Appendix Q
Searching and Sorting Arrays

Topics

Focus on Software Engineering: *Introduction to Search Algorithms*

CONCEPT A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses two algorithms for searching the contents of an array.

It's very common for programs not only to store and process data stored in arrays, but to search arrays for specific items. This section will show you two methods of searching an array: the linear search and the binary search. Each has its advantages and disadvantages.

The Linear Search

The *linear search* is a very simple algorithm. Sometimes called a *sequential search*, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will unsuccessfully search to the end of the array.

Here is the pseudocode for a function that performs the linear search:

Set found to false.
Set position to -1.
Set index to 0.

While found is false and index < number of elements
If list[index] is equal to search value
found = true.
position = index.
End If
Add 1 to index.
End While.
Return position.

The function `searchList` shown below is an example of C++ code used to perform a linear search on an integer array. The array `list`, which has a maximum of `numElems` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, -1 is returned indicating the value did not appear in the array.

```
int searchList(int list[], int numElems, int value)
{
    int index = 0;        // Used as a subscript to search array
    int position = -1;    // To record position of search value
    bool found = false;   // Flag to indicate if the value was found

    while (index < numElems && !found)
    {
        if (list[index] == value)    // If the value is found
        {
            found = true;            // Set the flag
            position = index;        // Record the value's subscript
        }
        index++;                     // Go to the next element
    }
    return position;                 // Return the position, or -1
}
```

Note: The reason –1 is returned when the search value is not found in the array is because –1 is not a valid subscript.

Program Q-1 is a complete program that uses the `searchList` function. It searches the five-element array `tests` to find a score of 100.

Program Q-1

```
// This program demonstrates the searchList function, which
// performs a linear search on an integer array.
#include <iostream>
using namespace std;
```

(program continues)

Program Q-1 ***(continued)***

```
// Function prototype
int searchList(int [], int, int);

const int arrSize = 5;

int main()
{
   int tests[arrSize] = {87, 75, 98, 100, 82};
   int results;

   results = searchList(tests, arrSize, 100);
   if (results == -1)
      cout << "You did not earn 100 points on any test\n";
   else
   {
      cout << "You earned 100 points on test ";
      cout << (results + 1) << endl;
   }
   return 0;
}

//*****************************************************************
// The searchList function performs a linear search on an        *
// integer array. The array list, which has a maximum of numElems *
// elements, is searched for the number stored in value. If the   *
// number is found, its array subscript is returned. Otherwise,   *
// -1 is returned indicating the value was not in the array.      *
//*****************************************************************

int searchList(int list[], int numElems, int value)
{
   int index = 0;        // Used as a subscript to search array
   int position = -1;    // To record position of search value
   bool found = false;   // Flag to indicate if the value was found

   while (index < numElems && !found)
   {
      if (list[index] == value)    // If the value is found
      {
         found = true;             // Set the flag
         position = index;         // Record the value's subscript
      }
      index++;                     // Go to the next element
   }
   return position;                // Return the position, or -1
}
```

Program Output

```
You earned 100 points on test 4
```

Inefficiency of the Linear Search

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order to find a value stored in the last element (so that the algorithm actually reads an element of the array 20,000 times).

In an average case, an item is just as likely to be found near the beginning of the array as near the end. Typically, for an array of N items, the linear search will locate an item in N/2 attempts. If an array has 50,000 elements, the linear search will make a comparison with 25,000 of them in a typical case. This is assuming, of course, that the search item is consistently found in the array. (N/2 is the average number of comparisons. The maximum number of comparisons is always N.)

When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the linear search should not be used on large arrays if the speed is important.

The Binary Search

The *binary search* is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be sorted in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm immediately tests *its* middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found, or there are no more elements to test.

Here is the pseudocode for a function that performs a binary search on an array:

Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first]
    and array[last].
    If array[middle] equals the desired value
        Set found to true.

Set position to middle.
Else If array[middle] is greater than the desired value
Set last to middle - 1.
Else
Set first to middle + 1.
End If.
End While.
Return position.

This algorithm uses three index variables: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's first and last elements. The subscript of the element halfway between `first` and `last` is calculated and stored in the `middle` variable. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so that only the top or bottom half of the array is searched the during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The function `binarySearch` shown in the following example is used to perform a binary search on an integer array. The first parameter, `array`, which has a maximum of `numElems` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, -1 is returned indicating the value did not appear in the array.

```
int binarySearch(int array[], int numElems, int value)
{
   int first = 0,                     // First array element
       last = numElems - 1,           // Last array element
       middle,                        // Mid point of search
       position = -1;                 // Position of search value
   bool found = false;                // Flag

   while (!found && first <= last)
   {
       middle = (first + last) / 2;   // Calculate mid point
       if (array[middle] == value)    // If value is found at mid
       {
           found = true;
           position = middle;
       }
       else if (array[middle] > value) // If value is in lower half
           last = middle - 1;
       else
           first = middle + 1;        // If value is in upper half
   }
   return position;
}
```

Program Q-2 is a complete program using the `binarySearch` function. It searches an array of employee ID numbers for a specific value.

Program Q-2

```
// This program demonstrates the binarySearch function, which
// performs a binary search on an integer array.
#include <iostream>
using namespace std;

// Function prototype
int binarySearch(int [], int, int);

const int arrSize = 20;

int main()
{
   int tests[arrSize] = {101, 142, 147, 189, 199, 207, 222,
                         234, 289, 296, 310, 319, 388, 394,
                         417, 429, 447, 521, 536, 600};
   int results, empID;

   cout << "Enter the employee ID you wish to search for: ";
   cin >> empID;
   results = binarySearch(tests, arrSize, empID);
   if (results == -1)
      cout << "That number does not exist in the array.\n";
   else*
   {
      cout << "That ID is found at element " << results;
      cout << " in the array.\n";
   }
   return 0;
}

//****************************************************************
// The binarySearch function performs a binary search on an     *
// integer array. array, which has a maximum of numelems        *
// elements, is searched for the number stored in value. If the *
// number is found, its array subscript is returned. Otherwise, *
// -1 is returned indicating the value was not in the array.    *
//****************************************************************

int binarySearch(int array[], int numelems, int value)
{
   int first = 0,             // First array element
      last = numelems - 1,    // Last array element
      middle,                 // Mid point of search
      position = -1;          // Position of search value
   bool found = false;        // Flag
```

(program continues)

Program Q-2 ***(continued)***

```
    while (!found && first <= last)
    {
       middle = (first + last) / 2;     // Calculate mid point
       if (array[middle] == value)      // If value is found at mid
       {
          found = true;
          position = middle;
       }
       else if (array[middle] > value)  // If value is in lower half
          last = middle - 1;
       else
          first = middle + 1;           // If value is in upper half
    }
    return position;
}
```

Program Output with Example Input

```
Enter the employee ID you wish to search for: 199
That ID is found at element 4 in the array.
```

WARNING! Notice that the array in Program Q-2 is initialized with its values already sorted in ascending order. The binary search algorithm will not work properly unless the values in the array are sorted.

The Efficiency of the Binary Search

Obviously, the binary search is much more efficient than the linear search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 1,000 elements. If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 500. If the item is not found on the second attempt, the number of elements that remains to be searched is 250. This process continues until the binary search has either located the desired item or determined that it is not in the array. With 1,000 elelments this takes no more than 10 comparisons. (Compare this to the linear search, which would make an average of 500 comparisons!)

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. (A power of 2 is 2 raised to the power of some number.) Simply find the smallest power of 2 that is greater than or equal to the number of elements in the array. For example, a maximum of 16 comparisons will be made on an array of 50,000 elements (2^{16} = 65,536), and a maximum of 20 comparisons will be made on an array of 1,000,000 elements (2^{20} = 1,048,576).

Focus on Problem Solving and Program Design: *A Case Study*

The Demetris Leadership Center (DLC, Inc.) publishes the books, videos, and audio cassettes listed in Table Q-1.

Table Q-1

Product Title	Product Description	Product Number	Unit
Six Steps to Leadership	Book	914	$12.95
Six Steps to Leadership	Audio cassette	915	$14.95
The Road to Excellence	Video	916	$18.95
Seven Lessons of Quality	Book	917	$16.95
Seven Lessons of Quality	Audio cassette	918	$21.95
Seven Lessons of Quality	Video	919	$31.95
Teams Are Made, Not Born	Book	920	$14.95
Leadership for the Future	Book	921	$14.95
Leadership for the Future	Audio cassette	922	$16.95

The manager of the Telemarketing Group has asked you to write a program that will help order-entry operators look up product prices. The program should prompt the user to enter a product number, and will then display the title, description, and price of the product.

Variables

Table Q-2 lists the variables needed:

Modules

The program will consist of the functions listed in Table Q-3.

Table Q-2

Variable	Description
`numProds`	A constant integer initialized with the number of products the Demetris Leadership Center sells. This value will be used in the definition of the program's array.
titleSize	A constant integer initialized with the maximum size of a product's title.
descSize	A constant integer initialized with the maximum size of a product's description.
minProdNum	A constant integer initialized with the lowest product number.
maxProdNum	A constant integer initialized with the highest product number.
`id`	Array of integers. Holds each product's number.
`title`	Array of strings, initialized with the titles of products.
`description`	Array of strings, initialized with the descriptions of each product.
`prices`	Array of `float`s. Holds each product's price.

Table Q-3

Function	Description
`main`	The program's main function. It calls the program's other functions.
`getProdNum`	Prompts the user to enter a product number. The function validates input and rejects any value outside the range of correct product numbers.
`binarySearch`	A standard binary search routine. Searches an array for a specified value. If the value is found, its subscript is returned. If the value is not found, –1 is returned.
`displayProd`	Uses a common subscript into the title, description, and prices arrays to display the title, description, and price of a product

Function `main`

Function `main` contains the variable definitions and calls the other functions. Here is its pseudocode:

do
Call getProdNum.
Call binarySearch.
If binarySearch returned -1
Inform the user that the product number was not found.
else
Call displayProd.
End If.
Ask the user if the program should repeat.
While the user wants to repeat the program.

Here is its actual C++ code.

```
int main ()
{
    int id[numProds] = {914, 915, 916, 917, 918, 919, 920,
                        921, 922};

    char title[numProds][titleSize] =
                  {   "Six Steps to Leadership",
                      "Six Steps to Leadership",
                      "The Road to Excellence",
                      "Seven Lessons of Quality",
                      "Seven Lessons of Quality",
                      "Seven Lessons of Quality",
                      "Teams Are Made, Not Born",
                      "Leadership for the Future",
                      "Leadership for the Future"
                  };

    char description[numProds][descSize] =
                  {   "Book", "Audio Cass.", "Video",
                      "Book", "Audio Cass.", "Video",
                      "Book", "Book", "Audio Cass."
                  };

    float prices[numProds] = {12.95, 14.95, 18.95, 16.95, 21.95,
                              31.95, 14.95, 14.95, 16.95};
    int prodNum, index;
    char again;

    do
    {
        prodNum = getProdNum();
        index = binarySearch(id, numProds, prodNum);
        if (index == -1)
            cout << "That product number was not found.\n";
        else
            displayProd(title, description, prices, index);
        cout << "Would you like to look up another product? (y/n) ";
        cin >> again;
    } while (again == 'y' || again == 'Y');
    return 0;
}
```

The named constants `numProds`, `titleSize`, and `descSize` will be defined globally and initialized to the values 9, 26, and 12 respectively. Notice that the arrays `id`, `title`, `description`, and `prices` are initialized with data.

The `getProdNum` Function

The `getProdNum` function prompts the user to enter a product number. It tests the value to ensure it is in the range of 914–922 (which are the valid product numbers). If an invalid value is entered, it is rejected and the user is prompted again. When a valid product number is entered, the function returns it. The pseudocode is shown below.

```
Display a prompt to enter a product number.
Read prodNum.
While prodNum is invalid
    Display an error messge.
    Read prodNum.
End While.
Return prodNum.
```

Here is the actual C++ code.

```
ιντ γετΠροδΝυμ()
{
  ιντ προδΝυμ;

    χουτ << ∀Εντερ τηε ιτεμ∋σ προδυχτ νυμβερ: ∀;
    χιν >> προδΝυμ;
    // ςαλιδατε ινπυτ.
    ωηιλε (προδΝυμ < μινΠροδΝυμ ¦¦ προδΝυμ > μαξΠροδΝυμ)
    {
        χουτ << ∀Εντερ α νυμβερ ιν τηε ρανγε οφ ∀ << μινΠροδΝυμ;
        χουτ <<∀ τηρουγη ∀ << μαξΠροδΝυμ << ∀.∴ν∀;
        χιν >> προδΝυμ;
    }
    ρετυρν προδΝυμ;
}
```

The `binarySearch` Function

The `binarySearch` function is identical to the function discussed earlier in this chapter.

The `displayProd` Function

The `displayProd` function has parameter variables named `title`, `desc`, `price`, and `index`. These accept as arguments (respectively) the `title`, `description`, and `price` arrays, and a subscript value. The function displays the data stored in each array at the subscript passed into `index`. Here is the C++ code.

```
void displayProd(char title[][titleSize], char desc[][descSize],
                 float price[], int index)
{
   cout << "Title: " << title[index] << endl;
   cout << "Description: " << desc[index] << endl;
   cout << "Price: $" << price[index] << endl;
}
```

The Entire Program

Program Q-3 shows the entire program's source code.

Program Q-3

```
// Demetris Leadership Center (DLC) product lookup program
// This program allows the user to enter a product number
// and then displays the title, description, and price of
// that product.
#include <iostream>
using namespace std;

const int numProds = 9;         // The number of products produced.
const int titleSize = 26;       // The max size of a title string
const int descSize = 12;        // The max size of a desc. string
const int minProdNum = 914;     // The lowest product number
const int maxProdNum = 922;     // The highest product number

// Function prototypes
int getProdNum();
int binarySearch(int [], int, int);
void displayProd(char [][titleSize], char [][descSize], float [], int);

int main()
{
   int id[numProds] = {914, 915, 916, 917, 918, 919, 920,
                       921, 922};

   char title[numProds][titleSize] =
             { "Six Steps to Leadership",
               "Six Steps to Leadership",
               "The Road to Excellence",
               "Seven Lessons of Quality",
               "Seven Lessons of Quality",
               "Seven Lessons of Quality",
               "Teams Are Made, Not Born",
               "Leadership for the Future",
               "Leadership for the Future"
             };
```

(program continues)

Program Q-3 *(continued)*

```
 char description[numProds][descSize] =
              { "Book", "Audio Cass.", "Video",
                "Book", "Audio Cass.", "Video",
                "Book", "Book", "Audio Cass."
              };

   float prices[numProds] = {12.95, 14.95, 18.95, 16.95, 21.95,
                             31.95, 14.95, 14.95, 16.95};
   int prodNum, index;
   char again;

   do
   {
       prodNum = getProdNum();
       index = binarySearch(id, numProds, prodNum);
       if (index == -1)
           cout << "That product number was not found.\n";
       else
           displayProd(title, description, prices, index);
       cout << "Would you like to look up another product? (y/n) ";
       cin >> again;
   } while (again == 'y' || again == 'Y');
   return 0;
}

//****************************************************
// Definition of getProdNum function                *
// The getProdNum function asks the user to enter a *
// product number. The input is validated, and when *
// a valid number is entered, it is returned.       *
//****************************************************
```

(program continues)

Program Q-3 ***(continued)***

```
int getProdNum()
{
   int prodNum;

    cout << "Enter the item's product number: ";
    cin >> prodNum;
    // Validate input
    while (prodNum < minProdNum || prodNum > maxProdNum)
    {
        cout << "Enter a number in the range of " << minProdNum;
        cout <<" through " << maxProdNum << ".\n";
        cin >> prodNum;
    }
    return prodNum;
}

//*****************************************************************
// Definition of binarySearch function                            *
// The binarySearch function performs a binary search on an      *
// integer array. array, which has a maximum of numElems         *
// elements, is searched for the number stored in value. If the *
// number is found, its array subscript is returned. Otherwise, *
// -1 is returned indicating the value was not in the array.     *
//*****************************************************************

int binarySearch(int array[], int numElems, int value)
{
    int first = 0,                         // First array element
        last = numElems - 1,               // Last array element
        middle,                            // Mid point of search
        position = -1;                     // Position of search value
    bool found = false;                    // Flag

    while (!found && first <= last)
    {
        middle = (first + last) / 2;       // Calculate mid point
        if (array[middle] == value)        // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)  // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;            // If value is in upper half
    }
    return position;
}
```

(program continues)

Program Q-3 ***(continued)***

```
//*************************************************************
// The displayProd function accepts three arrays and an int. *
// The arrays parameters are expected to hold the title,     *
// description, and prices arrays defined in main. The       *
// ndx parameter holds a subscript. This function displays   *
// the data in each array contained at the subscript.        *
//*************************************************************

void displayProd(char title[][titleSize], char desc[][descSize],
                 float price[], int index)
{
   cout << "Title: " << title[index] << endl;
   cout << "Description: " << desc[index] << endl;
   cout << "Price: $" << price[index] << endl;
}
```

Program Output with Example Input Shown in Bold

```
Enter the item's product number: 916 [Enter]
Title: The Road to Excellence
Description: Video
Price: $18.95
Would you like to look up another product? (y/n) y [Enter]
Enter the item's product number: 920 [Enter]
Title: Teams are Made, not Born
Description: Book
Price: $14.95
Would you like to look up another product? (y/n) n [Enter]
```

Focus on Software Engineering: *Introduction to Sorting Algorithms*

CONCEPT Sorting algorithms are used to arrange data into some order.

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Product codes could be sorted so all the products of the same color are stored together. To sort the data in an array, the programmer must use an appropriate *sorting algorithm*. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order. This section will introduce two simple sorting algorithms: the *bubble sort* and the *selection sort*.

The Bubble Sort

The bubble sort is an easy way to arrange data in *ascending* or *descending order*. If an array is sorted in ascending order, it means the values in the array are stored from lowest to highest. If the values are sorted in descending order, they are stored from highest to lowest. Let's see how the bubble sort is used in arranging the following array's elements in ascending order:

7	2	3	8	9	1
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged. After the exchange, the array shown above would appear as:

2	7	3	8	9	1
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

This method is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. The array above would then appear as:

2	3	7	8	9	1
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

Next, elements 2 and 3 are compared. In this array, these two elements are already in the proper order (element 2 is less than element 3), so no exchange takes place.

As the cycle continues, elements 3 and 4 are compared. Once again, no exchange is necessary because they are already in the proper order.

When elements 4 and 5 are compared, however, an exchange must take place because element 4 is greater than element 5. The array now appears as:

2	3	7	8	1	9
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

At this point, the entire array has been scanned, but its contents aren't quite in the right order yet. So, the sort starts over again with elements 0 and 1. Because those two are in the proper order, no exchange takes place. Elements 1 and 2 are compared next, but once again, no exchange takes place. This continues until elements 3 and 4 are compared. Because element 3 is greater than element 4, they are exchanged. The array now appears as

2	3	7	1	8	9
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

By now you should see how the sort will eventually cause the elements to appear in the correct order. The sort repeatedly passes through the array until no exchanges are made. Ultimately, the array will appear as

1	2	3	7	8	9
`Element 0`	`Element 1`	`Element 2`	`Element 3`	`Element 4`	`Element 5`

Here is the bubble sort in pseudocode:

Do
    Set swap flag to false.
    For count is set to each subscript in array from 0 through the next-to-last subscript
        If array[count] is greater than array[count + 1]
            Swap the contents of array[count] and array[count + 1].
            Set swap flag to true.
        End If.
    End For.
While any elements have been swapped.

The C++ code below implements the bubble sort as a function. The parameter `array` is an integer array to be sorted. `elems` contains the number of elements in `array`.

```
void sortArray(int array[], int elems)
{
    bool swap;
    int temp;

    do
    {
        swap = false;
        for (int count = 0; count < (elems - 1); count++)
        {
            if (array[count] > array[count + 1])
            {
                temp = array[count];
                array[count] = array[count + 1];
                array[count + 1] = temp;
                swap = true;
            }
        }
    } while (swap);
}
```

Inside the function is a `for` loop nested inside a `do-while` loop. The `for` loop sequences through the entire array, comparing each element with its neighbor, and swapping them if necessary. Anytime two elements are exchanged, the flag variable `swap` is set to `true`.

The `for` loop must be executed repeatedly until it can sequence through the entire array without making any exchanges. This is why it is nested inside a `do-while` loop. The `do-while` loop sets `swap` to `false`, and then executes the `for` loop. If `swap` is set to `true` after the `for` loop has finished, the `do-while` loop repeats.

Here is the starting line of the `for` loop:

```
for (int count = 0; count < (elems - 1); count++)
```

The variable `count` holds the array subscript values. It starts at zero and is incremented as long as it is less than `elems - 1`. The value of `elems` is the number of elements in the array, and `count` stops just short of reaching this value because the following line compares each element with the one after it:

```
if (array[count] > array[count + 1])
```

When `array[count]` is the next-to-last element, it will be compared to the last element. If the `for` loop were allowed to increment `count` past `elems - 1`, the last element in the array would be compared to a value outside the array.

Let's look at the `if` statement in its entirety:

```
if (array[count] > array[count + 1])
{
    temp = array[count];
    array[count] = array[count + 1];
    array[count + 1] = temp;
    swap = true;
}
```

If `array[count]` is greater than `array[count + 1]`, the two elements must be exchanged. First, the contents of `array[count]` are copied into the variable `temp`. Then the contents of `array[count + 1]` is copied into `array[count]`. The exchange is made complete when `temp` (which holds the previous contents of `array[count]`) are copied to `array[count + 1]`. Last, the `swap` flag variable is set to `true`. This indicates that an exchange has been made.

Program Q-4 demonstrates the bubble sort function in a complete program.

Program Q-4

```
// This program uses the bubble sort algorithm to sort an
// array in ascending order.
#include <iostream>
using namespace std;

// Function prototypes
void sortArray(int [], int);
void showArray(int [], int);
```

(program continues)

Program Q-4 ***(continued)***

```
int main()
{
    int values[6] = {7, 2, 3, 8, 9, 1};

    cout << "The unsorted values are:\n";
    showArray(values, 6);
    sortArray(values, 6);
    cout << "The sorted values are:\n";
    showArray(values, 6);
    return 0;
}

//**************************************************************
// Definition of function sortArray                            *
// This function performs an ascending order bubble sort on *
// array. elems is the number of elements in the array.        *
//**************************************************************

void sortArray(int array[], int elems)
{
    bool swap;
    int temp;

    do
    {
        swap = false;
        for (int count = 0; count < (elems - 1); count++)
        {
            if (array[count] > array[count + 1])
            {
                temp = array[count];
                array[count] = array[count + 1];
                array[count + 1] = temp;
                swap = true;
            }
        }
    } while (swap);
}

//****************************************************************
// Definition of function showArray.                             *
// This function displays the contents of array. elems is the *
// number of elements.                                           *
//****************************************************************

void showArray(int array[], int elems)
{
    for (int count = 0; count < elems; count++)
        cout << array[count] << " ";
    cout << endl;
}
```

Program Q-4 ***(continued)***

Program Output

```
The unsorted values are:
7 2 3 8 9 1
The sorted values are:
1 2 3 7 8 9
```

The Selection Sort

The bubble sort is inefficient for large arrays because items only move by one element at a time. The selection sort, however, usually performs fewer exchanges because it moves items immediately to their final position in the array. It works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order.

Let's see how the selection sort works when arranging the elements of the following array:

5	7	2	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The selection sort scans the array, starting at element 0, and locates the element with the smallest value. The contents of this element are then swapped with the contents of element 0. In this example, the 1 stored in element 5 is swapped with the 5 stored in element 0. After the exchange, the array would appear as

1	7	2	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The algorithm then repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at element 1. In this example, the contents of element 2 are exchanged with that of element 1. The array would then appear as

1	2	7	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value. This element's contents is exchanged with that of element 2, causing the array to appear as

1	2	5	8	9	7
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Next, the scanning begins at element 3. Its contents is exchanged with that of element 5, causing the array to appear as

1	2	5	7	9	8
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

At this point there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement:

1	2	5	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the selection sort algorithm in pseudocode:

For startScan is set to each subscript in array from 0 through the
next-to-last subscript
Set index variable to startScan.
Set minIndex variable to startScan.
Set minValue variable to array[startScan].

For index is set to each subscript in array from (startScan + 1)
through the next-to-last subscript
If array[index] is less than minValue
Set minValue to array[index].
Set minIndex to index.
End If.
Increment index.
End For.
Set array[minIndex] to array[startScan].
Set array[startScan] to minValue.
End For.

The following C++ code implements the selection sort in a function. It accepts two arguments: `array` and `elems`. `array` is an integer array and `elems` is the number of elements in the array. The function uses the selection sort to arrange the values in the array in ascending order.

```
void selectionSort(int array[], int elems)
{
    int startScan, minIndex, minValue;

    for (startScan = 0; startScan < (elems - 1); startScan++)
    {
        minIndex = startScan;
        minValue = array[startScan];
        for(int index = startScan + 1; index < elems; index++)
        {
            if (array[index] < minValue)
            {
                minValue = array[index];
                minIndex = index;
            }
        }
        array[minIndex] = array[startScan];
        array[startScan] = minValue;
    }
}
```

Inside the function are two `for` loops, one nested inside the other. The inner loop sequences through the array, starting at `array[startScan + 1]`, searching for the element with the smallest value. When the element is found, its subscript is stored in the variable `minIndex` and its value is stored in `minValue`. The outer loop then exchanges the contents of this element with `array[startScan]` and increments `startScan`. This procedure repeats until the contents of every element have been moved to their proper location.

Program Q-5 demonstrates the selection sort function in a complete program.

Program Q-5

```
// This program uses the selection sort algorithm to sort an
// array in ascending order.
#include <iostream>
using namespace std;

// Function prototypes
void selectionSort(int [], int);
void showArray(int [], int);

int main()
{
   int values[6] = {5, 7, 2, 8, 9, 1};

   cout << "The unsorted values are\n";
   showArray(values, 6);
   selectionSort(values, 6);
   cout << "The sorted values are\n";
   showArray(values, 6);
   return 0;
}

//****************************************************************
// Definition of function selectionSort.                         *
// This function performs an ascending order selection sort on  *
// array. elems is the number of elements in the array.         *
//****************************************************************

void selectionSort(int array[], int elems)
{
   int startScan, minIndex, minValue;

   for (startScan = 0; startScan < (elems - 1); startScan++)
   {
      minIndex = startScan;
      minValue = array[startScan];
```

(program continues)

Program Q-5 ***(continued)***

```
        for(int index = startScan + 1; index < elems; index++)
        {
            if (array[index] < minValue)
            {
                minValue = array[index];
                minIndex = index;
            }
        }
        array[minIndex] = array[startScan];
        array[startScan] = minValue;
    }
}
//****************************************************************
// Definition of function showArray.                             *
// This function displays the contents of array. elems is the   *
// number of elements.                                           *
//****************************************************************

void showArray(int array[], int elems)
{
    for (int count = 0; count < elems; count++)
        cout << array[count] << " ";
    cout << endl;
}
```

Program Output

```
The unsorted values are
5 7 2 8 9 1
The sorted values are
1 2 5 7 8 9
```

Focus on Problem Solving and Program Design: *A Case Study*

Like the previous case study, this is a program developed for the Demetris Leadership Center. Recall that DLC, Inc., publishes books, videos, and audio cassettes. (See Table Q-1 for a complete list of products, with title, description, product number, and price.) Table Q-4 shows the number of units of each product sold during the past six months.

The vice president of sales has asked you to write a sales reporting program that displays the following information:

- A list of the products in the order of their sales dollars (NOT units sold), from highest to lowest
- The total number of all units sold
- The total sales for the six-month period

Table Q-4

Product Number	Units Sold
914	842
915	416
916	127
917	514
918	437
919	269
920	97
921	492
922	212

Variables

Table Q-5 lists the variables needed:

Table Q-5

Variable	Description
`numProds`	A constant integer initialized with the number of products that DLC, Inc., sells. This value will be used in the defintion of the program's array.
`prodNum`	Array of `int`s. Holds each product's number.
`units`	Array of `int`s. Holds each product's number of units sold.
`prices`	Array of `float`s. Holds each product's price.
`sales`	Array of `float`s. Holds the computed sales amounts (in dollars) of each product.

The elements of the four arrays, `prodNum`, `units`, `prices`, and `sales` will correspond with each other. For example, the product whose number is stored in `prodNum[2]` will have sold the number of units stored in `units[2]`. The sales amount for the product will be stored in `sales[2]`.

Modules

The program will consist of the functions listed in Table Q-6.

Table Q-6

Function	Description
`main`	The program's `main` function. It calls the program's other functions.
`calcSales`	Calculates each product's sales.
`dualSort`	Sorts the `sales` array so the elements are ordered from highest to lowest. The `prodNum` array is ordered so the product numbers correspond with the correct sales figures in the sorted `sales` array.
`showOrder`	Displays a list of the product numbers and sales amounts from the sorted `sales` and `prodNum` arrays.
`showTotals`	Displays the total number of units sold and the total sales amount for the period.

Function `main`

Function `main` is very simple. It contains the variable definitions and calls the other functions. Here is the pseudocode for its executable statements:

Call calcSales.
Call dualSort.
Set display mode to fixed point output with 2 decimal places of precision.
Call showOrder.
Call showTotals.

Here is its actual C++ code:

```
int main ()
{
    int id[numProds] = {914, 915, 916, 917, 918, 919, 920,
                        921, 922};
    int units[numProds] = {842, 416, 127, 514, 437, 269, 97,
                           492, 212};
    float prices[numProds] = {12.95, 14.95, 18.95, 16.95, 21.95,
                              31.95, 14.95, 14.95, 16.95};
    float sales[numProds];

    calcSales(units, prices, sales, numProds);
    dualSort(id, sales, numProds);
    cout << fixed << showpoint << setprecision(2);
    showOrder(sales, id, numProds);
    showTotals(sales, units, numProds);
    return 0;
}
```

The named constant `numProds` will be defined globally and initialized to the value 9.

Notice that the arrays `id`, `units`, and `prices` are initialized with data. (It will be left as an exercise for you to modify this program so the user may enter these values.)

The `calcSales` Function

The `calcSales` function multiplies each product's units sold by its price. The resulting amount is stored in the `sales` array. Here is the function's pseudocode:

For index is set to each subscript in the arrays from 0 through the last subscript.
Set sales[index] to units[index] times prices[index].
End For.

And here is the function's actual C++ code:

```
void calcSales(int units[], float prices[], float sales[], int num)
{
    for (int index = 0; index < num; index++)
        sales[index] = units[index] * prices[index];
}
```

The `dualSort` Function

The `dualSort` function is a modified version of the selection sort algorithm shown in Program Q-5. The `dualSort` function accepts two arrays as arguments: the `sales` array and the `id` array. The function actually performs the selection sort on the `sales` array. When the function moves an element in the `sales` array, however, it also moves the corresponding element in the `id` array. This is to ensure that the product numbers in the `id` array still have subscripts that match their sales figures in the `sales` array.

The `dualSort` function is also different in another way: It sorts the array in descending order.

Here is the pseudocode for the `dualSort` function:

For startScan variable is set to each subscript in array from 0 through the next-to-last subscript
Set index variable to startScan.
Set maxIndex variable to startScan.
Set tempId variable to id[startScan].
Set maxValue variable to sales[startScan].
For index variable is set to each subscript in array from (startScan + 1) through the last subscript
If sales[index] is greater than maxValue
Set maxValue to sales[index].
Set tempId to tempId[index].
Set maxIndex to index.
End If.
End For.
Set sales[maxIndex] to sales[startScan].
Set id[maxIndex] = id[startScan].
Set sales[startScan] to maxValue.
Set id[startScan] = tempId.
End For.

Here is the actual C++ code for the `dualSort` function:

```
void dualSort(int id[], float sales[], int elems)
{
    int startScan, maxIndex, tempId;
    float maxValue;

    for (startScan = 0; startScan < (elems - 1); startScan++)
    {
        maxIndex = startScan;
        maxValue = sales[startScan];
        tempId = id[startScan];
        for(int index = startScan + 1; index < elems; index++)
        {
            if (sales[index] > maxValue)
            {
                maxValue = sales[index];
                tempId = id[index];
                maxIndex = index;
            }
        }
        sales[maxIndex] = sales[startScan];
        id[maxIndex] = id[startScan];
        sales[startScan] = maxValue;
        id[startScan] = tempId;
    }
}
```

Note: Once the `dualSort` function is called, the `id` and `sales` arrays are no longer synchronized with the `units` and `prices` arrays. Because this program doesn't use `units` and `prices` together with `id` and `sales` after this point, it will not be noticed in the final output. It is never a good programming practice to sort parallel arrays in such a way that they are out of synchronization. It will be left as an exercise for you to modify the program so all the arrays are synchronized and used in the final output of the program.

The `showOrder` Function

The `showOrder` function displays a heading and the sorted list of product numbers and their sales amounts. It accepts the `id` and `sales` arrays as arguments. Here is its pseudocode:

Display heading.
For index variable is set to each subscript of the arrays from 0 through the last subscript
    Display id[index].
    Display sales[index].
End For.

Here is the function's actual C++ code:

```
void showOrder(float sales[], int id[], int num)
{
    cout << "Product Number\tSales\n";

    cout << "----------------------------------\n";
    for (int index = 0; index < num; index++)
    {
        cout << id[index] << "\t\t$";
        cout << setw(8) << sales[index] << endl;
    }
    cout << endl;
}
```

The `showTotals` Function

The `showTotals` function displays the total number of units of all products sold and the total sales for the period. It accepts the `units` and `sales` arrays as arguments. Here is its pseudocode:

Set totalUnits variable to 0.
Set totalSales variable to 0.0.
For index variable is set to each subscript in the arrays from 0 through the last subscript
    Add units[index] to totalUnits[index].
    Add sales[index] to totalSales.
End For.
Display totalUnits with appropriate heading.
Display totalSales with appropriate heading.

Here is the function's actual C++ code:

```
void showTotals(float sales[], int units[], int num)
{
    int totalUnits = 0;
    float totalSales = 0.0;
    for (int index = 0; index < num; index++)
    {
        totalUnits += units[index];
        totalSales += sales[index];
    }
    cout << "Total Units Sold:  " << totalUnits << endl;
    cout << "Total Sales:      $" << totalSales << endl;
}
```

The Entire Program

Program Q-6 shows the entire program's source code.

Program Q-6

```
// This program produces a sales report for DLC, Inc.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void calcSales(int [], float [], float [], int);
void showOrder(float [], int [], int);
void dualSort(int [], float [], int);
void showTotals(float [], int [], int);

// numProds is the number of products produced.
const int numProds = 9;

int main()
{
    int id[numProds] = {914, 915, 916, 917, 918, 919, 920,
                        921, 922};
    int units[numProds] = {842, 416, 127, 514, 437, 269, 97,
                           492, 212};
    float prices[numProds] = {12.95, 14.95, 18.95, 16.95, 21.95,
                              31.95, 14.95, 14.95, 16.95};
    float sales[numProds];

    calcSales(units, prices, sales, numProds);
    dualSort(id, sales, numProds);
    cout << fixed showpoint << setprecision(2);
    showOrder(sales, id, numProds);
    showTotals(sales, units, numProds);
    return 0;
}

//****************************************************************
// Definition of calcSales. Accepts units, prices, and sales    *
// arrays as arguments. The size of these arrays is passed      *
// into the num parameter. This function calculates each        *
// product's sales by multiplying its units sold by each unit's *
// price. The result is stored in the sales array.              *
//****************************************************************

void calcSales(int units[], float prices[], float sales[], int num)
{
    for (int index = 0; index < num; index++)
        sales[index] = units[index] * prices[index];
}
```

(program continues)

Program Q-6 ***(continued)***

```
//************************************************************
// Definition of function dualSort. Accepts id and sales arrays *
// as arguments. The size of these arrays is passed into elems. *
// This function performs a descending order selection sort on  *
// the sales array. The elements of the id array are exchanged  *
// identically as those of the sales array. elems is the number *
// of elements in each array.                                   *
//************************************************************

void dualSort(int id[], float sales[], int elems)
{
    int startScan, maxIndex, tempId;
    float maxValue;

    for (startScan = 0; startScan < (elems - 1); startScan++)
    {
        maxIndex = startScan;
        maxValue = sales[startScan];
        tempId = id[startScan];
        for(int index = startScan + 1; index < elems; index++)
        {
            if (sales[index] > maxValue)
            {
                maxValue = sales[index];
                tempId = id[index];
                maxIndex = index;
            }
        }
        sales[maxIndex] = sales[startScan];
        id[maxIndex] = id[startScan];
        sales[startScan] = maxValue;
        id[startScan] = tempId;
    }
}

//*************************************************************
// Definition of showOrder function. Accepts sales and id arrays *
// as arguments. The size of these arrays is passed into num.    *
// The function first displays a heading, then the sorted list   *
// of product numbers and sales.                                 *
//*************************************************************
```

(program continues)

Program Q-6 ***(continued)***

```
void showOrder(float sales[], int id[], int num)
{
    cout << "Product Number\tSales\n";
    cout << "--------------------------------\n";
    for (int index = 0; index < num; index++)
    {
        cout << id[index] << "\t\t$";
        cout << setw(8) << sales[index] << endl;
    }
    cout << endl;
}

//*****************************************************************
// Definition of showTotals function. Accepts sales and id arrays *
// as arguments. The size of these arrays is passed into num.     *
// The function first calculates the total units (of all          *
// products) sold and the total sales. It then displays these     *
// amounts.                                                       *
//*****************************************************************

void showTotals(float sales[], int units[], int num)
{
    int totalUnits = 0;
    float totalSales = 0.0;
    for (int index = 0; index < num; index++)
    {
        totalUnits += units[index];
        totalSales += sales[index];
    }
    cout << "Total Units Sold:  " << totalUnits << endl;
    cout << "Total Sales:      $" << totalSales << endl;
}
```

Program Output

```
Product Number    Sales
---------------------------------
914               $10903.90
918               $ 9592.15
917               $ 8712.30
919               $ 8594.55
921               $ 7355.40
915               $ 6219.20
922               $ 3593.40
916               $ 2406.65
920               $ 1450.15
Total Units Sold:  3406
Total Sales:      $58827.70
```

If You Plan to Continue in Computer Science: *Sorting and Searching* `vectors` (Continued from Section 7.11)

CONCEPT The sorting and searching algorithms you have studied in this chapter may be applied to STL `vectors` as well as arrays.

Once you have properly defined an STL `vector` and populated it with values, you may sort and search the `vector` with the algorithms presented in this chapter. Simply substitute the `vector` syntax for the array syntax when necessary. Program Q-7, which illustrates this, is a modification of the case study in Program Q-6.

Program Q-7

```
// This program produces a sales report for DLC, Inc.
// This version of the program uses STL vectors instead of arrays.
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

// Function prototypes
void initVectors(vector<int> &, vector<int> &, vector<float> &);
void calcSales(vector<int>, vector<float>, vector<float> &);
void showOrder(vector<float>, vector<int>);
void dualSort(vector<int> &, vector<float> &);
void showTotals(vector<float>, vector<int>);

int main()
{
    vector<int> id;
    vector<int> units;
    vector<float> prices;
    vector<float> sales;

    // Must provide an initialization routine.
    initVectors(id, units, prices);

    // Calculate and sort the sales totals,
    // and display the results.
    calcSales(units, prices, sales);
    dualSort(id, sales);
    cout << fixed << showpoint << setprecision(2);
    showOrder(sales, id);
    showTotals(sales, units);
    return 0;
}
```

(program continues)

Program Q-7 *(continued)*

```
//***************************************************************
// Definition of initVectors. Accepts id, units, and prices      *
// vectors as reference arguments. This function initializes each *
// vector to a set of starting values.                           *
//***************************************************************

void initVectors(vector<int> &id, vector<int> &units,
                 vector<float> &prices)
{
   // Initialize the id vector
   for (int value = 914; value <= 922; value++)
      id.push_back(value);

   // Initialize the units vector
   units.push_back(842);
   units.push_back(416);
   units.push_back(127);
   units.push_back(514);
   units.push_back(437);
   units.push_back(269);
   units.push_back(97);
   units.push_back(492);
   units.push_back(212);

   // Initialize the prices vector
   prices.push_back(12.95);
   prices.push_back(14.95);
   prices.push_back(18.95);
   prices.push_back(16.95);
   prices.push_back(21.95);
   prices.push_back(31.95);
   prices.push_back(14.95);
   prices.push_back(14.95);
   prices.push_back(16.95);
}

//*************************************************************
// Definition of calcSales. Accepts units, prices, and sales    *
// vectors as arguments. The sales vector is passed into a     *
// reference parameter. This function calculates each product's *
// sales by multiplying its units sold by each unit's price. The *
// result is stored in the sales vector.                        *
//*************************************************************

void calcSales(vector<int> units, vector<float> prices,
               vector<float> &sales)
{
   for (int index = 0; index < units.size(); index++)
      sales.push_back(units[index] * prices[index]);
}
```

(program continues)

Program Q-7 ***(continued)***

```
//*****************************************************************
// Definition of function dualSort. Accepts id and sales vectors *
// as reference arguments. This function performs a descending   *
// order selection sort on the sales vector. The elements of the *
// id vector are exchanged identically as those of the sales     *
// vector.                                                        *
//*****************************************************************

void dualSort(vector<int> &id, vector<float> &sales)
{
    int startScan, maxIndex, tempid, elems;
    float maxValue;

    elems = id.size();
    for (startScan = 0; startScan < (elems - 1); startScan++)
    {
        maxIndex = startScan;
        maxValue = sales[startScan];
        tempid = id[startScan];
        for(int index = startScan + 1; index < elems; index++)
        {
            if (sales[index] > maxValue)
            {
                maxValue = sales[index];
                tempid = id[index];
                maxIndex = index;
            }
    }
    sales[maxIndex] = sales[startScan];
    id[maxIndex] = id[startScan];
    sales[startScan] = maxValue;
    id[startScan] = tempid;
    }
}

//******************************************************************
// Definition of showOrder function. Accepts sales and id vectors *
// as arguments. The function first displays a heading, then the  *
// sorted list of product numbers and sales.                      *
//******************************************************************

void showOrder(vector<float> sales, vector<int> id)
{
    cout << "Product Number\tSales\n";
    cout << "----------------------------------\n";
```

(program continues)

Program Q-7 ***(continued)***

```
    for (int index = 0; index < id.size(); index++)
    {
        cout << id[index] << "\t\t$";
        cout << setw(8) << sales[index] << endl;
    }
    cout << endl;
}

//*****************************************************************
// Definition of showTotals function. Accepts sales and id vectors  *
// as arguments. The function first calculates the total units (of  *
// all products) sold and the total sales. It then displays these   *
// amounts.                                                         *
//*****************************************************************

void showTotals(vector<float> sales, vector<int> units)
{
    int totalUnits = 0;
    float totalSales = 0.0;

    for (int index = 0; index < units.size(); index++)
    {
        totalUnits += units[index];
        totalSales += sales[index];
    }
    cout << "Total units Sold:  " << totalUnits << endl;
    cout << "Total sales:      $" << totalSales << endl;
}
```

Program Output

```
Product Number  Sales
----------------------------------
914             $10903.90
918             $ 9592.15
917             $ 8712.30
919             $ 8594.55
921             $ 7355.40
915             $ 6219.20
922             $ 3593.40
916             $ 2406.65
920             $ 1450.15
Total Units Sold:  3406
Total Sales:      $58827.70
```

There are some differences between this program and Program Q-6. First, the `initVectors` function was added. In Program Q-6, this was not necessary because the `id`, `units`, and `prices` arrays had initialization lists. `vector`s do not accept initialization lists, so this function stores the necessary initial values in the `id`, `units`, and `prices` vectors.

Now, look at the function header for `initVectors`:

```
void initVectors(vector<int> &id, vector<int> &units,
                 vector<float> &prices)
```

Notice that the `vector` parameters are references (as indicated by the `&` that precedes the parameter name). This brings up an important difference between vectors and arrays: By default, `vectors` are passed by value, whereas arrays are only passed by reference. If you want to change a value in a `vector` argument, it *must* be passed into a reference parameter. Reference `vector` parameters are also used in the `calcSales` and `dualSort` functions.

Also, notice that each time a value is added to a `vector`, the `push_back` member function is called. This is because the `[]` operator cannot be used to store a new element in a `vector`. It can only be used to store a value in an existing element or read a value from an existing element.

The code in this function appears cumbersome because it calls each `vector`'s `push_back` member function once for each value that is to be stored in the `vector`. This code can be simplified by storing the `vector` initialization values in arrays, and then using loops to call the `push_back` member function, storing the values in the arrays in the vectors. The following code shows an alternative initVectors function which takes this approach.

```
void initVectors(vector<int> &id, vector<int> &units,
                 vector<float> &prices)
{
   const int numProds = 9;
   int count;
   int unitsSold[numProds] = {842, 416, 127, 514, 437, 269, 97,
                              492, 212};
   float productPrices[numProds] = {12.95, 14.95, 18.95, 16.95,
                                    21.95, 31.95, 14.95, 14.95,
                                    16.95};

   // Initialize the id vector
   for (int value = 914; value <= 922; value++)
      id.push_back(value);

   // Initialize the units vector
   for (count = 0; count < numProds; count++)
      units.push_back(unitsSold[count]);

   // Initialize the prices vector
   for (count = 0; count < numProds; count++)
      prices.push_back(productPrices[count]);
}
```

Next, notice that the `calcSales`, `showOrder`, `dualSort`, and `showTotals` functions do not accept an argument indicating the number of elements in the `vectors`. This is not necessary because `vectors` have the `size` member function, which returns the number of elements in the `vector`. The following code segment, which is taken from the `calcSales` function, shows the `units.size()` member function being used to control the number of loop iterations.

```
for (int index = 0; index < units.size(); index++)
     sales.push_back(units[index] * prices[index]);
```