CHAPTER 6

Using Arrays in C++ Programs

After studying this chapter, you will be able to:

- Use arrays in C++ programs
- Search an array for a value
- Use parallel arrays in a C++ program

You should do the exercises and labs in this chapter after you have finished Chapter 6 of *Programming Logic and Design, Eighth Edition*, by Joyce Farrell. In this chapter, you learn how to use C++ to declare and initialize arrays. You then access the elements of an array to assign values and process them within your program. You also learn why it is important to stay within the bounds of an array. In addition, you study some programs written in C++ that implement the logic and design presented in *Programming Logic and Design.*

Array Basics

An **array** is a group of data items in which all items have the same data type, are referenced using the same variable name, and are stored in consecutive memory locations. To reference an individual element in an array, you use a subscript. Think of a **subscript** as the position number of a value within an array. It is important for you to know that in C++, subscript values begin with 0 (zero) and end with $n-1$, where n is the number of items stored in the array. You might be tempted to think that the first value in an array would be element number 1, but in fact it would be element number 0. The fifth element in an array would be element number 4.

To use an array in a C++ program, you must first learn how to declare an array, initialize an array with predetermined values, access array elements, and stay within the bounds of an array. In the next section you will focus on declaring arrays.

Declaring Arrays

Before you can use an array in a C++ program, you must first **declare** it. That is, you must give it a name and specify the data type for the data that will be stored in it. In some cases, you also specify the number of items that will be stored in the array. The following code shows how to declare two arrays, one named `cityPopulations` that will be used to store four `int`s, and another named `cities` that will be used to store four `string`s:

```
int cityPopulations[4];
string cities[4];
```

As shown, you begin by specifying the data type of the items that will be stored in the array. The data type is followed by the name of the array and then a pair of square brackets. Within the square brackets, you see an integer value that specifies the number of elements this array can hold.

As shown in Figure 6-1, the compiler allocates enough consecutive memory locations to store four elements of data type `int` for the array named `cityPopulations`. If `cityPopulations[0]` is stored at memory address 1000, then the address of `cityPopulations[1]` is 1004 because each `int` requires 4 bytes of memory. Similarly, `cityPopulations[3]` is at address 1012.

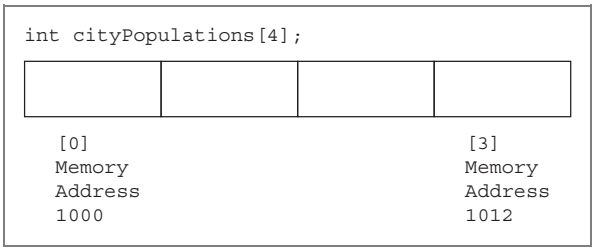

Figure 6-1 Memory allocation for the `cityPopulations` array

If an array is declared to store items of data type `double`, 8 bytes are allocated for each item in the array.

The `cityPopulations` array provides an example of how memory is allocated for arrays that contain primitive data types. Memory allocation is different for arrays of `string`s because a `string` is an object in C++, not a primitive data type. Using the Visual C++ compiler, 28 bytes of memory are allocated for a `string` object. This represents the size of the object.

The amount of memory allocated for a `string` object may be different on your system.

Additional memory for the characters that make up the `string` is allocated **dynamically** (as your program runs) when a value is stored in the `string` object. This is shown in Figure 6-2.

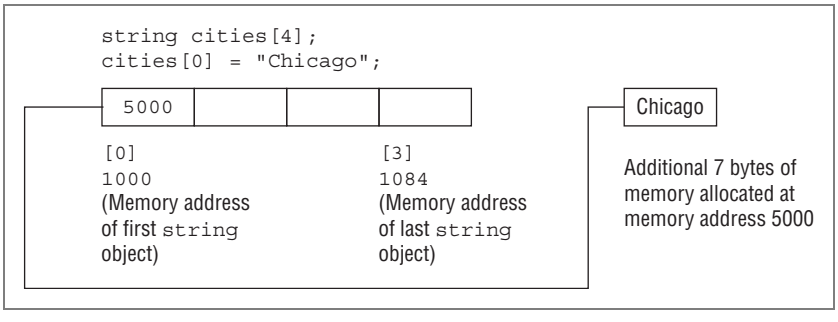

Figure 6-2 Memory allocation for the `cities` array

In Figure 6-2, the compiler allocates enough consecutive memory locations to store four `string` objects for the array named `cities`. If the address of `cities[0]` is 1000, the address of `cities[1]` is 1028, and the address of `cities[3]` is 1084 because each reference requires 28

bytes of memory. When a string is assigned to `cities[0]`, memory is dynamically allocated at another memory address. This address is then stored in the array. When the statement `cities[0] = "Chicago";` executes, the memory dynamically allocated for Chicago begins at address 5000, and then that address (5000) is stored in the first element of the array. An example of creating `string` objects is presented later in this chapter.

Dynamic memory allocation is an advanced topic that you will learn more about when you take additional courses in C++ programming.

Initializing Arrays

In C++, array elements are not automatically initialized to any value when the array is declared. Therefore, all of the elements of an array contain **garbage values**, which means they contain the values last stored at the memory location assigned to your array. Because these values are not useful in your program, it is a good idea to initialize arrays to all zeros for arrays that store numbers and to empty strings for arrays that store strings. Two double quotes with no space between, "", is the empty string in C++.

If you initialize an array when you declare it, you do not specify the array's size within the square brackets. Instead, you use an empty pair of square brackets.

You can and will sometimes want to initialize arrays with values that you choose. This can be done when you declare the array. To initialize an array when you declare it, use curly braces to surround a comma-delimited list of data items, as shown in the following example:

```
int cityPopulations[] = {9500000, 871100, 23900, 40100};
string cities[] = {"Chicago", "Detroit", "Batavia", "Lima"};
```

These initializations create four elements for each array and assign values to them.

You can also use assignment statements to provide values for array elements after an array is declared, as in the following example:

```
cityPopulations[0] = 9500000;
cities[0] = "Chicago";
```

A loop is often used to assign values to the elements in an array, as shown here:

```
int loopIndex;
for(loopIndex = 0; loopIndex < 3; loopIndex++)
{
    cityPopulations[loopIndex] = 12345;
    cities[loopIndex] = "AnyCity";
}
```

The first time this loop is encountered, `loopIndex` is assigned the value 0. Because 0 is less than 3, the body of the loop executes, assigning the value 12345 to `cityPopulations[0]` and the value `"AnyCity"` to `cities[0]`. Next, the value of `loopIndex` is incremented and takes on the value 1. Because 1 is less than 3, the loop executes a second time and the value 12345 is assigned to `cityPopulations[1]`, and `"AnyCity"` is assigned to `cities[1]`. Each time the loop executes, the value of `loopIndex` is incremented. This allows you to access a different location in the array each time the body of the loop executes.

Accessing Array Elements

You need to access individual locations in an array when you assign a value to an array element, print its value, change its value, assign the value to another variable, and so forth. In C++, you use an integer expression placed in square brackets to indicate which element in the array you want to access. This integer expression is the subscript.

Remember that subscript values begin with 0 (zero) in C++.

The following C++ program declares two arrays of data type `double`, initializes an array of data type `double`, copies values from one array to another, changes several values stored in the array named `target`, and prints the values of the arrays named `source` and `target`. You can compile and execute this program if you like. It is stored in the file named `ArrayTest.cpp`.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    double target[3];
    double source[] = {1.0, 5.5, 7.9};
    int loopIndex;
    // Copy values from source to target
    for(loopIndex = 0; loopIndex < 3; loopIndex++)
       target[loopIndex] = source[loopIndex];
    // Assign values to two elements of target
    target[0] = 2.0;
    target[1] = 4.5;
    // Print values stored in source and target
    for(loopIndex = 0; loopIndex < 3; loopIndex++)
    {
       cout << "Source " << source[loopIndex] << endl;
       cout << "Target " << target[loopIndex] << endl;
    }
}
```

Staying Within the Bounds of an Array

As a C++ programmer, you must be careful to ensure the subscript values you use to access array elements are within the legal bounds. Unlike most other programming languages, the C++ compiler does *not* check to make sure that a subscript used in your program is greater than or equal to 0 and less than the length of the array. For example, suppose you declare an array named `numbers` as follows:

```
int numbers[10];
```

In this case, the C++ compiler does not check to make sure the subscripts you use to access this array are integer values between 0 and 9.

When using a loop to access the elements in an array, be sure that the test you use to terminate the loop keeps you within the legal bounds, 0 to *n*–1, where *n* is the number of items stored in the array.

However, when you execute your program, if you use an array subscript that is not in the legal bounds, a garbage value is accessed. As an example, consider the highlighted operator in the following code, which is taken from the previous C++ program example:

```
double source[] = {1.0, 5.5, 7.9};
int loopIndex;
for(loopIndex = 0; loopIndex < 3; loopIndex++)
```

If you change the highlighted operator to <=, as shown below, your program will compile with no errors:

```
for(loopIndex = 0; loopIndex <= 3; loopIndex++)
```

A problem arises, however, when your program runs because the loop will execute when the value of `loopIndex` is 3. When you access the array element `source[3]`, you are outside the bounds of the array because there is no such element in this array, and a garbage value is stored at this location in the array.

Using Constants with Arrays

It is a good programming practice to use a named constant to help you stay within the bounds of an array when you write programs that declare and access arrays. The following example shows how to use a named constant:

It is a C++ convention to use all capital letters when naming constants.

```
const int NUM_ITEMS = 3;
double target[NUM_ITEMS];
int loopIndex;
for(loopIndex = 0; loopIndex < NUM_ITEMS; loopIndex++)
        target[loopIndex] = loopIndex + 10;
```

Exercise 6-1: Array Basics

Use what you have learned about declaring and initializing arrays to complete the following:

1. Write array declarations for each of the following:
   a. Six grade point averages.

   b. Seven last names.

   c. 10 ages.

2. Declare and initialize arrays that store the following:
   a. The whole numbers 2, 4, 6, 8, 10

   b. The last names Carlson, Matthews, and Cooper

   c. The prices 15.00, 122.00, and 7.50

3. Write an assignment statement that assigns the value 32 to the first element of the array of integers named `customerNumber`.

4. An array of `int`s named `numbers` is stored at memory location 4000. Where is `numbers[1]`? Where is `numbers[4]`?

Lab 6-1: Array Basics

In this lab, you complete a partially prewritten C++ program that uses an array. The program prompts the user to interactively enter eight batting averages, which the program stores in an array. The program should then find the minimum and maximum batting average stored in the array as well as the average of the eight batting averages. The data file provided for this lab includes the input statement and some variable declarations. Comments are included in the file to help you write the remainder of the program.

1. Open the source code file named `BattingAverage.cpp` using Notepad or the text editor of your choice.
2. Write the C++ statements as indicated by the comments.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `BattingAverage.cpp`.

5. Execute the program. Enter the following batting averages: **.299**, **.157**, **.242**, **.203**, **.198**, **.333**, **.270**, **.190**. The minimum batting average should be .157, and the maximum batting average should be .333. The average should be .2365.

Searching an Array for an Exact Match

One of the programs described in *Programming Logic and Design* uses an array to hold valid item numbers for a mail-order business. The idea is that when a customer orders an item, you can determine if the customer ordered a valid item number by searching through the array for that item number. This program relies on a technique called setting a flag to verify that an item exists in an array. The pseudocode and the C++ code for this program are shown in Figure 6-3.

```
start
   Declarations
      num item
      num SIZE = 6
      num VALID_ITEMS[SIZE] = 106, 108, 307,
         405, 457, 688
      num sub
      string foundIt
      num badItemCount = 0
      string MSG_YES = "Item available"
      string MSG_NO = "Item not found"
      num FINISH = 999
   getReady()
   while item <> FINISH
      findItem()
   endwhile
   finishUp()
stop

getReady()
   output "Enter item number or ", FINISH, " to quit"
   input item
return

findItem()
   foundIt = "N"
   sub = 0
   while sub < SIZE
       if item = VALID_ITEMS[sub] then
          foundIt = "Y"
      endif
      sub = sub + 1
   endwhile
   if foundIt = "Y" then
      output MSG_YES
   else
      output MSG_NO
      badItemCount = badItemCount + 1
   endif
   output "Enter next item number or ", FINISH, " to quit"
   input item
return

finishUp()
   output badItemCount, " items had invalid numbers"
return
```

Figure 6-3 Pseudocode and C++ code for the Mail Order program *(continues)*

(continued)

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

int main()
{
   // Declare variables
   int item, badItemCount = 0;
   const int SIZE = 6;
   int VALID_ITEMS[] = {106, 108, 307, 405, 457, 688};
   int sub;
   bool foundIt = false;
   const string MSG_YES = "Item Available";
   const string MSG_NO = "Item not found";
   const int FINISH = 999;

   // This is the work done in the getReady() function
   // Get user input
   cout << "Enter item number or " << FINISH << " to quit ";
   cin >> item;

   // Loop through array
   while(item != FINISH)
   {
      // This is the work done in the findItem() function
      foundIt = false;
      sub = 0;
      while(sub < SIZE)
      {
         // Test to see if this item is a valid item
         if(item == VALID_ITEMS[sub])
            foundIt = true;   // Set flag to true
         sub += 1;            // Increment loop index
      }
      // Test value of foundIt
      if(foundIt == true)
         cout << MSG_YES << endl;
      else
      {
         cout << MSG_NO << endl;
         badItemCount++;
      }
      cout << "Enter item number or " << FINISH << " to quit ";
      cin >> item;
   }
   // This is the work done in the finishUp() function
   cout << badItemCount << " items had invalid numbers." << endl;
   return 0;
}
```

Figure 6-3 Pseudocode and C++ code for the Mail Order program

Notice that the equality operator (==) is used when comparing the `int` value in the first `if` statement and the `bool` value in the second `if` statement.

The program can be found in the file named `MailOrder.cpp`. You may want to compile and execute the program to see how it operates.

As shown in Figure 6-3, when you translate the pseudocode to C++, you make a few changes. In both the pseudocode and the C++ code, the variable named `foundIt` is the flag. However,

in the C++ code, you assign the value `false` instead of the string constant `"N"` to the variable named `foundIt`. This is because the variable named `foundIt` is declared as a variable of the `bool` type. The `bool` data type is one of the primitive data types found in C++ and is only used to store `true` and `false` values.

In Figure 6-3, the array declaration and the lines of code that set the flag are highlighted.

The variable named `foundIt` could be declared as a `string` as it was in the pseudocode. It is better to use a `bool` in C++ for `true` and `false` values.

Exercise 6-2: Searching an Array for an Exact Match

In this exercise you use what you have learned about searching an array for an exact match. Study the following code, and then answer Questions 1–4. Note that this code may contain errors.

```
string apples[] = {"Gala", "Rome", "Fuji", "Delicious"};
int foundIt = false, i;
const int MAX_APPLES = 4;
string inApple;
cout << "Enter apple type:";
cin >> inApple;
for(i = 0; i <= MAX_APPLES; i++)
{
    if(inApple == apples[i])
    {
       foundIt = true;
    }
}
```

1. Is the `for` loop written correctly?

   _____

   If not, how can you fix it?

   _____

2. Which variable is the flag?

   _____

3. Is the flag variable declared correctly?

   _____

   If not, what should you do to fix it?

   _____

4. Is the comparison in the `if` statement done correctly?

   _____

   If not, how can you fix it?

   _____

Lab 6-2: Searching an Array for an Exact Match

In this lab, you use what you have learned about searching an array to find an exact match to complete a partially prewritten C++ program. The program uses an array that contains valid names for 10 cities in Michigan. You ask the user to enter a city name; your program then searches the array for that city name. If it is not found, the program should print a message that informs the user the city name is not found in the list of valid cities in Michigan.

The data file provided for this lab includes the input statements and the necessary variable declarations. You need to use a loop to examine all the items in the array and test for a match. You also need to set a flag if there is a match and then test the flag variable to determine if you should print the `"Not a city in Michigan."` message. Comments in the code tell you where to write your statements. You can use the Mail Order program in this chapter as a guide.

1. Open the source code file named `MichiganCities.cpp` using Notepad or the text editor of your choice.
2. Study the prewritten code to make sure you understand it.
3. Write a loop statement that examines the names of cities stored in the array.
4. Write code that tests for a match.
5. Write code that, when appropriate, prints the message `"Not a city in Michigan."`.
6. Save this source code file in a directory of your choice, and then make that directory your working directory.
7. Compile the source code file `MichiganCities.cpp`.
8. Execute the program using the following as input:

```
Chicago
Brooklyn
Watervliet
Acme
```

Parallel Arrays

As you learned in *Programming Logic and Design*, you use parallel arrays to store values and to maintain a relationship between the items stored in the arrays. Figure 6-4 shows that the student ID number stored in `stuID[0]` and the grade stored in `grades[0]` are related—student 56 received a grade of 99.

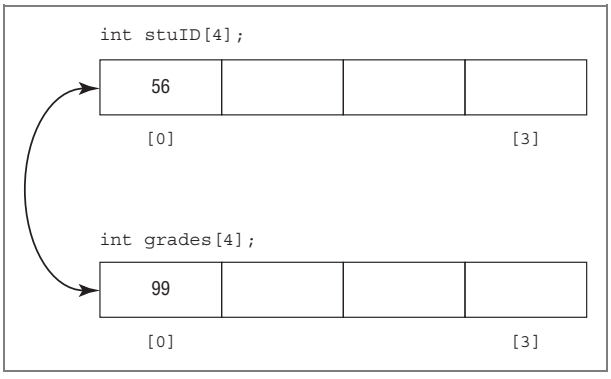

Figure 6-4 Parallel arrays

This relationship is established by using the same subscript value when accessing each array. Note that as the programmer, you must maintain this relationship in your code by always using the same subscript. C++ does not create or maintain the relationship.

Parallel arrays have the same number of elements, but they do not necessarily store items of the same data type.

One of the programs discussed in *Programming Logic and Design* is an expanded version of the Mail Order program discussed in the "Searching an Array for an Exact Match" section earlier in this chapter. In this expanded program, you need to determine the price of the ordered item and print the item number along with the price. You use parallel arrays to help you organize the data for the program. One array, `VALID_ITEMS`, contains six valid item numbers. The other array, `VALID_PRICES`, contains six valid prices. Each price is in the same position as the corresponding item number in the other array. When a customer orders an item, you search the `VALID_ITEMS` array for the customer's item number. When the item number is found, you use the price stored in the same location of the `VALID_PRICES` array, and then output the item number and the price. The complete C++ program is stored in the file named `MailOrder2.cpp`. The pseudocode and C++ code that search the `VALID_ITEMS` array, use a price from the `VALID_PRICES` array, and then print the ordered item and its price are shown in Figure 6-5.

In Figure 6-5, the declaration for the second array and the lines of code that use the variable named `price` are highlighted.

```
start
   Declarations
      num item
      num price
      num SIZE = 6
      num VALID_ITEMS[SIZE] = 106, 108, 307,
         405, 457, 688
      num VALID_PRICES[SIZE] = 0.59, 0.99,
         4.50, 15.99, 17.50, 39.00
      num sub
      string foundIt
      num badItemCount = 0
      string MSG_YES = "Item available"
      string MSG_NO = "Item not found"
      num FINISH = 999
   getReady()
   while item <> FINISH
      findItem()
   endwhile
   finishUp()
stop

getReady()
   output "Enter item number or ", FINISH, " to quit"
   input item
return

findItem()
   foundIt = "N"
   sub = 0
   while sub < SIZE
      if item = VALID_ITEMS[sub] then
         foundIt = "Y"
         price = VALID_PRICES[sub]
      endif
      sub = sub + 1
   endwhile
   if foundIt = "Y" then
      output MSG_YES
      output "The price of ", item, " is ", price
   else
      output MSG_NO
      badItemCount = badItemCount + 1
   endif
   output "Enter next item number or ", FINISH, " to quit"
   input item
return

finishUp()
   output badItemCount, " items had invalid numbers"
return
```

Figure 6-5 Pseudocode and C++ code for the Mail Order2 program *(continues)*

(continued)


```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      int item, badItemCount = 0;
      double price;
      const int SIZE = 6;
      int VALID_ITEMS[] = {106, 108, 307, 405, 457, 688};
      double VALID_PRICES[] = {0.59, 0.99, 4.50, 15.99, 17.50, 39.00};
      int sub;
      bool foundIt = false;
      const string MSG_YES = "Item Available";
      const string MSG_NO = "Item not found";
      const int FINISH = 999;

      // This is the work done in the getReady() function
      cout << "Enter next item number or " << FINISH << " to quit ");
      cin >> item;

      while(item != FINISH)
      {
         // This is the work done in the findItem() function
         foundIt = false;
         sub = 0;
         while(sub < SIZE)
         {
            if(item == VALID_ITEMS[sub])
            {
               foundIt = true;
               price = VALID_PRICES[sub];
            }
            sub++;
         }
         if(foundIt == true)
         {
            cout << MSG_YES << endl;
            cout << "The price of " << item <<
                    " is " << price << endl;
         }
         else
         {
            cout << MSG_NO << endl;
            badItemCount++;
         }
         cout << "Enter next item number or " << FINISH << " to quit ";
         cin >> item;
      }
      // This is the work done in the finishUp() function
      cout << badItemCount << " items had invalid numbers" << endl;
      return 0;
}
```

Figure 6-5 Pseudocode and C++ code for the Mail Order2 program

Exercise 6-3: Parallel Arrays

In this exercise, you use what you have learned about parallel arrays. Study the following code, and then answer Questions 1–4. Note that this code may contain errors.

```
string cities[] = "Chicago", "Rome", "Paris", "London";
int populations[] = 2695598, 2760665, 2190777, 7805417;
const int MAX_CITIES = 4;
int foundIt;
int i, x;
```

```
string inCity;
cout << "Enter city name: ";
cin >> inCity;
for(i = 0; i = MAX_CITIES; ++i)
{
    if(inCity == cities[i])
    {
       foundIt = i;
    }
}
cout << "Population for " << cities[foundIt] <<
        " is " << populations[foundIt] << endl;
```

1. Are the arrays declared and initialized correctly?

   _____

   If not, how can you fix them?

   _____

2. Is the `for` loop written correctly?

   _____

   If not, how can you fix it?

   _____

3. As written, how many times will the `for` loop execute?

   _____

4. How would you describe the purpose of the statement `foundIt = i;`?

   _____

Lab 6-3: Parallel Arrays

In this lab, you use what you have learned about parallel arrays to complete a partially completed C++ program. The program is described in Chapter 6, Exercise 7, in *Programming Logic and Design*. The program should either print the name and price for a coffee add-in from the Jumpin' Jive Coffee Shop or it should print the message `"Sorry, we do not carry that."`.

Read the problem description carefully before you begin. The data file provided for this lab includes the necessary variable declarations and input statements. You need to write the part of the program that searches for the name of the coffee add-in(s) and either prints the name and price of the add-in or prints the error message if the add-in is not found. Comments in the code tell you where to write your statements. You can use the expanded Mail Order2 program shown in Figure 6-5 as a guide.

1. Open the source code file named `JumpinJive.cpp` using Notepad or the text editor of your choice.
2. Study the prewritten code to make sure you understand it.
3. Write the code that searches the array for the name of the add-in ordered by the customer.

4. Write the code that prints the name and price of the add-in or the error message, and then write the code that prints the cost of the total order.
5. Save this source code file in a directory of your choice, and then make that directory your working directory.
6. Compile the source code file `JumpinJive.cpp`.
7. Execute the program using the following data and record the output:

```
Cream
Caramel
Whiskey
chocolate
Chocolate
Cinnamon
Vanilla
```

Remember that C++ is case sensitive, which means it distinguishes between uppercase letters and lowercase letters. This means, for example, that *cinnamon* is not the same as *Cinnamon*.