C++ FOR THE IMPATIENT

BRIAN OVERLAND

C++ for the Impatient

Brian Overland

Addison-Wesley

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

This address may be assigned to a standard pointer, but do not use **delete** with this pointer. Remember, also, that the address becomes invalid when the *unique_ptr* is destroyed.

6.7 Sample App: Sieve of Eratosthenes

The program in this section makes good use of a dynamically allocated array. The size of this array is determined at runtime, in response to the user. The program accesses elements by using the subscript operator, [], with a pointer.

This famous "Sieve" was invented around 250 B.C.E. by an ancient Greek scientist and mathematician named Eratosthenes. It turns out to be an excellent test for the performance of computers. The Sieve provides perhaps the fastest way possible to compile a large group of prime numbers. The technique is as follows.

1. Write out all the numbers from 2 to N, where N is the upper limit of numbers you want to examine.
2. Begin by assuming that all the numbers are prime; that is, they are not composite numbers (a composite number is a number that is the product of two other integers A and B, where neither is 1).
3. Beginning with 2, look at the current number. If this number is prime, cross out all the multiples of this number—except for the number itself. These crossed-out numbers are flagged as not being prime. For example, 2 is prime, so eliminate all the multiples of 2 beginning with 4, 6, 8, and so on.
4. Then advance to the next number and repeat step 3; continue until N is reached.
5. The numbers that are left—not crossed out—are prime. (This is why it is a "sieve": The primes are the numbers left over once the composite numbers—the non-primes—are eliminated. The numbers that "fall through," as if in a sieve, are the primes.)

The program in this section takes essentially the same approach, using a Boolean array and initializing all elements to **true**. Each element of the Boolean array corresponds to a number from 2 to N. If p[I] is **true**, that means I is a prime number. To keep the programming simple, the first two elements—representing 0 and 1—are included in the array but ignored.

Whenever a prime is found (that is, the corresponding element in the Boolean array is **true**), the process_prime function is called to do two things.

1. Print the number, followed by a tab.
2. Beginning with 2 times this number, flag all multiples by setting the corresponding elements in the Boolean array to **false**.

Here is the program listing.

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

void process_prime(bool *p, int max_n, int n);

int main() {
     int max_n = 0;
     cout << "Calculate primes up to what number? ";
     cin >> max_n;
     bool *p = new bool[max_n + 1];    // Allocate array.
     for (int i = 2; i <= max_n; ++i) {    // Init array.
          p[i] = true;
     }
     for (int i = 2; i <= max_n; ++i) {    // Find primes.
          if (p[i]) {
               process_prime(p, max_n, i);
          }
     }
     cout << endl;
     delete [] p;      // Not absolutely necessary, but
                       //   a good idea.
     cin.ignore();     // Consume last carriage return.
     cin.ignore();     // Wait for user to press ENTER.
     return 0;
}

// Process Prime function.
// Print the number n passed to the function, then
//   flag all the elements in the Boolean array (p)
//   that correspond to a multiple of n as false.
//   So, p[i] <= false if i is a multiple of n.
void process_prime(bool *p, int max_n, int n) {
     cout << n << "\t";
     for (int i = n + n; i <= max_n; i += n) {
          p[i] = false;
     }
}
```

Exercises

1. Upon program termination, a C++ program is expected to release any memory resources still in use. Given this fact, why might it still be a good idea (or at least not a bad one) to include a **delete** statement?
2. Section 4.6, "for Statements," contained another program for calculating all the primes from 2 up to a given number. (This number was set to 100 in that example, but it could have been any positive integer.) Which program—the one in Section 4.6 or the Sieve program listed here—is likely to execute more quickly at runtime? Make an argument for one or the other having better runtime performance.
3. Some of the earliest C and C++ compilers had no **bool** type, as well as no **true** and **false** keywords. Instead, programmers had to use **int** variables to store Boolean values. Adapt the program so that it uses an array of integers rather than an array of **bool**. Instead of **true** and **false**, use 1 and 0. Although most of the time this is the last thing you'd want to do, it is still a useful programming exercise to demonstrate that you understand the variables and operators involved.
4. Revise the Sieve program to report the number of primes found.
5. Revise the Sieve program further to report only the primes in a given range, from N to M. (So you'll have to prompt the user for both a lower and upper bound.) Have the program print only primes in this range, and have it total the number of primes in this same range.
6. If your compiler is C++11 compliant, rewrite the application to use a smart pointer.
7. Add statements to handle the possibility of insufficient space for dynamic memory allocation (which tends to be rare on today's computers, but still possible). You can either write exception-handling code for a **bad_alloc** exception or use the **nothrow** keyword as described in Section 6.3.6, to get back a null pointer if and when **memory** allocation fails.

◆ ◆ ◆ 7 CHAPTER 7

Classes and Objects

One of the central features of C++ is that of *classes,* the single most important ability that C++ adds to C. There are many ways to think of a class. One of the best ways is this: A class is a user-defined type. Once you declare it, you can use it just as you would **int**, **char**, or **double**.

By declaring a class, you create a fundamental new data type, extending C++ itself. You can, if you choose, define how the type responds to operators (such as +, -, or <). You can also provide services in the form of member functions, effectively creating intelligent data types. An object "knows" how to respond to function calls and retains that knowledge even when it is accessed through a pointer of a more general type. (This is called *polymorphism*.)

Above all, classes provide a rational and flexible way to group together closely related functions and data. For larger projects, this is a superior way to design programs. Class-based architecture and design—known as object-oriented programming (OOP)—meshes well with the needs of event-driven, database, and graphical user interface applications.

7.1 Overview: Structures, Unions, and Classes

C++ extends the C concept of data structure (called a "data record" in some languages) to the wider concept of *class*. For example, here is a declaration of a simple class named Point:

```
class Point {
public:
     double x, y;
};
```

The Point class is now declared as a type. Given this declaration, the program can create any number of individual Point variables, called *objects*. All of these objects share the same structure: Each Point object contains its own x and y settings.

```
Point   my_point, your_point, pt1, pt2;
```

Each of these objects (my_point, your_point, pt1, and pt2) has its own x and y members. With each object, you can access its x and y values by using the *object.member* syntax.

```
my_point.x = 0.0;            // Access my_point members.
my_point.y = 0.0;
your_point.x = 1.5;          // Access your_point members.
your_point.y = -2.5;
pt1_point.x = 0.0;           // Access pt1 members.
pt1_point.y = 3.1415;
pt2_point.x = 100.0;         // Access pt2 members.
pt2_point.y = 250.7;
```

The **struct**, **class**, and **union** keywords all can be used to declare classes in a similar manner. The following list compares and contrasts the uses of these keywords.

- **`struct`**

  Creates a **struct** data type that is backward-compatible with how this keyword is used in C. Everything stated in this chapter about the **class** keyword applies equally to the **struct** keyword, with this difference: Members of a **struct** class are public by default. (They also use public inheritance by default; see Section 7.7.2, "Base-Class Access Specifiers.") Otherwise, the **struct** keyword can be used just like the **class** keyword.

  If you are porting code from C, you should expect classes created with **struct** to work in C++ just as they did in C. However, you have the option of adding member functions and operator support to existing structures, which you couldn't do in C.

- **`class`**

  Creates a class just as the **struct** keyword does, but member access is private by default. In C++, the **class** keyword is the preferred keyword for declaring new classes, but **struct** can be used as well. As with the **struct** keyword, the **class** keyword declares a type that can optionally provide services in the form of member functions and operators.

- **`union`**

  Creates a data type in which members have public access by default; but in a union, all the members share the same address in memory. As a result, the same location can be used to store different kinds of data at different times. A union can also be used to read data in one format and then read out the same bit pattern in a different format. Section 7.9, "Unions," covers unions.

7.2 Basic Class Declaration Syntax

Class syntax in C++ can become complex, so the sections that follow focus on the fundamentals of declaring and using an elementary class. Later sections introduce more advanced concepts such as constructors, operator functions, and deriving one class from another (subclassing).

7.2.1 Declaring and Using a Class

Here is the basic syntax for declaring a class. Although the **class** keyword is recommended for creating new data types, you can optionally use **struct** instead of **class**; the difference is that **struct** classes have members with public access by default.

```
class name {
     declarations
};
```

Again, remember that in C++, a **struct** declaration also creates a class, but its members are public by default:

```
struct name {
     declarations
};
```

A class declaration ends in a semicolon (;) following the closing brace (}). The semicolon is both required and essential, as it enables the C++ compiler to distinguish easily between class declarations and function definitions.

The *declarations* are a series of zero or more function and/or data declarations. Each such item—or "member"—has a name that becomes part of the class's namespace.

A class declaration can create one or more instances (objects), or it can terminate immediately. To declare objects at the end of a declaration, use this syntax:

```
class name {
     declarations
} obj_decl, obj_decl…;
```

Alternatively, after a class is declared, you can create objects in a separate statement by using the class name just as you would a primitive type name:

```
class_name  obj_decl, obj_decl…;
```

Each member *declaration* within the class follows the rules for a standard data or function declaration described in earlier chapters—although initialization is not