CHAPTER 8

Advanced Array Techniques

After studying this chapter, you will be able to:

- Explain the need to sort data
- Swap data values in a program
- Create a bubble sort in C++
- Work with multidimensional arrays

In this chapter, you review why you would want to sort data, how to use C++ to swap two data values in a program, how to create a bubble sort in a C++ program, and how to use multidimensional arrays. You should do the exercises and labs in this chapter after you have finished Chapter 8 in *Programming Logic and Design, Eighth Edition.*

Sorting Data

Data records are always stored in some order, but possibly they are not in the order in which you want to process or view them in your program. When this is the case, you need to give your program the ability to arrange (sort) records in a useful order. For example, the inventory records you need to process might be stored in product-number order, but you might need to produce a report that lists products from lowest cost to highest cost. That means your program needs to sort the records by cost.

Sorting makes searching for records easier and more efficient. A human can usually find what he or she is searching for by simply glancing through a group of data items, but a program must look through a group of data items one by one, making a decision about each one. When searching unsorted records for a particular data value, a program must examine every single record until it either locates the data value or determines that it does not exist. However, when searching sorted records, the program can quickly determine when to stop searching, as shown in the following step-by-step scenario:

1. The records used by your program are sorted by product number.
2. The user is searching for product number 12367.
3. The program locates the record for product number 12368 but has not yet found product number 12367.
4. The program determines that the record for product number 12367 does not exist, and therefore stops searching through the list.

Many search algorithms require that data be sorted before it can be searched. (An **algorithm** is a plan for solving a problem.) You can choose from many algorithms for sorting and searching for data. In *Programming Logic and Design,* you learned how to swap data values in an array, and you also learned about the bubble sort. Both of these topics are covered in this book.

Swapping Data Values

When you swap values, you place the value stored in one variable into a second variable, and then you place the value that was originally stored in the second variable in the first variable. You must also create a third variable to temporarily hold one of the values you want to swap so a value is not lost. For example, if you try to swap values using the following code, you will lose the value of `score2`:

```
int score1 = 90;
int score2 = 85;
```

```
score2 = score1; // The value of score2 is now 90
score1 = score2; // The value of score1 is also 90
```

However, if you use a variable to temporarily hold one of the values, the swap is successful. This is shown in the following code:

```
int score1 = 90;
int score2 = 85;
int temp;
temp = score2; // The value of temp is 85
score2 = score1; // The value of score2 is 90
score1 = temp; // The value of score1 is 85
```

Exercise 8-1: Swapping Values

In this exercise, you use what you have learned about swapping values to answer the following question.

1. Suppose you have declared and initialized two `string` variables, `product1` and `product2`, in a C++ program. Now, you want to swap the values stored in `product1` and `product2` but only if the value of `product1` is greater than the value of `product2`. Write the C++ code that accomplishes this task. The declarations are as follows:

   ```
   string product1 = "hammer";
   string product2 = "wrench";
   ```

   ____

   ____

   ____

   ____

Lab 8-1: Swapping Values

In this lab, you complete a C++ program that swaps values stored in three `int` variables and determines maximum and minimum values. The C++ file provided for this lab contains the necessary variable declarations, as well as the input and output statements. You want to end up with the smallest value stored in the variable named `first` and the largest value stored in the variable named `third`. You need to write the statements that compare the values and swap them if appropriate. Comments included in the code tell you where to write your statements.

1. Open the source code file named `Swap.cpp` using the text editor of your choice.
2. Write the statements that test the first two integers, and swap them if necessary.
3. Write the statements that test the second and third integer, and swap them if necessary.
4. Write the statements that test the first and second integers again, and swap them if necessary.

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 `Swap.cpp`.
7. Execute the program using the following sets of input values, and record the output:

```
101     22    -23
630   1500      9
 21      2      2
```

Using a Bubble Sort

A bubble sort is one of the easiest sorting techniques to understand. However, while it is logically simple, it is not very efficient. If the list contains *n* values, the bubble sort will make *n – 1* passes over the list. For example, if the list contains 100 values, the bubble sort will make 99 passes over the data. During each pass, it examines successive overlapped pairs and swaps or exchanges those values that are out of order. After one pass over the data, the heaviest (largest) value sinks to the bottom and is in the correct position in the list.

In *Programming Logic and Design,* you learned several ways to refine the bubble sort. One way is to reduce unnecessary comparisons by ignoring the last value in the list in the second pass through the data, because you can be sure that it is already positioned correctly. On the third pass, you can ignore the last two values in the list because you know they are already positioned correctly. Thus, in each pass, you can reduce the number of items to be compared, and possibly swapped, by one.

Another refinement to the bubble sort is to eliminate unnecessary passes over the data in the list. When items in the array to be sorted are not entirely out of order, it may not be necessary to make *n – 1* passes over the data because after several passes, the items may already be in order. You can add a flag variable to the bubble sort, and then test the value of that flag variable to determine whether any swaps have been made in any single pass over the data. If no swaps have been made, you know that the list is in order; therefore, you do not need to continue with additional passes.

You also learned about using a constant for the size of the array to make your logic easier to understand and your programs easier to change and maintain. Finally, you have learned how to sort a list of varying size by counting the number of items placed in the array as you read in items.

All of these refinements are included in the pseudocode for the Score Sorting program in Figure 8-1. The C++ code that implements the Score Sorting logic is provided in Figure 8-2. The line numbers shown in Figure 8-2 are not part of the C++ code. They are provided for reference only.

```
start
   num SIZE = 100
   num scores[SIZE]
   num x
   num y
   num temp
   num numberOfEls = 0
   num comparisons
   num QUIT = 999
   String didSwap
   fillArray()
   sortArray()
   displayArray()
stop

fillArray()
   x = 0
   output "Enter a score or ", QUIT, " to quit "
   input scores[x]
   x = x + 1
   while x < SIZE AND scores[x] <> QUIT
      output "Enter a score or ", QUIT, " to quit "
      input scores[x]
      x = x + 1
   endwhile
   numberOfEls = x
   comparisons = numberOfEls - 1
return

void sortArray()
   x = 0
   didSwap = "Yes"
   while didSwap = "Yes"
      x = 0
      didSwap = "No"
      while x < comparisons
         if scores[x] > scores[x + 1] then
            swap()
            didSwap = "Yes"
         endif
         x = x + 1
      endwhile
   endwhile
return
```

Figure 8-1 Pseudocode for the Score Sorting program *(continues)*

(continued)

```
void swap()
   temp = scores[x + 1]
   scores[x + 1] = scores[x]
   scores[x] = temp
return

void displayArray()
   x = 0
   while x < numberOfEls
      output scores[x]
      x = x + 1
   endwhile
return
```

Figure 8-1 Pseudocode for the Score Sorting program

```cpp
// StudentScores.cpp - This program interactively reads a
// variable number of student test scores, stores the
// scores in an array, and then sorts the scores in
// ascending order.
// Input:  Interactive
// Output:  Sorted list of student scores.

#include <iostream>
using namespace std;

int main()
{
   // Declare variables.
   // Maximum size of array
   const int SIZE = 100;
   // Array of student scores.
   int scores[SIZE];
   int x;
   int temp;
   // Actual number of elements in array.
   int numberOfEls = 0;
   int comparisons;
   const int QUIT = 999;
   bool didSwap;

   // Work done in the fillArray function
   x = 0;
   cout << "Enter a score or " << QUIT << " to quit ";
   cin >> scores[x];
   x++;
```

Figure 8-2 C++ code for the Score Sorting program *(continues)*

(continued)

```
31       while(x < SIZE && scores[x-1] != QUIT)
32       {
33          cout << "Enter a score or " << QUIT << " to quit ";
34          cin >> scores[x];
35          x++;
36       }  // End of input loop.
37       numberOfEls = x-1;
38       comparisons = numberOfEls -1;
39
40       // Work done in the sortArray() function
41       didSwap = true;  // Set flag to true.
42       // Outer loop controls number of passes over data.
43       while(didSwap == true) // Test flag.
44       {
45          x = 0;
46          didSwap = false;
47          // Inner loop controls number of items to compare.
48          while(x < comparisons)
49          {
50             if(scores[x] > scores[x+1]) // Swap?
51             {
52                // Work done in the swap() function
53                temp = scores[x + 1];
54                scores[x+1] = scores[x];
55                scores[x] = temp;
56                didSwap = true;
57             }
58             x++;   // Get ready for next pair.
59          }
60          comparisons--;
61       }
62
63
64       // Work done in the displayArray() function
65       x = 0;
66       while(x < numberOfEls)
67       {
68          cout << scores[x] << endl;
69          x++;
70       }
71       return 0;
72    } // End of main() function.
```

Figure 8-2 C++ code for the Score Sorting program

The `main()` Function

As shown in Figure 8-2, the `main()` function (line 11) declares variables and performs the work of the program. The variables include the following:

- A constant named `SIZE`, initialized with the value 100, which represents the maximum number of items this program can sort
- An array of data type `int` named `scores` that is used to store up to a maximum of `SIZE` (100) items to be sorted

- An `int` variable named `x` that is used as the array subscript
- an `int` variable named `temp` that is used to swap the values stored in the array
- An `int` named `numberOfEls` that is used to hold the actual number of items stored in the array
- An `int` named `comparisons` that is used to control the number of comparisons that should be done
- An `int` constant named `QUIT`, initialized to 999, that is used to control the `while` loop
- A `bool` named `didSwap` that is used as a flag to indicate when a swap has taken place

After these variables are declared, the work done in the `fillArray()`function begins on line 27. The `fillArray()` work is responsible for filling up the array with items to be sorted. On line 41, the work done in the `sortArray()` function begins. This work is responsible for sorting the items stored in the `scores` array. Lastly, the work done in the `displayArray()` function begins on line 65 and is responsible for displaying the sorted scores on the user's screen.

The `fillArray()` Function

The work done in the `fillArray()` function, which begins on line 27 in Figure 8-2, is responsible for (1) storing the data in the array, and (2) counting the actual number of elements placed in the array. The `fillArray()` function assigns the value 0 to the variable named `x` and then performs a priming read (lines 28 and 29) to retrieve the first student score from the user and to also store the score in the array named `scores` at location `x` on line 29. Notice the array subscript variable `x` is initialized to 0 on line 27 because the first position in an array is position 0. Also, notice the variable named `x` is incremented on line 30 because it is used to count the number of scores entered by the user of the program.

On line 31, the condition that controls the `while` loop is tested. The `while` loop executes as long as the number of scores input by the user (represented by the variable named `x`) is less than `SIZE(100)` and as long as the user has not entered `999` (the value of the constant `QUIT`) for the student score. If `x` is less than `SIZE` and the user does not want to quit, there is enough room in the array to store the student score. In that case, the program retrieves the next student score, and stores the score in the array named `scores` at location `x` on line 34. The program then increments the value of `x` (line 35) to get ready to store the next student score in the array. The loop continues to execute until the user enters the value `999` or until there is no more room in the array.

When the program exits the loop, the value of `x-1` is assigned to the variable named `numberOfEls` on line 37. Notice that `x` is used as the array subscript and that its value is incremented every time the `while` loop executes, including when the user enters the value `999` in order to quit; therefore, `x` represents the number of student scores the user entered *plus one*. On line 38 the value of `numberOfEls -1` is assigned to the variable named `comparisons` and represents the maximum number of elements the bubble sort will compare on a pass over the data stored in the array. It ensures that the program does not attempt to compare item `x` with item `x+1`, when `x` is the last item in the array.

The `sortArray()` Function

The work done in the `sortArray()` function begins on line 41 and uses a refined bubble sort to rearrange the student scores in the array named `scores` to be in ascending order. Refer to Figure 8-1, which includes the pseudocode, and Figure 8-2, which includes the C++ code that implements the `sortArray()` function.

Line 41 initializes the flag variable `didSwap` to `true`, because, at this point in the program, it is assumed that items will need to be swapped.

The outer loop (line 43), `while(didSwap == true)`, controls the number of passes over the data. This logic implements one of the refinements discussed earlier—eliminating unnecessary passes over the data. As long as `didSwap` is `true`, the program knows that swaps have been made and that, therefore, the data is still out of order. Thus, when `didSwap` is `true`, the program enters the loop. The first statement in the body of the loop (line 45) is `x = 0;`. The program assigns the value 0 to `x` because `x` is used as the array subscript. Recall that in C++, the first subscript in an array is number 0.

Next, to prepare for comparing the elements in the array, line 46 assigns the value `false` to `didSwap`. This is necessary because the program has not yet swapped any values in the array on this pass. The inner loop begins on line 48. The test, `x < comparisons`, controls the number of pairs of values in the array the program compares on one pass over the data. This implements another of the refinements discussed earlier—reducing unnecessary comparisons. The last statement in the outer loop (line 60), `comparisons--;`, decrements the value of `comparisons` by 1 each time the outer loop executes. The program decrements `comparisons` because, when a complete pass is made over the data, it knows an item is positioned in the array correctly. Comparing the value of `comparisons` with the value of `x` in the inner loop reduces the number of necessary comparisons made when this loop executes.

On line 50, within the inner loop, adjacent items in the array are accessed and compared using the subscript variable `x` and `x+1`. The adjacent array items are compared to see if the program should swap them. If the values should be swapped, the program executes the statements that make up the work done in the `swap()` function on lines 53 through 55, which uses the technique discussed earlier to rearrange the two values in the array. Next, line 56 assigns `true` to the variable named `didSwap`. The last task performed by the inner loop (line 58) is adding 1 to the value of the subscript variable `x`. This ensures that the next time through the inner loop, the program will compare the next two adjacent items in the array. The program continues to compare two adjacent items and possibly swap them as long as the value of `x` is less than the value of `comparisons`.

The `displayArray()` Function

In the `displayArray()` function, you print the sorted array on the user's screen. Figure 8-1 shows the pseudocode for this function. The C++ code is shown in Figure 8-2.

The work done in the `displayArray()` function begins on line 65 of Figure 8-2. Line 65 assigns the value 0 to the subscript variable, `x`. This is done before the `while` loop is entered because the first item stored in the array is referenced using the subscript value 0. The loop in lines 66 through 70 prints all of the values in the array named `scores` by incrementing the value of the subscript variable, `x`, each time the loop body executes. When the loop exits, the statement `return 0;` (line 71) executes and ends the program.

Exercise 8-2: Using a Bubble Sort

In this exercise, you use what you have learned about sorting data using a bubble sort. Study the following C++ code, and then answer Questions 1–4.

```
int numbers[] = {432, -5, 54, -10, 36, 9, 65, 21, 24};
const int NUM_ITEMS = 9;
int j, k, temp;
int numPasses = 0, numCompares = 0, numSwaps = 0;
for(j = 0; j < NUM_ITEMS - 1; j++)
{
   numPasses++;
   for(k = 0; k < NUM_ITEMS - 1; k++)
   {
      numCompares++;
      if(numbers[k] > numbers[k+1])
      {
         numSwaps++;
         temp = numbers[k+1];
         numbers[k+1] = numbers[k];
         numbers[k] = temp;
      }
   }
}
```

1. Does this code perform an ascending sort or a descending sort? How do you know?

   ____

2. Exactly how many passes are made over the data in the array? Specify a number.

   ____

3. How many comparisons are made? Specify a number.

   ____

4. Do the variables named `numPasses`, `numCompares`, and `numSwaps` accurately keep track of the number of passes, compares, and swaps made in this bubble sort? Explain your answer.

   ____

Lab 8-2: Using a Bubble Sort

In this lab, you complete a C++ program that uses an array to store data for the village of Marengo. The program is described in Chapter 8, Exercise 5, in *Programming Logic and Design*. The program should allow the user to enter each household size and determine the mean and median household size in Marengo. The program should output the mean and median household size in Marengo. The file provided for this lab contains the necessary variable declarations and input statements. You need to write the code that sorts the household sizes in ascending order using a bubble sort and then prints the mean and median household size in Marengo. Comments in the code tell you where to write your statements.

1. Open the source code file named `HouseholdSize.cpp` using Notepad or the text editor of your choice.
2. Write the bubble sort.
3. Output the mean and median household size in Marengo.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file `HouseholdSize.cpp`.
6. Execute the program with the following input and record the output:
   Household sizes: `4, 1, 2, 4, 3, 3, 2, 2, 2, 4, 5, 6`

Using Multidimensional Arrays

As you learned in Chapter 8 of *Programming Logic and Design*, an array whose elements are accessed using a single subscript is called a **one-dimensional array** or a **single-dimensional array**. You also learned that a **two-dimensional array** stores elements in two dimensions and requires two subscripts to access elements.

In Chapter 8 of *Programming Logic and Design,* you saw how useful two-dimensional arrays can be when you studied the example of owning an apartment building with five floors with each floor having studio, one-bedroom, and two-bedroom apartments. The rent charged for these apartments depends on which floor the apartment is located as well as the number of bedrooms the apartment has. Table 8-1 shows the rental amounts.

Floor	Studio apartment	1-bedroom apartment	2-bedroom apartment
0	350	390	435
1	400	440	480
2	475	530	575
3	600	650	700
4	1000	1075	1150

Table 8-1 Rent schedule based on floor and number of bedrooms

In C++, declaring a two-dimensional array to store the rents shown in Table 8-1 requires two sets of square brackets. The first set of square brackets holds the number of rows in the array, and the second set of square brackets holds the number of columns. The declaration is shown below:

```
const int FLOORS = 5;
const int BEDROOMS = 3;
double rent[FLOORS][BEDROOMS];
```

The declaration shows the array's name, `rent`, followed by two sets of square brackets. The number of rows is included in the first set of square brackets using the constant value of `FLOORS(5)`, and the number of columns is included in the second set of square brackets using the constant value of `BEDROOMS(3)`.

As shown below, you can also initialize a two-dimensional array when you declare it by enclosing all of the values within a pair of curly braces and also enclosing the values (separated by commas) for each row within curly braces. Also, notice that each group of values within curly braces is separated by commas.

```
double rent[][] =  {{350, 390, 435},
                    {400, 440, 480},
                    {475, 530, 575},
                    {600, 650, 700},
                    {1000, 1075, 1150}};
```

To access individual elements in the `rent` array, two subscripts are required as shown below.

```
double myRent;
myRent = rent[3][1];
```

The first subscript (3) determines the row, and the second subscript (1) determines the column. In the assignment statement, `myRent = rent[3][1]`, the value 650 is assigned to the variable named `myRent`.

Remember that in C++, array subscripts begin with 0.

Figure 8-3 shows the pseudocode for a program that continuously displays rents for apartments based on renter requests for bedrooms and floor, and Figure 8-4 shows the C++ code that implements the program.

```
start
   Declarations
      num RENTS_BY_FLR_AND_BDRMS[5][3] = {350, 390, 435},
                                         {400, 440, 480},
                                         {475, 530, 575},
                                         {600, 650, 700},
                                         {1000, 1075, 1150}
      num floor
      num bedrooms
      num QUIT = 99
   getReady()
   while floor <> QUIT
      determineRent()
   endwhile
   finish()
stop
```

Figure 8-3 Pseudocode for a program that determines rent *(continues)*

(continued)

```
getReady()
   output "Enter floor "
   input floor
return

determineRent()
   output "Enter number of bedrooms "
   input bedrooms
   output "Rent is $",
           RENTS_BY_FLR_AND_BDRMS[floor][bedrooms]
   output "Enter floor "
   input floor
return

finish()
   output "End of program"
return
```

Figure 8-3 Pseudocode for a program that determines rent

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

int main()
{
   // Declare variables.
   const int FLOORS = 5;
   const int BEDROOMS = 3;
   double rent[FLOORS][BEDROOMS] = {{350, 390, 435},
                                    {400, 440, 480},
                                    {475, 530, 575},
                                    {600, 650, 700},
                                    {1000, 1075, 1150}};
   int floor;
   int bedroom;
   int QUIT = 99;

   // Work done in the getReady() function
   cout << "Enter floor or 99 to quit: ";
   cin >> floor;
   while(floor != QUIT)
   {
      // Work done in the determineRent() function
      cout << "Enter number of bedrooms: ";
      cin >> bedroom;
      cout << "Rent is $" << rent[floor][bedroom] << endl;
      cout << "Enter floor or 99 to quit: ";
      cin >> floor;
   }
   // Work done in the finish() function
   cout << "End of program" << endl;
   return 0;
} // End of main() function
```

Figure 8-4 C++ code for a program that determines rent

Exercise 8-3: Using Multidimensional Arrays

In this exercise, you use what you have learned about using multidimensional arrays to answer Questions 1–3.

1. A two-dimensional array declared as `int myNums[6][2];` has how many rows?

   ____

2. A two-dimensional array declared as `int myNums[6][2];` has how many columns?

   ____

3. Consider the following array declaration, `int myNums[6][2];`

   Are the following C++ statements legal?

```
number = myNums[5][3]; ________________
number = myNums[0][1]; ________________
number = myNums[1][2]; ________________
```

Lab 8-3: Using Multidimensional Arrays

In this lab, you will complete a C++ program that uses a two-dimensional array to store data for the Building Block Day Care Center. The program is described in Chapter 8, Exercise 10, in *Programming Logic and Design*. The day care center charges varying weekly rates depending on the age of the child and the number of days per week the child attends. The weekly rates are shown in Table 8-2.

	Days Per Week				
Age in Years	**1**	**2**	**3**	**4**	**5**
0	30.00	60.00	88.00	115.00	140.00
1	26.00	52.00	70.00	96.00	120.00
2	24.00	46.00	67.00	89.00	110.00
3	22.00	40.00	60.00	75.00	88.00
4 or more	20.00	35.00	50.00	66.00	84.00

Table 8-2 Weekly rates for Lab 8-3

The program should allow the user to enter the age of the child and the number of days per week the child will be at the day care center. The program should output the appropriate weekly rate. The file provided for this lab contains all of the necessary variable declarations, except the two-dimensional array. You need to write the input statements and the code that initializes the two-dimensional array, determines the weekly rate, and prints the weekly rate. Comments in the code tell you where to write your statements.

1. Open the source code file named `DayCare.cpp` using Notepad or the text editor of your choice.
2. Declare and initialize the two-dimensional array.
3. Write the C++ statements that retrieve the age of the child and the number of days the child will be at the day care center.
4. Determine and print the weekly rate.
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 `DayCare.cpp`.
7. Execute the program.