in n—is pushed back onto the stack.

```
num_stack.push(n);
```

That takes care of everything that has to be done in response to an operator. If the item read is not an operator, the action to be taken is a great deal simpler. All we do is translate the item into a floating-point number and push it onto the stack.

```
num_stack.push(atof(p));
```

What if the item doesn't contain a valid number? For example, what if it contains letters instead? That's not a problem: **atof** returns 0 in that case, and there is no harm in operating on 0. (Just don't try to divide by it!)

EXERCISES

Exercise 13.2.1. Extend the RPN calculator in Example 13.2 by adding a unary operator signified by a pound sign (#). Have this operator take the reciprocal of its one operand. That is, x should yield 1/x. Remember that all four of the existing operators are binary operators, taking two operands. But the grammar for a unary operator is as follows:

expression → *expression unary-op*

Exercise 13.2.2. Add a caret operator (^) to perform unary negation; that is, reverse the sign of the operand, changing positive numbers to negative and vice-versa.

Exercise 13.2.3. Revise the program so that it keeps prompting the user for another line of input to calculate until the user enters an empty line by just pressing ENTER. That is to say, continue the cycle of prompting for input, interpreting it as RPN, and printing the answer, until the user wants to quit. Don't quit after one go-round. (By the way, to make sure this program works correctly every time, you ought to clear the stack before beginning a new operation.)

Correct Interpretation of Angle Brackets

Because brackets (< and >) have multiple uses in C++, ambiguity is possible when you get into heavy use of templates. The following declaration creates a problem for C++ syntax:

```
list<stack <int>>   list_of_stacks;
```

This statement should create a list of stacks. This behavior is perfectly intelligible; C++ is being directed to create a container class that contains other containers within it, which, no matter how complex that sounds, is perfectly valid.

But in this case, traditional C++ is syntactically challenged. It normally interprets two right-angle brackets in a row (>>) as the right-shift operator, and that causes a baffling syntax error. (This same operator, incidentally, is overloaded as a stream data in-flow operator with objects such as **cin**.)

In traditional C++, therefore, it is necessary to insert a space between the two right-angle brackets to ensure correct interpretation.

```
list<stack <int> >   list_of_stacks;
```

In C++11 and later, however, adding this space is not necessary, because it specifies that the language must correctly interpret two right-angle brackets according to context.

Chapter 13 Summary

Here are the main points of Chapter 13:

- To enable use of the list template, use an **include** directive.

```
#include <list>
```

- Each use of the name “list” must be qualified as **std::list**, unless, of course, you place a **using** statement in your program.

```
using namespace std;
```

- You declare a list container class by using this syntax:

```
list<type>  list_name;
```

- Once a list is created, you can add items of the appropriate type to it by using **push_back** (push to back of list) and **push_front** (push to front of list).

```
#include <list>
using namespace std;
```

```
...
list<int> Ilist;
Ilist.push_back(11);
Ilist.push_back(42);
```

- You can access members of a list by creating an iterator, which is not a pointer but uses several of the same operators. For example:

```
list<int>::iterator iter;
```

- You can loop through a list by calling the list's **begin** and **end** functions. For example, the following code prints each item in the list, one to a line:

```
for (iter = Ilist.begin(); iter != Ilist.end(); i++)
     cout << *iter << endl;
```

- As with the **list** class, you turn on support for last-in-first-out (LIFO) stack classes with an **#include** statement.

```
#include <stack>
using namespace std;
...
stack<string> my_stack;
```

- The **push** function pushes an item onto the top of the stack.

```
my_stack.push("dog");   // Put onto top of stack
```

- To pop items off the top of the stack, use a "top and pop" technique.

```
string s = my_stack.top();  // Get top item
my_stack.pop();             // Remove top item
```

- Popping an empty stack is a fatal error, so be sure to check stack size by calling the **size** or **empty** function whenever you need to do so.

14 Object-Oriented Monty Hall

Do you like solving paradoxes and riddles? This chapter provides more experience in applying object orientation while attacking one of the more interesting riddles of our time.

In studying the sample program in this chapter, you'll learn the solution to one of the most perplexing and famous paradoxes of modern times—no kidding! Follow along to learn how to solve the paradox and see practical examples of C++ class-object syntax at work.

This chapter explores one of the most interesting uses for computers. When you have a question that can't be solved by people arguing, you can sometimes use a computer program, assuming it is written correctly, to run a simulation and see what happens. Isn't that better than just arguing forever?

What's the Deal?

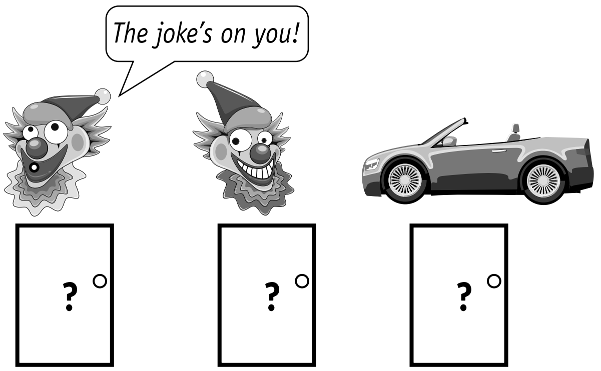
In the 1960s, the most popular game show in America was *Let's Make a Deal,* starring a charismatic host named Monty Hall. Monty's name will forever be associated with this entertaining show and the logical paradox it inspired.

This paradox—the logic of which I'll explore in detail in the second half of the chapter—is not really based on *Let's Make a Deal*, but on a hypothetical game show that never aired. (If it ever does air, you'll learn in this chapter how to maximize your chances of winning the game.) Monty Hall himself stated that there are differences between this hypothetical game and the real *Let's Make a Deal*. That's good for me, as it prevents conflicts with intellectual-property law.

Instead of playing *Let's Make a Deal*, we're going to play a hypothetical game called *Good Deal, Bad Deal*. The host will not be Monty Hall but rather a guy named "Monty Schmall."

Here's how the game works: Monty asks the contestant to pick one of three doors, behind each of which is a prize. Not all the prizes are good. Two out of three of them are stinkers—something worth a few dollars, if that much.

Behind just one of the doors is a fabulous prize, such as—in the booming voice of the announcer—“YOUR BRAND NEW CAR!”



Here’s where it starts to get interesting: After the contestant makes her initial selection, Monty Schmall shows her what’s behind one of the doors she *didn’t* pick. This door will be one of the ones with a stinker.

Now, having shown her this “bad” door, Monty asks another question. There are now two doors left: the one picked initially, and an alternative door, the one the contestant did not pick, the contents of which has not yet been revealed. Monty asks: “Do you want to stick with your initial choice, or do you want to switch to the remaining door?”

For example, if the contestant’s initial choice was Door No. 1, and Door No. 2 was revealed to have a stinker, does she want to stick with Door No. 1 or switch to Door No. 3?

For many people—including more than a few with PhD’s and other scholarly credentials—it’s obvious that the choice between Doors 1 and 3 is a 50/50 proposition, and that you can’t improve your chances by switching.

And yet these people are wrong. To find out why, we’re going to write a program to simulate the game show and see what happens. To summarize, the rules are as follows:

1. Behind one of three doors, randomly selected, is a valuable prize. The other two doors have worthless prizes, or “stinkers.”
2. The contestant makes an initial selection but is not told (at first) whether it’s the winning choice or not.
3. Monty then reveals one of the doors the contestant did not choose, revealing it to be a stinker. If the contestant’s first choice has the good prize, Monty (or his producers) must randomly decide which of the other two doors to reveal, because they both have stinkers.

4 After this reveal is done, there remain two doors: the contestant's first choice and another door yet to be opened. Monty asks: "Do you want to stick with your first choice, or switch to the remaining door?" Do you stick with Door No. 1 (assuming it's your first pick) or switch to Door No. 3?

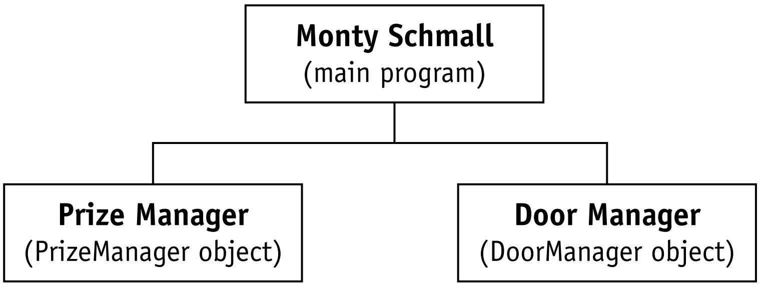
TV Programming: "Good Deal, Bad Deal"

It's fun to apply the object-oriented approach in modeling real-world objects. Pretend you're Monty Schmall. You're both the host and the executive producer of *Good Deal, Bad Deal*, but you can't do everything yourself. You need to delegate tasks to your staff.

With the object-oriented approach, we ask what the important pieces of data are and how we want to manipulate them. In this case, there are two major groups of data:

- The lists of good and bad prizes.
- The status of the doors, including: the identity of the door that has the one good prize instead of the stinkers, as well as which door should be revealed first and thereby eliminated.

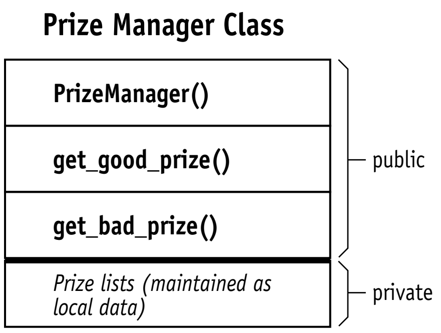
So there are the two groups of data—prize data and door-related data—that need to be managed. As executive producer, you delegate the data management to two of your associate producers. We can carry out this scheme by creating two classes, PrizeManager and DoorManager.



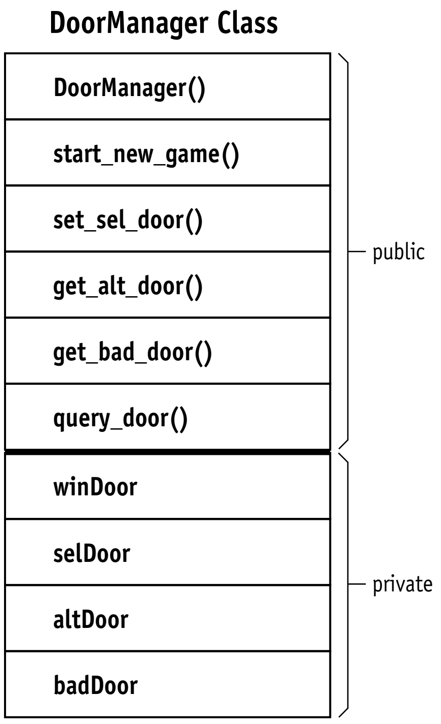
In object orientation, there's a one-to-many relationship between classes and objects. But in this case, there is just one object for each class. Each object will act like an associate producer.

```
PrizeManager prize_mgr;  // Create objects.
DoorManager  class_mgr;
```

The design of the Prize Manager class is simple. There is one constructor along with two public functions. The prize lists themselves are maintained as local data in the member functions.



The Door Manager's job is more elaborate. First, this producer has to determine in secret which door has the good prize. The other two doors will have stinkers. Later, depending on which door the contestant chooses, the Door Manager determines which door will be revealed as the "bad" door and which door will be offered as the alternative door.



This looks like a lot of members. But the process is straightforward. Here's how the functions are used:

- The start_new_game function determines which of three doors—0, 1, or 2—is the winning door (although to users of the class, these doors will be represented externally as 1, 2, and 3).
- After the contestant (that is, the user) selects a door, Monty calls the set_sel_door function to register that choice. The Door Manager uses that information to determine the alternative door (altDoor) as well as the "bad" door (badDoor), the one Monty will reveal to have a stinker.
- After the contestant makes her final choice, Monty calls the query_door function to determine whether the final choice is the winning door, that is, the one with the good prize.

The necessary information is represented internally with four data members, each of which is an integer containing a 0, 1, or 2 (which are translated into 1, 2, and 3 outside the class).

DATA MEMBER	USAGE
winDoor	Contains the number of the winning door. The DoorManager object chooses this number randomly at the start of each new game.
selDoor	Specifies the door that the contestant initially selects.
badDoor	Specifies the "bad" door, the door that's revealed as a stinker before the final choice is offered.
altDoor	Specifies the "alternative" door. Monty offers the contestant a chance to stick with her first choice (selDoor) or to switch to this door.

With this help from his producers, the job of Monty Schmall (represented by the main program) is relatively easy.

Example 14.1. *The PrizeManager Class*

The PrizeManager contains only two public functions, get_good_prize and get_bad_prize. This makes the class easy to write.

prizemgr.cpp

```
// Remember to include string, cstdlib, and ctime,
//  and using namespace std in the enclosing program.

class PrizeManager {
public:
    PrizeManager() {srand(time(NULL));}
    string get_good_prize();
    string get_bad_prize();
};

string PrizeManager::get_good_prize() {
    static const string prize_list[5] = {
       "YOUR BRAND NEW CAR!",
       "A BA-ZILLION DOLLARS!",
       "A EUROPEAN VACATION!",
       "A CONDO IN HAWAII!",
       "TEA WITH THE QUEEN OF ENGLAND!"
     };
     return prize_list[rand() % 5];
}

string PrizeManager::get_bad_prize() {
    static const string prize_list[8] = {
       "two week's supply of Spam.",
       "a crate of rotting fish heads.",
       "a visit from a circus clown.",
       "two weeks at a clown college.",
       "a ten-year-old VCR player.",
       "a lesson from a mime.",
       "psychoanalysis from a clown.",
       "a tour of the city dump."
     };
     return prize_list[rand() % 8];
}
```

How It Works

Remember that get_good_prize and get_bad_prize must be called through an object, because these are class functions.

```
    PrizeManager prize_mgr;
    cout << prize_mgr.get_good_prize() << endl;
```

Because this class consists of only two functions and their local variables, you might wonder why it needs to be a class at all. Couldn’t these two member functions be provided as global functions that are called directly, rather than through an object?

That would be true if there were no chance any data members would be added to this class in the future. However, when this class is revised and improved later in the chapter, it will be helpful to have the class organization, as you’ll see.

One of the fine points of the code is that it uses the **static** and **const** keywords to modify the declarations of the arrays of strings. Without these keywords, the string literals would be loaded into local-variable memory (on the stack), over and over again, every time the function was called. As local variables, these arrays have local visibility; but because they are static, they are loaded into memory only once.

Optimizing the Code

One potential drawback of the approach used so far is that the two arrays have “hard-coded” array sizes. The sizes have to be accurately determined by inspection, and (worse) if you add or delete elements, you have to make sure that you put in the new size correctly. Otherwise, the line of code that performs randomization will fail to execute correctly:

```
        return prize_list[rand() % 8];
```

If the hard-coded number (8) is too small, some prizes will never be chosen. But if this number is too large, the program will eventually fail, causing it to quit—if you’re in the Microsoft Visual Studio managed environment—or possibly do much worse.

So, even though it’s more work initially, a better approach is to use the compiler itself to determine the size of the array. Leave the array size blank, and then use the sizeof operator to determine the number of elements as follows:

```
    sizeof(prize_list) / sizeof(string)
```

This says: take the total size of prize_list and divide by the size of one element (a string object in this case). This produces the number of elements. Perhaps there should be a quicker and easier way to get the size of an array, but this is the only way currently supported in C++.

Now the two functions can be revised as follows. With this approach, strings can be added or removed as you like, and the size data will always be automatically correct.

```
string PrizeManager::get_good_prize() {
    static const string prize_list[ ]= {
       "YOUR BRAND NEW CAR!",
       "A BA-ZILLION DOLLARS!",
       "A EUROPEAN VACATION!",
       "A CONDO IN HAWAII!",
       "TEA WITH THE QUEEN OF ENGLAND!"
     };
     int sz = sizeof(prize_list) / sizeof(string);
     return prize_list[rand() % sz];
}

string PrizeManager::get_bad_prize() {
    static const string prize_list[ ] = {
       "two week's supply of Spam.",
       "a crate of rotting fish heads.",
       "a visit from a circus clown.",
       "two weeks at a clown college.",
       "a ten-year-old VCR player.",
       "a lesson from a mime.",
       "psychoanalysis from a clown.",
       "a tour of the city dump."
     };
     int sz = sizeof(prize_list) / sizeof(string);
     return prize_list[rand() % sz];
}
```

EXERCISES

Exercise 14.1.1. Test the PrizeManager class by writing a test program that includes the class declaration and then uses it to create an object. Finally, use that object by calling its two functions. Set up a loop that randomly calls one or the other of the functions until the end user enters a "quit" command.

Exercise 14.1.2. Modify the two PrizeManager functions by adding strings of your own. Then, make all modifications necessary for the functions to work correctly. (Note that to make sure the size values are correct, you can either change the sizes by hand or else use the technique described in the previous section.)

Exercise 14.1.3. It's probably not desirable to have the same prize two or more times in a row. Add a couple of data members and use them to prevent a prize already picked by a function from being picked in the next function call.

Example 14.2. The DoorManager Class

The Door Manager's principal tasks are 1) to secretly choose which of the three doors has the desirable prize, and 2) to decide—based on both the winning door and the contestant's selected door—which of the doors is revealed as a stinker in the middle of the game (the "bad" door).

doormgr.cpp

```
// The program that includes this class should also
//   include cstdlib and ctime.
//
class DoorManager {
public:
    DoorManager()  {srand(time(NULL)); }
    void start_new_game();
    void set_sel_door(int n);
    int get_alt_door() { return altDoor + 1; }
    int get_bad_door() { return badDoor + 1; }
    bool query_door(int n) {return n == (winDoor + 1); }
private:
    int winDoor;
    int selDoor, altDoor, badDoor;
};

void DoorManager::start_new_game() {
    winDoor = rand() % 3;
}

void DoorManager::set_sel_door(int n) {
    selDoor = n - 1;
    if (selDoor == winDoor) {
        if (rand() % 2) {  // Random true or false
            altDoor = (selDoor + 1) % 3;
            badDoor = (selDoor + 2) % 3;
        } else {
            badDoor = (selDoor + 1) % 3;
            altDoor = (selDoor + 2) % 3;
        }
```

▼ continued on next page

doormgr.cpp, cont.

```
        } else { //Else, if the selected door is not the
                    // winning door...

             // Alternative door MUST be the winning door!
            altDoor = winDoor;

             // Assign badDoor the number in {0, 1, 2}
             //  not equal to either selDoor or altDoor.
            badDoor = 3 - selDoor - altDoor;
        }
    }
```

How It Works

As with the PrizeManager class, only one instance of DoorManager is used in the program. This is a simple process: use the class to create one object, and then use that object to call member functions. For example:

```
    DoorManager door_mgr;
    door_mgr.new_game();
```

DoorManager is a good example of a C++ class, because it maintains some key pieces of data internally. Users of the class can access this private data only by calling member functions.

One reason this is necessary is that in order to keep the math simple, the doors are identified internally as 0, 1, and 2, even though these same doors are identified outside the class as 1, 2, and 3. If the contestant picks Door No. 3, *this is stored within the object* as 2, not 3. Such differences are invisible to the user of the class, who only knows about doors numbered 1, 2, and 3, and has no access to how things work inside the class.

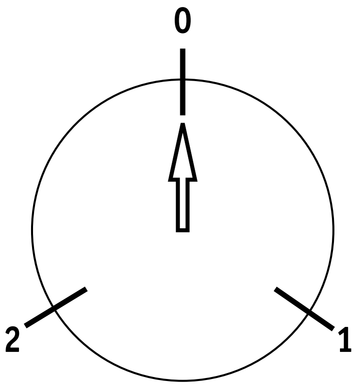
The DoorManager object has the responsibility for choosing the "bad" door, that is, the door revealed to have a stinky prize.

Consider the case in which the contestant chooses the correct door initially. This will happen one-third of the time. In that case, one of the remaining doors has to be randomly chosen as the "bad" door, and the other door will be the "alternative" door.

There are several ways to make this random selection, but modular arithmetic is most efficient. The MOD 3 operation (which is what "% 3" does) takes 0, 1, or 2 as input and produces the other two numbers in the set. So, for example, if selDoor is 1, altDoor and badDoor are assigned the values 0 and 2, but not necessarily in that order.

```
if (rand() % 2) {  // Random true or false
    altDoor = (selDoor + 1) % 3;
    badDoor = (selDoor + 2) % 3;
} else {
    badDoor = (selDoor + 1) % 3;
    altDoor = (selDoor + 2) % 3;
}
```

The MOD 3 operation (%3) is like using a special clock that has one hand and three positions: 0, 1, and 2. You can start at any of these numbers and moving the hand forward will produce the other two numbers.



Now consider the case in which the door initially selected by the contestant is not the winning door, which will happen two-thirds of the time. The "bad" door, the one that's going to be revealed by Monty, cannot be the initially selected door. Nor can the bad door be the winning door. By the process of elimination, therefore, we can assign all three doors: altDoor must necessarily be the winning door because that's the only result consistent with the rules.

```
 // Alternative door MUST be the winning door!
altDoor = winDoor;

 // Assign badDoor the number in {0, 1, 2}
 //  not equal to either selDoor or altDoor.
badDoor = 3 - selDoor - altDoor;
```

This point is absolutely critical, so let's go over it again. Remember, we are considering the case in which the selected door (selDoor) does not match the winning door—a condition that the Door Manager can detect as soon as the selection is made. The Door Manager then infers the following:

- selDoor is not the winning door (because we just assumed that).
- badDoor, by definition, cannot be the winning door.
- Whichever door remains is the winning door, but altDoor is never the same as either selDoor or badDoor; therefore, it *must* be the same as the winning door, because altDoor is the only door left.

The last line of code does the following, again for the case in which the selected door is not the winning door: the Door Manager uses the value selDoor (the door selected by the contestant) and altDoor (the door presented as an alternative to the contestant) to calculate the value of badDoor.

We know that the values 0, 1, and 2 are distributed uniquely among the three variables and no value is repeated. Therefore, the three variables must have a combined total of 3.

```
selDoor + badDoor + altDoor = 3
```

Then, applying seventh-grade algebra, we solve for the variable badDoor.

```
badDoor = 3 – selDoor – altDoor
```

This simple algebraic trick lets us calculate the value of badDoor because, by this point, the program has already determined the value of the other two variables.

EXERCISES

Exercise 14.2.1. If an object-oriented approach were not used with this example—if, instead, the Door Manager class were implemented as a series of separate data declarations and functions—what would be the risks to the program? Why might the Monty Schmall program fail badly?

Exercise 14.2.2. Write a program to test the DoorManager class by repeatedly making an initial selection (offering this choice to the end user), and then getting the value of badDoor and altDoor by calling the function members. Finally, use the query_door function to see if the contestant's choice was the winning one. Print out all these results. Run this program a few times to see if the results are consistent with the rules of the game.

Exercise 14.2.3. The door values are maintained internally as 0, 1, and 2. The class can use the values 1, 2, and 3 instead, but this requires the MOD arithmetic in the set_sel_door function to be slightly more complex. Revise all the class functions to store the door changes internally as 1, 2, 3 instead of 0, 1, 2. Although some of the code will be slightly more complex, several other functions will be shorter as a result, because no conversion between inner and outer numbers will need be made. (Hint: for the numbers 1, 2, and 3 to work correctly during the MOD operation, 1 should be subtracted, then the arithmetic operation applied, and then finally 1 should be added.)

Example 14.3. *The Full Monty Program*

The following program listing assumes that the PrizeManager and DoorManager classes are inserted where indicated.

monty.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;

// INCLUDE PRIZEMANAGER AND DOORMANAGER
//  DECLARATIONS HERE.

void play_game();
int get_number();

PrizeManager prize_mgr;
DoorManager door_mgr;

int main()
{
    cout << "Welcome to Good Deal, Bad Deal!" << endl;
    cout << "I'm your host, Monty Schmall." << endl;
    string s;
    while (true) {
        play_game();
        cout << "Play again? (Y or N): ";
        getline(cin, s);
        if (s[0] == 'N' || s[0] == 'n') {
            break;
```

▼ continued on next page

monty.cpp, cont.

```
            }
        }
        return 0;
    }

    void play_game() {
        string s;
        cout << "Which of three doors would you like";
          << " (1, 2, 3)? ";
        int n = get_number();
        door_mgr.set_sel_door(n);
        cout << "Before I reveal what's behind the door,"
          << " I'm going to show a door you DIDN'T"
          << " pick." << endl;
        cout << "Behind Door No. "
          << door_mgr.get_bad_door() << " is..."
          << prize_mgr.get_bad_prize() << endl << endl;
        cout << "Now, would you like to switch from Door"
          << " No. " << n << endl << " to Door No. "
          << door_mgr.get_alt_door() << "? (Y or N): ";
        getline(cin, s);
        if (s[0] == 'Y' || s[0] == 'y') {
             n = door_mgr.get_alt_door();
        }
        cout << endl << "Ok. You just won... ";
        if (door_mgr.query_door(n)) {
            cout << prize_mgr.get_good_prize();
        } else {
            cout << prize_mgr.get_bad_prize();
        }
        cout << endl << endl;
    }

    int get_number() {
        string sInput;
        while(true) {
             getline(cin, sInput);
             int n = stoi(sInput);
             if (n >= 1 && n <= 3) {
                 return n;
             }
             cout << "You must enter 1, 2, or 3. Re-enter:";
        }
    }
```

What follows now is a sample session, with end-user input in bold. Remember that this is a simulation of a hypothetical show, so it doesn’t have all the elements of a real TV show, obviously.

```
Welcome to Good Deal, Bad Deal!
I'm your host, Monty Schmall.
Which of three doors would you like (1, 2, 3)? 3
Before I show you what's behind the door,
I'm going to show you a door you DIDN'T pick.

Behind Door No. 2 is... a visit from a circus clown.

Now, would you like to switch from Door No. 3
to Door No. 1? (Y or N): Y

Ok, you just won... YOUR BRAND NEW CAR!

Play again? (Y or N): N
```

The program presents the same choices the user would have if he or she were a contestant on a real game; it’s also written so that you can repeatedly play the game and see results of different choices. If you play the game a sufficient number of times, it should become clear which strategy is best.

I contend that if you play the game enough and track the results, the winning strategy will eventually become clear. There is a particular strategy that wins the game two-thirds of the time!

How It Works

If you understand the purpose of the two classes, the main program should be straightforward. This part of the C++ code presumes that the PrizeManager and DoorManager classes have already been declared, and it uses the class names to create two objects, one of each type.

```
PrizeManager prize_mgr;
DoorManager  door_mgr;
```

Each of these objects takes over an important job, leaving relatively little for the main program to do except interact with the end user. The classes, as we’d expect with most object-oriented programming, have already done much of the heavy lifting.

Even with the character-based user interface assumed in this book, communication with the end user is not always trivial. For example, the following function is included so that you can query the user to input the number 1, 2, or 3. The function keeps prompting the user until he or she enters one of these numbers.

```
int get_number() {
    string sInput;
    while(true) {
         getline(cin, sInput);
         int n = stoi(sInput);
         if (n >= 1 && n <= 3) {
             return n;
         }
         cout << "You must enter 1, 2, or 3. Re-enter:";
    }
}
```

Note If you have a compiler that is more than a few years old, you may need to use the **atoi** function instead of **stoi**, which is supported by compilers that are compliant with C++11 and later. The alternative to using **stoi** is to use the expression "atoi(sInput.c_str())".

EXERCISES

Exercise 14.3.1. One mild defect of this program is that it sets the randomization seed (calling **srand**) more than once, which is clearly inefficient. Rewrite all the code necessary to ensure that **srand** is called once and only once.

Exercise 14.3.2. Revise the program so that it keeps a running count of which decisions were made (how often the user stuck with the first door, how often she switched) and, as the user is about to quit, summarizes the "win" percentage for each decision: What is the win percentage when the player sticks with the first door? What is the win percentage when the player switches?

Exercise 14.3.3. Revise the program so it doesn't interact with the end user except to report results. Run the game silently thousands of times to test 1) How often did picking Door No. 1 and sticking with that selection yield a good prize? 2) How often did picking Door No. 1 and switching yield a good prize? Do the same for each of the doors. Remember that there are six possible plays altogether. Run in "silent mode" a thousand times for each possible play and report the results in tabular form.

The Monty Hall Paradox, or What's Behind the Door?

In 1990, Marilyn vos Savant, known as the world's smartest woman due to her record-breaking I.Q. scores, presented the Monty Hall problem in her column for *Parade* magazine, and in answer to a reader's letter. For millions of readers, it was the first time they'd encountered this question. In fact, it was raised at least as early as 1975, when it was mentioned in a letter to *The American Statistician* magazine by University of California professor Steve Selvin.

This puzzle, or challenge, is the exact same one posed in this chapter, although I've added some humorous elements. The choice is simple. There are only two possible choices: stick with the first door you selected, or (after one of the stinkers has been revealed) switch to the remaining door.

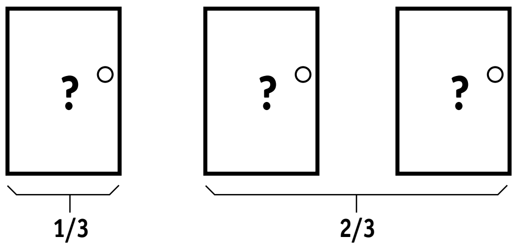
Almost everyone who considers this question for the first time insists on the same answer: it must be a 50/50 proposition. There are two doors left after one of the bad doors is revealed. Therefore, regardless of whether you stick to your first selection or switch to the remaining door, the probability is exactly 50% of winning the big prize. Right?

Wrong. It's not equal at all. In her column, Marilyn vos Savant tried to explain her reasoning, but thousands of people wrote in to say she was mistaken.

The correct answer, the one she gave, is that if you stay with your first choice, your chances of winning the game are only 1/3. But if you switch, your chances increase to 2/3, well above 50%!

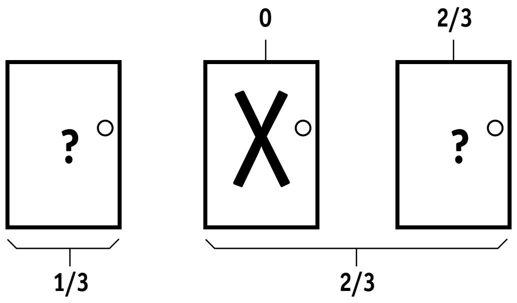
There are many ways to show this and yet people have great trouble accepting it. What is the difference between the two doors? The difference is that the initial choice was made with no information, so that choice has only a one-third probability of being right. But the choice of the remaining, or "alternative" door, contains within it the benefit of more information because one of the bad doors has been eliminated.

Imagine a picture of the three doors. For any door, the probability of it being the winning door is 1/3.

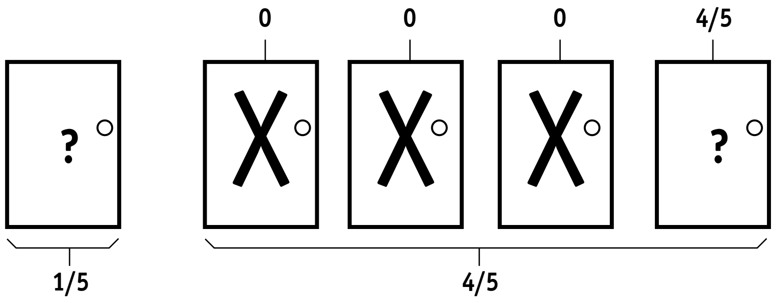


At this point, there's nothing in this scenario or picture that anyone would dispute, assuming the game isn't rigged. Right now, the chance of winning is obviously 1/3.

Now consider what happens after one of the stinkers is revealed. Suppose you chose Door No. 1 to begin with, and then Door No. 2 is revealed and eliminated. In that case, the probability of Door No. 1 being the winning choice doesn't change—there's no reason it should—but it becomes more likely that Door No. 3 is the right choice. Its probability goes from 1/3 to 2/3, while the probability of Door Number 2 being the winning door goes from 1/3 to 0.



The situation becomes even clearer if you consider a hypothetical game with five doors (still having only one good prize) in which Monty reveals three bad doors, not just one. You should be able to see that the probability that the alternative door turns out to be the winning door is 4/5.



Even that might not convince you, but you should be convinced by examining the C++ code in this chapter. Again, start by noting that the probability of the user's first choice being right is only 1/3, and it remains so. In that case, the allocation of the other two doors is random, as described.

But it's more likely that the first door the user chooses is not the winning door (a condition that has a probability of 2/3). In that case, the program executes the following statement:

```
altDoor = winDoor;
```

And there it is: the solution to the paradox and the proof that Marilyn was right! If the first door selected is not the winner (and remember, that's at a probability of 2/3), the program sets the alternative door to be the same as the winning door.

Logically, then, the probability of the alternative door being the winning door is 2/3. The smart move is to always switch doors in the final decision.

The reader may object that this is all my doing. By setting altDoor (the alternative door) to the value of winDoor, you may think I've rigged the game. But that's not so. The statement "altDoor = winDoor" necessarily follows as a consequence of the rules. There's no other way to write the program while observing all the rules presented on page 339. For reasons of pure logic, therefore, the alternative door must be the more likely winner.

As for Marilyn vos Savant's column, it became legendary and made famous what came to be called "the Monty Hall Paradox." It's a paradox not because there's no solution, but because so many people think that the solution is crazy. It runs counter to human intuition. But that just shows that intuition sometimes conflicts with the logical consequences of probability theory.

Within weeks, thousands of people wrote in, including some who were respected scholars, insisting that—in this case, at least—she had to be mistaken. So strongly did they feel, they seriously questioned whether she was really the smartest person in the world after all, and at least one letter made the sexist comment that only a male mind could handle problems in logic. Ms. Savant went on to devote three subsequent columns to defending her reasoning and still not everyone was convinced!

I like to think I've done a public service with this chapter. If you understand C++ and can follow the program I've described here, you'll find proof she was right. If you're still not convinced, run a simulation of a few thousand games, as described in Exercise 14.3.3, which should convince you beyond all doubt.

Improving the Prize Manager

A unique feature of this chapter is the fun we get to have with prizes. Other versions of the Monty Hall Paradox typically have one good prize and one bad: usually, a car (good) and a goat (not good). The humorous aspect of my version is that you can get all kinds of disappointing prizes, such as a visit from a circus clown, and the winning prizes can be (for example) a condo in Hawaii or tea with the Queen of England.

But by sheer chance, the user might see the same prizes over and over again. Producing a greater variety of prizes would make for a more entertaining game. We'd like to see the Prize Manager only select prizes not seen yet, until the list of prizes is exhausted.

A good way to adopt this behavior is the shuffle-up-and-deal approach, which selects elements of a prize list until they are used up, at which point the object automatically shuffles the list.

The shuffling algorithm is one described in Example 6.4 on page 148. We assume an array just big enough to hold all the prizes. Each index number, 0 to N − 1, has an equal chance of ending up in any position after shuffling.

```
For I = N − 1 Down to 2
    J = Random 0 to I
    Swap array[I] and array[J]
```

The algorithm, when correctly coded, takes an array of numbers from 0 to N − 1, which may start out in any order (provided that all the numbers are present), and produces an array of those same numbers, shuffled into newly randomized positions. If you analyze the algorithm carefully, you may realize there's always a chance that an element might get swapped with itself, but this has only a minor effect on performance, so it's not really worth worrying about. (Although you could test I and J for equality if you wanted, and if they are equal, don't bother to swap.)

The following code shows the revised PrizeManager class, rewritten so that it doesn't repeat prizes. It's necessary to have some new data members, such as an advancing index for each array. With this approach, when you get to the end of an array, it's time to "shuffle up and deal."

Notice that this C++ code uses two sets of arrays. Rather than shuffling the prize lists themselves, each of the two prize lists is controlled by an array filled with index numbers. The shuffle function randomizes the arrays of indexes, which are then used to select from the prize lists in what amounts to random order.

Prizemgr2.cpp

```
// Remember to include string, cstdlib, and ctime,
//  and using namespace std in the enclosing program.

class PrizeManager {
public:
    PrizeManager();
    string get_good_prize();
    string get_bad_prize();
```

Prizemgr2.cpp, cont.

```
private:
    int good_array[5];
    int bad_array[8];
    int good_index;
    int bad_index;
    void shuffle(int *p, int n);
};

PrizeManager::PrizeManager() {
     srand(time(NULL));
     for (int i = 0; i < 5; ++i) {
          good_array[i] = i;
     }
     for (int i = 0; i < 8; ++i) {
          bad_array[i] = i;
     }
     good_index = bad_index = 0;
     shuffle(good_array, 5);
     shuffle(bad_array, 8);
}

string PrizeManager::get_good_prize() {
     if (good_index >= 5) {
         shuffle(good_array, 5);
         good_index = 0;
     }
    static const string prize_list[5] = {
       "YOUR BRAND NEW CAR!",
       "A BA-ZILLION DOLLARS!",
       "A EUROPEAN VACATION!",
       "A CONDO IN HAWAII!",
       "TEA WITH THE QUEEN OF ENGLAND!"
     };
     return prize_list[good_array[good_index++]];
}

string PrizeManager::get_bad_prize() {
     if (bad_index >= 8) {
         shuffle(bad_array, 8);
         bad_index = 0;
     }
```

▼ continued on next page

Prizemgr2.cpp, cont.

```
      static const string prize_list[8] = {
         "two week's supply of Spam.",
         "a crate of rotting fish heads.",
         "a visit from a circus clown.",
         "two weeks at a clown college.",
         "a ten-year-old VCR player.",
         "a lesson from a mime.",
         "psychoanalysis from a clown.",
         "a tour of the city dump."
       };
       return prize_list[bad_array[bad_index++]];
   }

    void PrizeManager::shuffle(int *p, int n) {
       for (int i = n - 1; i > 1; --i) {
            j = rand() % (i + 1); // j = random 0 to i
            int temp = p[i];      // SWAP!
            p[i] = p[j];
            p[j] = temp;
       }
   }
```

Chapter 14 *Summary*

The main purpose of Chapter 14 has been to reinforce object-oriented concepts and show more examples of their use. But Chapter 14 also introduced, or gave greater emphasis, to several new ideas. These concepts and ideas are summarized here:

- Remember that object orientation is about modular programming. You can think of objects as assistants or coworkers to whom you delegate duties. Each has access to his or her own personal information and agrees to respond to certain requests.
- After declaring a class, you can create one or more objects. With some applications, you might have classes for which you only declare one object. But this is still perfectly valid.

```
PrizeManager  prz_manager;
```

- Class members are either public or private. (There is a third option, protected, mentioned in Chapter 16, “Polymorphic Poker.”) There are many advantages to keeping some data private, but a particularly strong one arises in situations in which an outside number (say, numbers running from 1 to 3) must be translated into an internal representation (say, 0 to 2). Because the data members are private, users of the class cannot “reach in” and refer to the data members directly. This can prevent a large source of errors.
- Remainder division (%) is useful for its support of modular arithmetic. One of the uses of modular arithmetic is to start with a number in a particular set (for example, 0, 1, or 2) and then advance to the other numbers in the same set.

```
doorAlt1 = (doorChoice + 1) %3;
doorAlt2 = (doorChoice + 2) %3;
```

- You can use the **sizeof** operator to let the compiler determine the size of an array. This has the benefits of making the program easier to maintain in the long run and also eliminating a source of potential errors.

```
int sz = sizeof(my_array) / sizeof(*my_array);
```

- Sometimes the best way to settle an argument is to just run a computer simulation of the problem, assuming the simulation is correctly written.

This page intentionally left blank

15 Object-Oriented Poker

Any hour of any day in Las Vegas, Nevada, you can hear them: the unending "*ding-ding-ding!*" sounds from what used to be "one-armed bandits," but these days are more likely to be machines playing video poker.

Now—at no expense!—you can bring the excitement of this classic game to your own computer, thanks to C++. Basically, each round of the game is a hand of draw poker, followed by a payout at the end. (Sorry, but I'm not set up to take your money, even if it were legal to do so. You'll have to be content with winning virtual dollars.)

Although it's possible to write this program without object orientation, I'm going to use this chapter to further demonstrate some more object-oriented features: how to return an object from a function, how to enable an object to display itself, how to manipulate arrays of objects, and finally, how to use the **vector** template, one of the most useful parts of the C++ Standard Template Library.

Winning in Vegas

The object of poker is to end up with the best hand of five cards you can. In the draw poker variant, used in video poker, you get a chance to improve your hand. Then, depending on how good that hand is, you get a payoff anywhere from one unit (the size of your bet, letting you break even) up to hundreds of times your bet. Jackpot!

In nearly all forms of poker, the more cards you have with matching rank, the better. Getting four of any rank is exceptional and earns a big payoff, although you won't see it often. Matching just two cards in rank gives you a pair, a much lower hand. It's not entirely worthless, but at least you break even rather than losing money. In between these hands is three of a kind, which, no surprise, pays off better than a pair (at 2 to 1) but nowhere near as well as four of a kind.

There are also a number of special hands. Most people find these easy to learn. Note that order of the cards never matters in poker. If you have one of the following hands, with your cards *in any order relative to each other*, you still have a high hand:

- **Full house:** three of a kind combined with a pair. For example: A-A-A-5-5 or 8-8-8-K-K. Valued just below four of a kind.
- **Flush:** all five cards of the same suit. Valued just below a full house.
- **Straight:** all five cards in a continuous sequence. For example, J-10-9-8-7. Even if these are arranged as 9-7-8-10-J, it's still a straight. Valued just below a flush, but above three of a kind.
- **Two pair:** fairly self-explanatory. Valued above a pair but below three of a kind.

In the second half of the chapter, I'm going to show how to write C++ code that looks at the player's hand and detects which of these hands are present. The rank of your final hand (after drawing) determines your payoff.

If you know anything about poker, or if you're just insightful, you may notice that it's logically possible for the same hand to be both a straight *and* a flush. This gives rise to two extra-special hands:

- **Straight flush:** A hand that's both a straight and a flush. For example, 6-5-4-3-2, all hearts. Remember that order never matters.
- **Royal flush:** A-K-Q-J-10, all in the same suit. This is the highest hand of all, except for five of a kind, which isn't possible with a standard deck and no wild cards. The payout of a royal flush is fantastic, as you'd imagine. Royal flushes are a subset of straight flushes, but of the highest rank.

To make things easier to digest, I'm going to follow a specific game plan in writing the poker application:

First, we'll develop the Deck and Card classes, laying the groundwork for the application. Second, we'll write the main program for the simplest version of the game, which uses the Deck and Card classes to play one round with no redraws.

Third, we'll improve the game by enabling the user to keep or redraw as many cards as she chooses, just as in draw poker. Finally, we'll write another class, Evaluator, which knows how to analyze any group of five cards and tell what kind of hand it is: a flush, four of a kind, full house, a pair, and so on. This evaluation of the hand will determine the payout.

And now, it's off to Vegas, as it were.

How to Draw Cards

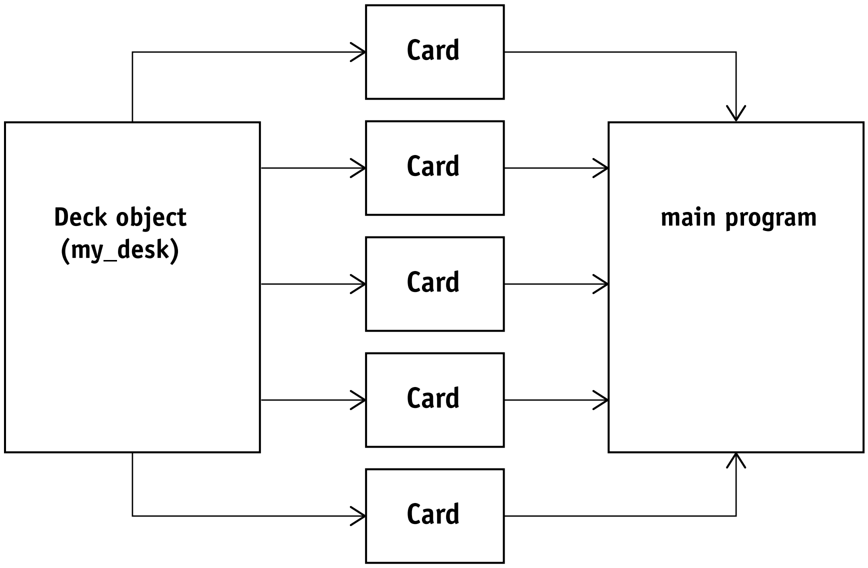
Object-oriented techniques, although not absolutely critical in this example, provide a superior way to analyze the problem. As with other examples in the last few chapters, we begin by asking: what are the major pieces of data in the program and how do we need to manipulate them?

Object orientation is helpful in design because you start with the big picture and fill in the details later. At the most general level, what's involved in a video poker game?

First, there's a deck of cards. Video poker is meant to simulate dealing from an actual deck, so that you would never see, for example, five aces of spades in a row, which would be impossible if a real deck were involved. Maintaining and shuffling are easy, because earlier chapters in this book introduced the basic techniques.

Another important kind of data is the individual card. This is a fairly small unit of data, but it does have both rank and suit information (the latter cannot be entirely ignored in poker because when it comes to flushes, suits do matter). This simple data type can be given some intelligence, so it knows how to display itself.

Conceptually, here's the flow of data in the overall program. The main program calls on the Deck class to supply five Card objects, and then the main program gives each of these objects the command, "Print yourself!"

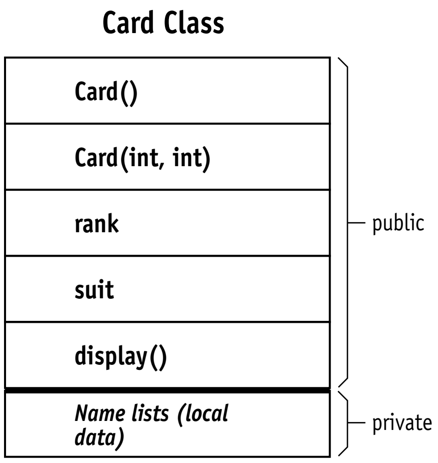


We need to design and implement two classes. Writing the main program should then be about as easy as getting a free drink at a Las Vegas casino.

The Card class is especially simple. Clearly, it needs rank and suit data members. If there was some need to protect this data from outside access, these members could be made private with access available through function members only. However, there isn't much to be gained by privatizing this data. Let's just provide direct access for a change.

The interesting members of the class are the functions. The constructors will enable us to create Card objects more easily, occasionally saving a line of code or two. The display function, although it could be made a global function, belongs in the class because of its tight association with the object. In short, it's a better way of organizing things.

Conceptually speaking, you can think of the display function as giving each Card object a kind of intelligence. It knows how to print itself.

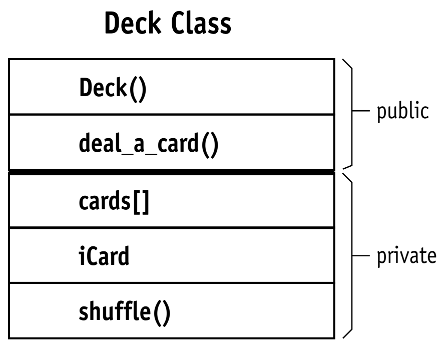


Now, let's move on to the Deck class. This shouldn't be difficult to write either.

By now, you should find the functions of the Deck class easy to write because its main duties are to shuffle up when needed, hiding this detail from the user of the class, and dealing a card when requested. Chapter 14, "Object-Oriented Monty Hall," ended with code that carried out these operations, and Chapter 6, "Arrays: All in a Row...," ended with a similar example.

The only substantially new thing about the Deck class is that its deal_a_card function returns a Card object rather than an integer.

The Deck class has the following structure—still relatively simple. If you've followed this book up until now, this class should be a breeze to implement.



Okay, enough theory. Let's examine the two classes in detail.

The Card Class

The Card class is basically a data record containing two integers, but it has some additional features. In the last few chapters, we've seen how useful constructors are, and the constructors for the Card class are going to save a few lines of code here and there.

In addition, this class supports a display function, which builds some intelligence into the data structure.

Here's the C++ code for the class, which is quite short:

card.cpp

```
// Remember to include string and
// using namespace std in the enclosing program.

class Card {
public:
    Card() {}
    Card(int r, int s) { rank = r; suit = s; }
    int rank;
    int suit;
    string display();
};

string Card::display() {
    static const string aRanks[] = {" 2", " 3", " 4",
```

▼ continued on next page

card.cpp, cont.

```
        " 5", " 6", " 7", " 8", " 9", "10", " J", " Q",
        " K", " A" };
    static const string aSuits[] = {
        "clubs", "diamonds", "hearts", "spades" };
    return aRanks[rank] + " of " + aSuits[suit] + ".";
}
```

The string-array data is stored in two local variables, effectively making it private. For efficiency's sake, these are both declared **static const**, as explained in Chapter 14, so that the data is loaded into memory only once, not every time the function is called.

The Deck Class

The Deck class is more elaborate than the Card class, but what it has to do is still fairly simple. Here's the code listing:

deck.cpp

```
// Remember to include string, cstdlib, ctime and
// using namespace std in the enclosing program.

class Deck {
public:
    Deck();
    Card deal_a_card();
private:
    int cards[52];
    int iCard;
    void shuffle();
};

Deck::Deck() {
    srand(time(NULL));
    for (int i = 0; i < 52; ++i) {
        cards[i] = i;
    }
    shuffle();
}
```

deck.cpp, cont.

```
void Deck::shuffle() {
    iCard = 0;
    for (int i = 51; i > 0; --i) {
        int j = rand() % (i + 1);
        int temp = cards[i];
        cards[i] = cards[j];
        cards[j] = temp;
    }
}

Card Deck::deal_a_card() {
    if (iCard > 51) {
        cout << endl << "RESHUFFLING..." << endl;
        shuffle();
    }
    int r = cards[iCard] % 13;
    int s = cards[iCard++] / 13;
    return Card(r, s);
}
```

This book has featured similar C++ code before. What's new here is that the deal_a_card has a class return type, Card, which means that the function must return a Card object:

```
Card deal_a_card();
```

The last line of the function definition returns just such an object. In this case, the constructor is called directly:

```
return Card(r, s);
```

The heart of the class is its ability to automatically "shuffle up and deal" as needed. Let's review the shuffling algorithm again.

Pseudocode

```
For I = 51 down to 1
    J = Random 0 to I
    Swap cards[I] and cards[J]
```

It's amazing that such a small algorithm can do so much. It uses the **for** statement, which in C++ is highly flexible. You can always use it to count down to something as well as up.

Index number 51 is the last position in the deck. The code causes swapping between that position and a position index by J, in which J can run from 0 all the way up to and including 51. This effectively says "swap positions with any card in the deck."

During the next iteration of the loop, I is set to 50 (the next highest number) and J can be any random number from 0 to 50. What this does is to select any card for this next-to-highest position *from all the remaining cards in the deck*. And so the loop continues. The third iteration selects a card from positions 0 to 49, and so on. Eventually, every position in the deck is filled with a randomly selected card.

Notice that the algorithm always carries out a swap, even though, on occasion, I and J may be the same.

The question of whether I and J should always be swapped—even though they might be equal—is a classic case for optimization analysis, an issue many programmers ignore but C++ programmers tend to think is important.

In this case, the potential inefficiency of swapping I and J unnecessarily must be compared to the cost of frequently performing a test for equality. Because I and J are integers, it isn't worth performing this test. But if I and J were more complex kinds of objects, it might be more efficient to run the test and swap I and J only if they're unequal.

```
if (i != j) {
    int temp = cards[i];
    cards[i] = cards[j];
    cards[j] = temp;
}
```

Finally, remember that to use a class, you must first create at least one object and then call its functions through that object. For example, the first statement below instantiates the Deck class directly, creating the object my_deck, and then the second statement uses my_deck to produce an instance of the Card class.

```
Deck my_deck;
Card crd = my_deck.deal_a_card()
```

Doing the Job with Algorithms

Chapter 13, "Easy Programming with STL," introduced the C++ Standard Template Library (STL) as a wonderful time-saving device. The library includes the collection templates, such as **list** and **stack**, which can contain almost any kind of base type. The other major category in the STL are the algorithms, which perform common programming tasks—again, operating on nearly any kind of base type.

Note Like many parts of the C++ library, each STL algorithm requires a **std::** prefix, unless you include the "using namespace std" statement, in which case you don't have to worry about the issue.

To use any of the STL algorithms, first make use of the **#include** directive.

```
#include <algorithm>
```

One of the most frequently used algorithms is **swap**, which switches the value of two arguments, provided that their types match precisely. (If the types do not match perfectly, the algorithm may fail, as the action then becomes ambiguous.) For example:

```
#include <algorithm>
using namespace std;
...
int lil = 1, big = 1000;
swap(lil, big);
cout << "big is now: " << big << endl;
```

By using **swap**, you have to add one additional line (**#include** <**algorithm**>) but you can save several lines of code. The shuffle function can then be reduced to:

```
void Deck::shuffle() {
    iCard = 0;
    for (int i = 51; i > 0; --i) {
        int j = rand() % (i + 1);
        swap(cards[i], cards[j]);
    }
}
```

But you can do even better than that. The **random_shuffle** algorithm does almost all the work of the function, saving even more lines of code. This algorithm assumes that the program has already set a random seed. Assuming it has, you can specify a range of elements to be shuffled randomly, as follows:

Keyword

random_shuffle(*beg_rage*, *end_range***);**

In this syntax, *beg_range* is an iterator or pointer to the beginning of the range of the collection (such as an array) to be shuffled; *end_range* is an iterator or pointer to one position past the last element in the target range. For example:

```
void Deck::shuffle() {
    iCard = 0;
    random_shuffle(cards, cards + 52);
}
```

Example 15.1. *Primitive Video Poker*

This version of the program is the simplest possible form of the game. It allows no redraws and it doesn't evaluate the hands. We're going to add those features later on in this chapter.

Poker1.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
using namespace std;

// INCLUDE THE DECK AND CARD CLASS
// DECLARATIONS AND DEFINITIONS HERE.

Deck my_deck;
Card aCards[5];
void play_game();

int main() {
    string s;
    while (true) {
        play_game();
        cout << "Play again? (Y or N): ";
        getline(cin, s);
        if (s[0] == 'N' || s[0] == 'n') {
            break;
        }
    }
    return 0;
}

void play_game() {
     for (int i = 0; i < 5; ++i) {
         aCards[i] = my_deck.deal_a_card();
         cout << i + 1 << ". ";
         cout << aCards[i].display() << endl;
     }
}
```

As with some other examples in this book, I've written the main function so that it keeps playing the game until the user wants to exit. Here's a sample session of this program:

```
1. A of clubs.
2. 10 of diamonds.
3. K of spades.
4. 3 of spades.
5. A of hearts.
Play again? N
```

This hand constitutes a pair: specifically a pair of aces (A). Order makes no difference, except for clarity and aesthetics. Consider that in a real poker game, you could lay out the cards in this order and the hand would still be a pair. One of the time-honored rules is "the cards speak for themselves," meaning that every poker player is honor-bound to recognize this hand as a pair even if the holder of this hand doesn't bother to put the aces next to each other.

On such questions did more than a few Riverboat gamblers reach for their guns. The cards speak for themselves. It shouldn't matter how you arrange them. That principle will matter later in this chapter, when the computer is taught how to recognize the value of hands; it must do so without expecting matching cards to be next to each other.

How It Works

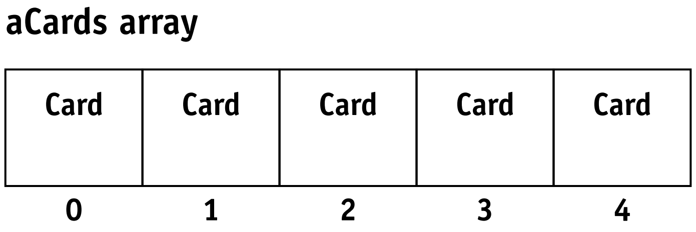
Most of the work of this program is done by the Card and Deck classes, so there's not much for the main program to do. The program creates the Deck object, which it then uses by requesting that object to deal cards.

```
Deck my_deck;
Card aCards[5];
```

Remember, to get a Card object we call the Deck object's deal_a_card member function.

```
aCards[i] = my_deck.deal_a_card();
```

The program, as you can see, puts every card dealt into an array, as well as displaying that card.



But why even bother with the array? I could have just printed the cards directly, as in the following code. This code looks similar to Example 15.1, but you will see no reference to members of an array.

```
void play_game() {
     for (int i = 0; i < 5; ++i) {
         Card crd = my_deck.deal_a_card();
         cout << i + 1 << ". ";
         cout << crd.display() << endl;
     }
}
```

In the next section, we're going to start programming in the ability for the end user to draw new cards, replacing each of those she wants to discard, while keeping others. We're going to need a place to hold all this information. That's why the array is needed.

EXERCISES

Exercise 15.1.1. Looking at the code, you should be able to tell what the default action is in response to the query "Play again?" What is this default action?

Exercise 15.1.2. Rewrite the program so that there is no default action. In other words, require the user to type either a "Y" or "N." Keep querying the user until she responds accordingly.

Exercise 15.1.3. Rewrite the program so that it shuffles up after each and every hand. (Hint: this might involve some revision to the Deck class.)

Exercise 15.1.4. If they are available on your system, use character-based graphics to print symbols for suits: ♣ ♦ ♥ ♠. You can then print out the card values in this simple format: "A ♠" (ace of spades). Not all systems have these symbols available. If they are, they are at ASCII character values 3, 4, 5, and 6. You can access these values as "\3", "\4", and so on. IBM machines and clones based closely on IBM support these.

The Vector Template

Chapter 13, "Easy Programming with STL," introduced two of the more useful templates from the C++ Standard Template Library (STL), the **list** and **stack** templates. Another one of the most useful templates is the **vector** template. As with the **list** and **stack**, we can build a vector container for any underlying type we want. For example:

```
vector<int>     vec_of_ints;
vector<string>  vec_of_strings;
vector<double>  vec_of_flts;
vector<Card>    vec_of_objs;
```

A vector, for nearly all intents and purposes, is like an array, only better. It can grow without limit, other than the limit of physical memory imposed by the computer itself.

After declaring an array, you can add elements to it by calling its **push_back** function. For example:

```
vector<int>  iVec;
iVec.push_back(10);
iVec.push_back(20);
iVec.push_back(30);
```

The result of these statements is a vector of integers (iVec) that holds the values 10, 20, 30, just as an integer array might hold these values. You can go ahead and index the vector just as you would an array:

```
cout << iVec[0] << " ";
cout << iVec[1] << " ";
```

You can use iterators to access a vector's elements, just as shown for **list** containers in Chapter 13. But an even simpler way is just to index a vector as you would an array, getting the length by calling the **size** function.

```
for (int i = 0; i < iVec.size(); ++i) {
     cout << iVec[i] << endl;
}
```

The code shown so far creates and prints out a vector that is currently of size 3. But new elements can be added at any time without fear of exceeding size limits (because a vector grows as needed). After this next statement, the size of the vector will be 4.

```
iVec.push_back(55);
```

Finally, one of the most convenient features of vectors is that you can clear their contents at any time, resetting the size of the vector to 0.

```
iVec.clear();   // Erase contents and start over.
```

C++14 ▶ The next paragraph applies to C++14 compilers only. (Actually, the feature was introduced in C++11 but it took time for some vendors, including Microsoft, to support it.)

With C++14-compliant compilers, you can initialize most STL containers, including **vector** containers, by using a comma-separated list, just as you can with arrays. For example:

```
vector<int>  iVec = {1, 2, 3, 4, 5];
```

Getting Nums from the Player

With most programs, one of the most important issues is user interface. Just what do we want the end user's experience to be?

We'd like to give the user the ability to select any combination of cards to keep from the five dealt, and (what amounts to the same thing, just from another angle) the ability to select any combination of cards to discard.

One way to do that is to query the user about each card separately.

```
Do you want to redraw card Number 1? Y
Do you want to redraw card Number 2? N
Do you want to redraw card Number 3? Y
...
```

But that would be boring, and who wants to bore people? We'd like to enable the user to enter all the requests on a single line, like this (again with user input in bold):

```
Enter #'s of cards to discard: 1, 3, 5
```

Yet we can do even better. Why require commas between numbers, or even spaces? Two-digit numbers are not a possibility here, because the only valid choices are all one-digit each: 1, 2, 3, 4, or 5. We should, therefore, let the user just run the choices together, which after a while, she'll want to do.

```
Enter #'s of cards to discard: 135
```

There's an easy way to support this entry system. You can index **string** objects to get individual characters, just as you can with C-strings. For example, you can scan a string for the digit characters "1" through "5," and print out each one found.

As explained in Chapter 8, "Strings: Analyzing the Text," if you index either a C-string or a C++ **string** object, you get a single value of type **char**. This value is actually a number that translates into a character when printed. Table E.1 on page 514 lists the ASCII character codes.

```
// Scan a digit string and print '1' thru '5' only.

for (int i = 0; i < sInput.size(); ++i) {
     int n = sInput[i] - '0';
     if (n >= 1 && n <= 5) {
          cout << n << " ";
     }
}
```

This loop works because although you may not know what the ASCII value of the digits are, you can rely on their relative order in the sequence. So, if a character is '0', subtracting '0' produces the number 0, obviously. But it's also true that if a character is '1', subtracting '0' produces the number 1. If a character is '2', subtracting '0' produces the number 2, and so on. Therefore, subtracting the ASCII value '0' converts to the numeric value of the digit.

We can use that fact to recognize the digits 1 through 5, and add the equivalent index number 0 through 4 to a vector of such numbers (because C++ has zero-based arrays).

```
for (int i = 0; i < sInput.size(); ++i) {
     int n = sInput[i] - '0';
     if (n >= 1 && n <= 5) {
          selVec.push_back(i - 1);
     }
}
```

Example 15.2. *Draw Poker*

Using the **vector** template and the ability to scan strings for individual digits, we can at last write the version of video poker that enables a player to discard any combination of cards and draw new ones.

Lines of code that are new or changed from Example 15.1 are printed here in bold.

Poker2.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <vector>
using namespace std;

// INCLUDE THE DECK AND CARD CLASS
// DECLARATIONS AND DEFINITIONS HERE.

Deck my_deck;
Card aCards[5];
bool aFlags[5];
vector<int> selVec;

void play_game();
bool draw();

main() {
    string s;
    while (true) {
        play_game();
        cout << "Play again? (Y or N): ";
        getline(cin, s);
        if (s[0] == 'N' || s[0] == 'n') {
            break;
        }
    }
    return 0;
}

void play_game() {
     for (int i = 0; i < 5; ++i) {
         aCards[i] = my_deck.deal_a_card();
         aFlags[i] = false;
         cout << i + 1 << ". ";
         cout << aCards[i].display() << endl;
     }
     cout << endl;
```

Poker2.cpp, cont.

```
     // Draw new cards, and then re-display

     if (draw()) {
         for (int i = 0; i < 5; ++i) {
             cout << i + 1 << ". ";
             cout << aCards[i].display();
             if (aFlags[i]) {
                 cout << " *";
             }
             cout << endl;
         }
         cout << endl;
     }
}

bool draw() {
    string sInput;
    selVec.clear();
    cout << "Input #'s of cards to redraw: ";
    getline(cin, sInput);
    if (sInput.size() == 0) {
        return false;
    }
    // Read input string, adding an
    //  element to selVec for each digit read.

    for (int i = 0; i < sInput.size(); ++i) {
        int n = sInput[i] - '0';
        if (n >= 1 && n <= 5) {
            selVec.push_back(n - 1);
        }
    }
    // For each number (0-4) in selVec, redraw
    //  the corresponding card.

    for (int i = 0; i < selVec.size(); ++i) {
        int j = selVec[i];  // Select a card
        aCards[j] = my_deck.deal_a_card();
        aFlags[j] = true;
    }
    return true;
}
```

Here's a sample session. You should notice the difference, which is that the player can redraw any combination of cards! As in earlier sample sessions, user input is in bold.

```
1. A of clubs.
2. 10 of diamonds.
3. K of spades.
4. 3 of spades.
5. A of hearts.

Enter #'s of cards to redraw: 234
1. A of clubs.
2. A of diamonds. *
3. 7 of diamonds. *
4. 7 of clubs. *
5. A of hearts.

Play again? N
```

Wow, a full house: three aces and two 7's! Wouldn't it be nice if the program could recognize that fact and reward you accordingly? That's what we're going to add to the program in the last part of the chapter.

How It Works

The first thing the new version of the program does is to declare some new data structures:

```
bool aFlags[5];
vector<int> selVec;
```

The aFlags array is a set of five flags, one corresponding to each card in the hand. When a flag is set to **true**, that means that card has been redrawn. This makes it easy to print an asterisk (*) next to each card that represents a redraw.

Next, the play_game function was changed so it calls the draw function, which gets input from the user as to what cards should be redrawn, if any. If the user presses ENTER but enters no input, it's assumed that she simply wants to keep the cards she has, and no further action is taken on this hand. A return value of **false** indicates this condition.

```
    if (draw()) {

      // Reprint the hand...

    }
```

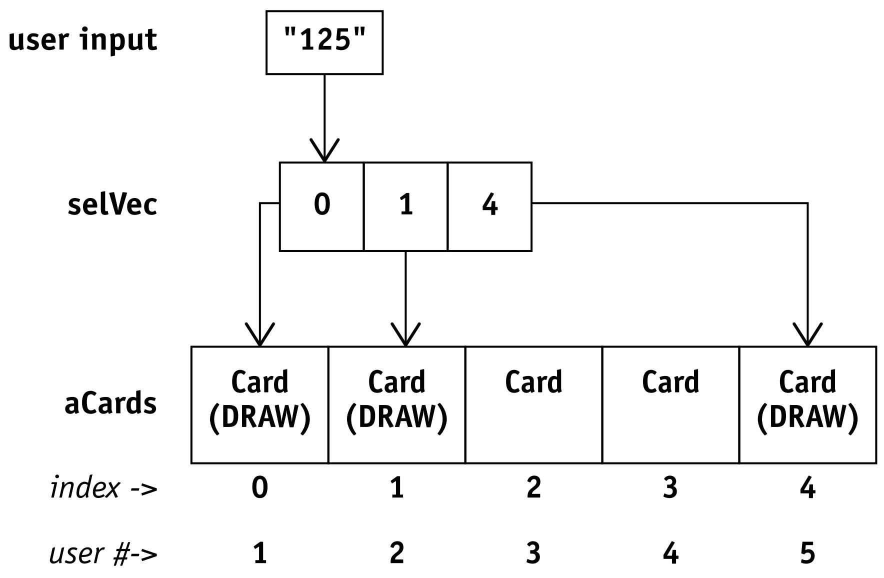
The draw function has two major things to do, each accomplished with a simple loop: 1) inquire from the user which cards are to be redrawn and 2) redraw those cards, identified by the user as cards 1 through 5, by requesting new cards from the Deck object. These actions can be broken down into the following pseudocode:

```
Prompt user for input string
For each character in input string
    If character is a digit 1 through 5
    Push N-1 onto selVec
For each element of selVec
    Set J to the current element of selVec
    Replace aCards[J] with a new card
```

These loops might be a little easier to understand with an example. Suppose the user enters the input string "125". The first loop subtracts 1 from each of these numbers and produces a vector with the following values:

0 1 4

The second loop then steps through this vector (which is very much like an array, remember) and redraws three cards, replacing corresponding elements of the Cards array: aCards[0], aCards[1], and aCards[4].



EXERCISES

Exercise 15.2.1. Rewrite the play_game function so that it prints an asterisk (*) next to redrawn cards at the beginning of each line. Do it in such a way that everything else lines up.

Exercise 15.2.2. Implement selVec with the list template introduced in Chapter 13. You'll probably want to call it something else, such as selList ("selection list"). Lists cannot be indexed the way vectors and arrays can, but you can step through a list using the techniques discussed in the first half of Chapter 13.

Exercise 15.2.3. Implement selVec as an ordinary C++ array. (Hint: you'll need to have an absolute limit, as well as adding a variable that tracks the current number of cards selected for redrawing.)

Exercise 15.2.4. Prevent any card from being redrawn more than once. Right now, the user can enter "333" and the result is that Card Number 3 will be replaced over and over. The ultimate effect will be the same as if it were drawn only once, but this is inefficient, and furthermore, if you're counting down the deck, there will be cards you never see. Prevent this from happening so that if the user types "333", the card will be redrawn, but only once.

How to Evaluate Poker Hands

Now we come to the most interesting challenge of all. It's not immediately obvious how a computer program could look at a group of (mostly) unarranged cards and detect whether there are three of a kind, a flush, or a pair, but it shouldn't be impossible, either.

And it's actually not that difficult. In most cases, what's involved is just counting repetitions of things. For example, if we count four of any rank, the hand is four of a kind. What we need is to count the occurrences of all 13 ranks and 4 suits, and track all this information somewhere.

This is where—as has so often been the case—arrays come to our rescue. Two simple arrays of integers can be used to track the counts.

```
int rankCounts[13];
int suitCounts[4];
```

After these arrays are initialized to all-zero values, it's easy to use them to count all the repetitions of ranks and suits.

```
    for (int i = 0; i < 5; ++i) {
         int r = aCards[i].rank;
         int s = aCards[i].suit;
         ++rankCounts[r];
         ++suitCounts[s];
    }
```

Consider the hand A-A-A-5-5, in any order (a full house). After the program counts the number of cards of each rank, the resulting rankCounts array—too big to fit on just one line—looks like this:

0	0	0	**2**	0	0	0	0	0	0	0	0	**3**
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**J**	**Q**	**K**	**A**

Or, suppose that the hand is A-K-Q-J-10, an ace-high straight, again, in any order. Here's what the rankCounts array looks like after counting each of these cards:

0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**J**	**Q**	**K**	**A**

Now we can evaluate hands by examining these two arrays, rankCounts and suitCounts. There's a lot to do here, only because there are so many kinds of hands to consider.

In the interest of modular design, we're going to put all the hand-evaluation code in yet another class, which I'll name "Eval." As with the other classes, we'll create just one object and call functions through that object.

What's gained by putting everything in a class? You could write all these functions and data separately. The main advantage is that when you look at the Eval class, it's clear that all these functions and data are intended to be used together, to be part of the same module, as it were. They aren't isolated pieces of the program.

And there's another advantage: Private members are protected from outside access, removing the temptation of class users to "reach in" and tinker with internals, creating hidden dependencies and potential bugs.

Here's the code listing for the Eval class. It may seem long, but most of the individual functions are relatively short and simple to understand.

Note There is one hand this class fails to recognize. In poker, an ace is either high or low, as most benefits the player. In practice, this almost always means an ace will be high, but there is one exception: the hand A-2-3-4-5, in any order, constitutes a 5-high straight and is nicknamed "a bicycle." Recognizing this hand is left as an exercise.

eval.cpp

```
// Note that enclosing program must include <string>
//  as well as using namespace std;
//
class Eval {
public:
    Eval(Card* pCards);
    string rank_hands();
private:
    int rankCounts[13];
    int suitCounts[4];
    int has_reps(int n);
    bool is_straight();
    bool verify_straight(int n);
    bool is_flush();
    bool is_two_pair();
};

Eval::Eval(Card* pCards) {
    for (int i = 0; i < 13; ++i) {  // Clear arrays
        rankCounts[i] = 0;
    }
    for (int i = 0; i < 4; ++i) {
        suitCounts[i] = 0;
    }
    for(int i = 0; i < 5; ++i) {    // Init arrays
        int r = pCards[i].rank;
        int s = pCards[i].suit;
        ++rankCounts[r];
        ++suitCounts[s];
    }
}
```

eval.cpp, cont.

```
string Eval::rank_hands() {
    string s;
    if (is_straight() && is_flush()) {
        if (ranksCount[12] && ranksCount[11]) {  // A&K
          s = "You have a ROYAL FLUSH! PAYOUT = 800";
        }else {
          s = "You have a STRAIGHT FLUSH! PAYOUT = 50";
        }
    } else if (has_reps(4)) {
        s = "You have FOUR OF A KIND!  PAYOUT = 25";
    } else if (has_reps(3) && has_reps(2)) {
        s = "You have a FULL HOUSE!  PAYOUT = 9";
    } else if (is_flush()) {
        s = "You have a FLUSH!  PAYOUT = 6";
    } else if (is_straight()) {
        s = "You have a STRAIGHT!  PAYOUT = 4";
    } else if (has_reps(3)) {
        s = "You have three of a kind.  PAYOUT = 3";
    } else if (is_two_pair()) {
        s = "You have two pair.  PAYOUT = 2";
    } else if (has_reps(2)) {
        s = "You have a pair.  PAYOUT = 1";
    } else {
        s ="You have no pair.  PAYOUT = 0";
    }
    return s;
}

// Has reps function.
// Return true if any rank is repeated
//   the specified number of times.

int Eval::has_reps(int n) {
    for(int i = 0; i < 13; ++i) {
        if (rankCounts[i] == n) {
            return true;
        }
    }
    return false;
}
```

▼ continued on next page

eval.cpp, cont.

```
// Is straight function.
// Look for first "singleton" in the ranks,
//   and then verify if it begins a straight.

bool Eval::is_straight() {
    for (int i = 0; i < 8; ++i) {
        if (rankCounts[i] == 1) {
            return verify_straight(i);
        }
    }
    return false;
}

bool Eval::verify_straight(int n) {
    for (int i = n + 1; i < n + 5; ++i) {
        if (rankCounts[i] != 1) {
            return false;
        }
    }
    return true;
}

bool Eval::is_flush() {
    for(int i = 0; i < 4; ++i) {
        if (suitCounts[i] == 5) {
            return true;
        }
    }
    return false;
}

bool Eval::is_two_pair() {
    int n = 0;
    for(int i = 0; i < 13; ++i) {
        if (rankCounts[i] == 2) {
            ++n;
        }
    }
    return n == 2;
}
```

Example 15.3. *Draw-Poker Payout!*

The following code listing presents the main program with the changes necessary to interact with the Eval class. (None of the class declarations and definitions are included here.) Only two lines need to be added to use Eval, and these are printed in bold.

Poker3.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <vector>
using namespace std;

// INCLUDE DECK, CARD, AND EVAL CLASS
// DECLARATIONS AND DEFINITIONS HERE.

Deck my_deck;
Card aCards[5];
bool aFlags[5];
vector<int> selVec;

void play_game();
bool draw();

main()
{
    string s;
    while (true) {
        play_game();
        cout << "Play again? (Y or N): ";
        getline(cin, s);
        if (s[0] == 'N' || s[0] == 'n') {
            break;
        }
    }
    return 0;
}
```

▼ *continued on next page*

Poker3.cpp, cont.

```
void play_game() {
     for (int i = 0; i < 5; ++i) {
         aCards[i] = my_deck.deal_a_card();
         aFlags[i] = false;
         cout << i + 1 << ". ";
         cout << aCards[i].display() << endl;
     }
     cout << endl;

     // Draw new cards, and then re-display

     if (draw()) {
         for (int i = 0; i < 5; ++i) {
             cout << i + 1 << ". ";
             cout << aCards[i].display();
             if (aFlags[i]) {
                 cout << " *";
             }
             cout << endl;
         }
         cout << endl;
     }
     Eval my_eval(aCards);
     cout << my_eval.rank_hand() << endl;
}

bool draw() {
    string sInput;
    selVec.clear();
    cout << "Input #'s of cards to redraw: ";
    getline(cin, sInput);
    if (sInput.size() == 0) {
        return false;
    }
    // Read input string, adding an
    //  element to selVec for each digit read.

    for (int i = 0; i < sInput.size(); ++i) {
        int n = sInput[i] - '0';
```

Poker3.cpp, cont.

```
          if (n >= 1 && n <= 5) {
              selVec.push_back(n - 1);
          }
      }
      // For each number (0-4) in selVec, redraw
      //  the corresponding card.

      for (int i = 0; i < selVec.size(); ++i) {
          int j = selVec[i];
          aCards[j] = my_deck.deal_a_card();
          aFlags[j] = true;
      }
      return true;

  }
```

Here's a sample session of this (finally) finished program. As in earlier sample sessions, user input is in bold.

```
1. J of clubs.
2. 10 of diamonds.
3. J of spades.
4. 3 of spades.
5. J of hearts.

Enter #'s of cards to redraw: 2 4
1. J of clubs.
2. J of diamonds. *
3. J of spades.
4. 7 of clubs. *
5. J of hearts.

You have FOUR OF A KIND! PAYOUT = 25
Play again? N
```

How It Works

All the work of evaluating the hand is done by the Eval class, so almost nothing needs to be added to the main program.

Within the Eval class, an enormous amount of work is done by the has_reps member function, even though it's relatively simple. All this function does is detect whether any rank of card managed to achieve the specified number of occurrences. For example, if it's passed the value 4, then it returns true if and only if there is an element of ranksCount equal to 4. This would indicate that a four of a kind is present.

```
int Eval::has_reps(int n) {
    for(int i = 0; i < 13; ++i) {
        if (rankCounts[i] == n) {
            return true;
        }
    }
    return false;
}
```

With this function in place, the rest of the Eval class is straightforward, even if it seems long. Only the detection of straights presents any real difficulty. There are several ways this problem can be solved, but I chose a way that's fairly easy to program. It amounts to a double algorithm. First, it's necessary to find where a straight might begin. Then, we verify whether or not the rest of the straight is present.

```
For I = 0 to up and including 8
   If aCards[I] equals 1
      Return the value of verify_straight(I)
Return false
```

In other words, starting with the first position in the array, try to find an element of the countRanks array exactly equal to 1. If such an element is found, verify that the next four cards after this one fill out a straight (so we're looking for another four singletons in a row). We do this by calling the verify_straight function.

```
For I = N + 1 up to but not including N + 5
   If aCards[I] does NOT equal 1
      Return false
Return true
```

EXERCISES

Exercise 15.3.1. Enable the rank_hands function to return payout information numerically. (Hint: you'll need to add another argument to the function declaration.) Then, keep track of the player's bank account for the duration of the game, telling her how much she has after each hand. Note that it costs 1 unit to play, therefore a payoff of 0 causes a loss of one unit, and a payoff of 1 is actually break-even. Start the initial bank at 100.

Exercise 15.3.2. Revise the rank_hands function so that it recognizes A-2-3-4-5 as a straight. This is called a "bicycle" and it is the lowest straight in poker, ranking just below 2-3-4-5-6. (Again, remember that order does not matter.)

Exercise 15.3.3. Revise the rank_hands function to recognize the special hands called Big Tiger and Little Tiger. A suggested payout is 4. These hands, which are only allowed when house rules permit them, rank just above a straight. Both hands are special no-pair hands. A Big Tiger has a king as its highest card and an 8 as its lowest. A Little Tiger has an 8 as its high card and a 3 as its lowest.

Chapter 15 *Summary*

The main purpose of this chapter has been to reinforce and provide more examples of class- and object-writing techniques presented earlier, but it did present some new ideas, summarized here:

- An object type—that is, a class—can be a return type just as any other type (such as primitive data) can. You declare the return type the same way: at the beginning of the function declaration. For example:

```
Card deal_a_card();
```

- To return an object from a function, it's often helpful to call a class constructor.

```
return Card(r, s);   // Return a Card object.
```

- You can have arrays of objects just as you can have arrays of primitive data. Such arrays can even be placed inside other class declarations—in which case you have an object containing other objects.

```
Card  aCards[5];
```

- You can optionally use the **swap** or **random_shuffle** algorithm instead of writing your own shuffling routine. To use a C++ STL algorithm, make sure to include <algorithm>.

```
#include <algorithm>
```

- The **vector** template is one of the most useful parts of the STL. It provides containers that are very similar to arrays and can be indexed just as arrays are, but they grow without limit. To use this template, include <vector>.

```
#include <vector>
```

- You can then build vector containers on any kind of type. For example:

```
vector<int>     iVec;
vector<double>  fVec;
```

- To populate a vector, you can call the **push_back** function, which adds an element to the end of the vector.

```
vector<int>    my_vec;
my_vec.pushback(100);
my_vec.pushback(200);
my_vec.pushback(1000);
```

- Then, using references to the size of the vector (obtained by calling its **size** function), you can use indexing to iterate through the vector.

```
for (int i = 0; i < my_vec.size(); ++i) {
     cout << my_vec[i] << endl;
}
```

- A vector can be cleared by calling its **clear** function.

```
my_vec.clear();
```

16 Polymorphic Poker

If the only thing that object orientation accomplished was to encourage a more modular programming style, grouping tightly related code and data together, it would still be worthwhile. But there's more.

The central idea in OOP is that of an intelligent data type. Simply making a function into a member function is only a beginning. Ideally, objects bring with them the knowledge of which function to call; you should be able to switch out one object, switch in another, and get new behavior without changing anything else. You can even switch objects at runtime and get new behavior.

Sound like a dream? But that's what this chapter is going to explore. We'll start by examining the Deck class used in the previous chapter, "Object-Oriented Poker."

Multiple Decks

When playing the video poker game of Chapter 15, there may be times you'll want to use a deck that behaves differently.

You might want to do this for testing purposes. While reading Chapter 15, for example, it may have occurred to you that royal flushes occur so rarely they are nearly impossible to get, unless you play the game for thousands of hours, and even then, you might not ever get one. The odds of getting one on the initial deal are 1 in 649,170!

This creates a problem for your testing department. The probability of a royal flush is less than one in half a million, so how are you going to verify that your program will recognize a royal flush when it does come up?

One answer is specialized decks. You could create a variation on the Deck class that produces cards from a stacked Deck, in which the first five cards are A-K-Q-J-10. Another approach is to use a pinochle deck, which only uses ace through nine, but it features two copies of each such card. High-value hands are more likely to occur with such a deck.

The declaration and implementation of a pinochle deck is as follows: lines altered from the standard Deck class are in bold. Remember this deck only has 48 cards in it, and we can imagine cards 24 through 47 repeating the ranks and suits of cards 0 to 23.

```
class PinochleDeck {
public:
    PinochleDeck();
    Card deal_a_card();
private:
    int cards[48];
    int nCard;
    void shuffle();
};

PinochleDeck:: PinochleDeck() {
    srand(time(NULL));
    for (int i = 0; i < 48; ++i) {
        cards[i] = i;
    }
    shuffle();
}

void PinochleDeck::shuffle() {
    nCard = 0;
    for (int i = 47; i > 0; --i) {
        int j = rand() % (i + 1);
        int temp = cards[i];
        cards[i] = cards[j];
        cards[j] = temp;
    }
}

Card PinochleDeck::deal_a_card() {
    if (nCard > 47) {
        cout << endl << "RESHUFFLING..." << endl;
        shuffle();
    }
    int r = (cards[nCard] % 6) + 7;    // r = 9 thru A

    // Divide deck in half (%24), then divide by 6
    //  to produce suit values 0 thru 3.
```

```
        int s = (cards[nCard++] % 24) / 6;
        return Card(r, s);
    }
```

How would you switch to this new version of the Deck class? You could add all this code to the program and then change the line

```
    Deck my_deck;
```

to this:

```
    PinochleDeck my_deck;
```

Then you'd have to recompile the program, and (assuming nothing was mistyped) this would work. Calls to the deal_a_card function should be correctly resolved because C++ lets every class have a function by this name if it wants to.

```
    aCards[i] =  my_deck.deal_a_card();
```

Switching Decks at Runtime

Unfortunately, the problems for your testing department have just begun. We've created a situation in which every time testers want to switch to the pinochle Deck class, *the entire program must be recompiled*. Maybe that's not such a big hardship if you're a one-person shop, but even then, you don't want to waste your time constantly rebuilding the program.

What you need, then, is a way to switch between decks at runtime. Ideally, you'd like to change the declaration of my_deck in response to run-time conditions, and then have the following line of code call the appropriate function:

```
    my_deck.deal_a_card()
```

But no matter how you try to trick the compiler, I can assure you that this won't work. The problem is much deeper than syntax. I can potentially declare my_deck to be an object of one of many different classes.

```
    Deck          my_deck;
    PinochleDeck  my_deck;
    StackedDeck   my_deck;
    DoubleDeck    my_deck;
```

But no matter how you may try to trick the compiler, when it comes to calling a member function, the compiler has to make a definite decision about what physical address in memory to bind to. Now there's not one deal_a_card function, there are many.

We can use the scope operator (::) to clarify which version of the function is meant to be called. But that's not going to help in this situation:

```
Deck::deal_a_card()
PinochleDeck::deal_a_card()
StackedDeck::deal_a_card()
DoubleDeck::deal_a_card()
```

There's a makeshift solution to the problem, but it's not a great one. First, you could use **#define** directives to indicate the various decks of decks:

```
#define DECK52    0
#define PIN_DECK  1
#define DBL_DECK  2
```

Then, you could include all the different deck types in the program—meaning you need to create one deck of each object type, even though you're only going to use one of these types. This is highly inefficient and wasteful of resources.

```
Deck          my_deck;
PinochleDeck  my_pin_deck;
DoubleDeck    my_dbl_deck;
```

Finally, whenever the program wanted to call the deal_a_card function, it would have to use an intermediate function with a **switch** statement to determine which version of the function to call.

```
Card get_a_card() {
     switch(deck_selector) {
     case DECK52:
          return my_deck.deal_a_card();
     case PIN_DECK:
          return my_pin_deck.deal_a_card();
     case DBL_DECK:
          return my_dbl_deck.deal_a_card();
     }
}
```

It may work, but it's got problems. We need to find a better answer.

Polymorphism Is the Answer

The solution offered at the end of the last section is a poor one. It may work in some cases, but it makes for extra coding, as well as inefficiency. Worse yet, every time a new deck type is added to the project, new lines of code have to be added to the main program and everything has to be recompiled.

And it gets worse still. If an object is heavily used throughout the main program, and not just for one function, you might need a proliferation of **switch** statements throughout your program.

What we'd really like would be a way to call the deal_a_card function and have it automatically call the appropriate implementation for that object, even if its precise type is not known ahead of time.

```
my_deck.deal_a_card();    // Always works!
```

In computer science, such a function is called *polymorphic*, meaning "many forms." More accurately, what's meant here is *unlimited forms*; there is no limit to the number of different ways that deal_a_card may be implemented by different classes. If the function is polymorphic then—as if by magic—the correct version of that function gets called at runtime.

In C++, polymorphic functions are supported but carefully controlled. There are two absolute requirements:

- The classes involved must be related through inheritance. One must be derived from another, or they must both be derived from a common base class.
- The function must be declared **virtual** in the base class.

To digest all this, you need to understand inheritance. It's a way of creating a class that automatically inherits all the members of another class, called the base class. For example, if you wanted to create a variation on the Deck class with one extra function, you could do it this way:

```
class MyDeckClass : public Deck {
public:
     int cards_remaining(); // Code to be provided.
};
```

In this example, MyDeckClass automatically has all the members that the Deck class does, plus one more. However, in this particular case, the class will probably fail miserably because it has no access to private members of Deck. That's why there's a third access level, **protected**, which grants access to all derived classes (including "descendant" classes).

The other requirement is that the function be declared **virtual**—but the **virtual** keyword need only be applied once per function, in the base class. There's an important general rule here:

Any function that might be overridden by a derived class should be declared virtual.

This is the most important rule regarding virtual functions. There are some others; for example, you can make an inline function into a virtual function, but the compiler will only expand such a function when it is "safe" to do so—that is, when the exact type of the object can be fixed at compile time.

Another rule is that constructors cannot be made virtual. Constructors, by the way, are problematic for inheritance in general. They are the only kind of member not automatically inherited (although it's possible to specify an inherited constructor in C++11 and later), so that generally speaking, each class that needs constructors must supply its own.

To declare a function virtual, just precede the function declaration with the keyword **virtual**.

Keyword

```
virtual function_declaration;
```

This should only be done in the base class. For example, if you wanted to make it possible for programmers (yourself or others), to derive classes from the Deck class and implement their own version of deal_a_card, declare it this way:

```
class Deck {
public:
    Deck();
    virtual Card deal_a_card();
private:
    int cards[52];
    int nCard;
    void shuffle();
};
```

Then, the PinochleDeck class needs to be derived from the Deck class. If this is done, it relates the two classes through inheritance. Polymorphism then becomes possible.

```
class PinochleDeck : public Deck {
public:
    PinochleDeck();
    Card deal_a_card();  // Automatically virtual! This
                         //  funct is virtual because
                         //  it was declared so in the
                         //  base class.
private:
    int cards[48];
    int nCard;
    void shuffle();
};
```

Another approach to polymorphism is to derive the deck classes from a common base class, or "interface." The interface, being an abstract class, cannot be instantiated. But you can pass the address of an object of a derived type to a pointer of the base type. That is to say, you can 1) create an object, 2) take its address, and 3) pass that address to a pointer, even though the pointer is of the base-type class (the interface). For example:

```
IDeck *pDeck;          // Point to the base type, IDeck.

// Create an object of derived type and assign its
//   address to the pointer, pDeck.

pDeck = new PinochleDeck;
...
aCards[i] = pDeck->deal_a_card();
```

This example uses the pointer-dereference-and-member-access operator (–>) introduced in Chapter 12, "Two Complete OOP Examples." This operator dereferences a pointer and then accesses a member, so the last statement in this example is equivalent to:

```
aCards[i] = (*pDeck).deal_a_card();
```

The important point here is that pDeck can be assigned to point to *any* object at runtime as long as the object's class is derived from IDeck. If that's the case, and if deal_a_card is declared **virtual**, then the call to deal_a_card will always do the right thing: it will call the deal_a_card function defined for the object's own class.

The importance of this feature can't be overstated. A pointer of interface (that is, base-class) type can point to an object of a derived class at compile time, or it can point to different kinds of objects at runtime in response to changing conditions, such as a user's selection.

```
IDeck *pDeck;

if (strSel == "standard") {
     pDeck = new Deck;
} else (strSel == "pinochle") {
     pDeck = new Pinochle_Deck;
}
```

In IDeck, the deal_a_card function is declared **virtual**. For that reason, no matter what object's address is assigned to pDeck (assuming it is a legal assignment), the following statement will always call the correct implementation of the function.

```
Card crd = pDeck->deal_a_card();
```

Example 16.1. *A Virtual Dealer*

Ideck.cpp

```
// THE CARD CLASS MUST BE DECLARED FIRST,
//  AS IDECK REFERS TO THAT TYPE. SEE
//  CHAPTER 15.

class IDeck {
public:
     virtual Card deal_a_card() = 0;
};

class PinochleDeck : public IDeck {
public:
    PinochleDeck();
    Card deal_a_card();
private:
    int cards[48];
    int nCard;
    void shuffle();
};

PinochleDeck::Deck() {
    srand(time(NULL));
    for (int i = 0; i < 47; ++i) {
        cards[i] = i;
    }
    shuffle();
}

void PinochleDeck::shuffle() {
    nCard = 0;
    for (int i = 47; i > 0; --i) {
        int j = rand() % (i + 1);
        int temp = cards[i];
```

Ideck.cpp, cont.

```
            cards[i] = cards[j];
            cards[j] = temp;
        }
}

Card PinochleDeck::deal_a_card() {
    if (nCard > 47) {
        cout << endl << "RESHUFFLING..." << endl;
        shuffle();
    }
    int r = (cards[nCard] % 6) + 7;    // r = 9 thru A

    // Divide deck in half (%24), then divide by 6
    //  to produce suit values 0 thru 3.

    int s = (cards[nCard++] % 24) / 6;
    return Card(r, s);
}
```

How It Works

The important part of this example is the first few lines. They set up an inheritance relationship and make deal_a_card into a virtual function, so that the correct version of that member function is always called at runtime.

```
class IDeck {
public:
     virtual Card deal_a_card() = 0;
};

class PinochleDeck : public IDeck {
...
```

You could declare any number of other decks and they would become related through this inheritance hierarchy, assuming that they also were derived from IDeck. To be instantiated, by the way, a derived class must first supply its own implementation of the deal_a_card function.

Note For technical reasons not worth explaining, the keyword “public” should precede the name of the base class in the first line of the derived class declaration. You can use “private” or “protected” in this context, but that’s an advanced technique that many programmers never actually use.

Remember that the ultimate purpose of inheritance and virtual functions is that 1) any object can be selected at runtime, provided that it's of a class derived from a common base type, and 2) the right implementation of each virtual function will be called.

A silly (but informative) example might be several "animal" classes derived from a common Animal class.

```
class IAnimal {                // Base class.
public:
     virtual void speak() = 0;
};

class Dog : public IAnimal {   // Derived class.
     void speak();
};

class Cat : public IAnimal {   // Derived class.
     void speak();
};
```

A pointer of type IAnimal can point to an object of either derived class, Dog or Cat. Then, calling the object's speak() function will always do the right thing, invoking either Dog::speak or Cat::speak as appropriate.

```
IAnimal *pAnimal;
pAnimal = new Dog();
...
pAnimal->speak();  // Calls Dog::speak.
```

If this same pointer is reassigned to point to a Cat object, then the same statement calls Cat::speak instead of Dog::speak.

```
pAnimal = new Cat();
...
pAnimal->speak();  // Calls Cat::speak.
```

In this case, calling either Cat::speak or Dog::speak as appropriate may seem trivial, since we can see which kind of object pAnimal points to in this simple example. But much more complex scenarios are possible, such as having an array of IAnimal pointers. In that case, every element of the array might point to an object of a different class.

```
pAnimal *zooArray[10];
// Initialize the array to point to different animals...
```

```
for (int i = 0; i < 10; ++i) {
     zooArray[i]->speak();
}
```

In this last example, the loop causes each animal in the zoo to "speak" correctly, even though each type of object may have a different implementation of the speak function. These implementations, it must be remembered, are provided by different classes and each class may be different as long as all are derived from IAnimal.

Once again, remember that –> is the pointer-dereference-and-member-access operator, so that the loop statement in this last example is equivalent to:

```
     (*zooArray[i]).speak();
```

EXERCISES

Exercise 16.1.1. Write at least one Deck class of your own and derive it from IDeck, so that it can be related to other Deck classes in an inheritance hierarchy. Then test this out and make sure the correct version is called. At the beginning of the program, allow the user to choose between the standard Deck class and the PinochleDeck class.

Exercise 16.1.2. Write a program that features the IAnimal interface just shown, along with derived classes, Dog, Cat, and Cow, all derived from IAnimal. Test the polymorphic aspect of this design by creating an array of different "animal" objects. Then call the speak() function for each element of the array.

Interlude

What Is the Virtual Penalty?

Although it's not necessary to know how virtual function calls are implemented by C++, it's useful to understand the trade-off: Virtual functions are more flexible, but there is a small penalty to be paid. If you're really sure that a certain function will never be overridden, there is no point in making it virtual.

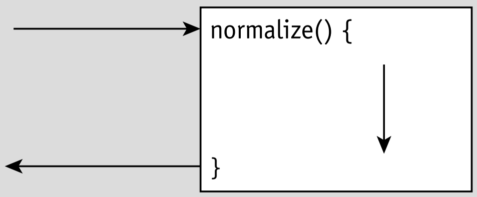
The penalty, however, is small, particularly in light of the speed and capacity of today's computers. There are actually two penalties: a performance penalty and a space penalty.

When a C++ program executes a standard function call, it does what I outlined in Chapter 5: it transfers control of the program to a specific address and returns when the function is done. This is a simple action.

▼ *continued on next page*

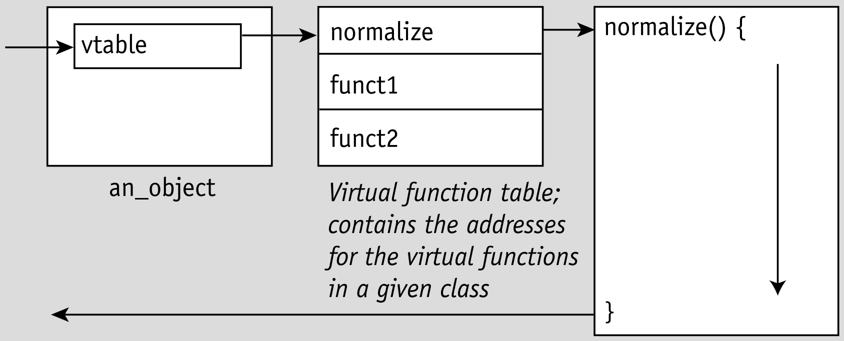
Interlude

▼ continued



Execution of a virtual function is more involved. Each object contains a hidden "vtable" pointer that points to a table of all the virtual functions for its class. (This pointer is typically called "vptr.") For example, all objects of class FloatFraction contain a vtable pointer to the table of virtual functions for FloatFraction. If a class has no virtual functions at all, by the way, its objects don't need to have a vtable pointer and that saves some space.

To call a virtual function, the program uses the vtable pointer (vptr) to make an indirect function call. This process, in effect, looks up the function address at runtime. (Remember, this is done under the covers and so is completely invisible to the C++ source code.) You can visualize the action this way:


vtable
an_object
normalize
funct1
funct2
Virtual function table; contains the addresses for the virtual functions in a given class
normalize() {
}

Because each object contains a vtable pointer, you can say that *the knowledge of how to carry out an action is built into the object itself*. The vtable pointer enables each object to have this "knowledge," because it points to implementations specific to its own class.

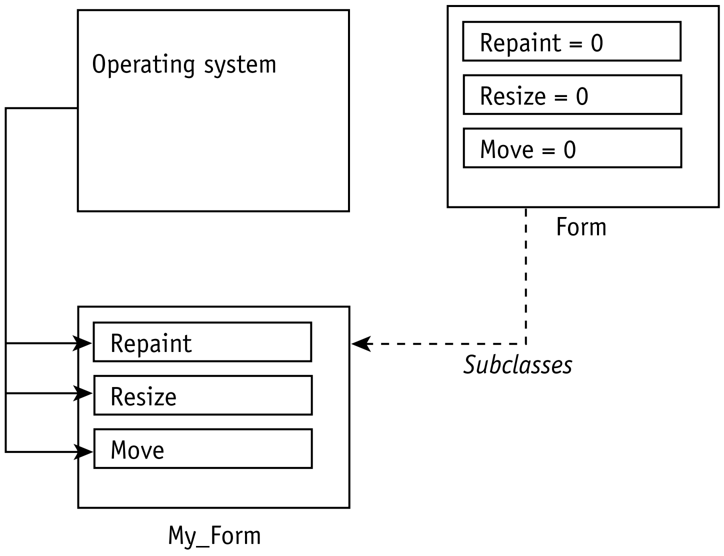
Clearly, the penalties are slight. The performance penalty arises from the greater time required to make an indirect function call (although that difference is measured in microseconds). The space penalty arises from the bytes taken up by vptr and the table itself. The moral: Make a function virtual if there's any chance it will be overridden. The cost is slight.

"Pure Virtual" and Other Abstract Matters

So, virtual functions matter. The issue is that of always getting the right implementation of a member function to execute even when that function is overridden in a derived class.

The implications of this ability go a long way. Inheritance hierarchies are deeply ingrained in development systems such as Microsoft Foundation Classes, Java, and Visual Basic.

With these systems, you subclass a general Form, Window, or Document class to create your own implementation. The operating system calls on your object (through your class declaration and implementation) to perform certain tasks—Repaint, Resize, Move, and so on. These actions are all virtual functions, which is what ensures that your functions are called using your implementation of your code.



Interfaces, or abstract classes, use *pure virtual functions*. A pure virtual function is neither required nor expected to have an implementation. You indicate a pure virtual function by using the notation =0. For example, a class might define normalize as follows:

```
class Number {
protected:
    virtual void normalize() =0;
};
```

Here, the normalize function is pure virtual. The declaration has no function definition.

Note It's possible, though not recommended, to give a definition for a function that is pure virtual… that is, to define it in the class in which the prototype includes "=0"! This may seem like a contradiction. The purpose of doing this would be to create a default implementation for the function but still have the effect of creating an abstract class, as explained in the next section. Usually, though, programmers will not provide a function definition in the base class if that function is pure virtual (=0).

Abstract Classes and Interfaces

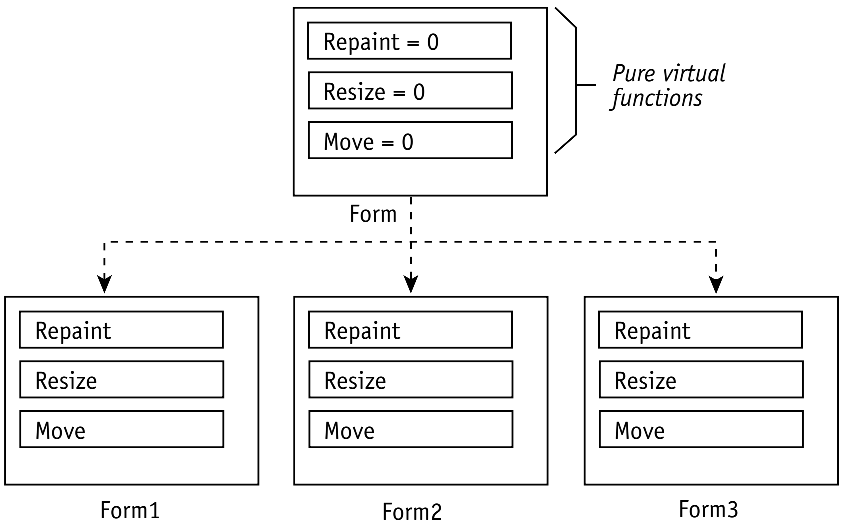
An abstract class is a class that has one or more pure virtual functions, that is, a function that includes "=0" in its prototype. An important rule is that abstract classes cannot be instantiated. This means you can't use the class to declare objects.

For example, if Number is an abstract class, trying to instantiate it produces an error.

```
Number a, b, c;   // ERROR: Number is abstract class
                  //  because it has a pure virtual
                  //  func. a, b, c cannot be created.
```

But an abstract class can be useful as a general pattern for its subclasses. Simply put, you use an abstract by deriving subclasses, implementing any virtual functions that need to be implemented, and then finally using the subclass to instantiate objects.

Suppose you have an inheritance hierarchy for Windows development and that this hierarchy includes an abstract Form class. You can subclass this to create individual, concrete forms.



Before you can use a subclass to instantiate (that is, create) objects, it must provide function definitions for all the pure virtual functions. A class that leaves even one of these functions unimplemented is abstract and therefore cannot be used to instantiate objects.

All this is useful in turn, because it gives you a way of specifying and enforcing a general set of services, or interface, according to the following rules:

- Each subclass is free to implement all these services (i.e., pure-virtual functions) in any way it wants.
- Every service needs to be implemented, or the class cannot be instantiated.
- Every class must strictly observe type information—return type and the type of each argument. This gives the inheritance hierarchy discipline so that really stupid actions (passing the wrong kind of data, for example) are flagged by the compiler.

The author of a subclass knows that he or she must implement the services defined in the interface—such as Repaint, Move, and Load in this case—but within that mandate, he or she is free. And because all these functions are virtual, the correct implementation is always executed, no matter how an object is accessed.

I'm about to show, I hope, an example of how all of this is useful.

Object Orientation and I/O

One of the best demonstrations of the power of object orientation (OOP) is the way it extends input/output through the use of the stream classes.

Once upon a time, there was the C language, which required the use of a library function called **printf** if you wanted print to the console. This function had cousins named **fprintf** (print to a text file) and **sprintf** (print to a string).

```
printf("Here's an int: %d", i);   // Print an integer
printf("Here's a flt pt: %f", x); // Print a double
```

The problem with these functions is that if you create your own data type—say a Fraction class or complex-number class—there is no way to extend **printf** to work with your class; **printf** and its cousins have a fixed set of data formats (%d, %f, %s, and so on), and these can never be modified.

You could, in theory, redefine **printf** by using **#define** to intercept calls, substitute your own function, and then call **printf** yourself through a function pointer when you needed to, but this is a horrendous "hack," requiring large amounts of ugly programming.

cout Is Endlessly Extensible

C++ introduced the I/O stream classes, although it still supports the old C functions for backward compatibility. The stream classes demonstrate the extensibility of OOP.

As you'll see in Chapter 18, making a class "printable" is simply a matter of writing an `operator<<` function. For example, you can write such a function for the Fraction class:

```
ostream &operator<<(ostream &os, const Fraction &fr) {
    os << fr.num << "/" << fr.den;
    return os;
}
```

This operator function makes Fraction objects printable in many contexts, not only with console output (**cout**) but also with any file or string using the I/O stream classes.

```
Fraction fr1(1, 2);        // fr1 = 1/2.

cout << fr1;                         // Print to console.
fout << "The value is: " << fr1;  // Print to file.
```

So, in theory, one might say that for any class of object, the following statement can be made to work smoothly:

```
cout << "The value of the object is " << an_object;
```

But cout Is Not Polymorphic

However, although this is not at first obvious, there is a limitation. Stream classes can work with an object only if its type is known at compile time. The client code must know all about the object's class.

But isn't that always true? How can you even refer to an object whose type isn't fully defined?

Actually, it's quite possible to refer to an object whose type isn't defined. For example, you could use a **void** pointer. If you use such a pointer and dereference it, **cout** will not know how to print the object.

```
void *p = &an_object;
cout << *p;            // ERROR! *p cannot be printed
```

Ideally, you ought to be able to specify a dereferenced pointer to an object (that is, an expression such as *p) and have the object always be printed in the correct format. Another way of saying this is, *the knowledge of how to print an object ought to be built into the object itself.*

This would involve not a *void** pointer but rather a pointer to a general interface, which we might call IPrintable:

```
IPrintable *p = &an_object;
```

The ability to use such pointers is important in systems programming. You might get a pointer to a new type of object over the Internet. You'd like to ensure that the correct function code is called, even if the object has a new type that the client code (the user of the object) knows nothing about.

In short, you'd like your programs to work seamlessly with new data classes to be defined in the future.

To do this, we can declare the abstract class IPrintable with one pure virtual function named print_me. In the next example, I show that any class that subclasses IPrintable and implements print_me can be correctly printed by **cout** (or any instance of an **ostream** class)—even if the class is newer than the client code—so that the specific class of the object isn't even known by the main program.

The following statements will work, even though nothing at all may be known about the class of an_object beyond the fact it subclasses IPrintable.

```
IPrintable *p = &an_object; // Object's class must
                            //  subclass Printable.

cout << *p;     // This will be printed in the
                //  correct format,
                //  as defined by
                //  the class of an_object.
```

There's an important rule that makes this code possible: a pointer to an object of subclass type can be passed to a pointer of base-class type. Or, to put it more simply:

✱ **Something specific (a subclass) can always be passed to something more general (a base class).**

The converse is not true (passing a base-class pointer to a subclass pointer) unless there is a conversion function to support it.

Example 16.2. *True Polymorphism: The IPrintable Class*

This next example demonstrates a way to work with output stream classes and objects (such as **cout**) that is truly polymorphic. By observing a general interface—realized here as the abstract class named IPrintable—you can correctly print any kind of object, even if the exact type of that object is not known at compile time.

This approach is polymorphic because a single function call can result in an unlimited number of implementations at runtime. The number of possible responses is theoretically infinite.

That may seem impossible, but I meant what I wrote. You can print an object without knowing its type or its function code because you (that is, the client code) don't need to know how to print the object. The knowledge of how to be printed is built into the object and its class.

Printme.cpp

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

class IPrintable {
    virtual void print_me(ostream &os) = 0;

    friend ostream &operator<<(ostream &os,
                         const IPrintable &pr);
};


// Operator<< function:
// All this does is cause virtual function print_me
//  to be called, sending output to the stream.
//
ostream &operator<<(ostream &os, const IPrintable &pr) {
    pr.print_me(os);
    return os;
};

// CLASSES SUBCLASSING PRINTABLE
//-------------------------------------------

class P_int : public IPrintable {
public:
    int n;

    P_int() {};
    P_int(int new_n) {n = new_n; };
```

Printme.cpp, cont.

```
    void print_me(ostream &os);      // override
};

class P_dbl : public IPrintable {
public:
    double val;

    P_dbl() {};
    P_dbl(double new_val) {val = new_val; };
    void print_me(ostream &os);     // override
};

// IMPLMENTATIONS OF PRINT_ME
//-------------------------------------------

void P_int::print_me(ostream &os) {
    os << n;
}

void P_dbl::print_me(ostream &os) {
    os << "  " << val << "f";
}

// MAIN FUNCTION
//-------------------------------------------
int main()
{
    IPrintable *p;
    P_int num1(5);
    P_dbl num2(6.25);

    p = &num1;
    cout << "Here is a number: " << *p << endl;
    p = &num2;
    cout << "Here is another:  " << *p << endl;
    return 0;
}
```

How It Works

The code in this example consists of three major parts:

- The abstract class, IPrintable, and a pointer, p, which is able to point to an object of any class derived from IPrintable
- The subclasses, P_int and P_dbl, which contain an integer and floating-point value, respectively, that tell how to print the object
- The main function, which puts these classes to the test

The IPrintable class is an abstract class that you can also think of as an interface that defines a single service: the virtual function print_me.

```
class IPrintable {
    virtual void print_me(ostream &os) = 0;

    friend ostream &operator<<(ostream &os,
                               const IPrintable &pr);
};
```

The idea of the class is simple: subclasses of IPrintable implement the function print_me to define how they send data to an output stream (**ostream**).

The IPrintable class also declares a global friend function. Chapter 18, “Operator Functions: Doing It with Class,” explains more about how to write such functions. For now, just accept that this function is valid.

This operator function converts an expression such as this

```
cout << an_object
```

into a call to the object’s own print_me function.

```
an_object.print_me(cout)
```

Because print_me is virtual, the correct version of print_me is always called no matter how the object is accessed.

```
Printable *p = &an_object;
//...
cout << *p;
```

If print_me were not a virtual function, this code would not work. In that case, the function IPrintable::print_me would be called… but since IPrintable doesn’t even implement print_me, the result would be a runtime error.

The actual implementations of print_me do little in this particular example, but that’s not important. Integers and floating-point values are easily printed.

I put in a small difference between them—printing a couple of extra spaces and an "f" suffix for the floating-point implementation—so you can notice that a different version of print_me is being called.

```
void P_int::print_me(ostream &os) {
    os << n;
}

void P_dbl::print_me(ostream &os) {
    os << "  " << val << "f";
}
```

Implementations of print_me for other classes can be much more interesting. Here, for example, is how you might implement print_me for the Fraction class:

```
void Fraction::print_me(ostream &os) {
    os << get_num() << "/" << get_den();
}
```

How is this useful? Well, you could have an array of objects of different types. As long as they all were instances of classes derived from Printable, you could print all of them, each in the correct format as determined by the objects' own classes.

The bottom line here is this: in a very real sense, the objects know how to print themselves! The line containing "cout <<" essentially says to each object "Print yourself." The amazing thing is that each object may (in effect) execute a different piece of code, tailor-made for its own particular subclass.

```
IPrintable array_of_objects[ARRAY_SIZE];
//...
for (int i = 0; i < ARRAY_SIZE; i++) {
     cout << array_of_objects[i]  << endl;
}
```

EXERCISES

Exercise 16.2.1. Write a version of the Point class from Chapter 10, so that it subclasses IPrintable and implements the print_me function. Then test the results. Implement the print_me function so that it displays output in the format "(x, y)".

Exercise 16.2.2. Revise the Fraction class from Chapters 10 and 11 so that it subclasses IPrintable and implements the print_me function. Then test the results by using code such as the following to print a Fraction object:

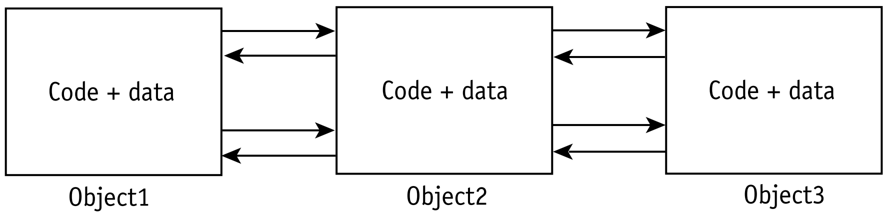
```
Fraction fract1(3, 4);
//...
```

```
IPrintable *p = &fract1;
cout << "The value is " << *p;
```

If all goes well, you should find that the Fraction object is printed in the correct format.

A Final Word (or Two)

When I was first learning about object-oriented programming way back in the 1980s, I developed the idea that object-oriented programming was all about creating individual, self-contained entities that communicate by sending messages to each other. The Smalltalk language, for example, is built around this idea.



These days, I still think that's not a bad way to get a grip on some of the major concepts. Individual, self-contained entities like to shield their contents; they therefore have encapsulation—the ability to keep their data private.

I'm not sure how well inheritance is demonstrated by this model, although you can make it fit. If each of the individual objects is like a microprocessor or chip (to think of it all in hardware terms), then ideally you should be able to pop out a chip, make some modifications or improvements to it, and pop it back in.

Above all, the model of "independent entities sending messages to each other" is a good way of illustrating what polymorphism and virtual functions are all about.

Recall what I said a little earlier about the IPrintable interface in the introduction to Example 16.2. Here, I paraphrase it in general terms:

You can use an object without knowing its type or what functions it calls because the knowledge of how to perform the service is built into the object itself, not the user of that object.

This principle is consistent with the idea of independent objects that communicate by sending messages. The user of an object doesn't need to tell the object how to do its job. What goes on inside another object is a mystery. You send a message, knowing that the object will respond in some appropriate way.

In essence, objects—independent units of code and data—are liberated from slavish dependence on the internal structure of other objects.

But the result is not anarchy. Object-oriented programming systems enforce discipline in the area of type checking. If you want to support an interface, you have to implement *all* the services (that is, virtual functions) of that interface, and you have to match the types in the argument lists exactly.

You can implement a function in a way that had not yet been written at the time the client code was written. Remember that the following code will always work correctly, without being revised or recompiled, even if the specific type of an_object changes—that is, if the pointer is reassigned to point to a different kind of object (provided only that the new object must also have a class derived from IPrintable).

```
IPrintable = &an_object;
cout << *p;
```

An (Even More) Final Word

But what does all this mean? Why does polymorphism matter? Is it because it contributes to code reuse? Well, yes. But it's not primarily that.

Object-oriented programming is really more about *systems*—graphical systems, network communication, and other aspects of the technology in which we daily become more enmeshed. Items in a graphical-user interface, or in a network, act like independent objects sending messages to each other.

Traditional programming techniques were developed for a different world, a world in which it could be a triumph just to submit a stack of punch cards and see your program have a successful beginning, middle, and end, rather than gagging and puking. In this world, you assumed that you were the only game in town.

Today's software has become more complex as it has become richer. The success of Microsoft Windows, for example, stems in part from its rich set of components. And the component model is not as easy to implement with traditional programming techniques. You want to be able to plug ever-newer software components into complex existing frameworks such as Windows.

Ultimately, this way of looking at things is closer to the reality of the great wide world. One of the most exalted claims made for object orientation is that "it more closely models the real world." That's an inflated claim but one with a nugget of truth. We *do* live in a complex world. We *do* interact with things and people independent from ourselves. We *do* need to trust in specialized knowledge that others can bring. And maybe if we could liberate software objects, giving them the independence and freedom to do what each of them knows how to do best, we might feel more encouraged to liberate ourselves.

Chapter 16 *Summary*

Here are the main points of Chapter 16:

- *Polymorphism* means that the knowledge of how to perform a service is built into the object itself, not the client (that is, the software that uses it). Consequently, the resolution of a single function call or operation can take unlimited different forms.
- Polymorphism is made possible by virtual functions.
- The address of a virtual function is not resolved until runtime. (This is also called *late binding.*) The class of an object—as known at runtime—determines which implementation of a virtual function is executed.
- To make a function virtual, precede its declaration in the class with the **virtual** keyword. For example:

```
virtual Card deal_a_card();
```

- Once a function is declared virtual, it is virtual in all subclasses. You don't need to use the **virtual** keyword more than once per function.
- You cannot make a constructor virtual. Technically, you can make an inline function into a virtual function, but the compiler cannot expand it as an inline function unless it's safe to do so. An example of when it would be safe would be a case in which the exact type could be determined at compile time. If it cannot be so determined, the function cannot be inlined.
- There is a small performance penalty and a small space penalty whenever a function is made virtual, but the advantages of making it a virtual function almost always make up for it.
- As a general rule, any member function that might be overridden should be declared virtual.
- A *pure virtual function* usually has no implementation (that is, no function definition) in the class in which it is declared. You declare a pure virtual function by using =0 notation. For example:

```
virtual void print_me() =0;
```

- A class with at least one pure virtual function is an *abstract class*. Such a class cannot be used to instantiate objects, although its subclasses can.

```
Number a, b, c;    // ERROR!
```

- Abstract classes are useful as a means to create a general interface—a list of services that a subclass provides by implementing all the virtual functions.
- In the final analysis, polymorphism is a way of liberating objects from slavish dependence on each other because the knowledge of how to perform a service is built into each individual object. Ultimately, it's this feature that gives object orientation its special flavor and makes it *object* oriented, rather than merely class oriented.

This page intentionally left blank

17 New Features of C++14

It's a mark of C++'s importance that the specification is updated every three years so that, like clockwork, you can expect interesting and useful new language features.

Professional programmers depend on C++ to write commercial software. The C++ community is therefore highly aware of ever-changing programming needs and what might help to write the best, most efficient, and most reliable software.

Many of the newest features are aimed at advanced programmers. There are new features in the spec helpful for writing templates and lambdas—functions defined "on the fly." You can read about these features in my book *C++ for the Impatient,* but they don't enter into elementary programing. This chapter focuses on features most likely to be useful to the beginning-to-intermediate programmer.

The Newest C++14 Features

Only a few features of C++14 are likely to be useful to the newer C++ programmer, but some of them really are interesting and they answer needs that programmers have had for many years.

Here's a summary of these new features:

- *Digit-group separators in literal constants.* It's always been difficult to write large constant numbers into a program, because you can't write them as (for example) "674,501"—but now you can.
- *String-literal suffix.* You can now append an "s" suffix to a literal string, which (by default) has the C-string type, an array of **char**. The "s" suffix gives the literal the same type as a **string** object.

- *Binary literals*. C++ has included support for writing numbers in hexadecimal and octal radix from the beginning. But some programmers have asked for binary (base-2) radix for a long time; their wish has been granted.

Digit-Group Separators

The newest C++ specification enables you to use the single quotation mark (') as a digit separator. This is among the most useful new features. New programmers make the mistake of entering digit-group separators all the time, incorrectly writing this:

```
1,320,000
```

instead of this:

```
1320000
```

This is a long-standing issue. A computer has no problem reading something like "1320000" because it scans the digits, reading one at a time. But reading such digits is difficult for humans, who read numbers by "chunking" groups together and saying, "Ah yes, it's over a million."

You might ask, "Why can't the compiler just look at the numbers and ignore the commas?" That would be a good feature, were it not for conflicts in syntax. Consider a function call taking several integers:

```
my_func(1,2,3);
```

If commas were accepted as digit-group separators, how would the compiler interpret the following?

```
My_func(1,200,333,500,100);
```

So instead of commas, the C++14 specification uses single-quote marks to separate groups of digits. The result is a little strange looking at first, but you should get used to it quickly. For example, the previous function call could be revised as follows:

```
my_func(1'200,333'500,100);
```

You should be able to see this function call is equivalent to the next one:

```
my_func(1'200, 333'500, 100);
```

Needless to say, you'll always be better off putting in spaces between arguments, but C++ has never required that and adding such a requirement would have created massive backward-compatibility problems.

This new feature—adding support for group separators—makes sense in an era in which programs are gaining the ability to handle larger and larger numbers. Later in the chapter, I present the **long long** integer type, mandated by the C++11 spec. But that type can be very difficult to initialize without group separators. For example, with the separators, you could initialize a number to 100 billion (that is, 100 thousand million) this way:

```
long long int big_n = 100'000'000'000;
```

Without the digit separators, you have to perform this initialization as follows:

```
long long int big_n = 100000000000;
```

You can see the problem. Missing even a single zero would cause the amount to be wrong by a factor of 10.

The digit separators can be used in any grouping pattern you like, and you can use them with any radix and with fractional amounts as well as integers. For example:

```
double pi = 3.141'592'653;
int goofy_num = 1'02'000'5;
```

When the compiler reads single-quotation marks in the context of a numeric literal, it basically ignores them. Very much like comments, digit separators are there only as an aid to humans reading and maintaining the code.

String-Literal Suffix

The "s" suffix, applied to a string literal, creates a true string object rather than a C-string.

```
"text"s
```

In most situations in which you need to use text strings, you're better off using the STL **string** object, which frees you from worrying about size limitations, provides many useful member functions, and enables you to write convenient lines of code like this:

```
#include <string>
using namespace::std;
...
string sFirst = "Elvis ";
string sLast = "Presley ";
string sFullName = sFirst + sLast;
```

But in C++, you can't get away from C-strings altogether. In order to maintain backward compatibility with C and with old software, string literals in C++ (assuming they have no special suffix) have C-string type. They're simple arrays of **type** char, terminated by a null value as explained in Chapter 8, "Strings: Analyzing the Text."

```
char str[] = "I am a null terminated C-string."
char *p  = "So am I."
```

This behavior—interpreting string literals as C-strings, not **string** objects—is occasionally a problem, because although you can write statements like

```
string sName = sFirst + " Adams";
```

you *can't* write this:

```
string sName = "John Quincy " + "Adams";
```

The problem with this last statement is that although a **string** object can interact with C-strings (making the earlier statement valid), C-strings themselves, having type **char***, have no behavior defined for the + operator. Instead, you must use the **strcat** function, which is much less elegant.

But with the "s" suffix, string literals become genuine **string** objects, so you can write statements like the following all day long, which is convenient when you want to span more than one physical line:

```
string sName = "John Quincy "s + "Adams"s;
```

There are other situations in which it's useful to specify a literal as having **string** type. Suppose you return a string from a function, but the function has **string** type, or worse, it has **auto** return-value type, meaning that the compiler has to deduce what the type of the function is. In that case, use of the "s" suffix may be extremely useful as a way of clarifying that you intended to return a true **string** object rather than an array of **char**.

```
return "Hello!"s;    // Return as a string object.
                     //  not a char*.
```

Binary Literals

In C++14, you can write numeric literals in binary radix by using a "0b" or "0B" prefix.

```
0bdigits
```

For example:

```
cout << 0b110 + 0b001;  // Prints 7 (= 0b111).
```

As everyone knows, all data inside a computer is made up of 1's and 0's. That you can read something other than 1's and 0's on the screen is due to a complex series of actions involving the operating system as well as the BIOS (Basic Input Output System). But if you could see the actual pattern of data at any memory location, it would be made up of 1's and 0's.

Many programmers have long wanted to be able to read and write data in this primitive form. That would make it easier to write bit masks, for example, which are used to turn individual bits on and off within a compact piece of data. For years, C and C++ programmers have had to settle for using hexadecimal and binary notation.

Now you can test bit masks directly. For example:

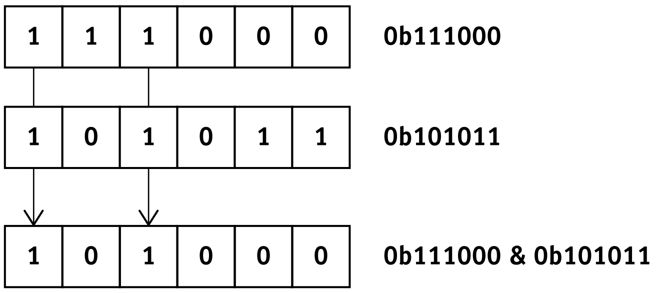
```
cout << data | 0b1111;  // Turn on low 4 bits.
cout << data & 0b1111;  // Mask out all but
                        //   low 4 bits.
```

To understand these statements, I need to briefly introduce the bitwise operators—which I've skipped until now because you rarely if ever need them in beginning-to-intermediate programming.

OPERATOR	NAME	DESCRIPTION
&	Bitwise AND	Sets a bit in result to 1 if both corresponding bits in the operands are 1. Other bits in result are set to 0.
|	Bitwise OR	Sets a bit in result to 1 if either corresponding bits in the operands is 1. Other bits in result are set to 0.
^	Bitwise exclusive OR	Sets a bit in result to 1 if either, but not both, of the corresponding bits in the operands is 1. Other bits in result are set to 0.
~	Bitwise negation	Takes one operand. Reverses the value of each bit, so each 1 in the operand becomes a 0 in the result, and vice-versa.

By "corresponding bits," I mean bits in the same position. So if 0b1100 is AND'ed with 0b0011, the result is 0b0000. But if they are OR'ed together, the result is 0b1111.

This is easier to see with a diagram. For example, suppose you use bitwise AND to combine two amounts, one equal to 0b111000 and the other equal to 0b101011. You can see how corresponding bits in each operand are combined using AND. A bit is set to 1 in the result only if both of the corresponding bits in the operands are 1.



Bitwise AND

Using the binary-radix notation, you can write statements such as the following, incorporating the single quote mark as digit separator as described in the previous section. Remember, this uses bitwise OR (|):

```
cout << 0b1111'0000 | 0b0000'1111 << endl;
```

The result of this statement is 1111 1111, which has 1's in all lower eight positions of an integer field, with the rest set to 0. The statement therefore prints:

```
255
```

The digit string "255" is not in binary-radix format, of course. An easy way to do that—to get output in binary radix as well—is to use the **bitset** template. To use this template, you don't specify an underlying type but rather a fixed number of bits. When printed, a bitset produces a digit string containing 1's and 0's.

As with other templates, you set up its usage with an **include** statement and a **using** statement:

```
#include <bitset>
using namespace std;
```

Then declare and initialize the bit set:

```
bitset<8>  my_bit_field(0b1111'0000 & 0b1100'0000);
cout << my_bit_field << endl;
```

This should print the following:

```
11000000
```

Example 17.1. *Bitwise Operations*

This next example is a simple program that prints the results of a series of bitwise operations. Both the operands and the results are printed out in binary (base 2) radix. Your challenge is to anticipate the output and see if your predictions match the results that get printed.

binaryr.cpp

```
#include <iostream>
#include <bitset>
using namespace std;

int main()
{
     bitset<8> a(0b1111'0000 & 0b1001'0000);
     bitset<8> b(0b1111'0000 | 0b1001'0000);
     bitset<8> c(0b1010'1010 & 0b1000'1111);
     bitset<8> d(0b1010'1010 | 0b1000'1111);
     cout << a << endl;
     cout << b << endl;
     cout << c << endl;
     cout << d << endl;
}
```

EXERCISES

Exercise 17.1.1. Write a program similar to this example, but use it to test the effect of the exclusive or operator (^) several times.

Exercise 17.1.2. Write a program similar to this example, but use it to test the effect of the bitwise negation operator (~) several times. Remember that negation is a unary operator—it takes just one operand.

Exercise 17.1.3. Write a program that takes any integer input, masks out all four of the lowest bits, and prints the result. Is there an operation involving modular division (%) that would always produce the same result?

Exercise 17.1.4. In what situations, if any, do logical operators (&&, ||, and !) always produce the same results as their corresponding bitwise operators, &, |, and ^? (Hint: for signed types, such as **int**, having a 1 in all positions produces an overall value of negative 1. With unsigned types, such as **unsigned long**, 1 in all positions produces the highest value in the range.)

Features Introduced in C++11

C++14, of course, is an upgrade to the most recent specification before it: C++11. Many of the newer features, therefore, are brought over from that slightly older specification. In some cases, a compiler manufacturer may have only recently gotten around to implementing full C++11.

The new C++11-specific features described in this chapter include the following:

- *The **long long int** type.* This new data type stores values far beyond the limit of the **long int** type, which is typically plus or minus 2 billion (thousand million). The **long long int** type is a 64-bit number that can store astronomical values far in excess of a billion, but with absolute precision, unlike **double**.
- *Ranged-base **for** ("for each").* This syntax is a variation on the C++ **for** keyword. Instead of explicitly setting begin and end points in a loop, you just say, "Process each item in the group." It's simpler, easier, and less error prone.
- *The **auto** and **decltype** keywords.* These can be convenient when working with complex, exotic types.
- *The **nullptr** keyword.* This is the new way to represent null (zero-value) pointers. Although not yet strictly required, it is now strongly recommended.
- *Strongly typed enumerations.* You can use this feature to refer to meaningful symbolic names rather than arbitrary numbers; it's another way to get rid of "magic numbers." This is a long-standing feature in C++ but greatly strengthened in C++11. Both weak and strong **enum** types are now supported.
- *Raw string literals.* This feature lets you enter string literals without having to use escape characters for " and \.

The long long Type

In the PC environment, the C and C++ languages have generally supported 16-bit integers as **short** and 32-bit as **long**. For some time now, **int**, which is the "natural" integer type, has been equivalent to **long**.

But it's not hard to go beyond the limit of 32-bit storage. One alternative is to switch to floating point, which—because of scientific notation—can store astronomically big and tiny values. But floating-point data cannot store big numbers with absolute precision.

Sixty-four bits is the future of computing and is the natural size for integers to migrate to, but the **long** type is already taken (32 bits). For that reason, two "longs" are needed.

TYPE	TYPICAL MEANING (SUPPORTED ON NEARLY ALL PCS)	C++ IMPLEMENTATION
`char`	Usually 8 bits, large enough to hold an ASCII character	Enough size to hold a standard character
`short int`	16 bits, limit of 64KB	At least 16-bits wide; size equal to or greater than **char** in size; no larger than **int**
`long int`	32 bits, limit of approximately 4 billion (plus or minus 2 billion)	Size equal to or greater than **int** in size
`long long int`	64 bits, limit of 4 billion *squared*	C++11 and later only; size greater than **long...** required to be at least 64 bits if supported.

Each of these types has an unsigned version, such as **unsigned short** and **unsigned long**. Unsigned values do not store negative values, but instead store twice as many positive values. So, what you lose in the negative range, you gain back by having a bigger positive range. Signed integers (the default) store both positive and negative values.

The syntax for declaring **long long** integers is similar to that for any type. Integer types have a quirk. Except with the **int** type itself, the keyword **int** is optional.

Keyword

```
long long variables;

long long int variables;    // int is optional
```

And there is also the unsigned version, which stores only non-negative values:

```
unsigned long long variables;

unsigned long long int variables;  // int is optional
```

For example:

```
long long i;        // i is uninitialized 64-bit int.
long long i = 0;    // i initialized to 0.
long long i, j, k;  // i, j, and k all 64-bit ints.
```

Interlude

Why a "Natural" Integer?

I use the **int** type throughout this book. For the current PC environment, **int** is equivalent to **long**, the 32-bit type. As the "natural" integer, **int** is intended to correspond to the processor size of the target environment so that programs run efficiently.

There is one downside: If you ever port your code to a smaller architecture (16 bits), programs that run beautifully on 32-bit architecture can break without warning. It's possible your **int** variables hold values larger than 64KB; if so, when ported to a 16-bit architecture, your programs will encounter serious bugs.

The upshot is that you are safe using **int** in programs for your own use only. But…

Code developed professionally at Microsoft avoids this approach. The **int** type is never used in serious commercial projects. Microsoft developers conscientiously stick to types with fixed sizes that have names like INT32, controlled and defined in header files. This is because they are developing software for many platforms to be used all over the world. For a beginner, such an approach is overkill. But keep these issues in mind if you ever intend to write commercial (widely distributed) software or port to different platforms.

Working with 64-Bit Literals (Constants)

For the most part, using the 64-bit type is as easy as declaring variables as **long long**. But there is also the issue of initialization. Fortunately, the following works just fine, even though the numeric literal, 0, has **int** type and n has **long long int** type:

```
long long n = 0;
```

This is OK, because C++ automatically promotes a smaller type, such as **int**, into a larger type such as **long long** and does so without complaint. But what if you need to initialize n with a larger value?

```
long long n = 123000123000456;  // ERROR
```

The problem is that the literal in this example is too large to be stored as a standard **int**, so you will likely get an error message. To ensure that such a large number (larger than 2 billion or so) can be stored, use the new "LL" suffix, which stands for **long long**, of course:

```
long long n = 123000123000456LL;  // LL used; no error
```

You can also use the "ULL" prefix for **unsigned long long**:

```
long long n = 123000123000456ULL;      // ULL used; OK.
```

Using a single-quote mark as digit separator—a new feature just introduced in C++14, and which I covered earlier—can be very helpful here:

```
long long n = 123'000'123'000'456ULL;  // ULL used; OK.
```

Accepting long long Input

Earlier, I used the **atoi** function to convert strings to integers. C++ compilers that support **long long** also provide a useful support function, **atoll**, to convert **char*** strings (C strings) to **long long** integers.

```
char *input_string[MAX_WIDTH + 1];
cin.get(input_string, MAX_WIDTH);

long long n = atoll(input_string);
```

The **long long** type presents another challenge: When numbers get to be this big, they are difficult for an end user to type in or read. This is a problem I discussed at length earlier with regard to the single-quote mark, which is now supported in C++ code for writing large numeric literals.

But you may want your user to be able to use commas as well. For example:

```
123,000,123,000,446,001
```

There's an easy solution: You can strip away these characters before converting to a number. The following function performs this task and I invite you to use it in your own programs. You're welcome.

Note To support the **atoll** function used here, you'll need to include <cstdlib>.

```
#define GROUP_SEP ','

long long read_formatted_input(string s) {
     for (int i = 0; i < s.size(); ++i) {
            if (s[i] == GROUP_SEP)
                 s.erase(i, 1);
        }
        return atoll(s.c_str());
    }
```

Formatting long long Numbers

Printing formatted numbers with the proper digit-group separators is more of a challenge, because the program has to make intelligent decisions about where to put those characters.

The STL **stringstream** class is helpful here. Using this special class, you can write to a string as you would write to the console or a file. Make sure you include both these files:

```
#include <string>
#include <sstream>
```

Now you can create and use a "string stream." You can write to the following object, s_out, just as you would write to **cout**. After you are done writing to the stream object, you convert it to an actual string by using the **str** member function.

```
stringstream s_out;
s_out << "The value of i answer is" << i << endl;
string s = s_out.str()
```

We now have enough techniques to write a function that takes a **long long** as input and returns a formatted string.

```
#define GROUP_SEP     ','
#define GROUP_SIZE    3

string output_formatted_string(long long num) {

     // Read data into string s.

     stringstream temp, out;
     temp << num;
     string s = temp.str();

     // Write first characters, in front of
     //  first separator (GROUP_SEP).
```

```
        int n = s.size() % GROUP_SIZE;
        int i = 0;
        if (n > 0 && s.size() > GROUP_SIZE) {
            out << s.substr(i, n) << GROUP_SEP;
            i += n;
        }

        // Handle all the remaining groups.

        n = s.size() / GROUP_SIZE - 1;
        while (n-- > 0) {
            out << s.substr(i, GROUP_SIZE) << GROUP_SEP;
            i += GROUP_SIZE;
        }
        out << s.substr(i); // Write the rest of digits.
        return out.str();   // Convert stream -> string.
    }
```

Again, I invite you to steal this code to use freely in your own programs.

The function is as long as it is because I've provided comments for readability. As always, if you are trying to type it in fast, the comments are optional on your part.

This example uses **substr**, an important function of the **string** class. Its first argument is a starting position (0-based, remember), and the second argument is the number of characters to select from that position onward. The **substr** function returns the indicated substring. When the second argument is omitted, it returns the substring from the indicated position forward to the end of the string.

The function takes numeric input such as 88123000567001LL and returns a string formatted with group separators, making it more readable to the end user. You can then print the resulting string on the console.

```
88,123,000,567,001
```

Example 17.2. *Fibonacci: A 64-Bit Example*

OK, enough preliminaries! Here's a practical example: Suppose you want to know what the fiftieth Fibonacci number is. It turns out that the answer is well outside the range of a standard **int** or **long** (32 bits), and is a perfect application for **long long**.

First, a quick refresher on Fibonacci numbers: This is a famous set of numbers, in which each member, after the first two, is equal to the total of the two numbers preceding it. The first few numbers in the series are:

```
1 1 2 3 5 8 13 21 34 55 89 144
```

This set of numbers has a formal mathematical definition:

```
F(0) = 1
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

At first glance, this definition is a perfect candidate for recursion. The formal definition translates smoothly into C++ code. (You'll notice my use of **long long**, which is needed here because Fibonacci numbers get large quickly.)

```
long long Fibo(long long n) {
     if (n < 2) {
         return 1;
     } else {
         return Fibo(n - 1) + Fibo(n - 2);
     }
}
```

But as beautiful as this version is, it's *incredibly* inefficient. You won't notice a problem running it on low values, say up to Fibo(30). But once you get around Fibo(40) or so, the delay becomes unacceptably long—even in this era of fast processors. This is because every increase in n geometrically increases the number of function calls.

The iterative version requires a few more lines of code. But unlike the recursive version, it can handle Fibo(50) at (what seems like) an instantaneous speed instead of taking hours.

```
long long Fibo(int n) {
     if (n < 2)
         return 1;
     long long temp1 = 1;
     long long temp2 = 1;
     long long total = 0;
     while (n-- > 1) {
          total = temp1 + temp2;
          temp2 = temp1;
          temp1 = total;
     }
     return total;
}
```

With this version, the issue is not processor time but the ability to hold large numbers. That's why **long long** is needed here, but note that even this huge range is exceeded before you reach Fibo(100).

Here is the complete program that prompts for a number and calculates Fibo(n), the Nth Fibonacci number:

fibo.cpp

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

using namespace std;

int long long Fibo(int n);
string output_formatted_string(long long num);

int main() {
     int n = 0;
     cout << "Enter a number: ";
     cin >> n;
     string s = output_formatted_string(Fibo(n));
     cout << "Fibo(" << n << ") = " << s << endl;
     return 0;
}

long long Fibo(int n) {
     if (n < 2)
         return 1;
     long long temp1 = 1;
     long long temp2 = 1;
     long long total = 0;
     while (n-- > 1) {
          total = temp1 + temp2;
          temp2 = temp1;
          temp1 = total;
     }
     return total;
}

#define GROUP_SEP     ','
#define GROUP_SIZE    3
```

▼ *continued on next page*

fibo.cpp, cont.

```
string output_formatted_string(long long num) {

     // Read data into string s.

     stringstream temp, out;
     temp << num;
     string s = temp.str();

     // Write first characters, in front of
     //  first separator (GROUP_SEP).

     int n = s.size() % GROUP_SIZE;
     int i = 0;
     if (n > 0 && s.size() > GROUP_SIZE) {
         out << s.substr(i, n) << GROUP_SEP;
         i += n;
     }

     // Handle all the remaining groups.

     n = s.size() / GROUP_SIZE - 1;
     while (n-- > 0) {
         out << s.substr(i, GROUP_SIZE) << GROUP_SEP;
         i += GROUP_SIZE;
     }
     out << s.substr(i); // Write the rest of digits.
     return out.str();   // Convert stream -> string.
}
```

For example, if the end user inputs 70, the program prints this result:

```
Fibo(70) = 308,061,521,170,129
```

How It Works

The main function gets a number from the console and passes it to a function, Fibo. The resulting **long long** integer result is passed to the print_formatted_string function, which produces a nicely formatted string result. This string is then printed.

```
string s = output_formatted_string(Fibo(n));
cout << "Fibo(" << n << ") = " << s << endl;
```

The rest of the code then works as described earlier. The program uses the iterative version of the Fibo function, which—although not as elegant as the recursive version—is infinitely more practical: it produces instantaneous results even for high numbers. The recursive version would take hours to calculate Fibo(50) if it didn't bring down the system first.

EXERCISES

Exercise 17.2.1. Write a version of the program that maintains an array of 70 integers of type **long long**. Then fill up this array with the first 70 Fibonacci numbers. (Note: In our nomenclature, we've dubbed the first number F(0), not F(1).) Instead of using the Fibo(0) function from Example 17.2, set F(0) and F(1) directly. Then write a loop to calculate each remaining array element, (F(2) to F(70)), by adding the values of the two preceding array elements. Print the array with calls to the output_formatted_string function already provided.

Exercise 17.2.2. Write a program that prompts for a number, which you then store as a **long long**. Permit the user to enter the number using optional digit separators. Then determine the first prime number bigger than the number entered and print this print number. If needed, refer to the prime-number-testing code in Chapters 2 and 4.

Localizing Numbers

In Chapter 8, I introduced the **#define** preprocessor directive. This directive is useful in minimizing the appearance of "magic numbers" (numbers whose use appears to be entirely arbitrary) from your program.

The syntax for the simple use of **#define** is as follows:

```
    #define  symbol_name  replacement_text
```

The result is that the C++ preprocessor replaces each occurrence of *symbol_name* it finds in the rest of the source file (outside of comments and printed strings) with the *replacement_text*.

If you want to compile the program to work correctly for end users in other countries, you may need to pay attention to formats. Large numbers have one format for Americans and British, and another for France and many other European countries. Still other countries use a dot (.), reserving the comma (,) as a decimal-point (radix) indicator.

```
    1,235,070,556   // American/UK format
    1 235 070 556   // Continental format
    1.235.070.556   // Alternative European format
```

I wrote the program in this chapter to provide the easiest possible control. The format used by the print_formatted_string function is determined by the two **#define** directives. You never need to change more than these two lines:

```
#define GROUP_SEP     ','
#define GROUP_SIZE    3
```

GROUP_SEP specifies the group separator: a comma, blank space, apostrophe, or dot, as appropriate; GROUP_SIZE sets the number of digits grouped together. China and Japan often use groupings of four; most countries use three.

Interlude

Who Was Fibonacci?

Fibonacci did not invent Fibonacci numbers, yet we can thank him for computers. He brought decimal numbers to Europe, and if we didn't understand the decimal system, we wouldn't understand binary numbers. Without binary numbers, no computers.

This great visionary was a man named Leonardo Bonacci, whose nickname was "son of Bonacci"—or in Italian, "Fibonacci." Born in 1170, he is considered the greatest European mathematician of the Middle Ages.

His most famous work, *Liber Abaci* (meaning "Book of Calculation"), introduced Europe to the decimal-number system devised in India and used by Arabs (hence, "Arabic numerals"). It also introduced Europe to a fascinating problem that Hindu mathematicians had asked and answered back in the sixth century: was there a series of numbers, they wondered, that described the population growth of a pair of rabbits in an ideal environment, free of famine and predators? Their answer was a series of numbers: 1, 1, 2, 3, 5, 8, 13, and so on, later dubbed "Fibonacci numbers" in the West.

Did the ancient Hindus realize the secrets of Nature that lay hidden in this deceptively simple series? It turns out the ratio of two consecutive such numbers converge toward a mysterious transcendental number, approximately 1.618, which the Greeks called the Golden Ratio. The Parthenon is designed on this ratio and, 2,000 years later, Leonardo da Vinci's famous *Vitruvian Man* would diagram how thoroughly the Golden Ratio matches the ratios of the human form: the length of a leg to the length of the torso, for example, and the length of the torso to the length of the body.

When we see so much of nature described this way, does that mean someone is trying to tell us something? All we know for sure is that there's something in the nature of existence that responds to the beauty of mathematics.

Range-Based “for” (For Each)

One of the most popular features introduced in C++11 (and still supported in C++14, obviously) is *range-based for*. This is a technique for writing less code—and getting fewer errors—when you use a **for** loop to process an array or other container.

Some other languages have had this feature for years. It says, “Process every item in the group” without having to worry about where you begin or end. The language automates the details of beginning and ending, provided only that this information is available to the compiler.

This approach has two benefits:

- It saves programming effort because it frees you from worrying about how to properly initialize and set terminal conditions in a **for** loop.
- It frees you from one of the most common sources of bugs in C++ programs: incorrectly setting loop conditions. Even the most experienced programmers commit this sin.

Here’s the general syntax. It comes in two forms. Look closely; the only difference you can see here is that of an ampersand (&).

Keyword

```
for(base_type&  variable : container)    // Reference
     statement

for(base_type  variable : container)     // By value
     statement
```

The *statement*, as always, can be a compound statement, or “block,” enclosed in curly braces ({}), and this is always recommended. The *variable* has scope limited to the *statement* or block.

In the first version of this syntax, the *variable* is a reference type, which means it has the ability to manipulate the data. If you want to manipulate the values in the container, use this version. The second version of the syntax provides access only to copies of values.

Here’s a practical example, using an array. This code fragment sets every member of my_array to 0:

```
int my_array[10];
for(int& i : my_array) {
     i = 0;
}
```

This next example sets every member of my_array to 5:

```
for(int& i : my_array) {
     i = 5;
}
```

Remember, if you don't intend to change any values, you can protect the data by dropping the ampersand (&). For example, this will print all the elements of my_array:

```
for(int i : my_array) {
     cout << i << endl;
}
```

Alternatively, you can protect values by using the ampersand (&), but declaring the loop variable **const**. This has the advantage of keeping i as a reference variable and thus avoiding the performance cost associated with copying.

```
for(const int& i : my_array) {
     cout << i << endl;
}
```

Here's an example that prints all the values of an array of type **double**. This employs d as a "value" variable, so that it results in each element of the array being copied. Inside the loop, the code can manipulate these copies however it wants without affecting the original values. (However, as just noted, this version incurs the cost of copying 100 elements.)

```
double float_pt_nums[100];
...
for(double d : float_pt_nums) {
     cout << d << endl;
}
```

But in this next example, which sets all the floating-point values to 0.0, the ampersand (&) is required.

```
for(double& d : float_pt_nums) {
     d = 0.0;
}
```

The C++ range-based **for** syntax is flexible. The *container* can be

- Any kind of array.
- An STL **string** object. (The elements within a string object are individual characters.) Base type is **char**.

- Instances of STL classes that define an iterator, such as **list** and **vector**.
- Initialized lists (see the example immediately following).

The range-based **for** syntax supports initialized lists using curly braces. For example, the following code fragment prints the first 12 Fibonacci numbers, one to a line. This is the simplest way to print a lot of numbers:

```
for(int& n : {1,1,2,3,5,8,13,21,34,55,89,144}) {
    cout << n << endl;
}
```

Range-based **for** does have limitations. Because range-based **for** cycles through a container without explicit index numbers, pointers, or iterators, it can be harder to do certain things. Every element tends to get treated the same way.

For example, what if you want to set an array to {0, 1, 2, 3, 4}? Using a standard **for** statement, you'd write something like this:

```
for(int i = 0; i < 5; i++) {
     array[i] = i;
}
```

Fortunately, with a little extra programming, this is still doable with range-based **for**:

```
int j = 0;
for (int& i : array) {
    i = j++;
}
```

There is one other drawback to range-based **for** (which isn't bad, considering all its advantages). The loop variable must be declared local to the loop. You do not have the option of declaring it outside the loop.

Example 17.3. *Setting an Array with Range-Based "for"*

The following example shows the use of range-based **for** in several contexts:

range_based_for.cpp

```
#include <iostream>
#include <cstdlib>

using namespace std;
```

▼ *continued on next page*

range_based_for.cpp, cont.

```
#define SIZE_OF_ARRAY 5

int main()
{
     int arr[SIZE_OF_ARRAY];
     int total = 0;

     // For each element, prompt for a value,
     //   store, and add to total.
     //
     for (int& n : arr) {
          cout << "Enter array value: ":
          cin >> n;
          total += n;
     }
     cout << "Here are the values: ";

     // Print each element.
     //
     for (int n : arr) {
          cout << n << endl;
     }

     cout << "Total is: " << total << endl;
     cout << "Now, I'm going to zero out ";
     cout << "the values. " << endl;

     // Set each element to 0.
     //
     for (int& n : arr) {
          n = 0;
     }

     cout << "Here are the values: ";
     for (int n : arr) {
          cout << n << endl;
     }
     return 0;
}
```

How It Works

There's nothing new in this example other than the use of the C++11 range-based **for** syntax. It's useful to compare what the statements would look like without it. For example, to print every element of the array, arr, you'd normally write this:

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

Not bad, but look how much more succinct the newer version is, using range-based **for**:

```
for (int n : arr) {
     cout << n << endl;
}
```

Remember, you can use any type, but the type of the variable and the base type of the container must match. For example, if arr_floating_pt is an array of elements of type **double**, you'd use a reference to (or rather a copy of) a **double**, not **int**:

```
for (double x : arr_floating_pt) {
     cout << x << endl;
}
```

Don't forget that to alter values within the container, you need the ampersand (&).

```
for (int& n : arr) {
     n = 0;
}
```

EXERCISES

Exercise 17.3.1. Instead of prompting the user with the direction, "Enter array value," prompt the user by printing "Enter array value #X of 5: " where X is the current array index. Do this while still using range-based **for**. (Hint: You'll need to set another variable, such as j, to 0 and then increment it.)

Exercise 17.3.2. Initialize an array to {1, 2, 3, 4, 5}. Then, use range-based **for** to double each element of the array. Print out the results to confirm that the statements worked as expected. (Hint: Don't forget to include the ampersand (&).)

The auto and decltype Keywords

For beginners, the **auto** keyword might not seem like a huge convenience, but when you start to write more complicated programs with exotic data types, you might find it saves you significant work.

Note ▶ The **auto** keyword used to have another use: to indicate an automatic (that is, stack-based) storage class. Local variables are always given this storage class anyway, unless specifically declared static. The use of **auto** to indicate storage class is now entirely obsolete (and no longer supported!).

When a variable is declared with the **auto** keyword, its type is determined by context—specifically, by the thing that initializes it. Once fixed, the variable's type does not change. **auto** is not a variable-data type. For example:

```
auto x1 = 5;         // x1 is an int.
auto x2 = 3.1415     // x2 has type double
auto x3 = "Hello";   // x3 has type char*
auto x4 = "Hi!"s;    // x4 has string
```

You may wonder, at first, what the point of the keyword is. Advanced C++ programmers sometimes use some exotic types. Consider a function return_pp_Fraction that returns a pointer to a Fraction object. In that case, you could declare and initialize x by writing this:

```
Fraction **x = return_pp_Fraction();
```

But you could instead write this:

```
auto x = return_pp_Fraction();
```

Even better is the use of the **auto** keyword within range-based **for**. Suppose weirdContainer is an array of pointers to pointers to Fraction objects. In that case, you could write this

```
for (Fraction& **x : weirdContainer) ...
```

or you could just write

```
for (auto& x : weirdContainer) ...
```

The **auto** keyword is so useful here that it can almost be stated as part of range-based **for** syntax. The *container* will determine the type of the *variable*, and therefore use of **auto** in this context guarantees that this type will always be correct.

```
for(auto&  variable : container)       // Reference
     statement

for(auto  variable : container)         // By value
     statement
```

The **auto** keyword is related to another C++11 keyword, **decltype**, which returns the type of its argument.

```
decltype(x)  y;   // Declare y to have same type as x.
```

The **auto** keyword is also useful in declaring the return type of a function, particularly a function that returns a complex type or a type that might change in a later version of the program. Remember that the return type will be determined by what the function actually returns—so you'll need to be careful about what you're returning. In the following case, though, the return type is obviously **int**:

```
auto func1() {
     if (x == y) {
          return 1;
     } else {
          return 2;
     }
}
```

But this next function has return type **char***:

```
auto func2() {
     if (x == y) {
          return "equal";
     } else {
          return "not equal";
     }
}
```

Obviously, all the **return** statements in a given function need to have precisely matching return types, or an ambiguity is created for the compiler.

The nullptr Keyword

The **nullptr** keyword provides the new, preferred way to represent a null pointer, meaning it "points nowhere." This is *not* the same as an uninitialized pointer! For example, consider the **strtok** function introduced in Chapter 8. One of the ways you use the function is by calling it with a null-pointer argument.

Traditionally, the technique for making a pointer hold a null value is to set it to 0 or the predefined constant NULL.

```
int *p = 0;       // p points "nowhere" for now
int *p2 = NULL;   //  so does p2.
```

For now, these techniques still work, and they may be supported for some time to come for the sake of backward compatibility. But using 0 to initialize or set pointers is bad form because the same value can be used to set ordinary (scalar) variables, which should not be the case.

```
int i = 0;       // 0 also used with pointers
```

The virtue of **nullptr** is that it is specific to pointers. It's part of the language and will work correctly for all programs in which it is appropriate. If your compiler supports **nullptr**, then whenever convenient you should start using it in place of NULL. For example:

```
p = strtok(the_string, ", ");
while (p != nullptr) {
      cout << p << endl;
      p = strtok(nullptr, ", ");
}
```

Note that when any pointer is set to null (that is **nullptr**), it is equivalent to **false** when it is tested directly as a condition. So this conditional test

```
while (p != nullptr) { // While p is not false
...
}
```

is equivalent to the following:

```
while (p) {          // While p is true (not null)
...

}
```

Strongly Typed Enumerations

The more you program, the more you want to get rid of "magic numbers," which are numeric literals that appear in the program for no apparent reason. It's much better to use meaningful symbolic names.

As a first approach, one might assign the numbers 1, 2, and 3 to the choices. Most of the time, the values don't matter. The most important thing is that the three numbers are used consistently to indicate the choices. Consider statements in a game of Rock, Paper, Scissors:

```
cout << "Enter Rock, Paper, or Scissors: "
cin >> input_str;
int c = input_str[0];

if (c == 'R' || c == 'r') {
     player_choice = 1;              // 1 = rock
} else if (c == 'P' || c == 'p') {
     player_choice = 2;              // 2 = paper
} else if (c == 'S' || c == 's') {
     player_choice = 3;              // 3 = scissors
}
```

The programmer has to remember that 1 means rock, 2 means paper, and 3 means scissors. Without comments, this kind of program code is nearly incomprehensible. We need to get rid of the "magic numbers" and replace them with meaningful names.

Our next attempt at trying to store this information is to use a series of **#define** directives.

```
#define ROCK     1
#define PAPER    2
#define SCISSORS 3
```

Now we represent ROCK as 1, PAPER as 2, and SCISSORS as 3. The code is instantly more readable.

```
if (comp == ROCK && player == SCISSORS)
   cout << "Rock smashes scissors. I WIN! " << endl;
```

This is a big improvement, but wouldn't it be nice to automate the assignment of these numbers? C++ allows you to do just that, with the **enum** keyword.

```
enum {rock, paper, scissors };
```

The effect of this declaration is to create rock, paper, and scissors as symbolic constants that are assigned values of three consecutive integers: 0, 1, and 2. (By default, they start at 0.) You can then assign values of this class and test them as appropriate.

```
if (c == 'R' || c == 'r') player = rock;
...
if (comp == rock && player == scissors)
   cout << "Rock smashes scissors. I WIN! " << endl;
```

In traditional C++, you can optionally declare an enumerated type by using a type name after **enum**. This is a *weak type*: You can assign enumerated values to integers but not the other way around.

```
enum Choice {rock, paper, scissors };

Choice your_pick = paper, my_pick = rock;
int i = my_pick;        // Ok.
my_pick = 1;            // Error! Requires cast.
my_pick = static_cast<Choice>(1);  // Ok.
```

enum Classes in C++11 Onward

C++11 and later compilers support a new way of using the **enum** keyword. By combining **enum** with **class**, you create not just symbolic names but also a class.

```
enum class Choice {rock, paper, scissors };
```

You can now declare variables of type Choice. Such variables can be set or initialized only to Choice values, namely, rock, paper, or scissors. The advantage is you can never accidentally assign a value that isn't from the group or accidentally mistake an enumerated value for an integer. Conversion between a strong **enum** value to an integer, or vice versa, requires a **static_cast** conversion.

```
Choice comp = Choice::rock;
Choice player = Choice::paper;
Choice x = 0;   // Error - 0 not in class Choice!
Choice y = 1;   // Error - 1 not in class Choice!
Choice me = static_cast<Choice>(0)  // Ok, cast used
int i = Choice::rock;  // Error -- requires cast.
```

Extended enum Syntax: Controlling Storage

The syntax described in the previous section is probably sufficient for nearly all uses of **class name** declarations, but occasionally you may need more control over storage. The full syntax is as follows:

```
enum class enumeration_type : storage_type {
     symbols
};
```

For example, you can specify that your symbols should be implemented by the C++ as unsigned long integers.

```
enum class Choice : unsigned long {
     rock, paper, scissors
};
```

Another flexibility of the syntax is that you can optionally specify values for the symbols, as follows:

```
enum class Numbers {
     zero,
     ten = 10;
     eleven,
     twelve,
     hundred = 100,
     hundred_and_one
};
```

By default, enumerations start at 0. Otherwise, if not explicitly assigned a value, each symbol gets the value of the previous symbol plus 1. Therefore, in the example just shown, the symbols have the values you'd expect.

```
Numbers oceans = Numbers::eleven;  // Assign oceans 11
```

Raw-String Literals

The string literal conventions introduced in Chapter 7, "Pointers: Data by Location," create a standard C-string of type **char***. (C++ also supports a **wchar_t*** format for wide-character strings often used in international applications.)

The standard convention for string literals supports special characters such as tab and newline, but it also forces certain ordinary characters—notably \ and "—to be "escaped" by means of the backslash. For example, to represent this string data in traditional C++

```
The "file" is c:\docs\a.txt.
```

you have to use this representation:

```
char s[] = "The \"file\" is c:\\docs\\a.txt.";
```

This at least has the virtue of being unambiguous to the compiler. But the C++11 specification supports a new "raw-string" convention, whereby everything between R"(and)" is considered part of the string and no characters need to be "escaped." Everything really *is* taken literally.

```
char s[] = R"(The "file" is c:\docs\a.txt.)";
```

The R prefix signals a C++ raw-string literal. This is more readable, don't you think? In general, the syntax is as follows:

```
R"(raw-string-text)"
```

Instead of using "(" and ")" to enclose the string, you can add another character (or string of characters, up to 16 in length) to further delimit the string. This example uses "R*(" and ")*":

```
char s[] = R"*(The "file" is c:\docs\a.txt.)*";
```

The delimiter character—in this case, *—takes on its special meaning only in the special contexts shown. Otherwise, it can be used literally within the string, as can every character.

Chapter 17 *Summary*

Here are the main points of Chapter 17:

- The new C++14 specification supports the use of an apostrophe (') as a digit separator for large numbers. These are for readability only and do not affect the value of the number. For example, you can initialize a number to 10 million this way:

```
int n = 10'000'000;
```

- In the C++14 specification, the "s" suffix is supported by the standard library to enable the specification of a string literal of genuine **string** type rather than **char***. This can be particularly helpful when used with functions that have **auto** return type.

```
return "Hello"s;
```

- Finally, the C++14 specification adds support for the binary radix (that is, base 2). Such numerals work well when you are performing binary operations.

```
cout << data | 0b1111;  // Turn on low 4 bits.
```

- The C++14 specification contains all the new features introduced in C++11, some of which are only now being implemented on all compilers.
- C++11 adds support for the **long long int**, a 64-bit integer. There is also a corresponding **unsigned long long int** type. In declaring variables of either type, the **int** keyword is optional.

```
long long x = 0;
unsigned long long y = 0;
```

- For numeric literals outside the range of long integers, compilers that are C++11 and later provide new numeric literal prefixes: LL for **long long** and ULL for **unsigned long long**.

```
long long x = 1230004560012LL;
```

- The **atoll** function takes string input and returns a **long long** (64-bit) integer.
- Range-based **for** is a new syntax that says, "Do this statement for each member of specified container." The container can be an array, STL **string** object, or any STL class that supports a **begin** and **end** function, such as the **list** template.
- This **for** syntax is simplest if you don't need to alter the contents of the container during the loop:

```
for(int n : my_array)     // Print each member of
    cout << n << endl;    //   my_array
```

- Use an ampersand (&) in the variable declaration if you want to enable change of values.

```
for(int& n : my_array)    // Set each member of
    n = 0;                //   my_array to 0
```

- The **auto** keyword declares a data item in which the type is determined by context. (But once declared, the type is fixed.) For example:

```
int my_int_array[NUM_ITEMS];

for (auto x : my_int_array)
    cout << x << endl;       // x has int type
```

- The **decltype** keyword returns the type of its argument.
- Use the **nullptr** keyword to initialize a pointer that "points nowhere."

```
int *p = nullptr;
```

- Compilers that are C++11 and later support both weak and strong **enum** (enumerated types). Use of **enum class** (see the following example) creates a strongly typed set of enumerated values in which a separate namespace is created and values cannot be assigned to or from another integer type without a cast.

```
enum class type_name { symbols };
```

- The R prefix is used in C++11-complaint compilers to permit raw-string literals in which no character needs to be escaped, not even quote marks (") and backslashes (\). The sequences "(and)" delimit the string. Here is the general syntax:

```
R"(raw-string-text)"
```

This page intentionally left blank

18 Operator Functions: Doing It with Class

One of the more interesting things you can do with C++ is to define how operators work with objects of your classes. You can define a Fraction type, for example, and then create a way for the following statements to be meaningful in C++:

```
Fraction fr1(1, 2), fr2(1, 4);
cout << fr1 + fr2 << endl;
```

Wouldn't it be nice if this would add 1/2 to 1/4 and then print the characters "3/4"? Well, you can make C++ do just that. And you can make such code even more readable by adding the Fraction(string) constructor shown at the end of Chapter 11. In that case, you can write this:

```
Fraction a = "1/2", b = "1/6";
cout << a + b << endl;    // Print "2/3".
```

In this scenario, the addition operator (+) has been made into an operator function. And the technique is called *operator overloading.*

The ability to write such functions is an appealing feature of C++, but it's only useful when you want to create what (in effect) becomes a new primitive data type. In practice, this is an advanced technique only a few C++ programmers get around to using, so I've saved it for last.

Introducing Operator Functions

Keyword

The basic syntax for writing class-operator functions is fairly simple.

```
return_type operator@(argument_list)
```

In this syntax, replace the symbol @ with a valid C++ operator, such as +, –, *, or /. You can use any operator symbol supported for C++ standard types. Normal precedence and associativity rules are enforced, as appropriate, for the symbol. (See Appendix A.)

You can define an operator function as either a member function or a global (that is, nonmember) function.

- If you declare an operator function as a member function, then the object through which the function is called corresponds to the left operand.
- If you declare an operator function as a global function, then each operand corresponds to a function argument.

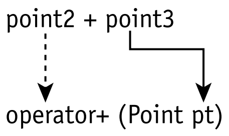
Here's how the + and – operator functions are declared inside the Point class:

```
class Point {
//...
public:

    Point operator+(Point pt);
    Point operator-(Point pt);
};
```

Given these declarations, you can apply operators to a Point object.

```
Point point1, point2, point3;
point1 = point2 + point3;
```

The compiler interprets this statement by calling the operator+ function through the left operand—point2 in this case. The right operand—point3 in this case—becomes the argument to the function. You can visualize the relationship this way:



What happens to point2? Is its value ignored? No. The function treats point2 as "this object," so that unqualified use to x and y refer to *point2's* copy of x and y. You can see how this works in this function definition:

```
Point Point::operator+(Point pt) {
    Point new_pt;
    new_pt.x = x + pt.x;
    new_pt.y = y + pt.y;
    return new_pt;
}
```

Unqualified use of data members x and y refers to values in the left operand (point2 in this case). The expressions pt.x and pt.y refer to values in the right operand (point3 in this case).

The operator function is declared with Point return type, which means it returns a Point object. This makes sense: if you add two points together, you should get another point, and if you subtract a point from another, you should get another point. But C++ allows you to specify any valid type for the return-value type.

If there is a Point(int, int) constructor, you can write the function more succinctly as follows:

```
Point Point::operator+(Point pt) {
    return Point(x + pt.x, y + pt.y);
}
```

The argument list can contain any type. Overloading is permitted here: You can declare an operator function that interacts with the **int** type, another operator function that interacts with the **float** type, and so on.

In the case of the Point class, it might make sense to permit multiplication by an integer. The declaration of the corresponding operator function (within the class) would look like this:

```
Point operator*(int n);
```

The function definition might reasonably look like this:

```
Point Point::operator*(int n) {
    Point new_pt;
    new_pt.x = x * n;
    new_pt.y = y * n;
    return new_pt;
}
```

Again, the function returns a Point object, although you could return anything you choose.

As a contrasting example, you could create an operator function that calculates the distance between two points and returns a floating-point (**double**) result. For this example, I've chosen the % operator, but you can choose any other binary operator defined in C++. The important thing here is that you can choose any return type that would be appropriate for the operation you're performing.

```
#include <cmath>

double Point::operator%(Point pt) {
    int d1 = pt.x - x;
```

```
        int d2 = pt.y - y;
        return sqrt(d1 * d1 + d2 * d2);
    }
```

Given this function definition, the following code would correctly print out the distance between points (20, 20) and (24, 23) as 5.0:

```
    Point pt1(20, 20);
    Point pt2(24, 23);
    cout << "Distance between points is :" << pt1%pt2;
```

Operator Functions as Global Functions

You can also declare operator functions as global functions. You no longer have all the relevant functions centered in the class declaration, but it's sometimes necessary to use this approach.

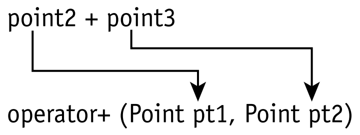
A global operator function is declared outside of any class. The types in the argument list determine what kinds of operands the function applies to. For example, the Point class addition-operator function can be written as a global function. Here's the declaration (the prototype), which should appear before the function is called:

```
    Point operator+(Point pt1, Point pt2);
```

Here's the function definition:

```
    Point operator+(Point pt1, Point pt2) {
        Point new_pt;
        new_pt.x = pt1.x + pt2.x;
        new_pt.y = pt1.y + pt2.y;
        return new_pt;
    }
```

You can visualize a call to this function this way:



Now both operands are interpreted as function arguments. The left operand (point2 in this case) gives its value to the first argument, pt1. There is no concept of "this object," and all references to Point data members must be qualified.

That can create a problem. If the data members are not public, this function cannot access them. One solution is to use function calls, if available, to get access to the data.

```
Point operator+(Point pt1, Point pt2) {
    Point new_pt;
    int a = pt1.get_x() + pt2.get_x();
    int b = pt1.get_y() + pt2.get_y();
    new_pt.set(a, b);
    return new_pt;
}
```

But that's not a pretty solution, and with some classes, it may not even work because the members might be completely inaccessible. A better solution is to declare a function as a *friend* function, which means that the function is global, but it has access to private members.

```
class Point {
//...
public:

    friend Point operator+(Point pt1, Point pt2);
};
```

Sometimes it's necessary to write an operator function as a global function. In a member function, the left operand is interpreted as "this object" in the function definition. But what if the left operand does not have an object type? What if you want to support an operation like this?

```
point1 = 3 * point2;
```

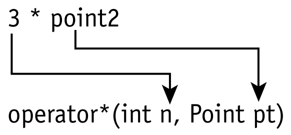
The problem here is that the left operand has **int** type, not Point type. The only way to support such an operation is to write a global function.

```
Point operator*(int n, Point pt) {
    Point new_pt;
    new_pt.x = pt.x * n;
    new_pt.y = pt.y * n;
    return new_pt;
}
```

To gain access to private data members, the function may need to be made a friend of the class.

```
class Point {
//...
public:
        friend Point operator*(int n, Point pt);
    };
```

You can visualize the call to the function this way:



Improve Efficiency with References

Every time an object is passed or returned as a value, a call to the copy constructor is issued and memory must be allocated. But you can minimize these actions by using reference types.

Here's a Point-class add function, along with an addition-operator (+) function that calls it, written without the use of reference types:

```
    class Point {
    //...
    public:

        Point add(Point pt);
        Point operator+(Point pt);
    };

    Point Point::add(Point pt) {
        Point new_pt;
        new_pt.x = x + pt.x;
        new_pt.y = y + pt.y;
        return new_pt;
    }

    Point Point::operator+(Point pt) {
        return add(pt);
    }
```

This is an obvious way to write these functions, but look how much an expression such as "pt1 + pt2" results in the creation of new objects:

- The right operand is passed to the operator+ function. A copy of pt2 is made and passed to the function.

- The operator+ function calls the add function. Now *another* copy of pt2 must be made and passed along to this function.
- The add function creates a new object, new_pt. This calls the default constructor. When that function returns, the program makes a copy of new_pt and passes it back to its caller (the operator+ function).
- The operator+ function returns the object to *its* caller, requiring yet another copy of new_pt to be made.

That's a lot of copying! Five new objects are created, involving one call to the default constructor and four calls to the copy constructor. This is extremely inefficient behavior.

Note In these days of super-fast CPUs, it may seem as if efficiency is not a factor. But you can never be sure how a class will be used, and some programs execute a loop thousands or even millions of times a second. So, when there's an easy way to make your code more efficient, you ought to take advantage of it.

You can eliminate two of these copy operations by using reference arguments. Here is the revised version, with altered lines in bold:

```
class Point {
//...
public:

    Point add(const Point &pt);
    Point operator+(const Point &pt);
};

Point Point::add(const Point &pt) {
    Point new_pt;
    new_pt.x = x + pt.x;
    new_pt.y = y + pt.y;
    return new_pt;
}

Point Point::operator+(const Point &pt)
    return add(pt);
}
```

One of the benefits of using reference types, such as Point&, is that the implementation of the function calls changes, but no other change is required in the source code.

I also use the **const** keyword here; this keyword prevents changes to the argument being passed. When the function got its own copy of the argument, it couldn't alter the value of the original copy, no matter what it did. The **const** keyword preserves data protection, so that you can't accidentally alter the value of an operand.

The use of references eliminates two instances of object copying. But each time one of these functions returns, it makes a copy of an object. You can cut down on this copying by making one or both of the functions inline. The operator+ function, which does nothing more than call the add function, is a good candidate for inlining.

```
class Point {
//...
public:

   Point operator+(const Point &pt) {return add(pt);}
};
```

When the operator+ function is inlined, operations such as "pt1 + pt2" are translated directly into calls to the add function. This saves another copy operation. Now most of the copying has been eliminated. The class is much more efficient.

Example 18.1. *Point Class Operators*

You now have all the tools you need to write efficient, useful operator functions for the Point class. The following code shows a complete declaration of the Point class, along with code that tests it by operating on objects.

Code brought over from Chapter 11 is left in normal font. New or altered lines are in bold.

point3.cpp

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

class Point {
private:             // Data members (private)
    int x, y;
```

point3.cpp, cont.

```
public:              // Constructors
    Point() {set(0,0);}
    Point(int new_x, int new_y) {set(new_x, new_y);}
    Point(const Point &src) {set(src.x, src.y);}

// Operations

    Point add(const Point &pt);
    Point sub(const Point &pt);
    Point operator+(const Point &pt) {return add(pt);}
    Point operator-(const Point &pt) {return sub(pt);}

// Other member functions

    void set(int new_x, int new_y);
    int get_x() const {return x;}
    int get_y() const {return y;}
};

int main()
{
    Point point1(20, 20);
    Point point2(0, 5);
    Point point3(-10, 25);
    Point point4 = point1 + point2 + point3;

    cout << "The point is " << point4.get_x();
    cout << ", " << point4.get_y() << "." << endl;
    return 0;
}

void Point::set(int new_x, int new_y) {
    if (new_x < 0) {
        new_x *= -1;
    }
    if (new_y < 0) {
        new_y *= -1;
    }
    x = new_x;
    y = new_y;
}
```

▼ *continued on next page*

point3.cpp, cont.

```
Point Point::add(const Point &pt) {
    Point new_pt;
    new_pt.x = x + pt.x;
    new_pt.y = y + pt.y;
    return new_pt;
}

Point Point::sub(const Point &pt) {
    Point new_pt;
    new_pt.x = x - pt.x;
    new_pt.y = y - pt.y;
    return new_pt;
}
```

How It Works

This example adds a series of operator functions to the Point class.

```
Point add(const Point &pt);
Point sub(const Point &pt);
Point operator+(const Point &pt) {return add(pt);}
Point operator-(const Point &pt) {return sub(pt);}
```

To review, the operator+ function is an inline function that translates expressions such as the following into calls to the add function:

```
Point point1 = point2 + point3;
```

This expression, in effect, gets translated into:

```
Point point1 = point2.add(point3);
```

The add function, in turn, creates a new point and initializes it by adding the coordinates of "this object" (point2 in this example) to the coordinates of the argument (point3). The operator- and sub functions work in a similar manner.

This example also adds the **const** keyword to the declarations of the get_x and get_y functions. In this context, the **const** keyword says, "The function agrees not to change any data member or call any function other than another const function."

```
int get_x() const {return x;}
int get_y() const {return y;}
```

This is a useful change. It prevents accidental changes to data members, it allows the functions to be called by other **const** functions, and it allows the functions to be called by functions that have agreed not to alter a Point object.

For example, suppose you declare a **const** Point object:

```
const Point p1, p2, p3;
cout << p1.get_x();   // Is this legal?
```

The important idea here is that **const** member functions enable this second statement to be successfully executed—and it's perfectly reasonable because all it tries to do is call a member function of p1 that doesn't change it.

The rule that applies to **const** member functions is:

* **If an object is declared const, only const member functions of this object can be called. If an object is not declared const, both const and non-const functions can be called.**

EXERCISES

Exercise 18.1.1. Write a test to see how many times the default constructor and the copy constructor are called. (Hint: Insert statements that send output to **cout**; you can span multiple lines if needed, as long as the function definitions are syntactically correct.) Then, run the program with and without the reference arguments (const Point &) changed back to ordinary arguments (Point). How much more efficient is the former approach?

Exercise 18.1.2. Write and test an expanded Point class that supports multiplication of a Point object by an integer. Use global functions, aided by **friend** declarations, as described in the previous section.

Exercise 18.1.3. Write a similar class but for a three-dimensional point (Point3D).

Example 18.2. *Fraction Class Operators*

This example uses techniques similar to those in Example 18.1 to extend basic operator support to the Fraction class. As before, the code uses reference arguments (const Fraction &) for efficiency.

Fract5.cpp

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

class Fraction {
private:
    int num, den;     // Numerator and denominator.
public:
    Fraction()  {set(0, 1);}
    Fraction(int n, int d)   {set(n, d);}
    Fraction(const Fraction &src);

    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num() const  {return num;}
    int get_den() const  {return den;}
    Fraction add(const Fraction &other);
    Fraction mult(const Fraction &other);
    Fraction operator+(const Fraction &other)
        {return add(other);}
    Fraction operator*(const Fraction &other)
        {return mult(other);}

private:
    void normalize();   // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);  // Lowest Common Denom.
};

int main()
{
    Fraction f1(1, 2);
    Fraction f2(1, 3);

    Fraction f3 = f1 + f2;

    cout << "1/2 + 1/3 = ";
    cout << f3.get_num() << "/";
    cout << f3.get_den() << "." << endl;
    return 0;
}
```

Fract5.cpp, cont.

```
// -------------------------------------------------
// FRACTION CLASS FUNCTIONS

Fraction::Fraction(Fraction const &src) {
    num = src.num;
    den = src.den;
}

// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
void Fraction::normalize(){

    // Handle cases involving 0

    if (den == 0 || num == 0) {
        num = 0;
        den = 1;
    }

    // Put neg. sign in numerator only.

    if (den < 0) {
        num *= -1;
        den *= -1;
    }

    // Factor out GCF from numerator and denominator.

    int n = gcf(num, den);
    num = num / n;
    den = den / n;
}

// Greatest Common Factor
//
int Fraction::gcf(int a, int b){
    if (b == 0)
        return abs(a);
```

▼ continued on next page

Fract5.cpp, cont.

```
    else
        return gcf(b, a%b);
}

// Lowest Common Multiple
//
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}

Fraction Fraction::add(const Fraction &other) {
    Fraction fract;
    int lcd = lcm(den, other.den);
    int quot1 = lcd/other.den;
    int quot2 = lcd/den;
    fract.set(num * quot1 + other.num * quot2, lcd);
    return fract;
}

Fraction Fraction::mult(const Fraction &other) {
    Fraction fract;
    fract.set(num * other.num, den * other.den);
    return fract;
}
```

How It Works

The add and mult functions are taken from previously existing code in the Fraction class. All I've done is change the type of the argument so that each of these functions uses reference arguments, providing a more efficient implementation.

```
Fraction add(const Fraction &other);
Fraction mult(const Fraction &other);
```

When the declarations of these functions change, the function definitions must change as well, to reflect the altered argument type. But this change affects only the function heading (shown in bold). The rest of the definitions stay the same.

```
Fraction Fraction::add(const Fraction &other) {
    Fraction fract;
    int lcd = lcm(den, other.den);
```

```
        int quot1 = lcd/den;
        int quot2 = lcd/other.den;
        fract.set(num * quot1 + other.num * quot2, lcd);
        return fract;
    }

    Fraction Fraction::mult(const Fraction &other) {
        Fraction fract;
        fract.set(num * other.num, den * other.den);
        return fract;
    }
```

The operator functions do nothing more than call the appropriate member function (add or mult) and return the value. This is because of how the inline operator+ and operator* functions are written. For example, when the compiler sees the expression

```
f1 + f2
```

it translates this expression by making the following function call:

```
f1.operator+(f2)
```

Similarly, when the compiler sees the expression

```
f1 * f2
```

it translates this function into:

```
f1.mult(f2)
```

The statements in the main function test the operator-function code by declaring fractions, adding them, and printing the results.

Optimizing the Code

The Fraction class has a useful Fraction(int, int) constructor. You can take advantage of this constructor, revising the add and mult functions to be more succinct so that they don't have to call the set function.

```
Fraction Fraction::add(const Fraction &other) {
    int lcd = lcm(den, other.den);
    int quot1 = lcd/den;
    int quot2 = lcd/other.den;
    return Fraction(num * quot1 + other.num * quot2,
      lcd);
}
```

```
Fraction Fraction::mult(const Fraction &other) {
  return Fraction(num * other.num, den * other.den);
}
```

There's another important way in which you can improve the class, which I mentioned earlier in regard to the Point class: you can declare most of the member functions to be **const** functions.

Which ones should be so declared? The answer is simple: if a member function does not change the object through which it is declared, it should ideally be declared as a **const** member function. In many programs this won't matter, but it does matter if there's any chance the class user will declare **const** objects.

```
const Fraction one_half(1, 2), one_third(1, 3);
```

These objects will only support calls to member functions that are also declared **const**. (Non-const objects can call both const and non-const functions.) Such member functions, in turn, are barred from making any changes to the object through which they are called.

So which member functions should be **const**? The answer is, most of them. The constructors should not be declared **const**, and neither should the functions that change the contents of the object: set and normalize. However, all the operator functions in this particular case should ideally be **const**, because although they might create a new object, they make no change to the object through which they are called.

To declare a member function **const**, place the keyword right after its declaration but before the semicolon or opening brace. For example:

```
    Fraction add(const Fraction &other) const;
    Fraction mult(const Fraction &other) const;
```

EXERCISES

Exercise 18.2.1. Revise the main function of the example so it prompts for a series of fraction values, exiting the input loop when 0 is entered for a denominator. Make the program track the sum of all the fractions entered and print the result.

Exercise 18.2.2. Write an operator– function (subtraction) for the Fraction class.

Exercise 18.2.3. Write an operator/ function (division) for the Fraction class.

Exercise 18.2.4. Revise the Fraction class so that every member function is declared **const**, except for the functions for which this would be inappropriate (set, normalize, and the constructors).

Working with Other Types

Thanks to overloading, you can write many different functions for each operator, in which each function works on different types. For example:

```
class Fraction {
//...
public:
    operator+(const Fraction &other);
    friend operator+(int n, const Fraction &fr);
    friend operator+(const Fraction &fr, int n);
}
```

Each of these functions deals with a different combination of **int** and Fraction operands, enabling you to support expressions like this:

```
Fraction  fract1;
fract1 = 1 + Fraction(1, 2) + Fraction(3, 4) + 4;
```

But there's an easier way to support operations with integers. All you really need is a function that converts integers to Fraction objects. If such an operation were in place, you'd only need to write one version of the operator+ function. In an expression such as the following, the compiler would convert the number 1 into Fraction format and then call the Fraction::operator+ function to add two fractions.

```
Fraction fract1 = 1 + Fraction(1, 2);
```

It turns out that such a conversion function is easy to write—it's supplied by the Fraction constructor that takes a single **int** argument! This is a simple constructor and it can be made an inline function for efficiency.

```
    Fraction(int n)  {set(n, 1);}
```

The Class Assignment Function (=)

When you write a class, the C++ compiler automatically supplies certain functions for you. I've introduced two of these so far, and this section introduces a third.

- The default constructor. The compiler version initializes nothing. Also, the compiler yanks this constructor away if you write any constructors of your own. To be safe, you should write your own default constructor, unless you want to force the class user to initialize the object.

- The copy constructor. The behavior of the automatic version is to perform a simple member-by-member copy of the source object.
- The assignment-operator function (=). This is the new one.

The compiler supplies an assignment-operator function if you don't. That's why you've been able to do operations such as this one:

```
f1 = f2;
```

Chapter 15, "Object-Oriented Poker," relied on this behavior, as it assigned objects returned by a function to an array of those objects. The data in a Card object was, in each case, copied directly into the array with base type Card.

The compiler-supplied operator= function is similar to the compiler-supplied copy constructor: it performs a simple member-by-member copy (or rather, assignment). But remember that the copy constructor creates a new object, so they are not identical.

To write your own assignment-operator function, use the following syntax:

```
class_name& operator=(const class_name &source_arg)
```

This function is similar to the copy constructor, but it should return a reference to an object of the class rather than create a new object. Here's what the operator= function might look like for the Fraction class:

```
class Fraction {
//...
public:
    Fraction& operator=(const Fraction &src) {
        set(src.num, src.den);
        return *this;
    }
};
```

This code involves the use of the keyword, **this**. The keyword in this case gives a pointer to the current object—that is, the object though which the member function was called.

Keyword

```
this
```

The effect of "return *this;" is to return the object itself! And yet, because of the way the function is declared, what gets returned is a reference, not a copy. That's exactly what is supposed to happen in a C++ assignment, by the way: the value produced by an assignment is actually a reference to the left operand, which is what makes it possible to write code such as the following:

```
int a, b, c;
a = b = c = 0;   // Assign 0 to all of these variables.
```

But for now, it's enough to know that for a class like this one, you don't need to write an assignment-operator function at all. The default behavior is adequate here, and the compiler always supplies this operator function if you don't.

An assignment operator would be needed in cases in which each object of the class owned resources in addition to its data members; resources that, for example, were allocated upon construction. This would be the case if you were to write your own version of the **string** class from scratch.

The Test-for-Equality Function (==)

The compiler supplies an assignment-operator (=) function if you don't write one, but test for equality is another matter. The compiler does *not* automatically supply an operator== function for your classes. So the following code does not work if you don't write the required function:

```
Fraction f1(2, 3);
Fraction f2(4, 6);

if (f1 == f2) {
    cout << "The fractions are equal.";
} else {
    cout << "The fractions are not equal.";
}
```

What this ought to do, of course, is print a message stating that the fractions are equal, even though different numbers (2/3 vs. 4/6) were entered.

Thanks to the normalize function we've written for the Fraction class, comparisons will work correctly. If the numerators and denominators are both equal, then the fractions are equal. Therefore, the operator== function can be written as follows:

```
bool Fraction::operator==(const Fraction &other) {
    if (num == other.num && den == other.den) {
        return true;
    } else {
        return false;
    }
}
```

This function definition can be made even more concise:

```
bool Fraction::operator==(const Fraction &other) {
    return (num == other.num && den == other.den);
}
```

The function definition is now short enough that it can be reasonably inlined.

```
class Fraction {
//...
public:
    int operator==(const Fraction &other) {
        return (num == other.num && den == other.den);
    }
};
```

A Class "Print" Function

It's annoying to have to keep writing essentially the same lines of code every time we want to print the contents of a fraction.

```
    cout << f3.get_num() << "/";
    cout << f3.get_den() << "." << endl;
```

The obvious way to reduce this work is to write a function. You can even name a member function "print," because it is not a reserved word in C++.

```
void Fraction::print() {
    cout << num << "/";
    cout << den;
};
```

But what you'd really like to do would be to print an object this way:

```
cout << fract;
```

The way to support such statements is to write an operator<< function that interacts with **cout**'s parent class, **ostream**. The function must be a global function, because the left operand is an object of the **ostream** class, and we don't have access to updating or altering **ostream** code.

The function should be declared as a friend of the Fraction class so that it has access to private members.

```
class Fraction {
//...
public:
friend ostream &operator<<(ostream &os, Fraction &fr);
};
```

This function returns a reference to an **ostream** object. This is necessary so that statements such as the following work correctly:

```
cout << "The fraction's value is " << fract << endl;
```

Here's a definition for the operator<< function. This will print the Fraction object in a pleasing form, such as "2/3".

```
ostream &operator<<(ostream &os, Fraction &fr) {
    os << fr.num << "/" << fr.den;
    return os;
}
```

This solution directs the Fraction output to be sent to any **ostream** object specified. For example, if outfile is a text-file output object, you can use it to print a fraction to the file.

```
outfile << fract;
cout << "The object " << fract;
cout << " was printed to a file." << endl;
```

Example 18.3. *The Completed Fraction Class*

Here is a more or less complete version of the Fraction class, along with code to test it. As before, only code that is new is shown in bold here.

Fract6.cpp

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

class Fraction {
private:
    int num, den;      // Numerator and denominator.
public:
    Fraction() {set(0, 1);}
    Fraction(int n, int d)  {set(n, d);}
    Fraction(int n)  {set(n, 1);}
    Fraction(const Fraction &src);
```

▼ *continued on next page*

Fract6.cpp, cont.

```
    void set(int n, int d)
        {num = n; den = d; normalize();}
    int get_num() const {return num;}
    int get_den() const {return den;}
    Fraction add(const Fraction &other);
    Fraction mult(const Fraction &other);
    Fraction operator+(const Fraction &other)
        {return add(other);}
    Fraction operator*(const Fraction &other)
        {return mult(other);}
    bool operator==(const Fraction &other);
    friend ostream &operator<<(ostream &os,
        Fraction &fr);

private:
    void normalize();   // Convert to standard form.
    int gcf(int a, int b);  // Greatest Common Factor.
    int lcm(int a, int b);  // Lowest Common Denom.
};

int main()
{
    Fraction f1(1, 2);
    Fraction f2(1, 3);

    Fraction f3 = f1 + f2 + 1;

    cout << "1/2 + 1/3 + 1 = " << f3 << endl;
    return 0;
}

// -------------------------------------------------
// FRACTION CLASS FUNCTIONS

Fraction::Fraction(Fraction const &src) {
    num = src.num;
    den = src.den;
}

// Normalize: put fraction into standard form, unique
//  for each mathematically different value.
//
```

Fract6.cpp, cont.

```
void Fraction::normalize(){

    // Handle cases involving 0

    if (den == 0 || num == 0) {
        num = 0;
        den = 1;
    }

    // Put neg. sign in numerator only.

    if (den < 0) {
        num *= -1;
        den *= -1;
    }

    // Factor out GCF from numerator and denominator.

    int n = gcf(num, den);
    num = num / n;
    den = den / n;
}

// Greatest Common Factor
//
int Fraction::gcf(int a, int b){
    if (b == 0)
        return abs(a);
    else
        return gcf(b, a%b);
}

// Lowest Common Multiple
//
int Fraction::lcm(int a, int b){
    int n = gcf(a, b);
    return a / n * b;
}
```

▼ continued on next page

Fract6.cpp, cont.

```
Fraction Fraction::add(const Fraction &other) {
    int lcd = lcm(den, other.den);
    int quot1 = lcd/den;
    int quot2 = lcd/other.den;
    return Fraction(num * quot1 + other.num * quot2,
      lcd);
}

Fraction Fraction::mult(const Fraction &other) {
  return Fraction(num * other.num, den * other.den);
}

bool Fraction::operator==(const Fraction &other) {
    return (num == other.num && den == other.den);
}

// ---------------------------------------------------
// FRACTION CLASS FRIEND FUNCTION

ostream &operator<<(ostream &os, Fraction &fr) {
    os << fr.num << "/" << fr.den;
    return os;
}
```

How It Works

This example adds just a few more capabilities to the Fraction class:

- A constructor that takes a single **int** argument
- An operator function that supports the test-for-equality operator (==)
- A global function that supports printing Fraction objects to an **ostream** object such as **cout**

A benefit of having a Fraction(int) constructor is that the program automatically converts integers to Fraction objects as needed.

```
Fraction(int n) {set(n, 1);};
```

The action of this function is to use whatever number is specified as numerator and use 1 for the denominator. 1 is converted to 1/1, 2 is converted to 2/1, 5 is converted to 5/1, and so on.

The other new extensions to the Fraction class incorporate code introduced in previous sections. First, the class declaration is expanded so that it declares two new functions.

```
        int operator==(const Fraction &other);
        friend ostream &operator<<(ostream &os, Fraction &fr);
```

The operator<< function is a global function but is also a friend of the Fraction class. It can therefore access private data (specifically, num and den).

```
    ostream &operator<<(ostream &os, Fraction &fr) {
        os << fr.num << "/" << fr.den;
        return os;
    }
```

EXERCISES

Exercise 18.3.1. Alter the `operator<<` function of the example so that it prints numbers in the format "(n, d)", where n and d are the numerator and denominator (num and den members), respectively.

Exercise 18.3.2. Write greater-than (>) and less-than (<) functions, and revise the main function of the example to test these functions. For example, test whether 1/2 + 1/3 is greater than 5/9. (Hint: Remember that A/B is greater than C/D if A * D > B * C.)

Exercise 18.3.3. Write an `operator<<` function for sending the contents of a Point object to an **ostream** object (such as **cout**). Assume the function has been declared as a friend function of the Point class. Write the function definition.

Exercise 18.3.4. Revise the class to declare all member functions **const**, except for those few (set, normalize, and the constructors) for which this would not be appropriate.

A Really Final Word (about Ops)

Writing class operator functions is an appealing side of C++. But it's only occasionally used in declaring new classes, even by advanced programmers.

Why isn't it used more? One reason is that it's potentially a good deal of work just to produce something that might be labeled "syntactic sugar." It's primarily a convenient trick that enables someone to write

```
    fract1 + fract2
```

instead of

```
fract1.add(fract2)
```

Admittedly, the operator-overloaded version (the first one) saves some typing. But some programming firms actively discourage their C++ programmers from writing operator functions because such usage doesn't improve efficiency at runtime and potentially hinders it.

Yet this book has been using the operator-function capability all along. Writing an operator function is also called *operator overloading*, which means redefining what an operator does when it's applied to a particular kind of object. We've been using this all along in the case of **cin** and **cout**.

```
cout << "Enter number to store in n: ";
cin >> n;
```

The so-called "stream operators," << and >>, are really the bit-shift operators from the old C language, redefined to do something different when applied to a stream object. Although this usage is convenient, it does not set a good precedent. (It's probably worth it, though, because the visual metaphor it creates in this case is so ideal.) Usually, the general, abstract meaning of an operator should remain the same, even when applied in a new context. For example, you'd expect the plus sign (+) to always perform some sort of addition—which, arguably, it does for **string** objects.

Why, then, did C++ designer Bjarne Stroustrup put the operator overloading capability into C++ from the beginning? No doubt it was to help achieve one of the design goals of C++: for the language was always more than "C with classes"; it was a way of building powerful and flexible data types, almost as if you were extending the language itself.

And that turns out to be one of the best ways to understand object orientation, at least as it is realized in C++. At a certain level, classes are extremely sophisticated user-defined types, empowered to do all kinds of interesting things and every bit as convenient as the primitive types—**int**, **double**, and so on. In the final analysis, there's probably no programming language in existence that offers all the possibilities and choices that C++ does.

Chapter 18 *Summary*

Here are the main points of Chapter 18:

- An operator function for a class has the following declaration, in which @ stands for any valid C++ operator.

  ```
  return_type operator@(argument_list)
  ```

- An operator function may be declared as a member function or a global function. If it is a member function, then (for a binary operator) there is one argument. For example, the operator+ argument for the Point class could have this declaration and definition:

```
class Point {
//...
public:
    Point operator+(Point pt);
};

Point Point::operator+(Point pt) {
    Point new_pt;
    new_pt.x = x + pt.x;
    new_pt.y = y + pt.y;
    return new_pt;
}
```

- Given this code, the compiler now knows how to interpret the addition sign when applied to two objects of the class. This expression produces another object of the Point class:

```
point1 + point2
```

- When an operator function is used this way, the left operand becomes the object through which the function is called, and the right operand is passed as an argument. So in the operator+ definition just shown, unqualified references to x and y refer to the values of the left operand.
- Operator functions can also be declared as global functions. For a binary operator, the function has two arguments. For example:

```
Point operator+(Point pt1, Point pt2) {
    Point new_pt;
    new_pt.x = pt1.x + pt2.x;
    new_pt.y = pt1.y + pt2.y;
    return new_pt;
}
```

- One drawback of writing the operator function this way is that it loses access to private members. To overcome this limitation, declare the global function as a friend of the class. For example:

```
class Point {
//...
public:
    friend Point operator+(Point pt1, Point pt2);
};
```

- If an argument takes an object but does not need to alter it, you can often improve the efficiency of a function by revising it to use a reference argument—for example, changing an argument of type "Point" to type "const Point&."
- A constructor with one argument provides a conversion function. For example, the following constructor enables automatic conversion of integer data into Fraction-class format:

```
Fraction(int n) {set(n, 1);};
```

- If you don't write an assignment-operator function (=), the compiler automatically supplies one for you. The behavior of the compiler-supplied version is to perform a simple member-by-member assignment.
- The compiler does not supply a test-for-equality function (==), so you need to write your own if you want to be able to compare objects. It's a good idea to use the **bool** return type, if your compiler supports it; otherwise, use the **int** return type for this function.
- To write a "print" function for a class, write a global operator<< function; the first argument should have the **ostream** type, so that the stream operator (<<) is supported for **cout** and other output-stream classes. You should first declare this function as a friend to your class. For example:

```
class Point {
//...
public:
  friend ostream &operator<<(ostream &os, Fraction &fr);
};
```

- In the function definition, the statements should write data from the right operand (fr in this case) to the ostream argument. Then the function should return the ostream argument itself. For example:

```
ostream &operator<<(ostream &os, Fraction &fr) {
    os << fr.num << "/" << fr.den;
    return os;
}
```

A Operators

Table A.1 lists C++ operators with their precedence, associativity, description, and syntax. The levels—to which I've assigned numbers—have no significance beyond the fact that all operators at the same level have equal precedence.

Associativity can be left-to-right or right-to-left. This matters when two operators are at the same level of precedence. For example, in the expression

```
*p++
```

the * and ++ operators are at the same level of precedence (level 2), so the order of evaluation is determined by associativity—in this case, right-to-left. The expression is therefore evaluated as if written this way:

```
*(p++)
```

meaning that it is the pointer p itself (not what it points to) that gets incremented.

Note that level-2 operators in this table are unary operators; that is, they operate on only one expression. Most other operators are binary, having two operands. Some operators (such as *) have both a unary and binary version—and they do very different things.

The items in the syntax column represent several kinds of expressions:

- *expr*: Any expression.
- *num*: Any numeric expression (including **char**).
- *int*: An integer (also includes **char**).
- *ptr*: A pointer (that is, an address expression).
- *member*: A member of a class.
- *lvalue*: An item that can legally be the target of an assignment. This includes a variable, an array element, a reference, or a fully dereferenced pointer. Literals and array names can never be lvalues.

Table A.1: C++ Operators by Precedence Level

LEVEL	ASSOCIATION	OPERATOR	DESCRIPTION	SYNTAX
1	L-to-R	()	Function call	*func*(*args*)
1	L-to-R	[]	Access array element	*array*[*int*]
1	L-to-R	–>	Access class member	*ptr*–>*member*
1	L-to-T	.	Access class member	*object*.*member*
1	L-to-R	::	Clarify scope	*class*::*name* ::*name*
2	*R-to-L*	!	Logical negation	!*expr*
2	*R-to-L*	~	Bitwise negation	~*int*
2	*R-to-L*	++	Increment	++*num* *num*++
2	*R-to-L*	--	Decrement	--*num* *num*--
2	*R-to-L*	–	Change sign of	–*num*
2	*R-to-L*	*	Get contents at (dereference)	*ptr
2	*R-to-L*	&	Get address of	&*lvalue*
2	*R-to-L*	**sizeof**	Get size of data in bytes	**sizeof**(*epxr*)
2	*R-to-L*	**new**	Allocate data object(s)	**new** *type* **new** *type*[*int*] **new** *type*(*args*)
2	*R-to-L*	**delete**	Remove data object(s)	**delete** *ptr* **delete** [] *ptr*
2	*R-to-L*	cast	Change type	(*type*) *expr*
3	L-to-R	.*	Pointer-to-member (rarely used)	*obj*.**ptr_mem*
3	L-to-R	–>*	Pointer-to-member (rarely used)	*ptr*–>**ptr_mem*
4	L-to-R	*	Multiply	*num* * *num*
4	L-to-R	/	Divide	*num* / *num*
4	L-to-R	%	Modulus (remainder)	*int* % *int*
5	L-to-R	+	Addition	*num* + *num* *ptr* + *int* *int* + *ptr*
5	L-to-R	–	Subtraction	*num* – *num* *ptr* – *int* *ptr* – *ptr*
6	L-to-R	<<	Left shift (bitwise); also stream op	*expr* << *int*
6	L-to-R	>>	Right shift (bitwise); also stream op	*expr*>>*int*

Table A.1: C++ Operators by Precedence Level (*continued*)

LEVEL	ASSOCIATION	OPERATOR	DESCRIPTION	SYNTAX
7	L-to-R	<	Is less than?	*num* < *num* *ptr* < *ptr*
7	L-to-R	<=	Is less than or equal?	*num* <= *num* *ptr* <= *ptr*
7	L-to-R	>	Is greater than?	*num* > *num* *ptr* > *ptr*
7	L-to-R	>=	Is greater than or equal?	*num* >= *num* *ptr* >= *ptr*
8	L-to-R	==	Test for equality	*num* == *num* *ptr* == *ptr*
8	L-to-R	!=	Test for inequality	*num* **!=** *num* *ptr* **!=** *ptr*
9	L-to-R	&	Bitwise AND	*int* **&** *int*
10	L-to-R	^	Bitwise XOR (exclusive OR)	*int* ^ *int*
11	L-to-R	|	Bitwise OR	*int* | *int*
12	L-to-R	&&	Logical AND	*expr* **&&** *expr*
13	L-to-R	||	Logical OR	*expr* || *expr*
14	R-to-L	?:	Conditional operator: evaluate expr1: if nonzero, evaluate and return expr2; otherwise, evaluate and return expr3	*expr1* **?** *expr2* **:** *expr3*
15	R-to-L	=	Assign	*lvalue* = *expr*
15	R-to-L	+=	Add and assign	*lvalue* += *expr*
15	R-to-L	−=	Subtract and assign	*lvalue* −= *expr*
15	R-to-L	*=	Multiply and assign	*lvalue* *= *expr*
15	R-to-L	/=	Divide and assign	*lvalue* **/**= *expr*
15	R-to-L	%=	Modular divide and assign	*lvalue* **%**= *expr*
15	R-to-L	>>=	Right shift and assign	*lvalue* >>= *expr*
15	R-to-L	<<=	Left shift and assign	*lvalue* <<= *expr*
15	R-to-L	&=	Bitwise AND and assign	*lvalue* **&**= *expr*
15	R-to-L	^=	Bitwise XOR and assign	*lvalue* ^= *expr*
15	R-to-L	|=	Bitwise OR and assign	*lvalue* |= *expr*
16	R-to-L	,	Join (evaluate both expressions and return expr2)	*expr1*, *expr2*

The rest of this appendix provides more detail on some of the operators, in order of precedence.

- The (::) scope operator
- The sizeof operator
- Cast operators
- Integer vs. floating-point division
- Conditional (?:) operator
- Assignment operators
- Join (,) operator

The Scope (::) Operator

This operator has several related uses. First, it can be used to refer to a symbol declared in a class or namespace.

```
class::symbol_name

namespace::symbol_name
```

The scope operator can also be used to refer to a global—or rather, an unqualified—name. This might be used, for example, to refer to a global symbol from within a class's member function when there is a name conflict.

```
::symbol_name
```

The sizeof Operator

The **sizeof** operator returns the size of the type of its operand in bytes.

- If **sizeof** is used on a pointer, it returns the width of the pointer itself (currently 4 bytes on today's 32-bit computers), not the base type.

```
double x = 0.0;
double *p = x;
cout << sizeof(p);   // Print 4.
cout << sizeof(x);   // Print 8.
```

- If **sizeof** is used on an array, it returns the total size of all the elements of the array. For example, if **sizeof(int)** is 4, then the following prints 40:

```
int arr[10];
cout << sizeof(arr);   // Print 40.
```

- **sizeof** can be used directly on a type name (including class names).

```
cout << sizeof(char);   // Print 1.
```

Old- and New-Style Type Casts

Key Syntax

For backward compatibility, C++ supports the old-style C type cast:

```
(type) expression
```

The C++ standards committee had long planned to deprecate this usage. A deprecated usage is one that the compiler advises the programmer against using, generating a warning message. However, because C++ is still being used to compile large amounts of C legacy code, the committee decided not to deprecate this type cast.

The four new-style casts are still preferred within the C++ community (see Table A.2). Admittedly, they are longer and take more work to put into programs, but they have the advantage of being more self-documenting. The habit of using these operators (and not the old-style C type cast) reduces the possibility of using an improper cast accidentally.

Table A.2: C++ Cast Operators (New Style)

CAST SYNTAX	DESCRIPTION
static_cast<*type*>(*expression*)	Recasts *expression* into the data format of *type*, such as casting **double** to **int** (and removing warning messages), or to cast to or from an **enum** type. Essentially, **static_cast** says, "Yes, I really do want to do this." For the cast to work, some sort of conversion must be possible between the types involved.
reinterpret_cast<*type*>(*expression*)	Recasts one pointer type to another or casts between a pointer type and int, or vice versa. This cast is potentially dangerous (so make sure you need it before using it) because it changes how data at a particular address is to be interpreted.
dynamic_cast<*type*>(*expression*)	Casts a base-class pointer to a subclass pointer after verifying that the object pointed to has the specified subclass *type*. Produces NULL if cast is not valid. Requires that the classes involved have one or more virtual functions. This is casting *downward* through an inheritance hierarchy; going the other way (assigning subclass pointer to a base-class pointer) is freely permitted and requires no cast.
const_cast<*type*>(*expression*)	Casts a non-**const** expression to a **const** type. It is your responsibility to make sure the expression is not one that will be changed.

Integer versus Floating-Point Division

Most of the operators in Table A.1 are self-explanatory, but special considerations apply to some types. When one integer is divided by another, the remainder is thrown away.

```
int quotient = 19 / 10;    // quotient  = 1
```

In this case, the fractional portion (0.9) is discarded. To get a remainder resulting from integer division, use the modulus (%) operator.

```
int remainder = 19 % 10;   // remainder = 9
```

In the following example, integer division is performed, throwing away much of the result, even though quotient has floating-point format and could have stored the result, 1.9:

```
double quotient = 19 / 10;    // quotient  = 1.0
```

However, if one of the operands has type **double** (which is the case when a decimal point is used in a numeric literal), the other operand is promoted to **double** and floating-point division is carried out.

```
double quotient = 19 / 10.0;   // quotient = 1.9
```

Bitwise Operators (&, |, ^, ~, <<, and >>)

Bitwise AND, OR, and exclusive OR (&, |, ^) operate on two integer expressions of the same width. If one operand has a different width (size) than the other, then the smaller of the two is promoted to the larger width. The action of these operators is to compare bit-n in one operand to bit-n in the other and set bit-n in the resulting integer. For example:

```
cout << hex;

cout << (0xe & 0x3);  // 1110 & 0011 -> 0010 (AND)
cout << endl;
cout << (0xe | 0x3);  // 1110 | 0011 -> 1111 (OR)
cout << endl;
cout << (0xe ^ 0x3);  // 1110 ^ 0011 -> 1101 (XOR)
```

Note that exclusive OR (also called XOR) means “either or but not both.”

Bitwise negation (~) is a unary operator that produces a result containing the reverse setting of each bit in its operand. For example:

```
cout << hex;

cout << (~(char)0xff); // 1111 1111 -> 0000 0000 (0)
cout << endl;
cout << (~(char)0x89); // 1000 1001 -> 0111 0110 (76)
```

When used on integers, double-angle brackets (<< and >>) are not stream operators but bit-shift operators:

```
integer << number_of_positions_to_shift
integer >> number_of_positions_to_shift
```

These operators originated in the C language solely to perform bit shifts, not I/O. In C++, they are *overloaded* to work with streams—in effect becoming stream input/output operators but only through redefinition. However, they retain the same precedence and associativity no matter how they are used.

Conditional Operator

The conditional operator (?) provides a way to perform if-then-else logic within an expression; it is another option for writing extremely compact code. A simple example is that of comparing x to 1 and then printing 1 or 0 depending on the result.

```
if (x == 1) {
    cout << 1 << endl;
} else {
    cout << 0 << endl;
}
```

With the conditional operator, this could be expressed as follows:

```
cout << (x == 1 ? 1 : 0) << endl;
```

The general form of the operator is

Key Syntax

```
condition ? expr1 : expr2
```

The *condition* is evaluated. If true (nonzero), then *expr1* is evaluated and returned as the value of the overall expression; otherwise, *expr2* is evaluated and returned as the value of the overall expression.

Precedence of the conditional operator is very low, so a conditional expression (like that just shown) typically needs to be enclosed in parentheses.

Assignment Operators

All the assignment operators return the value that was assigned, thus permitting multiple assignments such as the following:

```
x = y = z = 0;
```

Many assignment operators are provided as convenient shorthand for some operate-and-assign action. For example, the expression

```
i += 1;
```

is functionally equivalent to the following:

```
i = i + 1;
```

Similarly, for all the operators such as \=, *=, −=, and so on. Note that the expression

```
(i += 1)
```

is equivalent to

```
(++i)
```

because they both say, “Add 1 to the value of i, and then pass this value along to the larger expression.”

Join (,) Operator

The join, or comma, operator is a way of combining multiple expressions in the space of a single expression. This is useful in **for** statements in which there’s a need to initialize or increment more than one variable.

```
for (int i = 0, int j = 0; ; i++, j++) {
    ...
```

In general, the action of the comma operator is to evaluate both conditions on either side of the comma (,) and then return the value of the second expression. In addition to **for**, another useful situation is the following, where the comma is used to execute several actions at the top of a loop before finally testing the condition i < 10:

```
while (i = j + 1, cout << "i", i < 10) {
    i++;
}
```

This operator (,) has the lowest precedence of all C++ operators.

B Data Types

Although the C++ specification is somewhat general when it comes to ranges of types, certain ranges are (for all practical purposes) universal on computers with a 32-bit architecture. This includes all personal computers in use today, both PCs and Macs. However, some of these ranges are subject to change. When 64-bit architecture becomes standard, for example, you should expect **int** to be identified with 64-bit integers.

The **int** and **double** types are the "natural" sizes for integer and floating-point numbers, respectively, for the computer's own architecture. This means that when any integer type is used in an expression (such as **char**), it is automatically promoted to **int** provided that can be done without loss of information. There is never any reason to use **short** or **float** except where compact storage formats on disk or other data streams require it.

Table B.1 lists data types and their ranges on 32-bit computers and is followed by sections describing other issues in data-type storage. I use "billion" here in the American sense: a thousand million (1,000,000,000).

Here are some notes on version support:

- Some types are marked "ANSI." Almost all but the oldest compilers now support ANSI-required types, so unless your compiler is extremely out of date, it will support these.
- Some types are marked "C++11." Compilers that are compliant with C++11 or later support these, and that should include all compilers that claim to support C++14, such as Microsoft Community Edition.

Table B.1: C++ Intrinsic Data Types

TYPE	DESCRIPTION (FOR 32-BIT SYSTEMS)	RANGE (ON 32-BIT SYSTEMS)
char	1-byte integer (used to hold ASCII character value)	0 to 255
unsigned char	1-byte unsigned integer	0 to 255
signed char	1-byte signed integer	–128 to 127
short	2-byte integer	–32,768 to 32, 767
unsigned short	2-byte unsigned integer	0 to 65,535
Int	4-byte integer (but same as short on 16-bit systems)	Approx. ± 2 billion
unsigned int	4-byte unsigned integer (but same as unsigned short on 16-bit systems)	Approx. 4 billion
long	4-byte integer	Approx. ± 2 billion
unsigned long	4-byte unsigned integer	Approx. 4 billion
bool	Integer in which all nonzero values are converted to true (1); also holds false (0) (ANSI)	**true** or **false**
wchar_t	Wide character, for holding Unicode characters (ANSI)	Same as **unsigned int**
long long	64-bit signed integer (C++11)	Approx. ±9 x 10 to the 18th
unsigned long long	64-bit unsigned integer (C++11)	Approx. 1.8 x 10 to the 19th
float	Single-precision floating point	3.4 x 10 to the 38th
double	Double-precision floating point	1.8 x 10 to the 308th
long double	Extra-wide double-precision (ANSI)	At least as great as **double**

Precision of Data Types

All integer types have absolute precision at all times. That is one of their chief advantages. For example, you can be close to the top of the **long long** range, and adding 1 to the amount is accurately reflected in the new value, whereas adding 1 to a very high **float** number might have no effect, with the added value of 1 lost because of rounding.

- The **float** type has 7 (decimal) digits of precision.
- The **double** type has 15 (decimal) digits of precision.
- Values in **float** format can store the value 0.0 precisely. They can also store tiny values as close to zero as 1.175 x 10 to the –38th.
- Values in **double** format can store the value 0.0 precisely. They can also store tiny values as close to zero as 2.225074 x 10 to the –308th.

Data Types of Numeric Literals

In C++ (and programming generally), a *literal* is a series of characters that the compiler immediately recognizes as a fixed value upon reading. In the core language, these are always numbers and text strings. A literal is different from a symbol (usually a variable, class, or function name), which has to have a value assigned to it.

```
int i = 23;                     // 23 is a literal
int j = number_of_students;     // Not a literal
int k = MAX_PATH;               // Not a literal
```

In these statements, 23 is the only literal that appears. MAX_PATH may be changed into a literal during preprocessing (where a **#define** statement may replace it with a literal value such as 256), but it is not yet a literal.

All literals are constants, but not all constants are literals. For example, array names are constants in C and C++, but they are symbols, not literals.

The default numeric format is decimal (base 10). The default storage for whole numbers is **int** type. But several other numeric formats may be used with literals:

- The **0x** prefix specifies hexadecimal (base 16).
- A leading 0 specifies octal (base 8).
- With compilers that are fully C++14 compliant, the **0b** prefix specifies binary radix (base 2).
- Scientific notation indicates floating-point format: The literal is stored in **double** format.
- Use of a decimal point, even if followed by 0, indicates floating-point format; again, the literal is stored in **double** format.

Here's an example:

```
int a = 0xff;      // Assign 1111 1111 (256) to a.
int b = 0100;      // Assign octal 100 (64) to b.
double x = 3.14;   // Assign flt. pt. number
double y = 3.0;    // This also assigns flt pt.
double z = 1.6e5;  // Use of scientific notation:
                   //  1.6 times 10 to the fifth
```

In addition, several suffixes may affect how the value of a literal is stored. Storage of a literal can sometimes matter because it can affect the precision it has, what range is permitted, or what conversion has to be applied later on to copy the data to another location. Furthermore, some integer values cannot be represented without applying the proper suffix.

- The **L** suffix indicates storage of an integer in **long int** format. **long** is equivalent to **int** on most computers in use today.
- The **U** suffix indicates storage of an as **unsigned int**. (This doubles the range supported for positive numbers; see "Two's-Complement Format for Signed Integers" later in this appendix.)
- The **F** suffix indicates storage in **float** format (usually 4-byte floating point) rather than **double** (8-byte floating point). Usually this is unnecessary or even undesirable but might be necessary in a situation in which you were reading 4-byte floating-point numbers from a binary file, for example.
- If **long long** is supported, then the **LL** and **ULL** suffixes are supported for **long long** and **unsigned long long** formats, respectively. Some integer values are so large they cannot be written out as literals except with one of these suffixes (because otherwise the compiler attempts to fit integer literals into **int** format).

Also note that if your compiler is fully C++14 compliant, you can use single quote marks (') in numeric literals as digit-group separators. See Chapter 17 for more information.

String Literals and Escape Sequences

Ordinary string literals have **char*** format. They are translated into a **char*** array with one byte allocated for each character plus an extra byte for the null terminator.

```
char str[] = "This is a string. ";
```

Wide character strings are similar, but to indicate a wide-character literal, use an "L" prefix: this causes the compiler to allocate a **wchar_t** array, which includes two bytes for each character, including the null terminator.

```
wchar_t unicode_str[] = L"This is a Unicode string. ";
```

The appearance of a backslash in a string literal indicates that the backslash itself—along with the very next character—is not interpreted as a backslash but instead has special meaning as an *escape character*. Table B.2 summarizes the meaning of escape characters.

Table B.2: Escape Characters in C++

ESCAPE CHARACTER	MEANING
\'	Literal single quotation mark
\"	Literal double quotation mark (necessary because otherwise a quotation mark is recognized as terminating the string literal)
\\	Literal backslash
\a	Bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\nnn	ASCII character corresponding to number nnn, where nnn is an octal number (base 8)
\x*hh*	ASCII character corresponding to number hh, where hh is a hexadecimal number (base 16)

Two's-Complement Format for Signed Integers

Virtually all personal computers in use today (including Macs) use two's-complement format for storing signed integers. Two's complement is a technique for representing negative numbers along with positive numbers. Although the leftmost bit always indicates the sign of the quantity, it is not precisely the same as a sign bit.

With signed formats (for example, **int** as opposed to **unsigned int**), only the bottom half of the range is used to represent positive values, along with zero. The top half of the range represents negative values. Consequently, a pattern with leftmost bit set to 1 always indicates a negative value.

Here's how the format works. To take the negative of any number, follow these steps:

1. Reverse the setting of each bit (this is logical bitwise negation, also called the one's complement).
2. Add 1.

For example, to produce –1 for a single-byte number, first start with the bit pattern for 1. Remember, we're going to get the negative by reversing each bit and adding 1.

```
0000 0001
```

First, we reverse each bit:

```
1111 1110
```

Adding 1 then produces the *two's complement*. This is therefore the two's complement representation of –1:

```
1111 1111
```

And, in fact, for every signed integer format, the setting of all 1s in every bit position always signifies –1. If we were using unsigned format, this bit pattern would instead be interpreted as 255.

Using all 1s to represent –1 is mathematically sound. If you take the negative again, you get positive 1, exactly as expected. Remember, to get the negative of any signed number, reverse each digit and then add 1. You can see how this gets us back to positive 1:

```
  1111 1111     (This is -1)

  0000 0000     Reverse each bit.
+ 0000 0001     Add 1.
===========
  0000 0001
```

The –1 value, of course, is not the lowest possible negative value. The lowest signed value is always a bit pattern of 1 followed by all 0s:

```
1000 0000
```

With a signed, two's-complement format, this is interpreted as –128. (In unsigned format, it would be interpreted as positive 128.) Adding 1 in signed format produces –127, a slightly higher number:

```
1000 0001
```

The general rule is that any signed quantity with 1 in its leftmost bit is interpreted as a negative number.

Incidentally, taking the two's complement of 0 produces 0 itself. This is mathematically correct, because multiplying 0 by –1 should produce 0.

```
  0000 0000   // Start with 0
  1111 1111   // Reverse each bit (one's complement)
+         1   //  Now add 1 to get two's complement
===========
  0000 0000   // "Flips over," producing 0 again!
```

The advantage of using two's-complement format for representing signed numbers is that so many mathematical operations work smoothly with it and don't need to check a sign bit. With a few exceptions, the same machine instructions that work on unsigned integers work correctly, and without change, on signed integers. For example, if you add any number to its negative, you end up with zero, exactly as expected.

```
0000 0001       1
1111 1111  Add -1
=========
0000 0000  Result "flips over," producing 0.
```

This page intentionally left blank

C Syntax Summary

This appendix gives a general overview of the core C++ language.

Basic Expression Syntax

Except in the case of **void** expressions, an expression is something that produces a value. Expressions are the fundamental building blocks of statements, because an expression can be turned into a statement by adding a semicolon (;).

Smaller expressions can form part of larger expressions. For example, an expression is formed by addition:

```
expression + expression
```

Each of these can be any two smaller expressions that produce a numeric value. (Also, pointers may be added to integers; see Appendix A.) The result is an expression that can be used, in turn, in still larger expressions.

In C and C++, expressions can produce side effects. For example, the following decrements j by 1, multiplies the result by 3, and then assigns that result to both x and y:

```
x = (y = 3 * --j);
```

This statement contains one long expression terminated by a semicolon (;). Note that assignment is not a kind of statement but merely another kind of expression. Several of these expressions have side effects. First, --j decrements the value of j before using j in the assignment expression:

```
y = 3 * --j
```

Assignment is an expression with a side effect, in this case, setting the value of y. As with all assignment operators (see Appendix A), the value assigned is passed along to the larger expression, which in turn assigns this same value to x.

The following are all expressions:

```
literal
symbol
expression op expression                    // (binary op)
op expression                               // (unary op)
expression op                               // (unary op)
function(args)
```

In addition, C++ supports one trinary operator: the conditional operator.

Basic Statement Syntax

Statements are the building blocks of C++ programs because a program consists of one or more functions, and each function consists of zero or more statements.

The most common form of a statement in C++ is an expression terminated by a semicolon (;). Notably, semicolons are statement terminators, not statement separators as they are in Pascal.

```
expression;
```

It is also valid to have a statement with no expression. This is an empty statement.

```
;
```

Any number of statements can be grouped together to form a compound statement (also called a *block*). Remember that a compound statement is valid anywhere a single statement is valid.

```
{ statements }
```

Each of the control structures (covered in the next four sections) also defines a statement. Control structures can therefore be nested to any level.

In addition to these statements and the control structures (**if**, **while**, **do-while**, and **switch**), there are several "branching," or direct-transfer-of-control statements: **break**, **continue**, **return**, and **goto**.

A statement can be labeled with a symbolic name (following the same rules as variable names), as follows:

```
label: statement
```

This syntax (like that for control structures) is recursive so that a statement can have multiple labels—a fact that **switch-case** statements sometimes take advantage of.

Control Structures and Branch Statements

This section contains a section on each of the statements that controls execution in a C++ program.

The if-else Statement

The **if** statement has two forms. The first form is as follows, in which *condition* is an expression that evaluates to true (any nonzero value) or false (a zero value):

```
if (condition)
    statement
```

Standard practice is to use relational expressions (such as `n > 0`), which always produce **true** or **false**, or an expression of type **bool**. It can also be effective to use a pointer as a condition in C++. If the pointer is null (for example, because a file-open attempt failed), the condition equates to false; otherwise, it equates to true.

Pointers can be compared to NULL by using logical negation (!), meaning "not." For example, in this case, several statements are executed if the file was not successfully opened:

```
ofstream fout(silly_file_name);
if (!fout) {
    cout << "Could not print ";
    cout << silly_file_name;
    return -1;
}
```

An **if** statement can have an optional **else** clause:

```
if (condition)
    statement1
else
    statement2
```

The while Statement

The **while** statement has the following syntax:

```
while (condition)
     statement
```

The *condition* is a true/false expression; see the **if** statement for rules applying to conditions.

The action of the **while** statement is to first to evaluate the *condition*. If it is true, then the *statement* is executed, and then the condition is tested again. The cycle continues until *condition* is false or unless some other action terminates the loop, such as **break**.

For example, the following code fragment prints a message five times:

```
int n = 5;

while (n-- > 0) {
    cout << "Hello.";
    cout << endl;
}
```

The do-while Statement

The **do-while** statement has the following syntax:

```
do
    statement
while (condition);
```

The operation of the **do-while** statement is the same as the **while** statement, except that with the **do-while** statement, the enclosed statement is executed at least once; the condition is evaluated afterword.

The for Statement

The **for** statement provides a compact way of using three expressions (abbreviated *expr*) to control execution of a loop.

```
for (init_expr; condition_expr; increment_expr)
    statement
```

This expression is essentially the same as the following in its behavior (except, as noted later, with regard to the continue statement):

```
init_expr;
while (condition_expr) {
     statement
     increment _expr;
}
```

The *init_expression* can declare one or more variables; if it does, the variables declared are local to the **for** statement. For example, the following code fragment prints the whole numbers from 1 to 10:

```
for (int i = 1; i <= 10; ++i) {
    cout << i << endl;    // i local to this block.
}
```

For more information, see Chapter 3, which I have devoted entirely to the **for** statement.

The switch-case Statement

The **switch-case** statement is an alternative to the use of repeated if-else. It has this syntax:

```
switch (target_expression) {
    statements
}
```

Within the statements, you can place any number of statements labeled with the case keyword. A case statement has this syntax:

```
case constant: statement
```

It follows from the recursive nature of the syntax that a single statement may (optionally) have multiple labels. For example:

```
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    cout << "is a vowel";
```

You can also include an optional default label.

```
default: statement
```

Statement labels need to be unique within the scope of the **switch** statement but not necessarily the larger program.

The action of switch is to evaluate the *target_expression*. Control is then transferred to the case statement, if any, whose constant value matches the value of *target_expression*. If none of these values matches but there is a statement labeled default, then control is transferred there. If none of the values matches and there is no default statement, control passes to the first statement after the end of the switch statement.

For example, the following code fragment prints “is a vowel,” “may be a vowel,” or “is not a vowel,” depending on the value of c.

```
switch(c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        cout << "is a vowel";
        break;
    case 'y':
        cout << "may be a vowel";
        break;
    default:
        cout << "is not a vowel";
}
```

Once control is transferred to any statement within the block, execution is continues normally, falling through unless a **break** statement is encountered. For this reason, each “case block” should usually be terminated with a **break** statement.

The break Statement

The **break** statement transfers execution out of the nearest enclosing while, do-while, for, or switch-case statement. Execution is transferred to the first statement past the end of the block.

```
break;
```

The continue Statement

The **continue** statement causes execution to be transferred to the end of the current **while**, **do-while**, or **for** statement, effectively advancing to the next iteration (cycle) of the loop.

```
continue;
```

If **continue** is used inside a **for** loop, the *increment* portion of the **for** statement is executed as part of the action of advancing to the next iteration.

The goto Statement

Traditionally, using **goto** is strongly discouraged because its overuse tends to create something called *spaghetti code*, in which the flow of control in the program resembles a tangled heap of spaghetti; these days it might also be compared to the mess of interconnected wires in the back of a home-entertainment system.

But **goto** is still useful as a way of breaking out of a deeply nested loop, since a single **break** statement would not do the job.

```
goto label;
```

The label is a symbol (that is, a name) used to label a statement within the same function. Remember that labeled statements have this syntax:

```
label: statement
```

The return Statement

The **return** statement has two forms. The first, used with **void** functions, causes immediate exit from the current function. Control returns to the caller of the function.

```
return;
```

When within functions that are not **void**, the **return** statement must return a value of the appropriate type.

```
return value;
```

Note that when used within **main**, the **return** statement returns control to the operating system. In such a case, the return value may be a success or failure code (0 usually associated with no errors, or success).

```
return EXIT_SUCCESS;
```

The throw Statement

The **throw** statement raises an exception, which must then be handled by the nearest enclosing catch block; otherwise, the program terminates abruptly.

```
throw  exception_object;
```

The *exception_object* may have any type; **catch** statements handle an object by looking for that type. The matching catch block will have either the same type or a base-class type (that is, it must be an ancestor class of the type of object thrown).

Variable Declarations

A data declaration is a statement that creates one or more variables of a particular type. If the type is a class, the variables are objects. In the following syntax, *var_decl* is one or more variable declarations, separated by commas if there is more than one:

```
modifiers  type  var_decls;
```

Each *var_decl* is a variable declaration with an optional initialization. For the meaning of the optional *modifiers*, see the end of this section.
Either of these is valid:

```
var_name

var_name = init_expression
```

The *init_expression* is any valid expression of the corresponding type or an expression that can be converted to the type. If the variable is an array, *init_expression* may also be an aggregate:

```
{ init_expression, init_expression... }
```

For example, the following statements declare three variables of type **int**; two of them are initialized:

```
int i = 0, j = 1, k;
```

And this example declares a two-dimensional array and initializes it:

```
double[2][2] = {{1.5, 3.9}, {23.0, -8.1}};
```

If the variable being declared is an object (it has class type), it can be initialized through "function-style" syntax, which passes arguments along to the appropriate constructor.

```
Fraction fract1(1, 2);
```

The C++0x specification also allows you to use aggregate-style initialization with objects:

```
Fraction fract1{1, 2};
Fraction fract2 = {10, 3};
```

In a variable declaration, the variable name can be qualified with operators, including [], *, (), and &; these create arrays, pointers, pointers to functions, and references, respectively. To determine what kind of item has been declared, ask yourself what the item would necessarily represent in executable code. For example, the variable declaration

```
int **ptr;
```

means that **ptr, when it appears in executable code, is an item of type **int**; ptr itself is therefore a pointer to a pointer to an integer, because it would have to be dereferenced twice to produce an integer. Likewise, the following declaration creates a pointer to a function—and that function must take a single double argument and return a double return value.

```
double (*fPointer)(double);
```

Pointers to functions are used as callback functions (for example, passed to the qsort library function) and in tables of functions. A pointer to a function needs to be assigned the address of a function before being called.

```
double (*fPointer)(double);
fPointer = &sqrt;
...
int x = (*fPointer)(5.0);    // Call sqrt(5)
```

The exception to this general procedure is that the ampersand (&) creates a reference during declaration. Note that reference variables need to be initialized, or else need to be reference arguments; otherwise, they have nothing to refer to. References, unlike pointers, cannot be reassigned to refer to new data.

```
int n = 0;
int &silly = n;  // silly is a reference for n.
```

The optional *modifiers* to a variable declaration can include any of the following:

- **auto**: This is largely outmoded and unnecessary. It indicates automatic storage class, which local variables have by default. (Do not confuse with **auto** variable definitions permitted by C++14, in which the **auto** keyword is used in place of a type.)
- **const**: A variable declared **const** is prevented from being changed by being on the left part of an assignment, incremented, or decremented. Also, a pointer or reference to a **const** variable may not be passed to a function unless that function is also declared **const** or unless the function declares the argument as **const**. Pointers and references to a **const** type may not change the data to which they refer.

- **extern**: An **extern** declaration gives a variable visibility among all modules of the project. (In addition to an **extern** declaration, the variable also needs to be defined in exactly one module; you can do this by initializing it and declaring it without **extern**.)
- **register**: A suggestion to the compiler that it should dedicate a register (onboard processor memory) to the variable. Modern optimizing compilers do this anyway when it would improve program performance and therefore may ignore this modifier.
- **static**: When used with a local variable or data member, it indicates there is only one copy of the variable. In the case of local variables, this means that the function "remembers" the value between function calls. It's incompatible with recursive functions.
- **volatile**: A rarely used but occasionally important keyword; **volatile** is an indicator to the compiler that it must not place a variable in a register or make any assumptions about when it can change. This is most often used when a variable corresponds to a location that is being manipulated by some outside hardware device (such as a port).

Function Declarations

Before a function can be called by another function, it must first be declared or defined. It can be given a type declaration first (a prototype). The definition can then be placed anywhere in the source code or another module linked into the project.

A function prototype has this syntax:

```
modifiers  type  function_name(argument_list);
```

For the meaning of the optional *modifiers*, see the end of this section. The *type* specifies the return value of the function. A function can optionally have **void** type, specifying that it does not return a value.

The *argument list* contains one or more argument declarations separated by commas if there are more than one. The argument list may be left blank, indicating that the function has no arguments. (Unlike C, C++ does not permit the use of a blank argument list to mean an indeterminate list.)

Each entry in the argument list has the following form. Declaration syntax follows other rules specified for variable declarations (see previous section), and it permits an optional initializing expression indicating a default argument. But note that each *type* and *var_decl* must be one-to-one.

```
type  var_decl
```

A more complete syntax for a function prototype is therefore as follows, in which there are zero or more occurrences of *type var_decl*. If there are more than one, they are separated by commas:

```
modifiers  type  function_name( type var_decl, ...);
```

Syntax for a complete declaration, including function definition, is the same, except that it includes a block with zero or more statements. (Note that any *modifiers* previously declared in a prototype do not, as a general rule, need to be repeated in the definition.)

```
modifiers  type  function_name(argument_list) {
    statements
}
```

A function definition does not end with a semicolon (;) after the final brace—unlike a class declaration. Also, note that names of arguments (but not types) can be omitted from a prototype but not from a function definition.

The optional *modifiers* can include any of the following:

- **const**: A **const** function is restricted from changing the value of its arguments and from calling other functions not declared **const**. But this permits it to be called by other **const** functions.
- **inline**: A suggestion to the compiler that the function be made an inline function. Modern optimizing compilers do this on their own whenever it would improve speed and compactness, making this keyword less necessary. In addition, a member function defined within its class declaration is automatically inlined.
- **static**: In multiple-module projects, a function automatically has external linkage unless declared with static. (Each function still needs to be prototyped in any source file that uses it, however, thus making header files necessary.)
- **virtual**: Used with member functions only. Declaring a function virtual means that calls to the function are handled through indirect calls that involve a vtable, which, in practical terms, means that the destination of the function call is not resolved until runtime. In C++, the details of carrying this out are invisible to the programmer, so you call a virtual function exactly the way you'd call any other function. The **virtual** keyword needs to be applied only once, when the function is first declared in the base class.

Class Declarations

A class declaration extends the language by creating a new type. Once a class is declared, the class name can be used directly as a type name, just like an intrinsic data type such as **int**, **double**, **float**, and so on. The basic syntax for a class declaration is as follows:

```
class class_name {
    declarations
};
```

Unlike a function definition, a class declaration is always terminated by a semicolon (;) after the closing brace.

The declarations can include any number of data and/or function definitions. Within the declarations, the **public**, **protected**, and **private** keywords can occur, along with a colon (:) to indicate the access level of the declarations that follow it. For example, in the following class declaration, data members a and b are private; data member c, as well as function f1, are public.

```
class my_class {
private:
     int a, b;
public:
     int c;
     void f1(int a);
};
```

Note when you use the **class** keyword to declare a class, members are private by default.

Within a class declaration, constructors and destructors have the following special declarations. You can have any number of constructors differentiated by argument lists. You can have at most one destructor.

```
class_name (argument_list)       // Constructor
~class_name()                    // Destructor
```

The syntax for a subclass declaration includes the name of a base class. Although the use of **public** in this context is not required, it is strongly recommended. Without it, default base-class access level is private, which makes all inherited members private.

```
class class_name : public base_class {
    declarations
};
```

Most versions of C++ support multiple inheritance, in which you list more than one base class separated by commas. For example, in this example, the class Dog is derived from both Animal and Pets and therefore inherits all members of both classes:

```
class Dog : public Animal, public Pets {
...
}
```

Note ▶ The syntax here applies to the **struct** and **union** keywords as well as **class**. A **struct** class is the same as a class defined with the **class** keyword, except that with **struct**, members are public by default, and with **class**, members are private by default. Members of a **union** class are also public by default. Members of a union share the same address in memory. (Basically, a union can be used to create a "variable data type" class in which different data formats are in use at different times.)

Enum Declarations

The **enum** keyword can be used to create a series of symbolic names (symbols) each with a constant integer value. It has this general syntax, in which *name* is optional:

```
enum name  {
    symbol_decls
};
```

In this syntax, *symbol_decl* consists of one or more names, separated by commas if there are more than one. In addition, each may optionally have an assigned value:

```
symbol = assigned_value
```

If a symbol is not assigned a value (which must be a literal or other constant), its value is that of the previous symbol plus one. If it is the first symbol and not assigned a value, it is given the value zero.

For example, the following declaration creates enumerated constants rock, paper, scissors and gives them the values 0, 1, and 2.

```
enum {rock, paper, scissors};
```

Optionally, these can be given a type name, which creates a weakly typed enumeration. (For information on how to create strongly typed enumerations with C++14, see Chapter 17.)

```
enum Choice {rock, paper, scissors};
```

Now the word Choice can be used to declare variables just like any other type name. The underlying type of an enumeration is actually integer, and you can assign enumerated constants to integer variables. However, you can't go in the opposite direction without a cast.

```
Choice my_play = rock;
int n = paper;              // Ok without cast

// But this requires a cast...

Choice your_play = static_cast<Choice>(1);
```

D Preprocessor Directives

The C++ preprocessor can perform a number of useful actions before the regular compilation phase. For example, the **#define** directive can be used to replace all occurrences of a certain word with another—creating easy-to-interpret symbolic constants rather than arbitrary-looking numbers. The directives have other uses, the most important of which are probably including header files and making sure that such header files are compiled only once.

In addition to the directives listed here, C++ also supports a **#pragma** directive, but its use is entirely implementation defined. See your compiler documentation for more information.

This appendix covers the directives in alphabetical order, followed by a list of predefined compiler constants.

The #define Directive

The **#define** directive has three forms, each of which has a different use.

```
#define symbol_name

#define symbol_name  replacement_text

#define symbol_name(args)  replacement_text
```

The first version of **#define** is used to control compilation, by affecting the behavior of the **#ifdef** and related directives later. For example, you might use the following directive to indicate that the C++0x specification is supported. See **#if**, **#ifdef**, and **#ifndef** for more information.

```
#define CPLUSPLUS_0X
```

The second version is useful in creating predefined constants to help remove “magic numbers” (that is, arbitrary numbers) from your program. For example, you might define column width just once in your program:

```
#define COL_WIDTH  80
...
char input_string[COL_WIDTH+1];
```

One of the advantages of using this approach is that if you decide to change the column width, you only need to change it in one place (namely, the **#define** directive); the change is then automatically reflected throughout the source file wherever COL_WIDTH appears.

The third version is used to define macro functions, which take one or more arguments and expand them into larger expressions. The effect is something like inline functions, which are expanded into the body of the caller. Macro functions, however, have some limitations. They are usually limited to single expressions, and they have no type checking.

The following example macro function produces the maximum of two numbers. It takes advantage of the condition (?:) operator; see Appendix A for more information on this operator.

```
#define  MAX(A, B)  ((A)>(B) ? (A) : (B))
```

The extra parentheses, though not always necessary, help ensure that the expressions A and B are evaluated in their entirety before the other operators are applied (just in case A and B are complex expressions). Here is an example that uses this macro:

```
int x, y;
cout << "Enter a number: "
cin >> x;
cout << "Enter another: "
cin >> y;

cout << "The maximum is:" << MAX(x, y);
```

During preprocessing, the compiler expands the last line shown in the previous code into the following (in which I’ve removed the extra parentheses for clarity). The action of the conditional (?:) operator is to evaluate the first expression (in this case, x>y) and return x if true or y if false.

```
cout << "The maximum is:" << (x>y ? x : y);
```

The ## Operator (Concatenation)

The concatenation operator is used within macros to join text together, as in this macro, designed to generate file names:

```
#define  FILE(A, B)  myfile__##A.##B
```

The expression FILE(1, doc) should generate the following:

```
myfile__1.doc
```

The defined Function

This function is almost always used in conjunction with **#if** and **#elif**. It has this syntax:

```
defined(symbol_name)
```

If *symbol_name* is defined (it doesn't matter what value, if any, the symbol was given), the **defined** function returns true; otherwise, it returns false. For example, this function can be used to turn an **#if** directive into an **#ifdef** directive:

```
#if defined(CPLUSPLUS_0x)
```

For a more complete example, see the next section.

The #elif Directive

The **#elif** directive can be used as part of a conditional compilation block. **#elif** forms the beginning of an "else if" block. In the following example, different source code is compiled depending on whether the symbol CPLUSPLUS_0x has been defined, the symbol ANSI has been defined, or neither has been defined. It does not matter what value (if any) was assigned to these symbols.

```
#if defined(CPLUSPLUS_0x)

// Here you might place code supported by
// C++0x-compliant compilers only.

#elif defined(ANSI)
```

```
    // Here you might place code for compilers that are
    // ANSI compliant but not C++0x compliant.

    #else

    // Place code here for compilers not ANSI compliant.

    #endif
```

Given this conditional-compilation block, you can control what gets compiled by inserting or deleting one line that precedes this block, such as the following:

```
    #define ANSI   // Use ANSI features (but not C++0x)
```

The #endif Directive

The **#endif** directive forms the end of a conditional-compilation block. It is used in conjunction with **#if**, **#ifdef**, **#ifndef**, and **#elif**. Note that the syntax used here is not like C++ language syntax; the preprocessor has a language all its own that resembles Basic more than it does C++.

For examples of use, see any of the related sections: **#if**, **#ifdef**, **#ifndef**, and **#elif**.

The #error Directive

The **#error** directive generates an error message during compilation. For example:

```
    #ifndef __cplusplus < 199711
    #error C++ compiler out of date.
    #endif
```

The #if Directive

The **#if** directive is used to begin a conditional compilation block. It is mostly used in conjunction with the **defined** function. For example, although this is implementation specific, compilers may choose to refer to predefined constants such as `__win32__` or `__linix__` to indicate what operating system they are running on.

```
#if defined(__win32__)
char op_sys[] = "Microsoft Windows";
#endif
```

If the value following **#if** is true, the compiler reads and processes lines of code up until the nearest matching **#elif**, **#else**, or **#endif** directive; otherwise, it ignores those lines.

Another use for **#if** and **#endif** is to temporarily “comment out” large blocks of code. Notice that C-style comment symbols (/* and */) do not nest properly and—if you attempt to nest them—cause errors:

```
/*  (Begin a "commented-out" block...
/*
char op_sys[] = "Overco Operating System";

*/         // OOPS! This ends the first comment.
*/         // Syntax error!
```

However, **#if**/**#endif** pairs can be as deeply nested as you like. Each **#if**/**#endif** pair can be used to “comment out” lines of code that you do not want to compile at this particular time. One commented-out block can be placed inside another.

```
#if 1

// Do some stuff.
#if 1

// Do some more stuff.
#endif
...
#endif
```

The #ifdef Directive

The **#ifdef** directive begins a conditional compilation block. Although closely related to the **#if** directive, it is a more succinct way of expressing what the **#if** directive usually does. The following syntax

```
#ifdef symbol_name
```

is exactly equivalent to the following:

```
#if defined(symbol_name)
```

The first example in the previous section could be rewritten as follows:

```
#ifdef __win32__
char op_sys[] = "Microsoft Windows";
#endif
```

The #ifndef Directive

This directive is similar to the **#ifdef** directive but reverses its logical meaning; it enables conditional compilation only if the specified symbol has *not* been defined.

```
#ifndef symbol_name
```

This directive is widely used to avoid conflicts that can occur when multiple header files are in use.

For example, the following statements can be used to prevent compilation of a header file that has already been compiled. For example:

```
#ifndef FRACT_H
#define FRACT_H

// Here is the body of Fraction.h

#endif
```

The first time that the file Fraction.h is read in, it defines the symbol FRACT_H; that, in turn, prevents Fraction.h from being compiled a second time, no matter how many different files include it.

The #include Directive

The **#include** directive has two different versions. Both are almost always used to include *header files*, which contain function prototypes and symbolic constants needed to work with a project or a portion of the standard library.

```
#include <filename>

#include "filename"
```

In either case, the effect of **#include** is to suspend compilation of the current file and instead run the compiler on the named file until the end of that file is reached. Then the compiler returns to compiling the current file.

The first version (using angle brackets) searches for the named file in the directory or folder set aside for header files. In the case of MS-DOS and Windows, for example, this directory is indicated by the setting of an INC environment variable.

The second version (using quotation marks) searches for the named file in the same way but also searches the current directory or folder.

By convention, the first version is almost always used to include library header files or header files supplied by vendors (even though both versions would work), while the second version is used for header files for the project. For example:

```
#include <iostream>
#include <cmath>
#include "Fraction.h"
```

Note Each standard library file beginning with *c*, as in cmath, corresponds to a traditional .h file inherited from the C language. Another example is cctype, which corresponds to the ctype.h file used in C. With C++, the newer form is preferred, even if the older (C-style) headers still work for now.

The #line Directive

The #line directive has two forms.

```
#line  filename  number
#line  number
```

The effect is to reset the values of the __FILE__ and __LINE__ constants, as well as affecting how error messages are printed by the computer. For example:

```
#line myfile.cpp 100
```

The #undef Directive

The **#undef** directive undoes the definition of the named symbol, so it is no longer considered defined.

```
#undef  symbol_name
```

Predefined Constants

Table D.1 lists constants that all C++ preprocessors are required to support. Specific implementations of C++ may define others as well.

Table D.1: C++ Predefined Constants

CONSTANT	MEANING IF DEFINED
__cplusplus	Indicates support of a particular version of C++. Current, up-to-date compilers should define it as 199711 or greater.
__DATE__	Expands to a date string in the format mm dd yyyy.
__FILE__	Name of file being compiled.
__LINE__	Current line being compiled.
__STDC__	Defined by C compilers as 1. Usually defined by C++ compilers to indicate support for C, but this is implementation-defined.
__TIME__	Expands to a time string in the format hh:mm:ss.

E

ASCII Codes

This appendix presents ASCII codes according to decimal value, hexadecimal value, and corresponding character. Hexadecimal codes can be used to embed values into a string. You can also print characters by using a (char) cast. For example:

```
cout << "Hex code 7e is " << "\x7e" << endl;

// Print ASCII codes from 32 to 42.
for (int i = 32; i <= 42; i++)
    cout << i << ": " << (char) i << endl;
```

Some nonprintable characters have special meanings:

- **NUL**: null value
- **ACK**: acknowledgment signal (used in network communications)
- **BEL**: bell
- **BS**: backspace
- **LF**: linefeed
- **FF**: form feed (new page)
- **CR**: carriage return
- **NAK**: no acknowledgment
- **DEL**: delete

Table E.1 gives the standard ASCII codes.

Table E.1: Standard ASCII Codes

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
00	00	NUL	26	1a		52	34	4	78	4e	N	104	68	h
01	01		27	1b		53	35	5	79	4f	O	105	69	i
02	02		28	1c	FS	54	36	6	80	50	P	106	6a	j
03	03		29	1d	GS	55	37	7	81	51	Q	107	6b	k
04	04		30	1e	RS	56	38	8	82	52	R	108	6c	l
05	05		31	1f	US	57	39	9	83	53	S	109	6d	m
06	06	ACK	32	20	space	58	3a	:	84	54	T	110	6e	n
07	07	BEL	33	21	!	59	3b	;	85	55	U	111	6f	o
08	08	BS	34	22	"	60	3c	<	86	56	V	112	70	p
09	09		35	23	#	61	3d	=	87	57	W	113	71	q
10	0a	LF	36	24	$	62	3e	>	88	58	X	114	72	r
11	0b	CR	37	25	%	63	3f	?	89	59	Y	115	73	s
12	0c		38	26	&	64	40	@	90	5a	Z	116	74	t
13	0d		39	27	'	65	41	A	91	5b	[	117	75	u
14	0e		40	28	(	66	42	B	92	5c	\	118	76	v
15	0f		41	29	)	67	43	C	93	5d	]	119	77	w
16	10		42	2a	*	68	44	D	94	5e	^	120	78	x
17	11		43	2b	+	69	45	E	95	5f	-	121	79	y
18	12		44	2c	,	70	46	F	96	60	`	122	7a	z
19	13		45	2d	-	71	47	G	97	61	a	123	7b	{
20	14		46	2e	.	72	48	H	98	62	b	124	7c	|
21	15	NAK	47	2f	/	73	49	I	99	63	c	125	7d	}
22	16	SYN	48	30	0	74	4a	J	100	64	d	126	7e	~
23	17		49	31	1	75	4b	K	101	65	e	127	7f	DEL
24	18		50	32	2	76	4c	L	102	66	f			
25	19		51	33	3	77	4d	M	103	67	g			

Table E.2 lists extended ASCII codes; note that these are somewhat implementation dependent. They are most likely to be correct for computers of American manufacturers running Windows.

Table E.2: Extended ASCII Codes for American Equipment Manufacturers

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
128	80	Ç	154	9a	Ü	180	b4	┤	206	ce	╬	232	e8	Φ
129	81	ü	155	9b	¢	181	b5	╡	207	cf	╧	233	e9	Θ
130	82	é	156	9c	£	182	b6	╢	208	d0	╨	234	ea	Ω
131	83	â	157	9d	¥	183	b7	╖	209	d1	╤	235	eb	δ
132	84	ä	158	9e	₧	184	b8	╕	210	d2	╥	236	ec	∞
133	85	à	159	9f	ƒ	185	b9	╣	211	d3	╙	237	ed	φ
134	86	å	160	a0	á	186	ba	║	212	d4	╘	238	ee	ε
135	87	ç	161	a1	í	187	bb	╗	213	d5	╒	239	ef	∩
136	88	ê	162	a2	ó	188	bc	╝	214	d6	╓	240	f0	≡
137	89	ë	163	a3	ú	189	bd	╜	215	d7	╫	241	f1	±
138	8a	è	164	a4	ñ	190	be	╛	216	d8	╪	242	f2	≥
139	8b	ï	165	a5	Ñ	191	bf	┐	217	d9	┘	243	f3	≤
140	8c	î	166	a6	ª	192	c0	└	218	da	┌	244	f4	⌠
141	8d	ì	167	a7	º	193	c1	┴	219	db	█	245	f5	⌡
142	8e	Ä	168	a8	¿	194	c2	┬	220	dc	▄	246	f6	÷
143	8f	Å	169	a9	⌐	195	c3	├	221	dd	▌	247	f7	≈
144	90	É	170	aa	¬	196	c4	─	222	de	▐	248	f8	°
145	91	æ	171	ab	½	197	c5	┼	223	df	▀	249	f9	∙
146	92	Æ	172	ac	¼	198	c6	╞	224	e0	α	250	fa	·
147	93	ô	173	ad	¡	199	c7	╟	225	e1	ß	251	fb	√
148	94	ö	174	ae	«	200	c8	╚	226	e2	Γ	252	fc	ⁿ
149	95	ò	175	af	»	201	c9	╔	227	e3	ϖ	253	fd	²
150	96	û	176	b0	░	202	ca	╩	228	e4	Σ	254	fe	■
151	97	ù	177	b1	▒	203	cb	╦	229	e5	σ	255	ff	
152	98	ÿ	178	b2	▓	204	cc	╠	230	e6	µ			
153	99	Ö	179	b3	│	205	cd	═	231	e7	τ			

This page intentionally left blank

F

Standard Library Functions

The most commonly used library functions fall into a few categories: string functions, data-conversion functions, single character functions, math functions, time functions, and randomization functions. This appendix provides an overview. Note that I do not cover I/O functions such as printf or fprintf, because I assume you are using C++ stream classes.

For more information on the stream objects **`cin`**, **`cout`**, and the stream classes, see Appendix G.

String (C-String) Functions

To use these functions, include the file <cstring>. The functions apply to traditional C **char*** strings, not to the STL **`string`** class.

In Table F.1, s, s1, and s2 are null-terminated **char*** strings; or rather, each of these equates to the address of one of these strings. Also, n is an integer, and ch is a single character. Except where otherwise noted, each of these functions returns the address of its first argument.

Table F.1: Common String Functions

FUNCTION	ACTION
`strcat(``s1, s2`**`)`**	Concatenates the contents of `s2` onto the end of `s1`.
`strchr(``s, ch`**`)`**	Returns a pointer to the first occurrence of ch in string `s`; returns NULL if `ch` cannot be found.
`strcmp(``s1, s2`**`)`**	Performs a comparison between contents of `s1` and `s2`, returning a negative integer, 0, or a positive integer, depending on whether s1 appears before s2 in alpha order, has same contents as s2, or appears later than s2.
`strcpy(``s1, s2`**`)`**	Copies contents of `s2` into `s1`, replacing existing contents.

continue

Table F.1: Common String Functions (*continued*)

FUNCTION	ACTION
`strcspn(s1, s2)`	Searches *s1* for occurrence of any character in *s2*; returns index of first matching s1 character; returns the length of s1 if none found.
`strlen(s)`	Returns current length of s (not including null byte).
`strncat(s1, s2, n)`	Same action as **strcat** but copies at most *n* characters.
`strncmp(s1, s2, n)`	Same action as **strcmp** but compares at most *n* characters.
`strncpy(s1, s2, n)`	Same action as **strcpy** but copies at most *n* characters.
`strpbrk(s1, s2)`	Searches *s1* for occurrence of any character in *s2*; returns a pointer to first matching *s1* character; returns NULL if none found.
`strrchr(s, ch)`	Same action as **strpbrk** but searches *s1* in reverse order.
`strspn(s1, s2)`	Searches *s1* for first character that does not match any character in *s2*; returns index of this character; returns length of *s1* if none found.
`strstr(s1, s2)`	Searches *s1* for the first occurrence of substring *s2*; returns a pointer to substring found within *s1*; returns NULL if not found.
`strtok(s1, s2)`	Returns a pointer to the first token (substring) in *s1*, using delimiters specified in *s2*. Subsequent calls to this function with NULL for the first argument find the next token within the current string—the previously set value of *s1*. Specifying a non-null value for *s1* resets the tokenization process with a new string.

Data-Conversion Functions

To use the functions in Table F.2, include <cstdlib>.

Table F.2: Data-Conversion Functions

FUNCTION	ACTION
`atof(s)`	Reads a **char*** text string as a floating-point digit string and returns the equivalent **double**. The function skips past leading spaces and stops reading after the first character that doesn't form part of a valid floating-point representation (such as "1.5" or "2e1.2").
`atoi(s)`	Reads a **char*** text string as a digit string and returns the equivalent **int**. The function skips past leading spaces and stops reading after the first character that doesn't form part of a valid integer representation (such as "–27").
`atol(s)`	Reads a **char*** string and produces a **long** value; on 32-bit systems, this is equivalent to **atoi**.
`atoll(s)`	Similar to **atoi**, but it produces a **long long** integer value. Required in the C++0x specification

Single-Character Functions

To use any of the functions in Table F.3 or Table F.4, include <cctype>. Each of the functions in this first subgroup tests a single character and returns true or false.

Table F.3: Character-Testing Functions

FUNCTION	ACTION
`isalnum(ch)`	Is the character alphanumeric (a letter or digit)?
`isalpha(ch)`	Is the character a letter?
`iscntrl(ch)`	Is the character a control character? (These include backspace, linefeed, form feed, and tab, among others; these are nonprintable characters that perform actions.)
`isdigit(ch)`	Is the character a digit in the range 0 to 9?
`isgraph(ch)`	Is the character visible? (This includes printable characters other than a space.)
`islower(ch)`	Is the character a lowercase letter?
`isprint(ch)`	Is the character printable? (This includes space characters.)
`ispunct(ch)`	Is the character a punctuation character?
`isspace(ch)`	Is the character a white space? (This includes tab, newline, and form feed, in addition to the simple space character.)
`isupper(ch)`	Is the character an uppercase letter?
`isxdigit(ch)`	Is the character a hexadecimal digit? This includes digits in the range 0 through 9, as well as A through E and a through e.

Including <cctype> also adds declarations for the following two conversion functions.

Table F.4: Character Conversion Functions

FUNCTION	ACTION
`tolower(ch)`	Returns a lowercase letter if ch is an uppercase letter; otherwise, returns ch as is
`toupper(ch)`	Returns an uppercase letter if ch is a lowercase letter; otherwise, returns ch as is

Math Functions

To use any of the functions in Table F.5, include <cmath>. Each of the functions in this first takes an argument of type **double** and returns a **double** result, except where noted. Each of these functions returns the result of the operation, and none of them alter their argument or arguments.

In using these functions, keep in mind that an integer can be given where a **double** argument is required; the integer is promoted without C++ printing a warning message. However, to assign a double result to an integer variable causes a warning message to be issued if a cast is not used. (The new style cast useful in this case would be **`static_cast`**.)

Table F.5: Math Functions

FUNCTION	ACTION
`abs(`*`n`*`)`	Returns the absolute value of **`int`** argument *n*. Result has type **`int`**. For floating-point version, see **`fabs`**.
`acos(`*`x`*`)`	Arc cosine of *x*.
`asin(`*`x`*`)`	Arc sine of *x*.
`atan(`*`x`*`)`	Arc tangent of *x*.
`ceil(`*`x`*`)`	Rounds upward to nearest integer (but still returns result as a **`double`**).
`cos(`*`x`*`)`	Cosine of *x*.
`cosh(`*`x`*`)`	Hyperbolic cosine of *x*.
`exp(`*`x`*`)`	Raises the mathematical constant e to the power of *x*.
`fabs(`*`x`*`)`	Returns the absolute value of *x*.
`floor(`*`x`*`)`	Rounds *x* downward to the nearest integer (but still returns result as a **`double`**).
`log(`*`x`*`)`	Natural logarithm (base e) of *x*.
`log10(`*`x`*`)`	Base 10 logarithm of *x*.
`pow(`*`x`*`, `*`y`*`)`	Raises x to the power of y. (For example, pow(2,5) returns 32.)
`sin(`*`x`*`)`	Sine of *x*.
`sinh(`*`x`*`)`	Hyperbolic sine of *x*.
`sqrt(`*`x`*`)`	Square root of *x*.
`tan(`*`x`*`)`	Tangent of *x*.
`tanh(`*`x`*`)`	Hyperbolic tangent of *x*.

Randomization Functions

To use the functions in Table F.6, include both <cstdlib> and <ctime>. See Chapter 4 for more information on using these three functions.

Table F.6: Randomization Functions

FUNCTION	ACTION
rand()	Returns the next number (an integer) in the current random-number sequence. This sequence should first be set by calling **srand**. The number returned ranges in value between 0 and RAND_MAX (defined in <cstdlib>).
srand(*seed*)	Takes the seed number—an **unsigned int**—to start the random-number sequence used for calls to **rand**.
time(NULL)	Returns the system time. Calling this function is a good choice for getting the seed number to pass to **srand**.

Time Functions

To support the library functions in Table F.7, include <ctime>. The general procedure with these functions is to call the **time** function to get a **time_t** value representing the current time as a number. Then use that value as input to **gmtime** or **localtime**, which fills a **tm** structure listing specific information including month, day of the week, and so on. The declaration of the **tm** structure is shown at the end of this section.

Alternatively, you can call the **asctime** function to get a **char*** string describing the time in human-readable form—or **ctime**, which produces the same result more directly:

```
#include <ctime>
#include <iostream>

time_t t = time(NULL);   // Put time into t.
cout << ctime(&t);       // Display current time.
```

You can also use **strftime**, which returns a formatted time string.

Table F.7: Time Functions

FUNCTION	DESCRIPTION
asctime(*tm_ptr*)	Takes a pointer to a tm structure and returns a **char*** string in the format "Ddd Mmm DD HH:MM:SS YYYY\n" in which Ddd is a three-letter abbreviation for day of the week and Mmm is a three-letter abbreviation for the month. See also **ctime**.
clock()	Returns number of clock ticks on internal clock. To convert to seconds, divide by CLOCKS_PER_TICK predefined symbol.
ctime(*time_ptr*)	Takes a pointer to a time_t value (returned by the **time** function) and returns a **char*** string in the same format that **asctime** returns. This function is equivalent to calling **localtime** and then passing the resulting structure to **asctime**.
difftime(*t1*, *t2*)	Returns t1–t2 in seconds, in which *t1* and *t2* are time_t values.
gmtime(*time_ptr*)	Takes a pointer to a time_t value (returned by the **time** function) and returns a pointer to a tm structure using Greenwich Mean Time.
localtime(*time_ptr*)	Takes a pointer to a time_t value (returned by the **time** function) and returns a pointer to a tm structure using local time.
mktime(*tm_ptr*)	Converts tm structure pointed to by *tm_ptr* into a time_t value and returns that value. The tm_wday and tm_yday members of the tm structure are ignored.
strftime(*s*, *n*, *fmt*, *tm_ptr*)	Takes a pointer to a tm structure (*tm_ptr*), formats time data from that structure according to a format string *fmt*, and places the result in string *s*. See the next section for details. *n* is the maximum number of characters to write, including null.
time(*time_ptr*)	Returns the current time as a time_t value (usually **unsigned long**, although this is implementation dependent). If the argument is null, it is ignored. If it is not null, the return value is copied to the address specified.

Here is an example that uses some of these functions to print the day of the week as a number from 0 to 6:

```
#include <ctime>
...
time_t t = time(NULL);
tm *tm_pointer = localtime(&t);
cout << "The day is " << tm_pointer->tm_mday;
cout << endl;
```

The declaration of the tm structure is as follows:

```
tm struct tm {
    int tm_sec;     // Seconds, 0-59
```

```
        int tm_min;     // Minutes, 0-59
        int tm_hour;    // Hours, 0-23
        int tm_mday;    // Day of month, 1-31
        int tm_mon;     // Month,0-11
        int tm_year;    // Years since 1900
        int tm_wday;    // Days since Sunday, 0-6
        int tm_yday;    // Days since Jan. 1, 0-365
        int tm_isdst;   // Daylight Savings Time...
    }                     //   positive: DST in effect,
                          //   zero: not in effect,
                          //   negative: unknown
```

Formats for the strftime Function

The **strftime** function has the following declaration:

```
size_t strftime(
  char    *str,     // String to write to.
  size_t   n,       // Max. chars to write,
                    //   including null terminator.
  char    *fmt,     // Format string
  tm      *tm_ptr   // Ptr to tm structure.
);
```

The **strftime** function uses time data from the structure pointed to by *tm_ptr* and writes that data to string *str*. The *fmt* argument contains formatting characters determining what data to write and how to write it, using the format specifiers listed in Table F.8. For example, the following code fragment displays what day of the week it is:

```
#include <iostream>
#include <ctime>

char s[100];
time_t t = time(NULL);
tm *tm_ptr = localtime(&t);
strftime(s, 100, "Today is %A.", tm_ptr);
cout << s << endl;
```

Table F.8: Format Characters for strftime

FORMAT SPECIFIER	DESCRIPTION
%a	Day of the week, abbreviated.
%A	Day of the week, spelled out.
%b	Month name, abbreviated.
%B	Month name, spelled out.
%c	Complete month and day.
%D	Day of the month, 01 to 31.
%H	Hour, 00 to 23 (based on 24-hour clock).
%I	Hour, 00 to 11 (based on 12-hour clock).
%J	Day of the year, 000 to 366.
%m	Month, 01 to 12.
%M	Minute, 00 to 59
%p	a.m./p.m. designation.
%S	Seconds, 00 to 61 (up to two leap seconds)
%U	Week number, 01 to 53. Week 1 starts with first Sunday.
%w	Weekday, 0 to 6, in which Sunday is 0.
%W	Week number, 01 to 53. Week 1 starts with first Monday.
%x	Date.
%X	Time.
%y	Year, 00 to 09, within the century.
%Y	Year.
%Z	Time zone; blank if time zone is not known.
%%	Literal %.

G I/O Stream Objects and Classes

The objects and classes in this appendix support reading and writing to the console, as well as to files and strings. To read and write to the console, include <iostream>:

```
#include <iostream>
cout << "Hello, world." << endl;
```

To write data to a string, include <sstream>. String streams support a member function (in addition to the ones listed here), **str**, which returns the data in string format.

```
#include <sstream>

stringstream s_out;
s_out << "The value of i answer is " << i << endl;
string s = s_out.str()
```

Console Stream Objects

The objects listed in Table G.1 provide predeclared streams to which to read or write text to the console. Each of them supports the appropriate stream operator (<< or >>). For example:

```
cout << "n is equal to " << n << endl;
```

Table G.1: Stream Objects

FUNCTION	DESCRIPTION
cerr	Console error-message stream. By default it writes characters to the console just as **cout** does; however, this stream may be redirected without affecting **cout**. Use of this object is fairly rare.
cin	Console input stream. Reads input from the console as a stream of ASCII (8-bit) characters.
clog	Console log stream. Similar to **cerr** and **cout**, but intended to display runtime messages that are not necessarily errors. Use of this object is rare; many programmers never use it.
cout	Console output stream. Displays output to the console as a stream of ASCII (8-bit) characters.
wcerr	Wide-character error-message stream. Similar to **cerr** but writes text as a series of wide characters.
wcin	Wide Console Output. Similar to **cin** but reads input from the console as a stream of wide characters.
wclog	Wide-character log stream. Similar to **clog** but writes text as a series of wide characters.
wcout	Wide-character Console Output. Similar to **cout** but writes text as a series of wide characters.

I/O Stream Manipulators

The I/O manipulators in Table G.2 can be used with stream objects to modify how they read or write text. For example, the following statement writes "0x1f" to the console.

```
cout << hex << showbase << 31;  // Output 0x1f.
```

Some I/O manipulators affect both input and output streams, although many affect output only. For example, if **hex** is used, input is interpreted in hex format. In this code fragment, input of 10 would actually put the value 16 into n.

```
cin >> hex >> n;
```

Table G.2: Stream Manipulators

MANIPULATOR	DESCRIPTION
boolalpha	Writes out Boolean values **true** and **false** as "true" and "false" rather than 1 and 0 (the default).
dec	Switches to decimal format (the default) for integers.
fixed	Displays floating-point numbers in fixed-point format.
hex	Switches to hexadecimal format for integers.
left	Left-justifies output. (Matters only when a minimum print-field width has been specified: see the **width** function in Table G.4.)
noboolalpha	Turns off **boolalpha**, so that **true** and **false** are not printed.
noshowbase	Turns off **showbase**.
noshowpoint	Turns off **showpoint**. Floating-point 5 is written as "5."
nouppercase	Displays numeric data in lowercase letters, so that hexadecimal FF (for example) with **showbase** on is displayed as 0xff (the default).
nounitbuf	Turns off **unitbuf**.
oct	Switches to octal format for integers.
right	Right-justifies output. (Matters only when a minimum print-field width has been specified: see the **width** function in Table G.4.)
scientific	Displays all floating-point numbers in scientific notation.
showbase	When writing octal or hex format, displays the "0x" or "0" prefix.
showpoint	Always displays decimal point when writing out a floating-point number. For example, floating-point 5 is written as "5.0."
showpos	Shows the positive sign (+) for positive numbers.
uppercase	Displays numeric data in uppercase letters, so that hexadecimal FF (for example) with **showbase** on is displayed as 0XFF.
unitbuf	For output streams, causes output buffer to be flushed after each output operation.
endl	When sent to an output stream, this prints a character.
ends	When sent to an output stream, this outputs a null terminator. This manipulator is generally used only with strstream objects.
flush	Flushes the buffer, so that any text in the buffer is immediately written out to destination.

Input Stream Functions

The functions in Table G.3 can be called by input streams such as **cin** as well as input-file streams. For example:

```
    char input_str[COL_WIDTH];
    cin.getline(str, COL_WIDTH);
```

Table G.3: Input Stream Functions

FUNCTION	DESCRIPTION
get()	Gets the next character from the input stream
getline(*s*, *n*)	Copies line of input to string address *s*, getting no more than *n*-1 characters
peek()	Returns the next character without removing it from the stream
putback(*c*)	Puts character *c* back onto the input stream
read(*s*, *n*)	Binary read operation: reads *n* bytes and places data at address *s*, which needs to be cast to char* type if the data is not in char format.

Output Stream Functions

The functions in Table G.4 can be called by output streams such as **cout** as well as output-file streams and string streams. For example, the following code fragment prints a number n in a print field 20 characters wide and right-justifies it:

```
    cout.width(20);
    cout << right << n << endl;
```

Table G.4: Output Stream Functions

FUNCTION	DESCRIPTION
base(*n*)	Set the radix for output operations to n, which must be 8, 10, or 16.
fill(**c**)	Set the fill character used to fill print fields when the output is smaller that the width. (By default, this is a space.)
flush()	Flush the output buffer, causing output to be immediately printed.
precision(*n*)	Set the number of digits of precision for writing floating-point numbers.
put(*c*)	Output character c.
width(*n*)	Set minimum print-field width for the next output operation.
write(*s*, *n*)	Binary write operation: writes *n* bytes directly from data at address *s*, which needs to be cast to **char*** type if the data is not in **char** format.

File I/O Functions

To enable the member functions in this section, include <fstream>.

```
#include <fstream>
```

You can create a file of type **fstream**, **ifstream**, or **ofstream** and optionally attempt opening a file when declaring the file object—or you can declare the object first and then attempt opening with the open function. The default mode is text.

```
ofstream fout(filename);
```

Note that a file object contains NULL after a file-open attempt fails. See Chapter 8 for more information on the use of file-stream objects. File-stream objects support the member functions in Table G.5, as well as the ones shown in the earlier tables.

Table G.5: File I/O Functions

FUNCTION	DESCRIPTION
open(*file, mode*)	Instead of opening a file when a file-object is declared, you can call this function separately to open the indicated *file*, a **char*** string containing a file specification. The *mode* argument takes one or more of the flags listed in Table G.6.
close()	Closes the file.
eof()	Returns true if end-of-file marker has been reached.
is_open()	Returns true if file has been successfully opened.
seekg(*pos*)	Moves input-file pointer to indicated position (an offset from beginning of the file in bytes).
seekg(*off, dir*)	Moves input-file pointer by the indicated offset *off* (which may be positive or negative) in the direction indicated by *dir*. See Table G.7 for values.
seekp(*pos*)	Same as **seekg** but intended for use with output files.
seekp(*off,dir*)	Same as **seekg** but intended for use with output files.
tellg()	Returns file position—an offset from beginning of file in bytes.
tellp()	Same as **tellg** but intended for use with output files.

The flag values in Table G.6 are used with the mode argument of the open function. They can be combined through the bitwise OR operator (|). For example:

```
// Open named file (filename) for binary input.

fstream fout;
fout.open(filename, ios::binary | ios::in);
```

Table G.6: File Mode Flags

FUNCTION	DESCRIPTION
ios::binary	Open file in binary mode.
ios::in	Open file for input operations.
ios::out	Open file for output operations.
ios::ate	Position file pointer at end of file.
ios::app	(Append.) Position file pointer at end of file after each i/o operation.
ios::trunc	(Truncate.) Remove all existing contents before doing any other operations.

The constants in Table G.7 are used in conjunction with the **seekg** and **seekp** functions.

Table G.7: Seek Direction Flags

DIRECTION VALUE	DESCRIPTION
ios::beg	Seek operation is relative to beginning of the file.
ios::cur	Seek operation is relative to the current position; the offset moves the file pointer forward if positive or backward if negative.
ios::end	Seek operation moves file pointer forward beyond current end of file if offset positive; if negative, file pointer is moved backward from end of file.

H STL Classes and Objects

Although the Standard Template Library (STL) supports many useful templates, this appendix summarizes the use of five (the ones used in this book).

- The string class
- The bitset template
- The list template
- The vector template
- The stack template

The STL String Class

The features in this section require including <string>. STL string objects are declared simply as **string—**or **std::string** if the **std** namespace is not being used. The simple **string** class actually instantiates the template class **basic_string** for type **char**, so the functions listed here are also supported by **basic_string** classes:

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

basic_string<char>     s1;    // Equivalent to string
basic_string<wchar_t>  s2 = L"Hello";  // Wide string
wcout << s2;
```

Once declared, **string** objects support copying of contents (=), concatenation (+), and comparison of contents (<, >, and =). Unlike standard C-strings (**char*** strings), STL strings can be assigned data without worrying about size considerations.

```
#include <string>
using namespace std;
...
string your_dog = "Fido";
your_dog = "Montgomery";   // String automatically
                           // grows.

string my_dog = "Mr. " + your_dog;
```

The **string** objects can be indexed as if they were **char*** strings.

```
cout << "The third character is" << my_dog[2];
```

With most of the functions listed in Table H.1, *str* may be either a STL string or a **char*** string. The return value, except where otherwise noted, is a reference to the current string object (the object through which the function is called).

In all these functions, position numbers use zero-based indexing.

Table H.1: String Member Functions

FUNCTION	DESCRIPTION
append(*str*)	Appends string *str* onto current string
append(*str*, *n*)	Appends first *n* characters of *str* onto current string
append(*n*, *c*)	Appends *n* copies of character *c* onto current string
begin()	Returns an iterator pointing to the beginning of the string; this corresponds to a single character
clear()	Clears the contents of the string; has no return value
c_str()	Returns the C-string (**char***) equivalent of current string
empty()	Returns true if the string is currently empty; false otherwise
erase(*pos*, *n*)	Erases *n* characters starting at position *pos*
end()	Returns an iterator pointing to the end of the string (the position one past the last character)
insert(*pos*, *str*)	Inserts string *str* at position *pos*
insert(*iter*, *c*)	Inserts character *c* at position pointed to by iterator *iter*
find(*str*, *pos*)	Finds first occurrence of substring *str*, starting search at position *pos*; returns position of string found; returns **string::npos** if not found
find(*str*)	Finds first occurrence of substring *str*; returns position of string found; returns **string::npos** if not found
find(*c*, *pos*)	Finds first occurrence of character *c*, starting search at position *pos*; returns position of character found; returns **string::npos** if not found

Table H.1: String Member Functions (*continued*)

FUNCTION	DESCRIPTION
find(*c*)	Finds first occurrence of character *c*; returns position of character found; returns **string::npos** if not found
find_first_of(*s*, *pos*)	Finds first occurrence of a character that is also in string *s*, starting search at *pos*; if *pos* is omitted, searches from beginning of string; returns position of character found; returns **string::npos** if no such character found
find_first_not_of(*s*, *pos*)	Finds first occurrence of a character that is *not* in string *s*, beginning at *pos*; if *pos* is omitted, searches from beginning of string; returns position of character found; returns **string::npos** if every character in string is also in *s*
replace(*pos*, *n*, *str*)	Replaces *n* characters, starting at position, *pos* with *str*
replace(*iter1*, *iter2*, *str*)	Replaces characters in range from iterators *iter1* to *iter2*, with *str*
size()	Returns the current length of the string
substr(*pos*, *n*)	Returns substring at position *pos* of length *n*
swap(*str*)	Swaps existing contents with those of specified string, an STL string; has no return value

The <bitset> Template

To enable use of the **bitset** template, include <bitset>.

```
#include <bitset>
using namespace std;
```

The **bitset** template is unlike some other templates in that it is not built around a base type, but rather on a constant integer of fixed size. This size indicates the number of bits that the data type will hold (notably, in a compact format). For example:

```
bitset<8>   his_bits(255);  // Store 8 bits exactly.
bitset<16>  her_bits(108);  // Store 16 bits exactly.
```

The value used to initialize a bitset need not be a constant, but it should be an integer value, in this case, 255 or 108. The bitset enables access to individual bits of this value through indexing and other operations. Bit 0 is the least significant bit. Bit 1 is one position to its left, and so on. (Bitset indexing, unlike other indexing, appears to be in right-to-left ordering, because of the way we write numbers.)

```
if (his_bits[0]) {
     cout << "Lowest bit is 1.";
}
```

Printing a bitset results in a display of a string of "1" and "0" digit characters.

```
cout << his_bits << endl;
```

Table H.2 describes the principal **bitset** member functions.

Table H.2: Bitset Member Functions

MEMBER FUNCTION	DESCRIPTION
flip()	Performs bitwise negation, changing each 1 to a 0 and vice-versa
set(*n*)	Sets individual bit at position *n* to 1; if *n* is omitted, sets all bits to 1
test(***n***)	Returns value of the bit indexed by *n* (this function, unlike indexing, performs a range check on n)
reset(***n***)	Sets individual bit at position *n* to 0; if *n* is omitted, sets all bits to 0
to_string()	Returns value of the bitset, as a string object containing "1" and "0" digit characters
to_ulong()	Returns value of the bitset as an **unsigned long** integer
to_ullong()	Returns value of the bitset as an **unsigned long long** integer

The <list> Template

To enable use of the list template, include <list>.

```
#include <list>
using namespace std;
```

List classes can then be instantiated, creating a list container for the indicated base type. You can then add elements by calling any member functions such as **push_back**.

```
// Create several kinds of lists.

list<int>       list_of_int;
list<double>    list_of_double;
list<string>    list_of_string;
list<Point>     my_list;

list<int>       IList;  // Create list, add elements.
IList.push_back(5);
IList.push_back(225);
IList.push_back(100);
IList.sort();         // Sort list by value.
```

In C++11 and later (this includes the C++14 specification, of course), you can initialize a list more directly.

```
list<int>  IList = {2, 225, 100};
```

After declaring a list, you can declare an iterator and use it to step through the list.

```
list<int>::iterator  ii;   // Declare ii as iterator.

for(ii = IList.begin(); ii != IList.end(); ++ii){
     cout << *ii << endl;
}
```

In C++11 and later, you can use range-based **for** with any list container, even if created in another function, because C++ can always determine the beginning, end, and size of any container generated with **list**.

```
for (int i : list_of_int) {
     cout << i << endl;
}
```

See Chapter 17 for more information on range-based **for**.

Table H.3 lists the good majority of member functions supported for list templates. A few functions—merge, splice, and predicates—are complex and difficult to describe in a short summary. See your compiler documentation for more information on those.

Note that, unlike some containers, the list template has a built-in **sort** function, which can be very useful in many situations.

Table H.3: List Template Member Functions

MEMBER FUNCTION	DESCRIPTION
assign(*n*, *val*)	Replaces entire list contents with *n* copies of *val*, a data object of the base type
begin()	Returns an iterator that points to the beginning of the list
clear()	Erases contents of the list
empty()	Returns true if the list is empty; false otherwise
end()	Returns an iterator to the end of the list (one position past the last element)

continues

Table H.3: List Template Member Functions (*continued*)

MEMBER FUNCTION	DESCRIPTION
erase(*iter*)	Erases the element pointed to by iterator *iter*
erase(*iter1*, *iter2*)	Erases the elements in the range from *iter1* to *iter2*
front()	Returns the first element
insert(*iter*, *val*)	Inserts indicated value just before the position pointed to by *iter*; if *iter* is equal to **end**(), then element is inserted at end of list
insert(*iter*, *n*, *val*)	Inserts *n* elements just before the position pointed to by *iter*; each has the indicated value, *val*
pop_back()	Erases the last element in the list; behavior is "undefined" if list is empty (but don't count on your program still running)
pop_font()	Erases the first element in the list; behavior is "undefined" if list is empty
push_back(*val*)	Adds specified value, *val*, to back of the list
push_front(*val*)	Adds specified value, *val*, to the front of the list
rbegin()	Returns a (reverse) iterator that points to last element
rend()	Returns a (reverse) iterator that points to one position before the beginning of the list
reverse()	Reverses the current order of all elements in the list
sort()	Sorts elements in the list, using comparison operator (<) defined for the base type
unique()	Erases duplicate adjacent elements from the list

The <vector> Template

One of the most useful parts of the STL is the **vector** template, which allows you to build vector containers for any base type. A vector is similar to an array, but it can grow without limit. You can, for example, declare a vector this way:

```
vector<int>  my_vec(100, 0); // my_vec contains 100 0's.
```

You can then index this container just as you would an array, using zero-based indexes (and if it needs to grow, use **push_back** or **insert**).

```
my_vec[0] = 5;  // Assign 5 to first element.
```

To enable use of the list template, include <vector>.

```
#include <vector>
using namespace std;
```

Vector classes can then be instantiated, creating a list container for the indicated base type. You can then add elements by calling member functions such as **push_back**.

```
// Create several kinds of vectors.

vector<int>      iVec;
vector<double>   fVec;
vector<string>   strVec;
vector<Point>    pt_vector;

vector<int>        my_vec;
my_vec.push_back(10);
my_vec.push_back(150);
my_vec.push_back(-250);
```

In C++11 and later (this includes the C++14 specification, of course), you can initialize a vector directly.

```
vector<int>  iVec = {10, 150, -250};
```

After creating a vector, you can declare an iterator and use it to step through. (You can also use indexing to step through the vector, just as you can do with an array.)

```
vector<int>::iterator  ii;  // Declare ii as iterator.

for (ii = iVec.begin(); ii != iVec.end(); ++ii){
     cout << *ii << endl;
}
```

In C++11 and later, you can use range-based **for** with any vector container, even if it was created in another function, because C++ can always determine the beginning, end, and size of any vector.

```
for (int i : iVec) {
     cout << i << endl;
}
```

See Chapter 17 for more information on range-based **for**.

Table H.4 lists the principal member functions supported for vector templates.

Table H.4: Vector Template Member Functions

MEMBER FUNCTION	DESCRIPTION
assign(*n*, *val*)	Replaces vector contents with *n* copies of *val*, a data object of the base type
begin()	Returns an iterator that points to the beginning of the vector's elements
clear()	Erases entire contents of the list
empty()	Returns true if contents are empty; false otherwise
end()	Returns an iterator to the end of the vector (one position past the last element)
erase(*iter*)	Erases the element pointed to by iterator *iter*
erase(*iter1*, *iter2*)	Erases the elements in the range from *iter1* up to but not including *iter2*
front()	Returns the first element
insert(*iter*, *val*)	Inserts indicated value just before the position pointed to by *iter*; if *iter* is equal to **end**(), then element is inserted at end of list
insert(*iter*, *n*, *val*)	Inserts *n* elements just before the position pointed to by *iter*; each has the indicated value, *val*
pop_back()	Erases the last element; behavior is "undefined" if the vector is empty
pop_font()	Erases the first element; behavior is "undefined" if the vector is empty
push_back(*val*)	Adds specified value, *val*, to the end of the vector
rbegin()	Returns a (reverse) iterator that points to last element
rend()	Returns a (reverse) iterator that points to one position before the beginning of the vector
size()	Returns the current number of elements

The <stack> Template

To enable use of the stack template, include <stack>.

```
#include <stack>
using namespace std;
```

Stack classes can then be instantiated, with each declaration creating a stack that uses the indicated type.

```
stack<int>      stack_of_int;
stack<double>   stack_of_double;
stack<string>   stack_of_string;
stack<Point>    my_stack;
```

The stack template creates a simple last-in-last-out (LIFO) mechanism supporting just a few member functions. Because the template doesn't provide **begin** and **end** functions for iteration, a stack class is not a full container class; it cannot be used with ranged-based **for** supported by the C++0x specification.

Once you declare a stack class, you can push and pop members. Note that popping an STL stack actually requires two operations: **top** to get the top member of the stack and **pop** to remove it. So, remember to "top and pop."

```
stack<string>  beats;
beats.push("John");
beats.push("Paul");
beats.push("George");
beats.push("Ringo");
cout << beats.top() << endl;  // Print "Ringo".
a_stack.pop();
cout << beats.top() << endl;  // Print "George".
a_stack.pop();
```

Note that attempting to pop or remove an item from an empty stack produces "undefined" results, which usually means your program will journey off to the Twilight Zone and stay there. So, be careful not to pop empty stacks.

Table H.5 summarizes the most frequently-used stack template functions.

Table H.5: Stack Template Member Functions

FUNCTION	DESCRIPTION
push(*val*)	Pushes value (of stack's underlying type) onto top of stack
top()	Returns data (of stack's underlying type) from top of stack, but does not remove data; for that, use **pop**
pop()	Removes top item of stack; does not return data
size()	Returns the number of items currently stored in the stack
empty()	Return true if stack is empty; false otherwise

This page intentionally left blank

I

Glossary of Terms

This section provides an overview of terminology used in this book. For more detailed information on these topics, see the index.

abstract class: A class that cannot be used to create objects but that may still be useful as a general pattern (in other words, interface) for other classes. An abstract class has at least one pure virtual function. Such a function is declared virtual and uses the syntax "= 0;" for its implementation.

access level: The level—private, protected, or public—that determines who or what can access members of a given class. Public members are freely accessible outside the class (although references to such members have to be properly qualified). Private members are accessible within the class, but never from outside, and protected members are accessible within the class and any derived classes.

address: The numeric location of a piece of data or program code in memory. This location is often called a "physical location" in memory (although opening the computer to try to find this location won't help you). Addresses, when displayed, are usually shown in hexadecimal notation (base 16) and aren't very meaningful except in the context of a program. The CPU understands only numbers, not words or letters: Numbers used by the CPU to access locations within main memory are addresses.

ANSI: American National Standards Institute. ANSI C++ is a recent specification of C++ that includes many features (such as the **bool** type and new-style cast operators) required for compilers to be up-to-date. However, it is superseded by the C++11 and C++14 specifications, which are even newer.

application: A complete, functioning program, as seen from the user's point of view. A word processor is an application; so is a spreadsheet. Basically, any compiled, tested program that does something useful is an application. The C++ compiler is a kind of application, although useful only to a specialized audience (programmers). You should think of the compiler as a tool, and the program, once compiled, as the application.

argument: A value passed to a function; also known as a parameter.

array: A data structure made up of multiple elements, in which each element has the same base type. Individual elements are accessed through an index number as well as the array name. For example, an array declared as int arr[5] is an array of five int values, accessed as arr[0] through arr[4]. In C and C++, index numbers run from 0 to N − 1, where N is the size of the array.

ASCII: A coding system, adopted by convention, which assigns each printable character (and some nonprintable characters) a unique number in the 1-byte range (0 to 255). Consequently, in C++ a character is stored as **char** (a 1-byte integer), and a string is stored in an array of **char**. Use of ASCII code is built into the computer and low-level software so that when you type an H, for example, the ASCII code for H is put into the data stream and an H is displayed on the screen. (These details are far below the level of most programs, which means you don't have to worry about how all this happens.)

Some such coding system is necessary for programs to handle text, because at the level of machine code (the computer's native language), only numbers are understood. There are some other character coding systems (EBCDIC), but ASCII is universally adopted among personal computers and most minicomputer systems.

associativity: The rule (either left-to-right or right-to-left) that determines how to evaluate an expression that combines two or more operators at the same level of precedence. For example, in the expression *p++, the operators associate right-to-left, so the expression is equivalent to *(p++). This means the pointer gets incremented to point to the next element, rather than incrementing the element itself.

backward compatibility: The policy that states new versions of the C++ compiler should continue to support old programs even while introducing new features. C++ was intended to be largely backward compatible with the C language, although it is not quite 100 percent so. Lack of backward compatibility can cause big problems for programmers, who may find that programs that compiled and ran perfectly well before are suddenly "broken" just because they updated their compiler, as in, "The specification committee broke my programs!" The committee tries to avoid that at nearly all costs.

base class: A class from which you derive another class. All members of the base class are inherited by the derived class, except for constructors.

bit: A single digit equal to 0 or 1, stored in the CPU or in memory. Eight bits make up a byte. You cannot get access to individual bits except through bit fields, bitset templates, and bitwise operations.

bitset: A special kind of collection, supported by the STL, which represents a set of bits in compact form. With a bitset, you can access individual bits one at a time by indexing into the bitset. The least significant bit is indexed as bit 0. See Chapter 17 and Appendix H for more information.

bitwise operations: Operations that compare individual bits in one operand to individual bits in another. Such operations are useful in creating compact bit storage, as well as in creating bit masks, which can be used to mask out combinations of several bits at a time or (by using bitwise OR) to set a group of selected bits to 1. For a contrast, see also *logical operations*.

block: A group of statements put between two curly braces ({}). Blocks are executed as a unit—usually either all or none of the statements are executed. Blocks can define a level of visibility or scope, so that a local variable declared within a block is visible only within that block. See also *compound statement*.

Boolean: A true/false value or true/false operation. In ANSI C++ and later, the **bool** type is fully supported. Where a Boolean value is expected (for example, inside an **if** condition), any nonzero value is converted to 1 and interpreted as true. Note that data objects declared with **bool** can be assigned the special values **true** and **false**. (These are equal to 1 and 0 respectively, but intended to be used in Boolean operations. When any nonzero value is assigned to a Boolean type, it is converted to "true," or rather, 1.)

byte: A group of eight bits stored together. Memory in a computer is organized in bytes, so each byte has its own unique address.

C++11, C++14: These are the most up-to-date specifications for C++ as of this writing. Some of the C++11 features—such as user-defined literals and range-based **for**—are features that C++ programmers have requested for years. C++14 includes all these features, as well as adding other convenient tools such as binary radix and group separators. See Chapter 17 for more information.

C-string: The old text-string type supported by the C language and still fully supported in C++: the format consists of an array of **char** and includes a null-terminating byte. Use of the STL **string** class offers many advantages over using C-strings, but string literals are still stored as C-strings, and some of the old-style string functions (such as **strtok**) in the standard library are still useful. See Chapter 8 for more information.

callback: A function whose address you give to another process or function so that it can *call back* your function at the specified address. An example may help clarify: To call the C++ standard library function **qsort**, you need to pass a **qsort** address of a comparison function you supply. **qsort** then calls this function in order to help determine the proper ordering of any two elements of an array.

With C++ and other object-oriented languages, callback functions are made largely obsolete through the use of virtual functions, which provide a safer and more structured way of doing what callbacks do.

cast: Also known as type cast (or occasionally as data cast). An operation that changes the type of an expression. When a type with a smaller range is assigned to a variable of greater range, C++ automatically promotes the smaller type, and no cast is usually necessary. However, when going in the reverse direction, such as assigning a floating-point number to an integer variable, a cast is necessary to avoid a warning message.

Casts are also useful in other situations. In writing to a binary file, a pointer of base type char is expected, so you need to recast other address types to **char***. If you are using the new-style type cast, this requires **reinterpret_cast**, because it involves pointers. For examples using the new-style casts, as well as the old "C-style" cast, see Appendix A.

class: A user-defined data type, or a data type defined in a library. In C++, a class can be declared by using the **class**, **struct**, or **union** keyword. In traditional programming, a user-defined type, or struct, can contain any number of data members. Object-oriented programming in C++ fully supports structs but adds the ability to declare functions members as well (also called "methods"). Once a class is declared, you can use it to create any number of class instances, called *objects*. See *object-oriented programming*.

code: Another word for "the program" before it's compiled to produce an application. When C++ programmers speak of "code," they are usually referring to the C++ source code, which is the group of C++ statements that make up the program. This use of the word *code* originated in the early days of computer programming, when all programming had to be done in machine code. In those days, each instruction was encoded in a unique pattern of 1s and 0s. Computer languages such as C and C++ are a thousand times more readable, but the term "code" lives on.

compiler: The language translator that reads your C++ program and produces machine code and (ultimately) an executable file that can be run on the computer. This executable file is also called an *application*.

compound statement: A group of zero or more statements (typically more than one) enclosed in curly braces ({}). Also called a *block* or *statement block*. One of the fundamental rules of C and C++ syntax is that in any place a single statement is valid, so is a compound statement. For example, you can use this syntax to put any number of statements inside the body of a **while** loop, so that each time through the loop, multiple statements are executed. Note that a compound statement, or block, can define a level of visibility for local variables, so that any such variable declared within it is local to that block only.

constant: A value that is not allowed to change. All literals are constants, but not all constants are literals. In C and C++, an array name is a symbol (that is, a name), but it is a constant that evaluates to the address of the first element.

constructor: A special member function that is automatically called when an object (an instance of a class) is created. Implicitly, a constructor returns an instance of the class, although constructors never have an explicit return type. (Syntactically speaking, every function in C++ must have a return type—even if it is void—with the two exceptions being constructors and destructors.) Usually, a constructor performs initialization of some kind; note, however, that the compiler-supplied default constructor (see *default constructor*) performs none.

control structure: A way to control what happens next in a program rather than just "go onto the next statement," which is usually what happens. Control structures can make decisions, repeat operations, or transfer execution to a new program location. The **if**, **while**, **do-while**, **for**, and **switch-case** statements are all control structures.

copy constructor: A constructor in which an object is initialized from another object of the same type. For each class, the compiler provides a copy constructor if you don't write one: It performs a simple member-by-member copy.

CPU: Central processing unit. Although nearly all computers in use today have coprocessors that do some of the work (notably floating-point operations and graphics), a personal computer generally has one and only one central processor, which is a silicon "chip" that evaluates each machine-code instruction in a program. (Remember that all programs are translated into machine code before being run.) The central processor's execution of these instructions, one by one, is what drives a computer program to make decisions, perform arithmetic, copy values to memory, and so on.

data member: A data field of a class; each object of the class has its own copy of the data member, unless the member is declared static.

declaration: A statement that provides type information for a variable, class, member, or function. A data declaration—unless it is an **extern** declaration—creates a variable, causing the compiler to allocate memory for it. A function declaration can be either a *prototype* (which contains type information only) or a *definition* (which tells what the function does). In C++, every variable and function except for **main** must be declared before being used; note, however, that **#include** directives bring in declarations for large portions of the standard library.

default constructor: A constructor that has no arguments. For each class, the compiler provides an automatic default constructor if you write no constructor of your own. But if you write *any* constructor, the compiler takes away the automatic default constructor, and then you cannot create objects without initializing them. Such behavior can catch you by surprise, but it is useful when you want to force the user of the class to initialize new objects.

You can avoid this behavior by writing your own default constructor. The compiler-supplied default constructor performs no initialization. Other than allocating memory for the object (which all constructors do implicitly), it is a no-op.

definition, function: A series of statements that tells what a function does. When a function is called, program control is transferred to these statements.

deprecate: A deprecated feature is one that the C++ specifications committee strongly discourages from being used, so much so that the compiler generates a warning message that the programming feature in question may not always be supported. For a long time, the committee had intended to deprecate old-style C casts but changed its mind when it became clear how much C legacy code had been maintained with C++.

dereference: The process of getting data that a pointer points to. If p is a pointer that points to n, then the expression *p "dereferences" the pointer by getting the data stored in n. In theory, you can have a pointer to a pointer to a pointer; in that case, the expression ***p fully dereferences the pointer, producing data of the base type.

derived class: See *subclass*.

destructor: Not as lethal as it sounds. A destructor is a member function that performs cleanup and termination activities when an object is destroyed. The destructor is called just before an object is about to be removed from memory. The declaration of a destructor is ~*class_name*(). Writing a destructor is not required but is a good idea when objects of a class have ownership of resources (such as memory and file handles) that need to be given back so other processes can use them.

directive: A general command to the compiler. Directives affect how the compiler interprets the program but do not correspond directly to runtime actions. For example, the **#include** directive causes the compiler to include the contents of another source file. Unlike most statements, directives do not end with a semicolon (;).

encapsulation: The ability to hide or protect contents, exposing the underlying functionality of what is protected by providing a general (and ideally easy-to-use) interface. For example, by declaring a file-stream object, you gain the ability to read and write to files without having to deal with the operating system's low-level file commands. Encapsulating complex operations and data into classes with consistent, easy-to-use interfaces is a general goal of object-oriented programming.

end user: The person who runs a program, as opposed to writing it. Most programs are designed for end users (typically referred to as *users*) who are not experts and have no knowledge of programming. Ironically, though, the first user of a program—the first person to try it—is almost always the programmer him- or herself.

exception: An unusual occurrence at runtime, typically (but not always) a runtime error. What all exceptions have in common is that they disrupt the normal flow of the program and require immediate action. If an exception is not handled, the program terminates abruptly (and quite rudely) without so much as an explanation to the user. An example of an exception is attempting to divide by zero. C++ provides the **try**, **catch**, and **throw** keywords to let you centralize exception handling in your programs.

floating point: A data format that can store fractional portions of numbers as well as storing numbers in a much larger range than integer types (such as **int**, **short**, and **long**). On a computer, floating-point numbers are stored internally in base 2 but displayed in decimal format. Rounding errors are possible. Many ratios—such as 1/3—cannot be stored precisely in floating-point format, although they can be approximated to a certain precision. The principal floating-point type in C++ is **double**, which stands for "double precision."

Rounding errors are always possible with floating-point numbers, because each floating-point format has a limited precision; in some cases, very high integers may not be stored precisely, although a **long int** or **long long int** might be able to store the same number with absolute precision. In general, you should never use a floating-point type where an integer type would be sufficient.

GCF (greatest common factor): The highest integer that divides evenly into each of two numbers. For example, the greatest common factor of 12 and 18 is 6; the greatest common factor of 300 and 400 is 100.

global variable: A variable shared by all functions in the same source file (or at least all functions whose definitions appear after the declaration of the global variable). You declare a global variable in C++ by declaring it outside of any function. In a multiple-module program, you can even share a global variable among all the functions in the program by using **extern** declarations. A global variable is visible from the point where it is declared until the end of the file. A global variable automatically has a static storage class.

header file: A file that contains a series of declarations and, optionally, directives; it is intended to be included (through the use of **#include** directives) in multiple files. This technique saves programmers from having to enter all the needed declarations into each module of a project, and from having to declare prototypes for library functions. Remember that classes, variables, and functions all have to be declared in C++ before being used.

IDE: Integrated development environment; a text editor from which you can run the compiler as well as test your program.

implementation: A word with many different meanings, but one of the most common is: an implementation of a virtual function that provides a function definition, thereby implementing that function with a particular response.

index: The number used to refer to an element of an array. Index numbers are also used with **char*** strings (which are really arrays), STL string objects, and any container class that supports the brackets operator [ ]. Note that C++ uses zero-based indexing, rather than one-based, in virtually all contexts.

indirection: Accessing data indirectly through a pointer. For example, if pointer ptr points to a variable amount, then the expression "*ptr = 10" changes the value of the amount through indirection.

infinite loop: A loop that apparently continues forever because (for example) the **while** condition is always true. Usually, an infinite loop represents a fatal error in a program, unless there is some kind of exit condition, such as a break or return statement.

inheritance: The ability to give one class the attributes of another, previously declared class. This is done through subclassing. The new class automatically has all the members declared in the base class, except for constructors. See *subclass*.

inline function: A function whose statements are inserted into the body of the function that calls it. In a normal function call, program control jumps to a new location and then returns when execution is complete. This does not happen with an inline function. Instead, the inline function call is expanded by being replaced with the statements defined for it, much like a "macro."

When a member function is defined *within* a class declaration rather than outside the class declaration, the function is automatically made an inline function.

instance/instantiation: An instance of something is a concrete realization of a general category. For example, the Sears Tower is an instance of the category "building." In C++, the word *instance* is usually synonymous with the word *object*. Any individual value or variable is an instance of some type. The number 5 is an instance of **int**, and the number 3.1415927 is an instance of **double**. Every object is an instance of some class. When a class is *instantiated*, it means the class has been used to create an object.

integer: A whole number or, rather, a number with no fractional part. This includes the numbers 1, 2, 3, and so on, as well as 0 and negative numbers –1, –2, –3, and so on. Theoretically, there are an infinite number of integers, but on a computer, any integer type has a finite range, just as other data types do.

interface: This is another word with many different meanings, depending on the context. In this book, I've used it to refer to a general set of services that different subclasses can implement, each in their own way. In C++, you can use an abstract class to define an interface.

iteration/iterative: Computing by using repeated statements (loops). An iterative—as opposed to a recursive—solution is one that repeats a series of statements over and over, usually in a **while** loop, **do-while** loop, or **for** loop. Iteration is the process of repeating a group of statements by jumping back up to the top of the loop after the bottom is reached.

iterator: An object or variable used to cycle through the elements of a container, typically in a **for** loop. In the case of arrays, you can use a simple loop counter as a primitive iterator, but be careful to set its beginning and ending limits carefully. With STL container classes such as <list>, iterators provide a safe and convenient way to cycle through all the elements, although range-based **for** is typically even better. See Chapter 13 for more information.

keyword: A word (such as **if**, **for**, **while**, **return**, or **do**) that has special meaning to the C++ language. Function and variable names that you come up with yourself, or are provided by a library, are not and cannot be keywords.

late binding: Assigning an address to a function at runtime rather than compile or link time. Usually, when a function call is made in a program, the address of the function must be bound to a target address. Late binding causes this decision to be delayed until runtime, at which point the target address may change. This is how C++ and other languages enable member functions to be polymorphic. The exact type of an object pointed to may not be known until runtime, at which point a different implementation of the function is called depending on the object's class.

LCM: Lowest common multiple, the lowest number that two numbers can be divided into evenly. For example, the lowest common multiple of 20 and 30 is 60, because 20 and 30 both divide evenly into 60 (leaving no remainder). Another way of saying this is to say that 20 and 30 are both factors of 60.

LIFO: Last-in-first-out; this is a system of data management that is characteristic of any kind of *stack*. The last item to be put on the stack is the first item to be popped off. This is why stack operations are usually referred to as "pushing" and "popping."

literal: A fixed number such as 5, –100, or 3.1415927, or a text string such as "Mary had a little lamb." A literal is something the compiler recognizes as having a specific value upon reading it in a source file. Its value is fully determined at compile time.

A literal value, unlike a symbol, is not looked up in a table; therefore, it can be used immediately without being initialized. (More often, it *is* the initialization!) All literals are constants, but not all constants are literal.

local variable: A variable that is private to a particular function or statement block. The benefit of local variables is that each function can have its own variable x (for example), but changes to x within one function won't interfere with the value of x in any other function. We can say that the local variable x is *visible* only within its function.

In C++, every block (or compound statement) can declare local variables of its own. Such variables are not visible outside the block.

Most local variables combine two traits: local visibility and an automatic storage class, meaning they are allocated on the stack as temporary variables. Local variables declared **static** retain local visibility but have a static storage class, being loaded into the program only once, at startup.

logical operations: Operations that are intended to create complex Boolean (true/false) expressions. For example, logical AND (&&) produces true if and only if both operands are true. For the purposes of conditional evaluation, any nonzero expression or operand is considered "true." Therefore, the logical expression "5 && 2" evaluates to true, but the bitwise expression "5 & 2" combines the bit patterns 101 with 010 and thereby produces zero (false). In C++, logical expressions use short-circuit logic, so that if the first expression in op1 && op2 is false, the second operand is never evaluated.

loop: A group of statements repeated over and over: The image of "loop" comes from the way control cycles back to the top, each time the bottom is reached.

loop counter: A variable used to control the number of times a loop is executed. The loop counter is omitted in range-based **for**.

lvalue: A "left value," meaning a value that can appear on the left side of an assignment statement. In other words, an lvalue is something you can assign a value to. Variables are lvalues; literals are not. Other examples of lvalues include array members, most class data members (if they are ordinary variables, not array names), and fully dereferenced pointers. Array names, as opposed to array members, are not lvalues because they are constants.

machine code: The computer's own internal language. Such a language (different for every make and model of processor) consists of a unique pattern of 1s and 0s for each possible action; this is why programs are referred to as *code* because each machine instruction is a code for a different operation. The term *code* originated in the 1950s and stuck.

Few programmers ever write in machine code anymore—or even assembly language, which is similar to machine code but uses intelligible names for instructions, such as COPY, JUMP, or JNZ ("jump if not zero"), instead of bit patterns. Because a language like Basic or C++ is closer to human language and frees the programmer from having to worry about the processor's architecture, a programmer writing in C++ can generally accomplish the same task that a machine-code or assembly-code programmer could, only many, many times faster.

main function: The main function is the starting point of a C++ program, and it does not need to be declared before being used. In a console application, a main function is required; other types of applications may use other kinds of starting points. In a console application, main is the only function guaranteed to be run. Other functions in a program are executed only when called.

main memory: The memory in which all computer programs run; it is also called "RAM." Although programs are stored in a disk file or network location, a computer must download a program into main memory before it can run that program. This area is volatile and impermanent; but, generally speaking, it is the only memory that the CPU can access directly. Main memory must be shared with other programs running at the same time, including the operating system.

member: An item declared inside a class. Data members are similar to the fields of a record or structure. Member functions define operations exclusive to the class, which (generally speaking) operate on members of the class.

member function: A function declared within a class. Member functions are sometimes called *methods* in other languages. When you call a member function, it applies to the object through which it was called. But note that all objects of the same class support the same member functions, as well as data members.

memory: Although memory can either be volatile or persistent (in other words, stored in a disk file or other semipermanent medium), the term *memory* itself usually refers to main memory.

method: See *member function*.

module: A large, semi-independent division of a program. A module corresponds to one source file. In the largest programs, multiple modules—each in its own source file—may be compiled and linked together. Object orientation encourages an approach that is highly modular but uses class declarations to delineate modules rather than requiring separate source files.

nesting: Placing one control structure inside another, or one declaration inside another.

newline: A signal to the monitor that says to start a new line of text.

null pointer: A pointer with a zero value. This indicates a "pointer to nowhere": not an uninitialized pointer (which is a dangerous item that could point anywhere at all) but rather a pointer that's specifically set to have no current association with any piece of data. A pointer can be compared to NULL, or rather **nullptr**, to see if it currently points to any meaningful address.

In C++11 and later, **nullptr** is a keyword that has a zero value but always has a pointer—or rather address—type. The **nullptr** keyword, when supported, should be used in preference to NULL for pointer initialization and comparisons. See Chapter 12 for examples.

object: A unit of data that can have behavior (in the form of member functions) as well as an internal state. The concept stems for the old concept of "data record" but is much more flexible. By writing member functions, you can define an object's ability to respond to requests. Furthermore, because of polymorphism in C++, the knowledge of how to carry out an operation is built into the object itself, not the user of the object. The type of an object is its class, and for a given class you can declare any number of objects.

However, even though an object combines both state (data) and behavior (functions), all function code is shared by objects of the same class and, therefore, the class is where member functions are declared and defined.

object code: The machine code generated by the compiler and stored in an intermediate file to be linked into the final executable file (EXE on Windows systems). This term has no connection to objects and object-oriented programming whatsoever. The similarity of the names is frankly unfortunate.

object-oriented programming (OOP)**:** An approach to program design and coding that makes data objects more central, enabling you to define objects by what they do as much as by what data they contain. The object-oriented approach starts by asking, "What kinds of data does the program need, and what kinds of operations can be defined on each such data object?"

As a design methodology, the object-oriented approach has significant advantages. It's easier to break down a big project into its major components when using this approach. Large programs tend to be better organized and more comprehensible with object orientation, because they don't consist of dozens of isolated functions and isolated data structures, but rather modules (or rather, *classes*) of closely related code and data working together. This is extremely helpful in making larger programs easier to read and maintain.

one-based indexing: A system of indexing arrays and strings, which starts at index number 1. C and C++ use zero-based indexing instead, in virtually every context.

OOPS: A user's cry when he spills water on his laptop. But seriously, it's an acronym for object-oriented programming systems. See *object-oriented programming.*

operand: An expression involved in a larger expression through some operator. For example, in the expression x + 5, the items x and 5 are operands.

operator: A special symbol (usually a single character such as +) that combines one or more subexpressions into a larger expression. Some operators are unary, meaning they take only one operand; others are binary, meaning they take two operands. In the expression x + *p, the plus sign is a binary operator and, in this case, the asterisk (*) is a unary operator.

overloading: The reuse of a name or symbol for different—although usually related—purposes. Function overloading lets you use the same name any number of times to define different functions, as long as each function has a different argument list. Operating overloading lets you define how standard C++ operators (such as *, +, and <) work with objects of your own classes.

persistent memory: Memory that forms a semipermanent record so that after the program finishes or the computer is turned off, the data hangs around. Computers provide persistent memory in the form of disk files and other media, such as memory sticks.

pointer: A variable that contains the address of another variable, array, or function. A pointer can also be a null pointer, in which case it "points nowhere." Pointers have many uses in C++, as described in Chapter 7. In general, pointers are valuable because they give you a way of passing a handle to a hunk of data without having to copy all the data itself; you have to copy only the pointer value: that is, the address. Pointers also make dynamic-memory allocation possible, as well as making it possible to create linked lists, trees, and other data structures in memory.

polymorphism: A multisyllabic term meaning "many forms"; in computer programming terms, this is the ability to call a function and have it respond in infinitely many ways at runtime: The implementation to be called depends on the object, and the object (which can change in response to runtime conditions) brings with it its own function code.

A more compact way of saying this is: the knowledge of how to respond to a function call is built into the object itself, not the code that uses the object. Therefore, existing software can interact smoothly with new software yet to be written. In C++, polymorphism is made possible through virtual functions. See Chapter 16 for more information.

precedence: The rules that determine which operations to carry out first in a complex expression. For example, in the expression 2 + 3 * 4, multiplication (*) is carried out first, because multiplication has higher precedence than addition. See Appendix A for the precedence of all operators.

processor: See *CPU.*

program: A group of commands (or rather statements) that, taken together, perform useful actions. A word processor is a program and so is a spreadsheet. (From the standpoint of a user, a program is usually called an *application*.) C++ is a language for writing computer programs using a distinct set of keywords and syntax; this version of the program (called *source code*) is then translated into machine-readable form (machine code) that is run directly on the computer itself. See *application*.

prototype: A function declaration that gives type information only. A prototype is a declaration but not a definition. (Remember that a definition tells what the function *does*.)

pure virtual function: A function that has no implementation—that is, no definition—in the class in which it is declared. In place of an implementation, a pure virtual function has the syntax "=0;".

radix: The base of a number system. Decimal radix (base ten) is the default numbering system in most contexts. C++ also provides octal (0 prefix) and hexadecimal (0x prefix) for numeric literals. C++14 finally adds binary radix (0b prefix), in which all digits are 1s and 0s.

range: The high and low limits of what can be stored in a particular type of variable. For example, the limit of an **unsigned char** (one byte) is 0 to 255.

range-based "for": This is C++'s version of the "for-each" syntax that several other computer languages have. This language feature (introduced in the C++11 spec) enables you to process every element of a container without having to initialize beginning and ending conditions. As such, range-based **for** can prevent a major source of errors. See Chapter 17 for more information.

In C++11 and later, range-based **for** works with STL containers—as long as they have **begin** and **end** functions—and it works with arrays on a limited basis. The limitation with arrays is that if an array is declared local to one function and passed to another, the second function will have no knowledge of how large the array is, and range-based **for** in that case will fail. However, range-based **for** works well with arrays declared as global variables.

recursion: A programming technique in which a function calls itself. This sounds like a logical paradox leading to an infinite regress. Yet the technique is perfectly sound as long as there is a terminating condition—some condition under which the function no longer calls itself. At that point, the function calls, all stored on the stack, begin returning one by one. See Chapter 5 for several examples of recursion. A recursive approach is typically less efficient than an iterative approach, but some problems, notably the Tower of Hanoi puzzle in Chapter 5, are far more difficult to solve without recursion.

reference: A variable or argument that serves as an alias for another variable or argument. To use an analogy, "Mark Twain" was an alias for Samuel Clemens. The different names referred to the exact same individual. In this sense, "Mark Twain" is a reference to Samuel Clemens.

References behave in a way not so different from pointers but without pointer syntax. One big difference between pointers and references (aside from syntax) is that once a reference is assigned, it cannot be made to refer to something else. Note that passing a variable by reference causes the function to refer to the original variable itself, not a copy, so changes made to such an argument have lasting side effects.

scope: The area over which a variable is visible in a program. Local variables have local scope, which means that changes to a variable inside a function have no effect on code outside the function; therefore, every function can have its own local variable i, for example, without affecting the value of i in other functions. Scope can also be defined by namespaces and classes, in which case the scope operator (::) can enable visibility of a symbol outside its ordinary namespace.

source file: A text file containing C++ statements (and optionally, comments and/or directives).

stack: The word *stack* has at least two distinct meanings in computer programming. There is a special area of memory set aside called "the stack," in which the computer places the addresses of functions to return to, as well as the values of arguments and local variables during each function call. This management of the stack is usually invisible to the C++ programmer. In addition, some programs use a stack-like mechanism for their own purposes, and such a data type is provided by the STL. What all stacks have in common is that they use a last-in-first-out (LIFO) system of data management, so that the first item to be popped off a stack is the last item that was pushed on top of it.

Standard Template Library (STL): A library of templates supported by recent versions of C++. STL includes the easy-to-use **string** class, which provides many advantages over the old C-string type. It also includes a simplified stack class, as well as many useful container classes. This book introduces the STL **string** class as well as the **bitset**, **list**, **vector**, and **stack** templates.

statement: A basic unit of syntax in a C++ program. A C++ statement is roughly analogous to a command or sentence. As with sentences in natural languages, there is no fixed length for a C++ statement. It can be terminated at any time—usually with a semicolon (;)—but it can become as complex as you like. A function definition consists of zero or more statements.

In C and C++, an enormous amount of work can theoretically be done inside a single expression, which is finally terminated by a semicolon to form a statement. Why, then, bother with multiple statements? Because a statement is a unit of execution. In a complex expression, it can be difficult to predict what order things happen in… but statements are executed one after another, in the order in which placed in the program, except where control structures change the flow.

The moral, therefore, is not to let complex expressions get out of hand, but to use reasonable-length statements to control the order in which things happen.

statement block: See *compound statement*.

static storage class: Technically, global variables have static storage class, as do local variables declared with the **static** keyword. Such data objects—unlike automatic variables (put on the stack) or dynamically allocated data (created with **new**)—are created just once, at the beginning of the program. In the case of local static, they are only visible in one function but still have a lifetime co-extensive with that of the program itself.

STL: See *Standard Template Library*.

storage class: The manner in which a variable is stored on the computer: *Static* storage class maintains just one copy of the variable in the data area used by the program; *automatic* storage class (used by default for local variables) allocates space for the variable on the stack. This enables each instance of the function to maintain its own copy of the variable. It also means that the data is re-allocated and re-initialized, each and every time the function is called.

string: A series of text characters, which you can use to represent names, words, phrases… anything consisting of printable or unprintable characters. C++ compilers support C-strings, which are arrays of **char***, as well as the newer STL **string** type, which defines behavior for assignment (=), test-for-equality (==), and concatenation (+).

The STL **string** type, in general, is usually much easier to use than the C-string type, especially as you don't have to worry about size limitations. Such strings grow as needed, subject only to available memory.

string literal: A text string enclosed in quotation marks, such as:

"Here comes the sun."

When C++ reads a text string from the source file (assuming it is not inside a comment), it stores the characters as a C string, which is an array of **char** that includes an extra byte for the null terminator. The string name is then associated with the address of this data. Note that the backslash (\) signals an escape character in a C++ string. To represent an actual backslash, use two: \\. (See Appendix B for a list of escape characters.)

subclass: A class that inherits, or is derived from, another class (called the *base class*). The subclass inherits all the members of the base class except for constructors. Any declarations within the subclass create new or overridden members. Note: it is unsafe to override a function not declared **virtual**.

symbol: A variable, class, or function name. Unlike a literal, a symbol is just a name, and it derives its meaning and value according to context. Usually, it has no set value until assigned a value or initialized. Symbols (or *symbolic names*) must adhere to C++ naming rules: A symbol must begin with a letter or underscore (_), and the remaining characters must be letters, numbers, or uses of the underscore (_).

template: A generalized class, usually a container of some sort, that is built around a more specific class. For example, the STL list class can be used to create lists of any type: list<int>, list<float>, list<double>, and so on. Templates make use of a generalized algorithm or solution and apply to different kinds of data. Recent C++ compilers support the ability to define new templates; they also supply many useful templates in the Standard Template Library (STL).

text string: See *string*.

token: A word, symbol, or operator produced through lexical analysis. In plainer terms, if you input a line of text, each item separated by spaces or comma is usually regard as a token. So in the input string "amt = 3 + 15", "amt", "3", "+", and "15" are all tokens.

two's complement: The most common format, especially on today's personal computers, for storing signed (as opposed to unsigned) integers. The leftmost bit in such a number is 1 if the number is negative; it is 0 if the number is nonnegative. See Appendix B for more information.

variable: A named location for storing program data. Each variable has a unique location in program memory (its address). Other attributes of a variable include its type (for example, **int**, **double**, or **float**), its visibility (local or global), and its storage class.

vector: An array that can grow without limit. The STL supports such a mechanism in the form of the **vector** template. See Chapter 15 for examples.

virtual function: A function whose address is not determined until runtime, through a process called *late binding*. In C++, virtual functions are closely connected to the concept of polymorphism. Virtual functions have special flexibility: you can safely override a virtual function in a subclass, knowing that no matter how an object is accessed, the right version of the function will always be called. So, for example, a function call such as ptr->vfunc() will call the object's own version of vfunc rather than the base-class version. See *late binding*.

In ancient Rome, virtu meant "manliness"; in modern parlance, virtual means "to have the behavior of." This example of (otherwise regrettable) sexism has one redeeming feature: It suggests that for a man to take his place in society, his behavior was paramount; in ancient Rome, a man had to earn the right to be a consul, praetor, senator, or man of respect, and that meant correct (and "manly") behavior. Thus, the connecting thread in the word "virtual" over the millenia is that it focuses on *behavior*. In the computer world, we say that if something is virtual, it emulates the behavior of the real thing. For example, a virtual function call looks just like an ordinary function call and is used just like such a function call. It is *virtual* because to the caller, it is just as good as a "real" function call, even though the actual address is not fixed until runtime.

visibility: Essentially the same as scope. Global variables are visible from the place they are declared, onward to the end of the source file. Local variables are visible only within the functions they are declared. See *scope*.

zero-based indexing: A system of indexing arrays and strings that starts at 0 and runs to N − 1, where N is the size of the container. The second element is indexed as 1, the third is indexed as 2, and so on. Although this technique may seem less reasonable at first, it makes sense when you think of indexes as being like offsets. Thus, the first element is always 0 units distant from the beginning. C and C++ use zero-based indexing for arrays and strings, and for almost every other context imaginable.

Index