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!

Praise for Previous Editions of *C++ Without Fear*

"Dear Mr. Overland, I am in love with your *C++ Without Fear* book. It keeps me awake for hours during the night. This is a really, really good job, especially for beginners. I am a physics undergraduate student, and I needed to learn C since I am in the CERN CMS group—so I needed to do some data analyzing for some of the datas produced at CMS. I must say that thanks to you, I got most of the overall idea in just a few hours. I'm sure there are many, many others out there who are having fun with this as much as I do, while learning. Thank you very much with all my heart. Best wishes, Laura."

—Lara Vural, particle physics student, Istanbul, Turkey

"It's hard to tell where I began and ended with your book. I felt like I literally woke up and knew how to write C-style code. The book was written in a tone and pace, with its interludes and philosophical excursions, that was exactly what I needed. Even now when I see it on my bookshelf between Fante's *Ask the Dust* and *Tao Te Ching*, it just seems like my talisman, my passport, out of where I was to a new place. Again, I can't overstate the confidence that your book gave me. I knew instinctively that if I busted my tail over the next 18 months I'd be in a better place.

"And that's exactly what happened. After demonstrating my skill as a programmer and teammate I was offered a full-time job. I've never looked back. Again, Brian, thanks for the knowledge, enthusiasm, inventiveness, and humanity you injected into your book."

—Danny Grady, senior programming analyst at a Fortune 500 company

"*C++ Without Fear* breaks down writing software into logical steps, while always maintaining an easy-to-understand connection with the real world, as the reader goes from thinking about their solution, to pseudo code, to writing in C++. Other books should take note."

—Kevin Oke, awarding-winning co-founder of LlamaZOO Interactive, Victoria, BC, Canada

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:

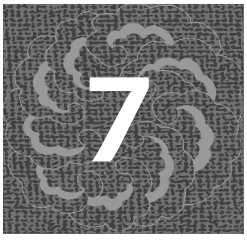
```
string band[ ] = {"John", "Paul", "George", "Ringo"};
```

- 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];
```



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).

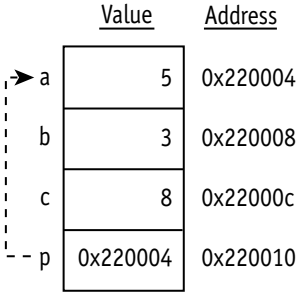
	Value	Address
a	5	0x220004
b	3	0x220008
c	8	0x22000c

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*