09GaddA2005 Page 567 Thursday, September 15, 2005 1:15 PM

9

Searching and Sorting Arrays

9.1 Focus on Software Engineering: *Introduction to Search Algorithms*

CONCEPT

A search algorithm is a method of locating a specific item of data in a collection of data.

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 index < number of elements and found is false
    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`, which follows, 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;       // Used 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. Any other nonvalid subscript value could also have been used to signal this.

Program 9-1 is a complete program that uses the `searchList` function. It searches the five-element `tests` array to find a score of 100.

Program 9-1

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

```

(program continues)

Program 9-1 ***(continued)***

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

const int SIZE = 5;

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

    results = searchList(tests, SIZE, 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) << ".\n";
    }
    return 0;
}

//*****************************************************************
//                          searchList                            *
// This function performs a linear search on an integer array.   *
// The list array, which has numElems elements, is searched for  *
// the number stored in value. If the number is found, its array *
// subscript is returned. Otherwise, -1 is returned.             *
//*****************************************************************
int searchList(int list[], int numElems, int value)
{
    int index = 0;              // Used as a subscript to search array
    int position = -1;          // Used 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 contained 20,000 elements, the algorithm would have to look at all 20,000 elements in order to find a value stored in the last element or to determine that a desired element was not in the array.

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 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 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 than the desired value then the value (if it is in the list) will be found somewhere in the first half of the array. If it is less than the desired value then the 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 whose elements are stored in ascending order.

```
Set first to 0
Set last 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 first and 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` in the following example C++ code is used to perform a binary search on an integer array. The first parameter, `array`, which has `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,                        // Midpoint of search
         position = -1;                 // Position of search value
    bool found = false;                 // Flag

    while (!found && first <= last)
    {
       middle = (first + last) / 2;     // Calculate midpoint
       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 9-2 is a complete program using the `binarySearch` function. It searches an array of employee ID numbers for a specific value.

Program 9-2

```
// This program performs a binary search on an integer array
// whose elements are in ascending order.
#include <iostream>
using namespace std;

// Function prototype
int binarySearch(int [], int, int);
const int SIZE = 20;

int main()
{
    int tests[SIZE] = {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, SIZE, 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;
}

//*******************************************************************
//                          binarySearch                            *
// This function performs a binary search on an integer array       *
// with numElems elements whose values are stored in ascending      *
// order. The array is searched for the number stored in the        *
// value parameter. If the number is found, its array subscript     *
// returned. Otherwise, -1 is returned.                             *
//*******************************************************************
int binarySearch(int array[], int numElems, int value)
{
    int  first = 0,                 // First array element
         last = numElems - 1,       // Last array element
         middle,                    // Midpoint of search
         position = -1;             // Position of search value
    bool found = false;             // Flag

```

(program continues)

Program 9-2 ***(continued)***

```
    while (!found && first <= last)
    {
       middle = (first + last) / 2;        // Calculate midpoint
       if (array[middle] == value)         // If value is found at midpoint
       {
          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 Shown in Bold

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

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 locates the desired value or determines that it is not in the array. With 1,000 elements in the array, this takes a maximum of 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 the number of elements in the array. That will tell you the maximum number of comparisons needed to find an element, or to determine that it is not present. For example, a maximum of 16 comparisons will be made to find an item in an array of 50,000 elements (2^{16} = 65,536), and a maximum of 20 comparisons will be made to find an item in an array of 1,000,000 elements (2^{20} = 1,048,576).

9.2 Demetris Leadership Center Case Study—Part 1

The Demetris Leadership Center (DLC, Inc.) publishes the books, videos, and audio cassettes listed in Table 9-1.

Table 9-1 DLC Product Line

PRODUCT NUMBER	PRODUCT TITLE	PRODUCT DESCRIPTION	UNIT PRICE	UNITS SOLD
914	Six Steps to Leadership	Book	$12.95	842
915	Six Steps to Leadership	Audio cassette	$14.95	416
916	The Road to Excellence	Video	$18.95	127
917	Seven Lessons of Quality	Book	$16.95	514
918	Seven Lessons of Quality	Audio cassette	$21.95	437
919	Seven Lessons of Quality	Video	$31.95	269
920	Teams are Made, Not Born	Book	$14.95	97
921	Leadership for the Future	Book	$14.95	492
922	Leadership for the Future	Audio cassette	$16.95	212

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 then display the title, description, and price of the product.

Structure

The program will use an array of `ProdStruct` structures to hold the product information. The structure has one member for each field of information to be stored. Table 9-2 lists the members of the `ProdStruct` structure.

Table 9-2 Members of the `ProdStruct` Structure

MEMBER	DATA TYPE	DESCRIPTION
`id`	`int`	Product number
`title`	`string`	Title
`description`	`string`	Description
`price`	`double`	Unit price
`sold`	`int`	Units sold during the past 6 months

Variables

Table 9-3 lists the variables to be used in the program.

Table 9-3 Variables Used in the DLC Sales Program

VARIABLE	DATA TYPE	DESCRIPTION
`product`	`ProdStruct`	Array of structures holding the product data. There is one array element for each product carried by DLC.
`NUM_PRODS`	`const int`	Number of products carried by DLC
`MIN_PROD_NUM`	`const int`	Lowest valid product number
`MAX_PROD_NUM`	`const int`	Highest valid product number
`prodNum`	`int`	Product number input by the user
`index`	`int`	Array index used to hold the location of a record
`again`	`char`	Holds user's choice (y/n) to do another search

Modules

The program will consist of the functions listed in Table 9-4.

Table 9-4 Functions in the DLC Sales Program

FUNCTION	DESCRIPTION
`main`	The program's main function. It calls the program's other functions.
`getProdNumber`	This function accepts, validates, and returns a product number input by the user.
`binarySearch`	A binary search routine modified to search through an array of `ProdStruct` structures searching for a specific value stored in the `id` member of the structure. If the desired `id` value is found, the subscript of the element holding the structure is returned. If the value is not found, –1 is returned.
`displayProd`	Displays the product `id`, `title`, `description`, and `price` members of the structure whose array index is passed to the function.

Function `main`

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

```
Initialize numProds, minProdNum, and maxProdNum
Set up the product array and initialize it with all the product records
Do
   Call getProdNum          // Returns the id the user wants
   Call binarySearch        // Returns the index of the desired record
```

9

```
   If binarySearch returned -1
      Display a message that product was not found
   Else
      Call displayProd       // Displays the record data
   End If
   Ask the user if the program should search for another record
   Input again
While again equals 'y' or 'Y'
```

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 to 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 as follows:

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

The `binarySearch` Function

The `binarySearch` function is identical to the function discussed earlier in this chapter, with two changes. First, instead of receiving an array of integers as the earlier `binarySearch` function did, the revised search function receives an array of `ProdStruct` structures. Its function header looks like this:

```
int binarySearch(ProdStruct array[], int numElems, int value)
```

Second, because each array element is now a structure holding an entire set of data fields, the value being searched for can no longer be compared to an entire array element. Instead, it must be compared to one of the members, or fields, of the structure. The specific field being searched on is sometimes called the *key field*. In this program, the data passed to the `value` parameter is a product id number. Therefore, the key field for the search is the `id` field. The two lines of code in the earlier `binarySearch` function that compared the `value` parameter to an array element are modified to compare the parameter to the `id` field of an array element, as shown here:

```
if (array[middle].id == value)

else if (array[middle].id > value)
```

The `displayProd` Function

The `displayProd` function has two parameters, one to receive the `product` array and one to receive the index of the structure within the array whose data members are to be displayed. It displays the `id`, `title`, `description` and `price` fields of this array element.

The Entire Program

Program 9-3 shows the entire program's source code.

Program 9-3

```
// This program manages an array of product structures. It allows
// the user to enter a product number, then finds and displays
// information on that product.
#include <iostream>
#include <string>
using namespace std;

struct ProdStruct
{
   int    id;                    // Product number
   string title,                 // Product title
          description;           // Product description
   double price;                 // Product unit price
   int    sold;                  // Units sold during the past 6 months

  // Default constructor for a ProdStruct structure
   ProdStruct()
   {  price = id = sold = 0;
      title = description = "";
   }
   // Constructor to set initial data values
   ProdStruct(int i, string t, string d, double p, int s)
   {  id = i;
      title = t;
      description = d;
      price = p;
      sold = s;
   }
};
// Function prototypes
int getProdNum(int, int);
int binarySearch(ProdStruct [], int, int);
void displayProd(ProdStruct [], int);

```

(program continues)

Program 9-3 ***(continued)***

```
int main()
{
    const int NUM_PRODS = 9,        // Number of products carried by DLC
              MIN_PROD_NUM = 914,   // Minimum product number
              MAX_PROD_NUM = 922;   // Maximum product number

    ProdStruct product[NUM_PRODS] =
    {
      ProdStruct(914, "Six Steps to Leadership",  "Book",  12.95,  842),
      ProdStruct(915, "Six Steps to Leadership",  "Audio cassette",
                                                           14.95,  416),
      ProdStruct(916, "The Road to Excellence",   "Video", 18.95,  127),
      ProdStruct(917, "Seven Lessons of Quality", "Book",  16.95,  514),
      ProdStruct(918, "Seven Lessons of Quality", "Audio cassette",
                                                           21.95,  437),
      ProdStruct(919, "Seven Lessons of Quality", "Video", 31.95,  269),
      ProdStruct(920, "Teams are Made, Not Born", "Book",  14.95,   97),
      ProdStruct(921, "Leadership for the Future","Book",  14.95,  492),
      ProdStruct(922, "Leadership for the Future","Audio cassette",
                                                           16.95,  212)
    };

    int prodNum,      // Product number the user wants
        index;        // Array pos where that product's record is found
    char again;       // Does user want to look up another record (y/n)?

    do
    {
        // Get the desired product number
        prodNum = getProdNum(MIN_PROD_NUM, MAX_PROD_NUM);

        // Find the array index of the record for that product
        index = binarySearch(product, NUM_PRODS, prodNum);

        if (index == -1)
            cout << "That product number was not found.\n";
        else
            displayProd(product, index);

        cout << "\nWould you like to look up another product? (y/n) ";
        cin  >> again;
    } while (again == 'y' || again == 'Y');
    return 0;
}

```

(program continues)

Program 9-3 ***(continued)***

```
//*************************************************************
//                         getProdNum                         *
// Passed in: legal minumum and maximum product numbers      *
// Returned : a valid product number                          *
//                                                            *
// The getProdNum function accepts, validates, and returns a *
// product number input by the user.                          *
//*************************************************************
int getProdNum(int min, int max)
{
    int prodNum;

    cout << "Enter the item's product number "
         << min << " - " << max << ": ";
    cin  >> prodNum;

    // Validate input
    while (prodNum < min || prodNum > max)
    {
        cout << "Invalid product number.\n"
             << "Enter the item's product number "
             << min << " - " << max << ": ";
        cin  >> prodNum;
    }
    return prodNum;
}

//*********************************************************************
//                          binarySearch                              *
// Passed in: the product array, its size, and the product ID being  *
//              searched for                                          *
// Returned : the array index for the record with that ID            *
//                                                                    *
// This binarySearch function is modified to perform a binary search *
// on the id field of an array of ProdStruct structures, looking for *
// a record (i.e., an array element) whose id member matches value,  *
// which holds the product id passed to the function. If the record  *
// is found, its array subscript is returned. If it is not found, -1 *
// is returned.                                                       *
//*********************************************************************
int binarySearch(ProdStruct array[], int numElems, int value)
{
    int  first = 0,                        // First array element
         last = numElems - 1,              // Last array element
         middle,                           // Midpoint of search
         position = -1;                    // Position of search value
    bool found = false;                    // Flag

```

(program continues)

9

Program 9-3 *(continued)*

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

//**********************************************************************
//                              displayProd                            *
// Passed in: the product array and the index of a specific element   *
//            of that array                                            *
//                                                                     *
// This function displays four fields (i.e., structure members) of    *
// the product array element whose index is passed to the function.   *
//**********************************************************************
void displayProd(ProdStruct product[], int index)
{
    cout << "\nID:          "  << product[index].id;
    cout << "\nTitle:       "  << product[index].title;
    cout << "\nDescription: "  << product[index].description;
    cout << "\nPrice:       $" << product[index].price << endl;
}
```

Program Output with Example Input Shown in Bold

```
Enter the item's product number 914 - 922: 900[Enter]
Invalid product number.
Enter the item's product number 914 - 922: 916[Enter]

ID:          916
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 914 - 922: 920[Enter]

ID:          920
Title:       Teams are Made, Not Born
Description: Book
Price:       $14.95

Would you like to look up another product? (y/n) n[Enter]
```

Checkpoint

9.1 Describe the difference between the linear search and the binary search.

9.2 On average, with an array of 20,000 elements, how many comparisons will the linear search perform? (Assume the items being search for are consistently found in the array.)

9.3 With an array of 20,000 elements, what is the maximum number of comparisons the binary search will perform?

9.4 If a linear search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

9.3 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. Mailing label records could be sorted by ZIP code. 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. Bubble sort works by comparing each element with its neighbor and swapping them if they are not in the desired order. Let's see how it arranges 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 appears as

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

This process is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. The array now appears as

9

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. However, 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. This is called the first *pass* of the sort. Notice that the largest value is now correctly placed in the last array element. However, the rest of the array is not yet sorted. So the sort starts over again with elements 0 and 1. Because they 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

Notice that this second pass over the array elements has placed the second largest number in the next to the last array element. This process will continue, with the sort repeatedly passing through the array and placing one number in order on each pass, until the array is fully sorted. 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. Notice that it uses a pair of nested loops. The outer loop, a `do-while` loop, iterates once for each pass of the sort. The inner loop, a `for` loop, holds the code that does all the comparisons and needed swaps during a pass. If two elements are exchanged, the `swap` flag variable is set to `true`. The outer loop continues iterating, causing additional passes to be made, until it finds the `swap` flag `false`, meaning that no elements were swapped on the previous pass. This is how the algorithms "knows" that the array is now fully sorted.

```
Do
    Set swap flag to false
    For count = 0 to the next-to-last array 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 the swap flag is true     // a swap ocurred on the previous pass
```

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

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

    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);
}
```

Let's look at more closely at the `for` loop that handles the comparisons and exchanges during a pass. Here is its starting line:

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

The variable `count` holds the array subscripts. 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.

Here is 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]` is 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]`) is copied to `array[count + 1]`. Last, the `swap` flag variable is set to `true`. This indicates that an exchange has been made.

Program 9-4 demonstrates the bubble sort function in a complete program.

Program 9-4

```
// 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);

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

   cout << "The unsorted values are:\n";
   showArray(values, SIZE);

   sortArray(values, SIZE);

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

//**************************************************************
//                         sortArray                           *
// This function performs an ascending-order bubble sort on   *
// array. The parameter elems holds the number of elements    *
// in the array.                                               *
//**************************************************************
void sortArray(int array[], int elems)
{
   int temp;
   bool swap;

   do
   {  swap = false;
      for (int count = 0; count < (elems - 1); count++)
      {
```

(program continues)

Program 9-4 (continued)

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

//**************************************************************
//                          showArray                          *
// This function displays the contents of array. The parameter *
// elems holds the number of elements in the array.            *
//**************************************************************
void showArray(int array[], int elems)
{
    for (int count = 0; count < elems; count++)
        cout << array[count] << " ";
    cout << endl;
}
```

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 repeated data swaps are often required to place a single item in its correct position. The selection sort, like the bubble sort, places just one item in its correct position on each pass. However, it usually performs fewer exchanges because it moves items immediately to their correct position in the array. Like any sort, it can be modified to sort in either ascending or descending order. An ascending sort 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 the smallest value, so it is swapped with the 5 stored in element 0. This completes the first pass and the array now appears 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. For the second pass, the algorithm begins the scan at element 1. It locates the smallest value in the unsorted part of the array, which is the 2 in element 2. Therefore, element 2 is exchanged with element 1. The array now appears 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 and will exchange this element's contents 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 = 0 to the next-to-last array subscript
    Set index to startScan
    Set minIndex to startScan
    Set minValue to array[startScan]
```

```
        For index = (startScan + 1) to the last subscript in the array
            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`. The parameter `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;
    }
}
```

As with bubble sort, selection sort uses a pair of nested loops, in this case two `for` loops. 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. For N pieces of data this requires N-1 passes.

Program 9-5 demonstrates the selection sort function in a complete program.

Program 9-5

```
// 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()
{
    Const int SIZE = 6;
    int values[SIZE] = {5, 7, 2, 8, 9, 1};

    cout << "The unsorted values are\n";
    showArray(values, SIZE);

    selectionSort(values, SIZE);

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

//****************************************************************
//                         selectionSort                         *
// This function performs an ascending-order selection sort on *
// array. The parameter elems holds 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];

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

```

(program continues)

Program 9-5 ***(continued)***

```
//*****************************************************************
//                       showArray                                *
// This function displays the contents of array. The parameter *
// elems holds the number of elements in the array.               *
//*****************************************************************
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
```

9.4 Demetris Leadership Center Case Study—Part 2

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 9-1 for a complete list of products, with product number, title, description, price and sales figures.)

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

Structures

In addition to the `ProdStruct` structure, described in Table 9-2, the program will need a `SalesStruct` structure to hold the product number and dollar sales amount of a product. Table 9-5 lists the members of the `SalesStruct` structure.

Table 9-5 Members of the `SalesStruct` Structure

MEMBER	DATA TYPE	DESCRIPTION
`id`	`int`	Product number of a product
`dollarAmt`	`double`	Dollar amount of sales for that product

Variables

Table 9-6 lists the variables the program will use.

Table 9-6 Variables Used in the DLC Sales Report Program

VARIABLE	DATA TYPE	DESCRIPTION
`NUM_PRODS`	`const int`	Number of products carried by DLC
`Product`	`ProdStruct`	Array of structures holding the product data described in Table 9-2. There is one array element for each product carried by DLC.
`sales`	`SalesStruct`	Array of structures holding the sales data described in Table 9-5. There is one array element for each product carried by DLC.

Modules

The program will consist of the functions listed in Table 9-7.

Table 9-7 Functions in the DLC Sales Report Program

FUNCTION	DESCRIPTION
`main`	The program's `main` function. It calls the program's other functions.
`calcSales`	Calculates each product's sales.
`sortBySales`	Sorts the `sales` array so the elements are ordered by `dollarAmt` from highest to lowest.
`showOrder`	Displays a list of the product numbers and their dollar sales amounts.
`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 its pseudocode:

```
Initialize numProds
Set up the product array and
   initialize it with all the product records
Call calcSales
call sortBySales
Call showOrder
Call showTotals
```

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 = 0 to the last array subscript
    Set sales[index].id to product[index].id
    Set sales[index].dollarAmt to
       product[index].price * product[index].sold
End For
```

The `sortBySales` Function

The `sortBySales` function is a modified version of the selection sort algorithm shown in Program 9-5. This version of the function accepts an array of `SalesStruct` elements, rather than an array of integers, and it sorts them in descending order based on the value of the `dollarAmt` structure member. Notice that when an array element needs to be moved, the entire structure can be moved together. It is not necessary to move each of its data members individually. Here is the pseudocode for the `sortBySales` function.

```
For startScan = 0 to the next-to-last subscript
    Set maxIndex to startScan
    Set maxValue to sales[startScan]
    For index = (startScan + 1) to the last array subscript
        If sales[index].dollarAmt is greater than maxValue.dollarAmt
            Set maxValue to sales[index]
            Set maxIndex to index
        End If
    End For
    Set sales[maxIndex] to sales[startScan]
    Set sales[startScan] to maxValue
End For
```

The `showOrder` Function

The `showOrder` function displays a heading and the sorted list of product numbers and their sales amounts. Here is its pseudocode:

```
Display heading
For index = 0 to the last array subscript
    Display sales[index].id
    Display sales[index].dollarAmt
End For
```

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 to 0
Set totalSales to 0.0
For index = 0 to the last array subscript
   Add product[index].sold to totalUnits
   Add sales[index].dollarAmt to totalSales
End For
Display totalUnits with appropriate heading
Display totalSales with appropriate heading
```

The Entire Program

Program 9-6 shows the program's source code.

Program 9-6

```
// This program produces a sales report for the Demetris
// Leadership Center. It uses an array of structures.
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct ProdStruct
{
   int    id;                 // Product number
   string title,              // Product title
          description;        // Product description
   double price;              // Product unit price
   int    sold;               // Units sold during the past 6 months

   // Default constructor for a ProdStruct structure
   ProdStruct()
   {  price = id = sold = 0;
      title = description = "";
   }

   // Constructor to set initial data values
   ProdStruct(int i, string t, string d, double p, int s)
   {  id = i;
      title = t;
      description = d;
      price = p;
      sold = s;
   }
};

```

(program continues)

Program 9-6 ***(continued)***

```
struct SalesStruct
{
    int     id;                 // Product number
    double  dollarAmt;          // Dollar amount of sales in past 6 months
};

// Function prototypes
void calcSales(ProdStruct[], SalesStruct [], int);
void sortBySales(SalesStruct [], int);
void showOrder(SalesStruct [], int);
void showTotals(ProdStruct[], SalesStruct [], int);

int main()
{
   const int NUM_PRODS = 9;    // Number of products carried
   ProdStruct product[NUM_PRODS] =
   {
     ProdStruct(914, "Six Steps to Leadership",   "Book",     12.95,  842),
     ProdStruct(915, "Six Steps to Leadership",   "Audio cassette",
                                                              14.95,  416),
     ProdStruct(916, "The Road to Excellence",    "Video",    18.95,  127),
     ProdStruct(917, "Seven Lessons of Quality", "Book",     16.95,  514),
     ProdStruct(918, "Seven Lessons of Quality", "Audio cassette",
                                                              21.95,  437),
     ProdStruct(919, "Seven Lessons of Quality", "Video",    31.95,  269),
     ProdStruct(920, "Teams are Made, Not Born", "Book",     14.95,   97),
     ProdStruct(921, "Leadership for the Future","Book",     14.95,  492),
     ProdStruct(922, "Leadership for the Future","Audio cassette",
                                                              16.95,  212)
    };
    SalesStruct sales[NUM_PRODS];

    calcSales(product, sales, NUM_PRODS);
    sortBySales(sales, NUM_PRODS);
    cout << fixed << showpoint << setprecision(2);
    showOrder(sales, NUM_PRODS);
    showTotals(product, sales, NUM_PRODS);
    return 0;
}

//**************************************************************
//                         calcSales                           *
// Passed in: the product array, the sales array, and the      *
//            size of the arrays                               *
//                                                             *
// This function uses data in the product array to get the     *
// product id and to calculate the product dollarAmt to be     *
// stored for each product in the sales array.                 *
//**************************************************************
```

(program continues)

9

Program 9-6 ***(continued)***

```
void calcSales(ProdStruct product[], SalesStruct sales[], int numProds)
{
    for (int index = 0; index < numProds; index++)
    {  sales[index].id = product[index].id;
       sales[index].dollarAmt = product[index].price * product[index].sold;
    }
}

//*************************************************************
//                          sortBySales                       *
// Passed in: the sales array and its size                    *
//                                                            *
// This function performs a selection sort, arranging array   *
// elements in descending-order based on the value of the     *
// dollarAmt structure member.                                *
//*************************************************************
void sortBySales(SalesStruct sales[], int elems)
{
    int startScan, maxIndex;
    SalesStruct maxValue;   // Holds structure with largest dollarAmt so far

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

//**************************************************************
//                          showOrder                          *
// Passed in: the sales array and its size                     *
//                                                             *
// This function displays the product number and dollar sales *
// amount of each product DLC sells.                           *
//**************************************************************
```

(program continues)

Program 9-6 ***(continued)***

```
void showOrder(SalesStruct sales[], int numProds)
{
    cout << "Product ID \t  Sales\n";
    cout << "----------------------------\n";
    for (int index = 0; index < numProds; index++)
    {
        cout << sales[index].id << "\t\t  $";
        cout << setw(8) << sales[index].dollarAmt << endl;
    }
    cout << endl;
}

//****************************************************************
//                         showTotals                             *
// Passed in: the product array, the sales array, and the        *
//            size of the arrays                                  *
//                                                                *
// This function calculates and displays the total quantity      *
// of items sold and the total dollar amount of sales.           *
//****************************************************************
void showTotals(ProdStruct product[], SalesStruct sales[], int numProds)
{
    int totalUnits = 0;
    double totalSales = 0.0;

    for (int index = 0; index < numProds; index++)
    {
        totalUnits += product[index].sold;
        totalSales += sales[index].dollarAmt;
    }
    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
```

Checkpoint

9.5 True or false: Any sort can be modified to sort in either ascending or descending order.

9.6 What one line of code would need to be modified in the bubble sort to make it sort in descending, rather than ascending order? How would the revised line be written?

9.7 After one pass of bubble sort, which value is in order?

9.8 After one pass of selection sort, which value is in order?

9.9 Which sort usually requires fewer data values to be swapped, bubble sort or selection sort?

9.5 Sorting and Searching Vectors

CONCEPT

The sorting and searching algorithms you have studied in this chapter can be applied to STL vectors as well as to 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 9-7 is an object-oriented version of Program 9-6 that uses STL vectors instead of arrays.

Program 9-7

```
// This program produces a sales report for the Demetris Leadership Center.
// This is an object-oriented version of the program that uses STL vectors
// instead of arrays and that reads the product data in from a file.
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <vector>                // Needed to use vectors
using namespace std;

struct ProdStruct
{
    int    id;                   // Product number
    string title,                // Product title
           description;          // Product description
    double price;                // Product unit price
    int    sold;                 // Units sold during the past 6 months

    // Default constructor for a ProdStruct structure
    ProdStruct()
    {  price = id = sold = 0;
       title = description = "";
    }
```

(program continues)

Program 9-7 ***(continued)***

```
    // Constructor to set initial data values
    ProdStruct(int i, string t, string d, double p, int s)
    {  id = i;
       title = t;
       description = d;
       price = p;
       sold = s;
    }
};

struct SalesStruct
{
    int     id;                // Product number
    double  dollarAmt;         // Dollar amount of sales in past 6 months
};

const int NUM_PRODS = 9;       // Number of products carried

class ProdMgr
{
private:
    // Declare vectors to hold product and sales records
    vector<ProdStruct> product;
    vector<SalesStruct> sales;

    // Private function prototypes
    void initVectors();
    void calcSales();
    void sortBySales();

public:
    // Default constructor
    ProdMgr()
    {  initVectors();
       calcSales();
       sortBySales();
       cout << fixed << showpoint << setprecision(2);
    }
    // Public function prototypes
    void showOrder();
    void showTotals();
};

```

(program continues)

Program 9-7 ***(continued)***

```
//*****************************************************************
//                    ProdMgr::initVectors                        *
// This function sets the size of the product and sales vectors   *
// and initializes the product vector with the product id, title, *
// description, price, and units sold for each item DLC sells.    *
//*****************************************************************
void ProdMgr::initVectors()
{
   product.resize(NUM_PRODS);
   sales.resize(NUM_PRODS);

   ifstream fileData;
   fileData.open("product.dat");
   if (!fileData)
      cout << "Error opening data file";
   else
   {  for (int index = 0; index < product.size(); index++)
      {
         fileData >> product[index].id;
         fileData.ignore();                          // Skip newline char
         getline(fileData, product[index].title);
         getline(fileData, product[index].description);
         fileData >> product[index].price;
         fileData >> product[index].sold;
      }
      fileData.close();
   }
}

//************************************************************
//                   ProdMgr::calcSales                      *
// This function uses data in the product vector to get the  *
// product id and to calculate the product dollarAmt to be   *
// stored for each product in the sales vector.              *
//************************************************************
void ProdMgr::calcSales()
{
   for (int index = 0; index < product.size(); index++)
   {  sales[index].id = product[index].id;
      sales[index].dollarAmt = product[index].price * product[index].sold;
   }
}

```

(program continues)

Program 9-7 ***(continued)***

```
//*************************************************************
//                     ProdMgr::sortBySales                   *
// This function performs a selection sort, arranging vector *
// elements in descending-order based on the value of the     *
// dollarAmt structure member.                                *
//*************************************************************
void ProdMgr::sortBySales()
{
   int startScan, maxIndex;
   int elems = sales.size();
   SalesStruct maxValue;  // Holds structure with largest dollarAmt so far

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

//**************************************************************
//                  ProdMgr::showOrder                         *
// This function displays the product number and dollar sales *
// amount of each product DLC sells.                           *
//**************************************************************
void ProdMgr::showOrder()
{
   cout << "Product ID \t  Sales\n";
   cout << "----------------------------\n";
   for (int index = 0; index < sales.size(); index++)
   {
      cout << sales[index].id << "\t\t  $";
      cout << setw(8) << sales[index].dollarAmt << endl;
   }
   cout << endl;
}

```

(program continues)

Program 9-7 ***(continued)***

```
//*****************************************************************
//                        ProdMgr::showTotals                     *
// This function calculates and displays the total quantity of *
// items sold and the total dollar amount of sales.               *
//*****************************************************************
void ProdMgr::showTotals()
{
   int totalUnits = 0;
   double totalSales = 0.0;

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

/***************************** CLIENT PROGRAM ************************/

int main()
{
   ProdMgr DLCsales;          // Create a ProdMgr object

   DLCsales.showOrder();
   DLCsales.showTotals();
   return 0;
}
```

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
```

Notice the similarities and differences between Program 9-7 and Program 9-6. The code in Program 9-7 that works with vectors is almost identical to the code in Program 9-6 that uses arrays. The differences lie in other things.

First, notice the addition of the `initVectors` function. In Program 9-6 this was not needed because the array sizes were declared when the arrays were created and the data to fill the `product` array was specified in an initialization list. However, vectors declared in a class declaration cannot be given an initial size. Declaring their size actually acquires memory to hold the stated number of starting elements, and this must not be done until after an object of the class has been created. Second, whether declared in a class or not, vectors do not accept initialization lists. Therefore, the needed vector elements are acquired in the `initVectors` function by using the `resize` vector member function. The `initVectors` function then fills the `product` vector with data read in from a file.

Second, notice that none of the class functions in Program 9-7 have parameters to receive the `product` or `sales` vectors. This is because these vectors are declared as members of the `ProdMgr` class and can therefore be directly accessed by all `ProdMgr` functions. If we were to write a non-object-oriented program that uses vectors, it would be necessary to pass the vectors to the functions that use them, as we did with the arrays in Program 9-6. Moreover, it would be necessary to use a reference parameter to pass any vector whose data was to be altered by the function. This was not necessary in Program 9-6 when passing arrays because arrays are always passed by reference, whereas vectors, by default, are passed by value.

Finally, notice that in the Program 9-7 the `calcSales`, `showOrder`, `sortBySales`, 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.

9.6 Additional Case Studies

The following case studies, which contain applications of material introduced in Chapter 9, can be found on the student CD.

Creating an Abstract Array Data Type—Part 2

The `IntList` class, begun as a case study in Chapter 8, is extended to include array searching and sorting capabilities.

Serendipity Booksellers Software Development Project—Part 9: *Adding Searching Capabilities*

In Part 9 of this ongoing project, you will add searching capabilities to the program so that the `addBook` function can locate an empty array element in which to place a new entry and the `lookUpBook`, `editBook`, and `deleteBook` functions can call a `findBook` function to locate a specific book they need to work with.

9.7 Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. The __________ search algorithm steps sequentially through an array, comparing each item with the search value.
2. The __________ search algorithm repeatedly divides the portion of an array being searched in half.
3. The __________ search algorithm is adequate for small arrays but not large arrays.
4. The __________ search algorithm requires that the array's contents be sorted.
5. The *average* number of comparisons performed by linear search to find an item in an array of N elements is __________.
6. The *maximum* number of comparisons performed by linear search to find an item in an array of N elements is __________.
7. A linear search will find the value it is looking for with just one comparison if that value is stored in the __________ array element.
8. A binary search will find the value it is looking for with just one comparison if that value is stored in the __________ array element.
9. In a binary search, after three comparisons have been made, only _________ of the array will be left to search.
10. The maximum number of comparisons that a binary search function will make when searching for a value in a 2,000-element array is _________.
11. If an array is sorted in _________ order, the values are stored from lowest to highest.
12. If an array is sorted in _________ order, the values are stored from highest to lowest.
13. Bubble sort places _________ number(s) in place on each pass through the data.
14. Selection sort places _________ number(s) in place on each pass through the data.
15. To sort N numbers, bubble sort continues making passes through the array until _________.
16. To sort N numbers, selection sort makes _________ passes through the data.
17. Why is selection sort more efficient than bubble sort on large arrays?
18. Which sort, bubble sort or selection sort, would require fewer passes to sort a set of data that is already in the desired order?
19. Complete the following table by calculating the average and maximum number of comparisons the linear search will perform, and the maximum number of comparisons the binary search will perform.

ARRAY SIZE →	50 ELEMENTS	500 ELEMENTS	10,000 ELEMENTS	100,000 ELEMENTS	10,000,000 ELEMENTS
Linear Search (Average Comparisons)					
Linear Search (Maximum Comparisons)					
Binary Search (Maximum Comparisons)					

Algorithm Workbench

20. Assume that `empName` and `empID` are two parallel arrays of size `numEmp` that hold employee data. Write a pseudocode algorithm that sorts the `empID` array in ascending ID number order (using any sort you wish), such that the two arrays remain parallel. That is, after sorting, for all indexes in the arrays, `empName[index]` must still be the name of the employee whose ID is in `empID[index]`.
21. Assume an array of structures is in order by the customerID field of the record, where customer IDs go from 101 to 500.
    A) Write the most efficient pseudocode algorithm you can to find the record with a specific customerID if every single customer ID from 101 to 500 is used and the array has 400 elements.
    B) Write the most efficient pseudocode algorithm you can to find a record with a customer ID near the end of the IDs, say 494, if not every single customer ID in the range of 101 to 500 is used and the array size is only 300.

Soft Skills

Deciding how to organize and access data is an important part of designing a program. You are already familiar with many structures and methods that allow you to organize data. These include one-dimensional arrays, vectors, multidimensional arrays, parallel arrays, structures, classes, arrays of structures, and arrays of class objects. You are also now familiar with some techniques for arranging (i.e., sorting) data and for locating (i.e., searching for) data items.

22. Team up with two to three other students and jointly decide how you would organize, order, and locate the data used in the following application. Be prepared to present your group's design to the rest of the class.

    The program to be developed is a menu-driven program that will keep track of parking tickets issued by the village that is hiring you. When a ticket is issued the program must be able to accept and store the following information: ticket number, officer number, vehicle license plate state and number, location, violation code (this indicates which parking law was violated), and date and time written. The program must store information on the amount of the fine associated with each violation code. When a ticket is paid the program must be able to accept and store the information that it has been paid, the amount of the payment, and the date the payment was received. The program must be able to accept inquiries such as displaying the entire ticket record when a ticket number is entered. The program must also be able to produce the following reports:

- A list of all tickets issued on a specific date, ordered by ticket number
- A list of all tickets for which payment was received on a specific date and the total amount of money collected that day
- A report of all tickets issued in a one-month period, ordered by officer number, with a count of how many tickets each officer wrote
- A report of all tickets that have not yet been paid, or for which payment received was less than payment due, ordered by vehicle license number

Programming Challenges

These programming challenges can all be written either with or without the use of classes. Your instructor will tell you which approach you should use.

myCodeMate

1. Charge Account Validation

Write a program that lets the user enter a charge account number. The program should determine if the number is valid by checking for it in the following list:

5658845	4520125	7895122	8777541	8451277	1302850
8080152	4562555	5552012	5050552	7825877	1250255
1005231	6545231	3852085	7576651	7881200	4581002

Initialize a one-dimensional array with these values. Then use a simple linear search to locate the number entered by the user. If the user enters a number that is in the array, the program should display a message saying the number is valid. If the user enters a number not in the array, the program should display a message indicating it is invalid.

2. Lottery Winners

A lottery ticket buyer purchases ten tickets a week, always playing the same ten 5-digit "lucky" combinations. Write a program that initializes an array with these numbers and then lets the player enter this week's winning 5-digit number. The program should perform a linear search through the list of the player's numbers and report whether or not one of the tickets is a winner this week. Here are the numbers:

```
13579    26791   26792    33445   55555
62483    77777   79422    85647   93121
```

3. Lottery Winners Modification

Modify the program you wrote for Programming Challenge 2 (Lottery Winners) so it performs a binary search instead of a linear search.

4. Annual Rainfall Report

Write a program that displays the name of each month in a year and its rainfall amount, sorted in order of rainfall from highest to lowest. The program should use an array of structures, where each structure holds the name of a month and its rainfall amount. Use a constructor to set the month names. Make the program modular by calling on different functions to input the rainfall amounts, to sort the data, and to display the data.

5. Hit the Slopes

Write a program that can be used by a ski resort to keep track of local snow conditions for one week. It should have a 7-element array of structures, where each structure holds a date and the number of inches of snow in the base on that date. The program should have the user input the name of the month, the starting and ending date of the 7-day period being measured, and then the seven base snow depths. The program should then sort the data in ascending order by base depth and display the results. Here is a sample report.

```
Snow Report December 12 - 18
    Date    Base
     13     42.3
     12     42.5
     14     42.8
     15     43.1
     18     43.1
     16     43.4
     17     43.8
```

6. String Selection Sort

Modify the `selectionSort` function presented in this chapter so it sorts an array of strings instead of an array of `int`s. Test the function with a driver program. Use Program 9-8 as a skeleton to complete.

Program 9-8

```
// Include needed header files here.

int main()
{
   const int SIZE = 20;

   string name[SIZE] =
   {"Collins, Bill",  "Smith, Bart",  "Michalski, Joe", "Griffin, Jim",
    "Sanchez, Manny", "Rubin, Sarah", "Taylor, Tyrone", "Johnson, Jill",
    "Allison, Jeff",  "Moreno, Juan", "Wolfe, Bill",    "Whitman, Jean",
    "Moretti, Bella", "Wu, Hong",     "Patel, Renee",   "Harrison, Rose",
    "Smith, Cathy",   "Conroy, Pat",  "Kelly, Sean",    "Holland, Beth"};

   // Insert your code to complete this program.
}
```

7. Binary String Search

Modify the `binarySearch` function presented in this chapter so it searches an array of strings instead of an array of `ints`. Test the function with a driver program. Use Program 9-8 as a skeleton to complete. (The array must be sorted before the binary search will work.)

8. Search Benchmarks

Write a program that has an array of at least 20 integers. It should call a function that uses the linear search algorithm to locate one of the values. The function should keep a count of the number of comparisons it makes until it finds the value. The program then should call a function that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.

9. Sorting Benchmarks

Write a program that uses two identical arrays of at least 20 integers. It should call a function that uses the bubble sort algorithm to sort one of the arrays in ascending order. The function should count the number of exchanges it makes. The program should then call a function that uses the selection sort algorithm to sort the other array. It should also count the number of exchanges it makes. Display these values on the screen.

10. Sorting Orders

Write a program that uses two identical arrays of just 8 integers. It should display the contents of the first array, then call a function to sort the array using an ascending order bubble sort modified to print out the array contents after each pass of the sort. Next the program should display the contents of the second array, then call a function to sort the array using an ascending order selection sort modified to print out the array contents after each pass of the sort.

11. Using Files—String Selection Sort Modification

Modify the program you wrote for Programming Challenge 6 so it reads in the 20 strings from a file. The data can be found in the `names.dat` file.

12. Using Vectors—String Selection Sort Modification

Modify the program you wrote for Programming Challenge 11 so it stores the names in a vector of strings, rather than in an array of strings. Create the vector without specifying a size and then use the `push_back` member function to add an element holding each string to the vector as it is read in from a file. Instead of assuming there are always 20 strings, read in the strings and add them to the vector until there is no data left in the file. The data can be found in the `names.dat` file.