PRENTICE HALL

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

C++ WITHOUT FEAR

SECOND EDITION

- **Learn** programming basics fast
- **Write** your first C++ programs

NEW FEATURES

- **Even more** figures, examples, and exercises
- **Even more** puzzles and games
- **An expanded** 75-page language reference
- **Instructions** for downloading free C++ software

BRIAN OVERLAND

Updated for C++0x!

C++ Without Fear

Second Edition

This page intentionally left blank

C++ Without Fear

Second Edition

A Beginner's Guide That Makes You Feel Smart

Brian Overland

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Overland, Brian R.
C++ without fear : a beginner's guide that makes you feel smart /
Brian Overland.—2nd ed.
p. cm.
Includes index.
ISBN 978-0-13-267326-6 (pbk. : alk. paper)
1. C++ (Computer program language) I. Title.
QA76.73.C153O838 2011
005.13'3—dc22

2011004218

ISBN-13: 978-0-13-267326-6
ISBN-10: 0-13-267326-6
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Third printing, August 2012

For Colin

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Many years ago, when I had to learn C overnight to make a living as a programmer (this was before C++), I would have given half my salary to find a mentor, a person would say, "Here are the potholes in the road...errors that you are sure to make in learning C. And here's how to steer around them." Instead, I had to sweat and groan through every error a person could make.

I'm not just talking about programmers who can write or writers who can program. Each of those is rare enough. Much rarer still is the person who is programmer, writer, and *teacher*—someone who will steer you around the elementary gotchas and enthusiastically communicate the "whys" of the language, including why this stuff is not just useful but, in its own way, kind of cool.

It's hard to find such a person. But way back then, I swore this is the person I'd become.

Later, at Microsoft, I started in tech support and testing and worked my way into management. But my most important job (I felt) was explaining new technology. I was sometimes the second or third person in the world to see a new feature of a programming language, and my job was to turn a cryptic spec into readable prose for the rest of the universe to understand. I took the goal of "make this simple" as not just a job but a mission.

About This Book: How It's Different

What's different about this book is that I'm an advocate for you, the reader. I'm on your side, not that of some committee. I'm aware of all the ways you are "supposed" to program and why they are supposed to be better (and I do discuss those issues), but I'm mostly concerned about telling you what *works*.

This book assumes you know nothing at all about programming—that you basically know how to turn on a computer and use a mouse. For those of you more knowledgeable, you'll want to breeze through the first few chapters.

The creators of C and C++—Dennis Ritchie and Bjarne Stroustrup, respectively—are geniuses, and I'm in awe of what they accomplished. But although C and C++ are great languages, there are some features that beginners (and even relatively advanced programmers) never find uses for, at least not for the first few years. I'm not afraid to tell you that information up front: what language features you can and should ignore. At the same time, I'm also eager to tell you about the elegant features of C++ that can save you time and energy.

This is a book about practical examples. It's also a book about having fun! The great majority of examples in this book either are useful and practical or—by using puzzles and games—are intrinsically entertaining.

So, have no fear! I won't bludgeon you to death with overly used (and highly *abused*) terms like *data abstraction*, which professors love but which forever remain fuzzy to the rest of us. At the same time, there are some terms—*object orientation* and *polymorphism*—that you will want to know, and I provide concrete, practical contexts for understanding and using them.

Onward to the Second Edition

The first edition has sold increasingly well over the years. I believe that's a testament to the variety of learning paths it supplied: complete examples, exercises, and generous use of conceptual art. The second edition builds on these strengths in many ways:

- **Coverage of new features in C++0x**: This is the new specification for C++ that will be standard by the time you have this book in your hands. Compiler vendors either have brought their versions of C++ up to this standard or are in the process of doing so. This book covers well over a dozen new features from this specification in depth.
- **Examples galore, featuring puzzles and games**: By the end of Chapter 2, you'll learn how to enter a program, barely a page long, that not only is a complete game but even has an optimal strategy for the computer. Just see whether you can beat it! But this is only the beginning. This edition features puzzles and games, much more so than the first edition.
- **Generous use of conceptual art**: The use of clarifying illustrations to address abstract points was one of the biggest strengths of the first edition. This edition has substantially more of these.
- **Even more exercises**: These encourage the reader to learn in the best way...by taking apart an example that works, analyzing it, and figuring out how to modify it to make it do your own thing.

- **No-nonsense syntax diagrams:** Programming and playing games is fun, but sometimes you need straightforward information. The syntax diagrams in this book, accompanied by loads of examples, clarify exactly how the language works, statement by statement and keyword by keyword.
- **Introduction to Standard Template Library (STL):** Although I lacked the space to do a complete STL manual, this edition (unlike the first) introduces you to the wonders of this exciting feature of C++, showing how it can save you time and energy and enable you to write powerful applications in a remarkably small space.
- **Expanded reference:** The appendixes in the back are intended as a mini desk reference to use in writing C++ programs. This edition has significantly expanded these appendixes.
- **Essays, or "interludes" for the philosophically inclined:** Throughout the book, I detour into areas related to C++ but that impact the larger world, such as computer science, history of programming, mathematics, philosophy, and artificial intelligence. But these essays are set aside as sidebars so as not to interfere with the flow of the subject. You can read them at your leisure.

"Where Do I Begin?"

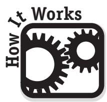
As I mentioned, this book assumes you know nothing about programming. If you can turn on a computer and use a menu system, keyboard, and mouse, you can begin on page 1. If you already have some familiarity with programming, you'll want to go through the first two or three chapters quickly.

If you already know a lot about C or C++ and are mainly interested in the new features of C++0x, you may want to go straight to Chapter 10, "New Features of C++0x."

And if you know C and are now starting to learn about object orientation with the C++ language, you may want to start with Chapter 11, "Introducing Classes: The Fraction Class."

Icons, Icons, Who's Got the Icons?

Building on the helpful icons used in the first edition, this edition provides even more—as signposts on the pages to help you find what you need. Be sure to look for these symbols.



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.)

After each full programming example, I provide at least one exercise, and usually several, that builds 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 can be found on the book's Web site (*www.informit.com/title/9780132673266*).

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

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.

This icon indicates a place where a keyword of the language is introduced and its usage clearly defined.

C++0x ▶ This icon is used to indicate sections that apply only to versions of C++ compliant with the new C++0x specification. Depending on the version of C++ you have, either these sections will apply to you or they won't. If your version is not C++0x-compliant, you'll generally want to skip these sections.

What Is Not Covered?

Relatively little, as it turns out. The two features not covered at all are bit fields and unions. Although these features are useful for some people, their application tends to be highly specialized—limited to a few special situations—and not particularly useful to people first learning the language. Of course, I encourage you to learn about them on your own later.

Another area in which I defer to other books is the topic of writing your own template classes, which I touch on just briefly in Chapter 16. Without a doubt, the ability to write new template classes is one of the most amazing features of state-of-the-art C++, but it is a very advanced and complex topic. For me to cover it adequately and exhaustively could easily have taken another 400 or 500 pages!

Fortunately, although templates and the Standard Template Library (STL) are advanced subjects, there are some good books on the subject—for example, *C++ Templates: The Complete Guide*, by David Vandevoorde and Nicolai M. Josuttis; *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*, by David R. Musser, Gillmer J. Derge, and Atul Saini; and *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers.

And remember that Chapter 16 does introduce you to using STL, which provides extremely useful, existing templates for you to take advantage of.

Getting Started with C++: A Free Compiler

Although this edition doesn't come with a CD with a free compiler on it, that is no longer necessary. You can download some excellent shareware (that is, free) versions of C++ from the Internet that not only have a free compiler (that's the application that translates your programs into machine-readable form) but also a very good development environment. And they install easily.

To download this free software, start by going to the book's Web site: *www.informit.com/title/9780132673266*.

As mentioned earlier, you will also find downloadable copies of all the full program examples in the book, as well as answers to exercises.

A Final Note: Have Fun!

Once again, there is nothing to fear about C++. Yes, there are those nasty potholes I started out discussing, but remember, I'm going to steer you around them. Admittedly, C++ is not a language for the weak of heart; it assumes you know exactly what you're doing. But it 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 a book about enjoying the ride.

This page intentionally left blank

Acknowledgments

I am likely to leave many deserving people out this time, but a few names cry out for special mention. The book's editor, Peter Gordon, not only took the initiative in arranging for the new edition but did a lovely job of nursing the book through all its stages along with the author's ego. His long-suffering right hand, Kim Boedigheimer, was a better person than we all deserved, coming to the rescue again and again and kindly aiding the author. I'd also like to extend a special thanks to Kim Wimpsett and Anna Popick, who unexpectedly have been an absolute delight to work with in getting the book through its final tense stages.

Closer to home in the Seattle area: I also want to make special mention to veteran Microsoft programmers John R. Bennett and Matt Greig, who provided superb insights about the latest directions of C++. Some of the more interesting new sections in the book came about directly as a result of extended conversations with these experts.

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.

4 Functions: Many Are Called

The most fundamental building block in the programming toolkit is the function—often known as *procedure* or *subroutine* in other languages. A function is a group of related statements that accomplish a specific task. Once you define a function, you can execute it whenever you need to do so.

Understanding functions is a crucial step to programming in C++: Without functions, it would be a practical impossibility to engage in serious programming projects. Imagine how difficult it would be to write a word processor, for example, without some means of dividing the labor. Functions make this possible.

The Concept of Function

If you've followed the book up until this point, you've already seen use of a function—the **sqrt** function, which takes a single number as input and returns a result.

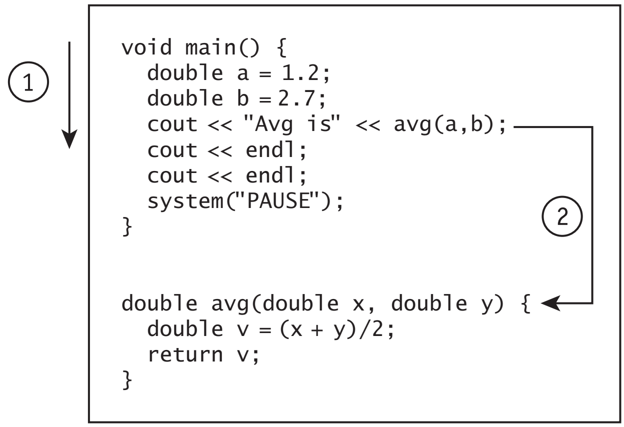
```
double sqrt_of_n = sqrt(n);
```

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:

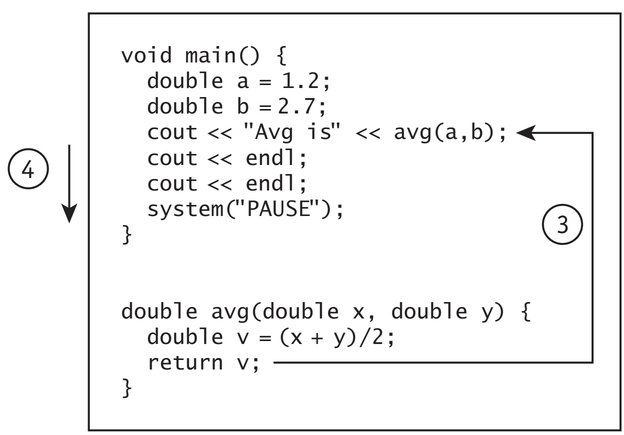
```
cout << avg(1.0, 4.0);
```

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 then is transferred back to the caller.

This may sound like a foreign language if you're not used to it. It's easy to see in a conceptual 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.)



The function runs until it encounters **return**, at which point: 3) execution returns to the caller of the function, which in this case prints the value that was returned. Then, 4) execution resumes normally inside **main**, and the program continues until it ends.



Note that only **main** is guaranteed to be executed. Other functions run only as called. But there are many ways a function can be called. For example, main can call a function A, which in turn calls B and C, which in turn calls 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

A function declaration (or *prototype*) provides type information only. It has this 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

The function definition tells what the function does. It uses this syntax:

```
return_type   function_name (argument_list) {
    statements
}
```

Most of this looks like a declaration. 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

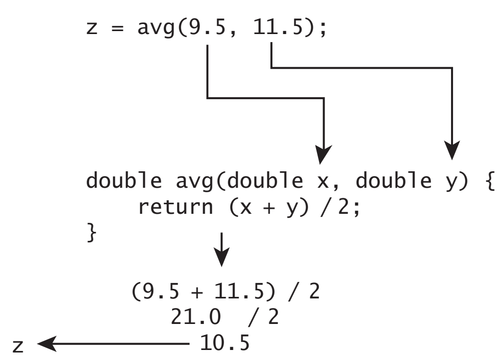
Once a function is declared and defined, it can be used—or rather, *called*—any number of times, from any function. For example:

```
n = avg(9.5, 11.5);
n = avg(5, 25);
n = avg(27, 154.3);
```

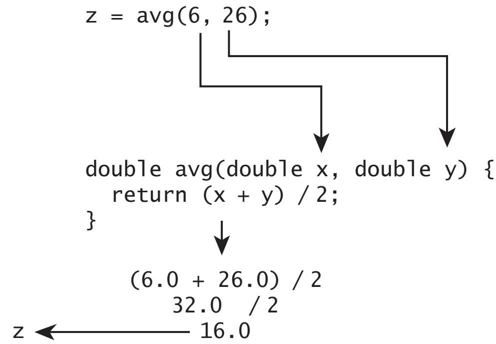
A function call is an expression: As long as it returns a value other than **void**, it can be used inside a larger expression. For example:

```
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.



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**.)



Example 4.1. *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);

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;
```

▼ continued on next page

avg.cpp, cont.

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

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);

    system("PAUSE");
    return 0;
}

// print_results function definition
//
void print_results(double a, double b) {
    cout << "Average is: " << avg(a, b) << endl;
}
```

▼ continued on next page

avg2.cpp, cont.

```
// 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. The program creates a flow of control as follows:

main() → print_results() → avg()

EXERCISES

Exercise 4.1.1. 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 4.1.2. 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 4.2. *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, is_prime.

prime2.cpp

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

prime2.cpp, cont.

```
// Function must be declared before being used.
bool prime(int n);

int main() {
    int i;

// Set up an infinite loop; break if user enters 0.
// Otherwise, evaluate n from prime-ness.

   while (true) {
     cout << "Enter num (0 = exit) and press ENTER: ";
     cin >> i;
     if (i == 0)            // If user entered 0, EXIT
        break;
     if (prime(i))                   // Call prime(i)
        cout << i << " is prime" << endl;
     else
        cout << i << " is not prime" << endl;
    }
    system("PAUSE");
    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) {
    int i;

    for (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 3.2 on page 75, you'll see only a few differences.

```
bool prime(int n) {
    int i;

    for (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, return
true.
}
```

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 >> i;
      if (i == 0)            // If user entered 0, EXIT
          break;
      if (prime(i))                    // Call prime(i)
         cout << i << " is prime" << endl;
      else
         cout << i << " is not prime" << endl;
     }
```

The rest of the loop calls the prime function and prints the result of the prime-number test. Note that the prime function, in this case, returns a true/false value, and so the call to prime(i) can be used as an if/else condition.

Exercises

EXERCISES

Exercise 4.2.1. 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 4.2.2. 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 4.2.3. Write a program that finds the first prime number greater than 1 billion (1,000,000,000).

Exercise 4.2.4. 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 4.2). 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 was 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. If you look back at Example 2.3, you'll see that i is local; **main** and prime each declare their own version of i.

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 functions interfering with each other as little 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 variation 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 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 often the best way to communicate information *between* 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.

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. And as you'll see, it's less crazy than it sounds. The technique of a function calling itself is called *recursion*. The obvious problem is the same one 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 4.1.1 (page 90)? 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.

```
factorial(4)
   ↓
   4 * factorial(3)
          ↓
          3 * factorial(2)
                 ↓
                 2 * factorial(1)
                        ↓
                        1
```

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.

Example 4.3. *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 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 was input, we'd get this:

```
2, 2, 3, 3
```

If 99 was input, we'd get this:

```
3, 3, 11
```

And if a prime number was input, the result would be the number itself. For example, if 17 was input, the output would 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;
```

▼ continued on next page

prime3.cpp, cont.

```
    get_divisors(n);
    cout << endl;
    system("PAUSE");
    return 0;
}

// Get divisors function
//  This function prints all the divisors of n,
//  by finding the lowest divisor, i, and then
//  rerunning itself on n/i, the remaining quotient.

void get_divisors(int n) {
     int i;
     double sqrt_of_n = sqrt(n);

     for (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: ";
    cin >> n;

    cout << endl;
    system("PAUSE");
    return 0;
}
```

The get_divisors function is the interesting part of this program. It has a **void** return value, meaning that it doesn't pass back a value. But it still uses the **return** statement to exit early.

```
void get_divisors(int n) {
     int i;
     double sqrt_of_n = sqrt(n);

     for (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 (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.

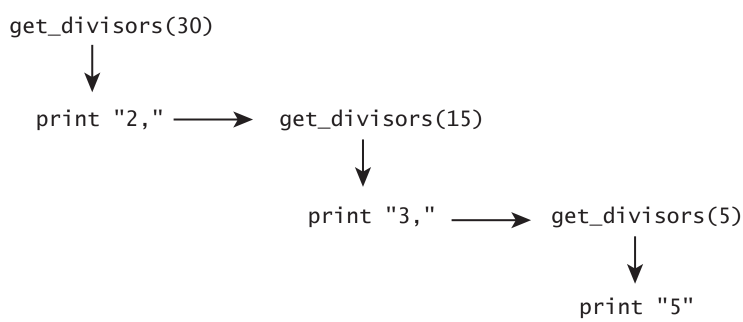
If no divisors are found, that means the number being tested is prime. The correct response is to print this number and stop.

```
cout << n;
```

For example, suppose that 30 is input. 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

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.

Interlude

▼ continued

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 4.3.1. Rewrite the **main** function for Example 4.3 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 4.2, on page 90.)

Exercise 4.3.2. 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 4.3.3. Modify Example 4.3 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 4.4. 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 11, 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.

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.
- 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 terminal case, in which B equals 0, is valid assuming A is nonzero. You can see that A divides evenly into both itself and 0, but nothing larger can divide into A. (Note that 0 can be divided evenly by any whole number except itself.) For example, 997 is the greatest common factor for the pair (997, 0). Nothing larger divides evenly into both.

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;
```

gcf.cpp, cont.

```
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;

    system("PAUSE");
    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 profound influence on mathematicians and philosophers ever since.

Interlude

▼ continued

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 4.4.1. 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 4.4.2. 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.

▼ continued on next page

Interlude

▼ continued

If these 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*—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 4.5. 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 problem that illustrates recursion beautifully, solving a problem that would be very difficult to solve otherwise.

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.

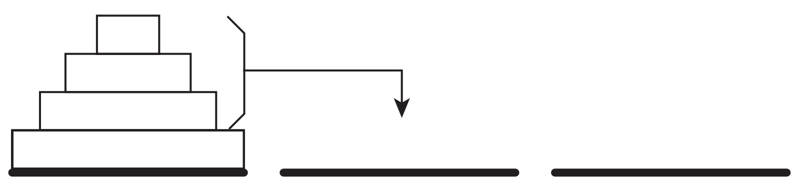
It sounds easy, until you try it! Consider a stack four rings high: You start by moving the top ring from the first stack, but where do you move it, and what do you do after that?

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.

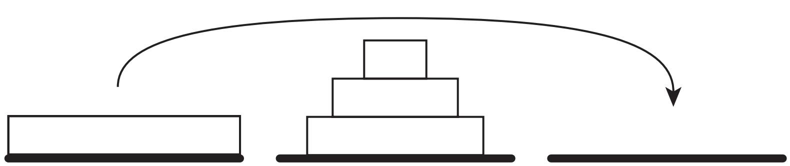
This is easier to envision graphically. First, the algorithm moves N–1 rings from the source stack to the "other" stack ("other" being the stack that is neither source nor destination for the current move). In this case, N is 4 and N–1 is 3, but these numbers will vary.

1. Move N–1 rings from source to "other."



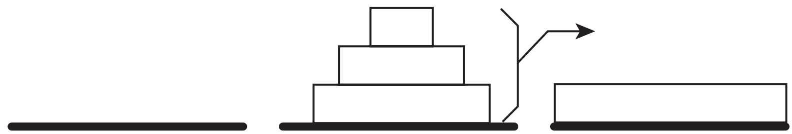
After this recursive move, at least one ring is left at the top of the source stack. This top ring is then moved: This is a simple action, moving one ring from source to destination.

2. Move one ring from source to destination, directly.



Finally, we perform another recursive move, moving N–1 rings from "other" (the stack that is currently neither source nor destination) to the destination.

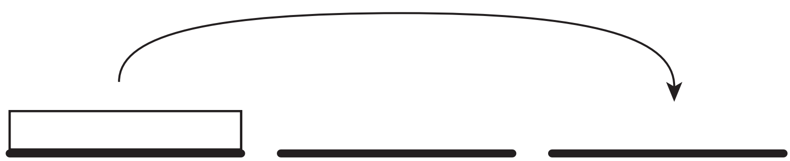
3. Move N–1 rings from "other" to destination.



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.

It's important, also, to solve the terminal case, N = 1. But that's trivial. 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 <cstdlib>
#include <iostream>

using namespace std;
void move_rings(int n, int src, int dest, int other);

int main()
{
  int n = 3;  // Stack is 3 rings high

  move_rings(n, 1, 3, 2); // Move stack 1 to stack 3
  system("PAUSE");
```

tower.cpp, cont.

```
    return 0;
  }

  void move_rings(int n, int src, int dest, int other) {
    if (n == 1) {
      cout << "Move from " << src << " to " << dest
           << endl;
    } else {
      move_rings(n - 1, src, other, dest);
      cout << "Move from " << src << " to " << dest
           << endl;
      move_rings(n - 1, other, dest, src);
    }
  }
```

4

How It Works

The program is brief 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 correctness 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);
cout << "Move from " << src << " to " << dest
     << endl;
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, which is not used now but will be at the next level).

EXERCISES

Exercise 4.5.1. 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 4.5.2. Instead of printing the "Move" message directly on the screen, have the move_ring function call yet another function, 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 4.6. *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 <cmath>
#include <cstdlib>
#include <ctime>
using namespace std;

int rand_0toN1(int n);

int main() {
    int n, i;
    int r;

    srand(time(NULL)); // Set seed for random numbers.

    cout << "Enter number of dice to roll: ";
    cin >> n;

    for (i = 1; i <= n; i++) {
        r = rand_0toN1(6) + 1; // Get a number 1 to 6
        cout << r << " ";      // Print it
    }
    system("PAUSE");
    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

The beginning of the program has to include a number of files to support the functions needed for random-number generation:

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

Make sure you include the last three here—cmath, cstdlib, and ctime—whenever you use random-number generation.

Random-number generation is a difficult problem in computing, because computers follow deterministic rules—which, by definition, are nonrandom. The solution is to generate what's called a *pseudorandom* sequence by taking a number and performing a series of complex transformations on it.

To do this, the program needs a number as random as possible to start off the sequence. So, we're back where we started, aren't we?

Well, fortunately no. You can take the system time and use it as a *seed*: That is the first number in the sequence.

```
srand(time(NULL));
```

NULL is a predefined value that means a data address set to nothing. You don't need to worry about it for now. The effect in this case is simply to get the current time.

C++0x ▶ The C++0x specification provides the **nullptr** keyword, which should be used in preference to NULL if you have a C++0x-compliant compiler.

A program that uses random numbers should call **srand** first. System time changes too quickly for a human to guess its exact value, and even a tiny difference in this number causes big changes in the resulting sequence. This is a practical application of what chaos theorists call the Butterfly Effect.

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 (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;
}
```

This is one of the simplest functions we've seen yet! Calling **rand** produces a number anywhere in the range of the **int** type, which, on 32-bit systems, can be anywhere in the range of roughly plus or minus two billion. But we want much smaller numbers.

The solution is to use your old friend, the remainder-division operator (%), to divide by n and return the remainder. No matter how large the amount being divided, the result must be a number from 0 to n–1, which is exactly what the function is being asked to provide.

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 4.4.1. 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 4.4.2. 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.

The Subtraction Game example at the end of Chapter 2 can be improved. Right now, when the user plays optimal strategy, the computer responds by choosing 1. We can make this more interesting by randomizing the computer's response in these situations. The following program makes the necessary changes, putting altered lines in bold:

nim2.cpp

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
```

▼ continued on next page

nim2.cpp, cont.

```
using namespace std;

int rand_0toN1(int n);

int main() {
    int total, n;

    srand(time(NULL)); // Set seed for random numbers.

    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 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;
         }
```

nim2.cpp, cont.

```
            total = total - n;
            cout << "New total is " << total << endl;
            if (total == 0) {
                cout << "You win!" << endl;
                break;
            }
        }
        system("PAUSE");
        return 0;
}

int rand_0toN1(int n) {
    return rand() % n;
}
```


Chapter 2 presented an exercise: Alter this program so that it permits any number from 1 to N to be subtracted each time, where N is set at the beginning. That problem is left as an exercise for this version as well. (You can even prompt the end user for this value before the game starts. As always, the computer should win whenever the user does not play perfect strategy.)

The last full example in Chapter 10 presents a game of Rock, Paper, Scissors that can be programmed even with C++ compilers that are not fully C++0x compliant. To use the example in Chapter 10 (Example 10.3) with weak, rather than strong, enumerations, replace this line in Chapter 10:

```
enum class Choice { rock, paper, scissors };
```

with this:

```
enum Choice { rock, paper, scissors };
```

Also, remove the **using** statement:

```
using namespace Choice;
```

Chapter 4 *Summary*

Here are the main points of Chapter 4:

- 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 (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.
- The addition-assignment operator (+=) provides a concise way to add a value to a variable. For example:

```
n += 50;                // n = n + 50
```

- 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!
```

Index

Symbols

Numbers

A

C

D

M

O

P

T