Array Processing

C++ provides many ways to access arrays. If you have programmed in other computer languages, you will find that some of C++'s array indexing techniques are unique. Arrays in the C++ language are closely linked with *pointers*. Chapter 26, "Pointers," describes the many ways pointers and arrays interact. Because pointers are so powerful, and because learning about arrays provides a good foundation for learning about pointers, this chapter attempts to describe in detail how to reference arrays.

This chapter discusses the different types of array processing. You learn how to search an array for one or more values, find the highest and lowest values in an array, and sort an array into numerical or alphabetical order.

This chapter introduces the following concepts:

- Searching arrays
- Finding the highest and lowest values in arrays
- Sorting arrays
- Advanced subscripting with arrays

Many programmers see arrays as a turning point. Gaining an understanding of array processing makes your programs more accurate and allows for more powerful programming.

Searching Arrays

Arrays are one of the primary means by which data is stored in C++ programs. Many types of programs lend themselves to processing lists (arrays) of data, such as an employee payroll program, a scientific research of several chemicals, or customer account processing. As mentioned in the previous chapter, array data usually is read from a disk file. Later chapters describe disk file processing. For now, you should understand how to manipulate arrays so you see the data exactly the way you want to see it.

Array elements do not always appear in the order in which they are needed.

Chapter 23, "Introducing Arrays," showed how to print arrays in the same order that you entered the data. This is sometimes done, but it is not always the most appropriate method of looking at data.

For instance, suppose a high school used C++ programs for its grade reports. Suppose also that the school wanted to see a list of the top 10 grade-point averages. You could not print the first 10 grade-point averages in the list of student averages because the top 10 GPAs might not (and probably will not) appear as the first 10 array elements. Because the GPAs would not be in any sequence, the program would have to sort the array into numeric order, from high GPAs to low, or else search the array for the 10 highest GPAs.

You need a method for putting arrays in a specific order. This is called *sorting* an array. When you sort an array, you put that array in a specific order, such as in alphabetical or numerical order. A dictionary is in sorted order, and so is a phone book.

When you reverse the order of a sort, it is called a *descending sort*. For instance, if you wanted to look at a list of all employees in descending salary order, the highest-paid employees would be printed first.

Figure 24.1 shows a list of eight numbers in an array called `unsorted`. The middle list of numbers is an ascending sorted version of `unsorted`. The third list of numbers is a descending version of `unsorted`.

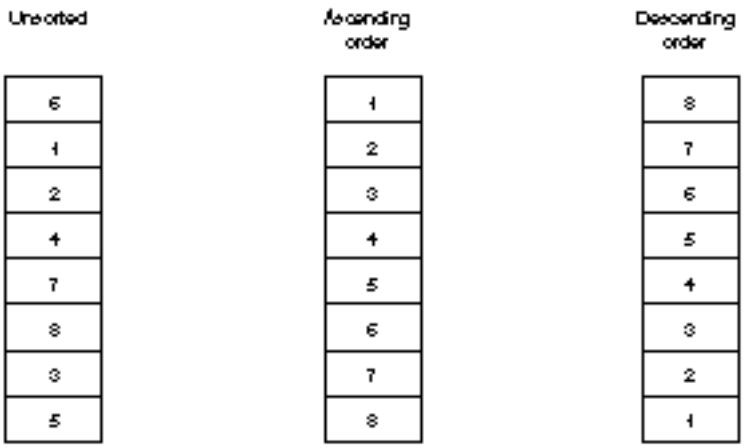


Figure 24.1. A list of unsorted numbers sorted into an ascending and a descending order.

Before you learn to sort, it would be helpful to learn how to search an array for a value. This is a preliminary step in learning to sort. What if one of those students received a grade change? The computer must be able to access that specific student's grade to change it (without affecting the others). As the next section shows, programs can search for specific array elements.

NOTE: C++ provides a method for sorting and searching lists of strings, but you will not understand how to do this until you learn about pointers, starting in Chapter 26, "Pointers." The sorting and searching examples and algorithms presented in this chapter demonstrate sorting and searching arrays of numbers. The same concepts will apply (and will actually be much more usable for "real-world" applications) when you learn how to store lists of names in C++.

Searching for Values

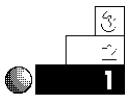
You do not have to know any new commands to search an array for a value. Basically, the `if` and `for` loop statements are all you need. To search an array for a specific value, look at each element in that array, and compare it to the `if` statement to see whether they match. If they do not, you keep searching down the array. If you run out of array elements before finding the value, it is not in the array.

You do not have to sort an array to find its extreme values.

You can perform several different kinds of searches. You might have to find the highest or the lowest value in a list of numbers. This is informative when you have much data and want to know the extremes of the data (such as the highest and lowest sales region in your division). You also can search an array to see whether it contains a matching value. For example, you can see whether an item is already in an inventory by searching a part number array for a match.

The following programs illustrate some of these array-searching techniques.

Examples



1. To find the highest number in an array, compare each element with the first one. If you find a higher value, it becomes the basis for the rest of the array. Continue until you reach the end of the array and you will have the highest value, as the following program shows.

Identify the program and include the I/O header file. You want to find the highest value in an array, so define the array size as a constant, then initialize the array.

Loop through the array, comparing each element to the highest value. If an element is higher than the highest value saved, store the element as the new high value. Print the highest value found in the array.

```
// Filename: C24HIGH.CPP
// Finds the highest value in the array.
#include <iostream.h>
const int SIZE = 15;
void main()
```

```
{
   // Puts some numbers in the array.
   int ara[SIZE]={5,2,7,8,36,4,2,86,11,43,22,12,45,6,85};
   int high_val, ctr;

   high_val = ara[0];           // Initializes with first
                                        // array element.
   for (ctr=1; ctr<SIZE; ctr++)
     {                  // Stores current value if it is
                        // the higher than the highest.
       if (ara[ctr] > high_val)
         { high_val = ara[ctr]; }
     }

   cout << "The highest number in the list is "
        << high_val << "\n";
   return;
}
```

The output of the program is the following:

```
The highest number in the list is 86.
```

You have to save the element if and only if it is higher than the one you are comparing. Finding the smallest number in an array is just as easy, except that you determine whether each succeeding array element is less than the lowest value found so far.

2. The following example expands on the previous one by finding the highest and the lowest value. First, store the first array element in *both* the highest and the lowest variable to begin the search. This ensures that each element after that one is tested to see whether it is higher or lower than the first.

   This example also uses the `rand()` function from Chapter 22, "Character, String, and Numeric Functions," to fill the array with random values from 0 to 99 by applying the modulus operator (`%`) and 100 against whatever value `rand()` produces. The program prints the entire array before starting the search for the highest and the lowest.


```
// Filename: C24HILO.CPP
// Finds the highest and the lowest value in the array.
#include <iostream.h>
#include <stdlib.h>
const int SIZE = 15;
void main()
{
   int ara[SIZE];
   int high_val, low_val, ctr;

   // Fills array with random numbers from 0 to 99.
   for (ctr=0; ctr<SIZE; ctr++)
     { ara[ctr] = rand() % 100; }

   // Prints the array to the screen.
   cout << "Here are the " << SIZE << " random numbers:\n";
   for (ctr=0; ctr<SIZE; ctr++)
     {  cout << ara[ctr] << "\n"; }

   cout << "\n\n";       // Prints a blank line.
   high_val = ara[0];    // Initializes first element to
                         // both high and low.
   low_val  = ara[0];

   for (ctr=1; ctr<SIZE; ctr++)
     {                 // Stores current value if it is
                       // higher than the highest.
       if (ara[ctr] > high_val)
         { high_val = ara[ctr]; }
       if (ara[ctr] < low_val)
         { low_val = ara[ctr]; }
     }

   cout << "The highest number in the list is " <<
           high_val << "\n";
   cout << "The lowest number in the list is " <<
           low_val << "\n";
   return;
}
```

Here is the output from this program:

```
Here are the 15 random numbers:
46
30
82
90
56
17
95
15
48
26
4
58
71
79
92
The highest number in the list is 95
The lowest number in the list is 4
```

3. The next program fills an array with part numbers from an inventory. You must use your imagination, because the inventory array normally would fill more of the array, be initialized from a disk file, and be part of a larger set of arrays that hold descriptions, quantities, costs, selling prices, and so on. For this example, assignment statements initialize the array. The important idea from this program is not the array initialization, but the method for searching the array.

NOTE: If the newly entered part number is already on file, the program tells the user. Otherwise, the part number is added to the end of the array.

```
// Filename: C24SERCH.CPP
// Searches a part number array for the input value. If
```

```
// the entered part number is not in the array, it is
// added. If the part number is in the array, a message
// is printed.
#include <iostream.h>
const int MAX = 100;
void fill_parts(long int parts[MAX]);

void main()
{
   long int search_part;            // Holds user request.
   long int parts[MAX];
   int ctr;
   int num_parts=5;          // Beginning inventory count.

   fill_parts(parts);    // Fills the first five elements.
   do
   {
     cout << "\n\nPlease type a part number...";
     cout << "(-9999 ends program) ";
     cin >> search_part;
     if (search_part == -9999)
       { break; }             // Exits loop if user wants.
     // Scans array to see whether part is in inventory.
     for (ctr=0; ctr<num_parts; ctr++) // Checks each item.
     { if (search_part == parts[ctr])       // If it is in
                                            // inventory...
          { cout << "\nPart " << search_part <<
                    " is already in inventory";
            break;
          }
       else
        { if (ctr == (num_parts-1) )      // If not there,
                                          // adds it.
          { parts[num_parts] = search_part;   // Adds to
                                               // end of array.
            num_parts++;
            cout << search_part <<
                 " was added to inventory\n";
```

```
            break;
             }
         }
         }
   } while (search_part != -9999);      // Loops until user
                                        // signals end.
   return;
}

void fill_parts(long int parts[MAX])
{
   // Assigns five part numbers to array for testing.
   parts[0] = 12345;
   parts[1] = 24724;
   parts[2] = 54154;
   parts[3] = 73496;
   parts[4] = 83925;
   return;
}
```

Here is the output from this program:

```
Please type a part number...(-9999 ends program) 34234
34234 was added to inventory

Please type a part number...(-9999 ends program) 83925

Part 83925 is already in inventory

Please type a part number...(-9999 ends program) 52786
52786 was added to inventory

Please type a part number...(-9999 ends program) -9999
```

Sorting Arrays

There are many times when you must sort one or more arrays. Suppose you were to take a list of numbers, write each number on a separate piece of paper, and throw all the pieces of paper into the air. The steps you take—shuffling and changing the order of the

pieces of paper and trying to put them in order—are similar to what your computer goes through to sort numbers or character data.

Because sorting arrays requires exchanging values of elements back and forth, it helps if you first learn the technique for swapping variables. Suppose you had two variables named `score1` and `score2`. What if you wanted to reverse their values (putting `score2` into the `score1` variable, and vice versa)? You could not do this:

```
score1 = score2;   // Does not swap the two values.
score2 = score1;
```

Why doesn't this work? In the first line, the value of `score1` is replaced with `score2`'s value. When the first line finishes, both `score1` and `score2` contain the same value. Therefore, the second line cannot work as desired.

To swap two variables, you have to use a third variable to hold the intermediate result. (This is the only function of this third variable.) For instance, to swap `score1` and `score2`, use a third variable (called `hold_score` in this code), as in

```
hold_score = score1;   // These three lines properly
score1 = score2;       // swap score1 and score2.
score2 = hold_score;
```

This exchanges the values in the two variables.

There are several different ways to sort arrays. These methods include the *bubble sort*, the *quicksort*, and the *shell sort*. The basic goal of each method is to compare each array element to another array element and swap them if the higher value is less than the other.

The lowest values in a list "float" to the top with the bubble sort algorithm.

The theory behind these sorts is beyond the scope of this book, however, the bubble sort is one of the easiest to understand. Values in the array are compared to each other, a pair at a time, and swapped if they are not in back-to-back order. The lowest value eventually "floats" to the top of the array, like a bubble in a glass of soda.

Figure 24.2 shows a list of numbers before, during, and after a bubble sort. The bubble sort steps through the array and compares pairs of numbers to determine whether they have to be swapped. Several passes might have to be made through the array before it is

finally sorted (no more passes are needed). Other types of sorts improve on the bubble sort. The bubble sort procedure is easy to program, but it is slower compared to many of the other methods.

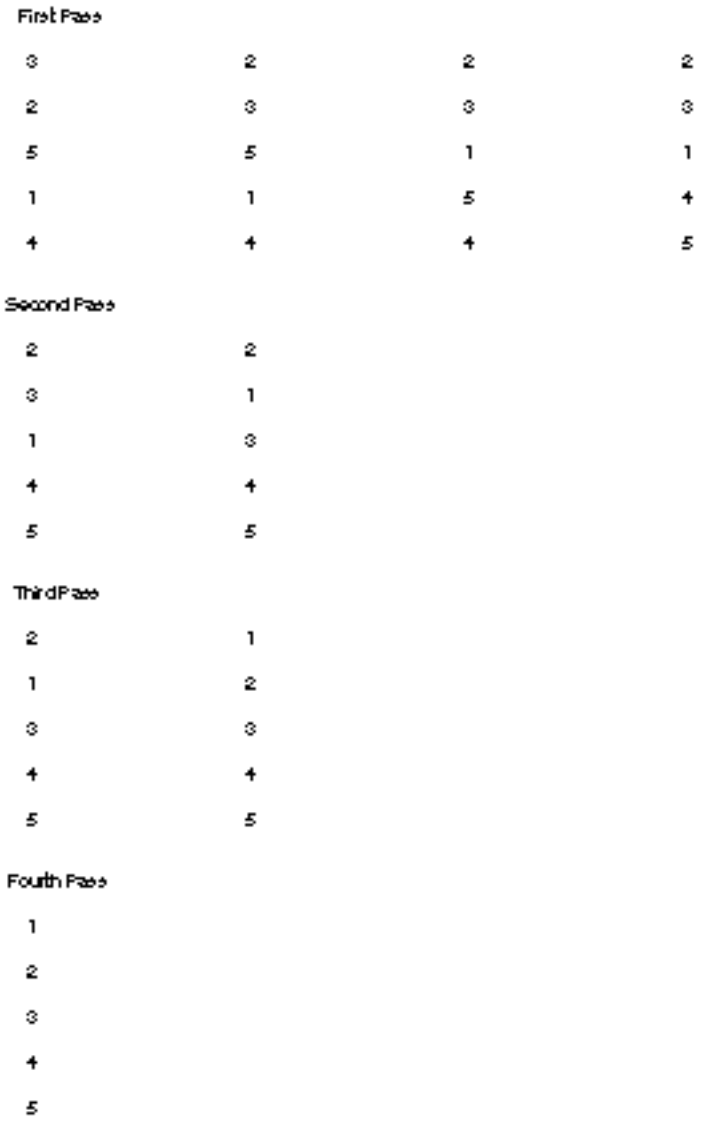


Figure 24.2. Sorting a list of numbers using the bubble sort.

The following programs show the bubble sort in action.

Examples

1. The following program assigns 10 random numbers between 0 and 99 to an array, then sorts the array. A nested `for` loop is perfect for sorting numbers in the array (as shown in the `sort_array()` function). Nested `for` loops provide a nice mechanism for working on pairs of values, swapping them if needed. As the outside loop counts down the list, referencing each element, the inside loop compares each of the remaining values to those array elements.

```
// Filename: C24SORT1.CPP
// Sorts and prints a list of numbers.
const int MAX = 10;
#include <iostream.h>
#include <stdlib.h>
void fill_array(int ara[MAX]);
void print_array(int ara[MAX]);
void sort_array(int ara[MAX]);

void main()
{
   int ara[MAX];

   fill_array(ara);    // Puts random numbers in the array.

   cout << "Here are the unsorted numbers:\n";
   print_array(ara);   // Prints the unsorted array.

   sort_array(ara);    // Sorts the array.

   cout << "\n\nHere are the sorted numbers:\n";
   print_array(ara);   // Prints the newly sorted array.
   return;
}

void fill_array(int ara[MAX])
{
```

```
   // Puts random numbers in the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { ara[ctr] = (rand() % 100); }   // Forces number to
                                       // 0-99 range.
   return;
}

void print_array(int ara[MAX])
{
   // Prints the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { cout << ara[ctr] << "\n"; }
   return;
}

void sort_array(int ara[MAX])
{
   // Sorts the array.
   int temp;            // Temporary variable to swap with
   int ctr1, ctr2;            // Need two loop counters to
                              //   swap pairs of numbers.
   for (ctr1=0; ctr1<(MAX-1); ctr1++)
      { for (ctr2=(ctr1+1); ctr2<MAX; ctr2++) // Test pairs.
           { if (ara[ctr1] > ara[ctr2])    // Swap if this
               { temp = ara[ctr1]; // pair is not in order.
                 ara[ctr1] = ara[ctr2];
                 ara[ctr2] = temp;   // "Float" the lowest
                                     // to the highest.
               }
           }
      }
   return;
}
```

The output from this program appears next. If any two randomly generated numbers were the same, the bubble sort would work properly, placing them next to each other in the list.

```
Here are the unsorted numbers:
46
30
82
90
56
17
95
15
48
26


Here are the sorted numbers:
15
17
26
30
46
48
56
82
90
95
```

To produce a descending sort, use the less-than (<) logical operator when swapping array elements.

2. The following program is just like the previous one, except it prints the list of numbers in descending order.

   A descending sort is as easy to write as an ascending sort. With the ascending sort (from low to high values), you compare pairs of values, testing to see whether the first is greater than the second. With a descending sort, you test to see whether the first is less than the second one.

```
// Filename: C24SORT2.CPP
// Sorts and prints a list of numbers in reverse
// and descending order.
const int MAX = 10;
#include <iostream.h>
#include <stdlib.h>
void fill_array(int ara[MAX]);
```

```
void print_array(int ara[MAX]);
void sort_array(int ara[MAX]);

void main()
{
   int ara[MAX];

   fill_array(ara);    // Puts random numbers in the array.

   cout << "Here are the unsorted numbers:\n";
   print_array(ara);          // Prints the unsorted array.

   sort_array(ara);                    // Sorts the array.

   cout << "\n\nHere are the sorted numbers:\n";
   print_array(ara);      // Prints the newly sorted array.
   return;
}

void fill_array(int ara[MAX])
{
   // Puts random numbers in the array.
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { ara[ctr] = (rand() % 100); }      // Forces number
                                          // to 0-99 range.
   return;
}

void print_array(int ara[MAX])
{
   // Prints the array
   int ctr;
   for (ctr=0; ctr<MAX; ctr++)
      { cout << ara[ctr] << "\n"; }
   return;
}

void sort_array(int ara[MAX])
{
   // Sorts the array.
   int temp;           // Temporary variable to swap with.
```

```
   int ctr1, ctr2;           // Need two loop counters
                             //   to swap pairs of numbers.
   for (ctr1=0; ctr1<(MAX-1); ctr1++)
      { for (ctr2=(ctr1+1); ctr2<MAX; ctr2++) // Test pairs
           // Notice the difference in descending (here)
           // and ascending.
           { if (ara[ctr1] < ara[ctr2]) // Swap if this
               { temp = ara[ctr1]; // pair is not in order.
                 ara[ctr1] = ara[ctr2];
                 ara[ctr2] = temp;  // "Float" the lowest
                                    // to the highest.
               }
            }
      }
   return;
}
```

TIP: You can save the previous programs' sort functions in two separate files named `sort_ascend` and `sort_descend`. When you must sort two different arrays, `#include` these files inside your own programs. Even better, compile each of these routines separately and link the one you need to your program. (You must check your compiler's manual to learn how to do this.)

You can sort character arrays just as easily as you sort numeric arrays. C++ uses the ASCII character set for its sorting comparisons. If you look at the ASCII table in Appendix C, you will see that numbers sort before letters and that uppercase letters sort before lowercase letters.

Advanced Referencing of Arrays

The array notation you have seen so far is common in computer programming languages. Most languages use subscripts inside brackets (or parentheses) to refer to individual array elements. For instance, you know the following array references describe the first

and fifth element of the array called sales (remember that the starting subscript is always 0):

```
sales[0]
sales[4]
```

C++ provides another approach to referencing arrays. Even though the title of this section includes the word "advanced," this array-referencing method is not difficult. It is very different, however, especially if you are familiar with another programming language's approach.

There is nothing wrong with referring to array elements in the manner you have seen so far, however, the second approach, unique to C and C++, will be helpful when you learn about pointers in upcoming chapters. Actually, C++ programmers who have programmed for several years rarely use the subscript notation you have seen.

An array name is the address of the starting element of the array.

In C++, an array's name is not just a label for you to use in programs. To C++, the array name is the actual address where the first element begins in memory. Suppose you define an array called amounts with the following statement:

```
int amounts[6] = {4, 1, 3, 7, 9, 2};
```

Figure 24.3 shows how this array is stored in memory. The figure shows the array beginning at address 405,332. (The actual addresses of variables are determined by the computer when you load and run your compiled program.) Notice that the name of the array, amounts, is located somewhere in memory and contains the address of amounts[0], or 405,332.

You can refer to an array by its regular subscript notation, or by modifying the address of the array. The following refer to the third element of amounts:

```
amounts[3] and (amounts + 3)[0]
```

Because C++ considers the array name to be an address in memory that contains the location of the first array element, nothing keeps you from using a different address as the starting address and referencing from there. Taking this one step further, each of the following also refers to the third element of amounts:

```
(amounts+0)[3] and (amounts+2)[1] and (amounts-2)[5]
(1+amounts)[2] and (3+amounts)[0] and (amounts+1)[2]
```

You can print any of these array elements with `cout`.

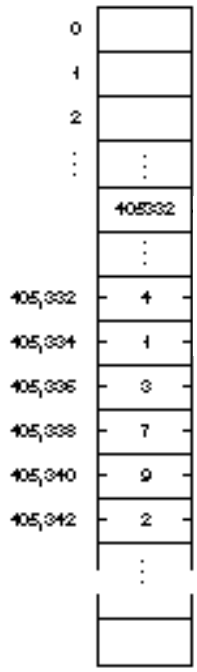


Figure 24.3. The array name `amounts` holds the address of `amounts[0]`.

When you print strings inside character arrays, referencing the arrays by their modified addresses is more useful than with integers. Suppose you stored three strings in a single character array. You could initialize this array with the following statement:

```
char names[]={'T','e','d','\0','E','v','a','\0','S','a','m','\0'};
```

Figure 24.4 shows how this array might look in memory. The array name, `names`, contains the address of the first element, `names[0]` (the letter *T*).

CAUTION: The hierarchy table in Appendix D, "C++ Precedence Table," shows that array subscripts have precedence over addition and subtraction. Therefore, you must enclose array names in parentheses if you want to modify the name as shown in these examples. The following are not equivalent:

`(2+amounts)[1]` and `2+amounts[1]`

The first example refers to `amounts[3]` (which is 7). The second example takes the value of `amounts[1]` (which is 1 in this example array) and adds 2 to it (resulting in a value of 3).

This second method of array referencing might seem like more trouble than it is worth, but learning to reference arrays in this fashion will make your transition to pointers much easier. An array name is actually a pointer, because the array contains the address of the first array element (it "points" to the start of the array).

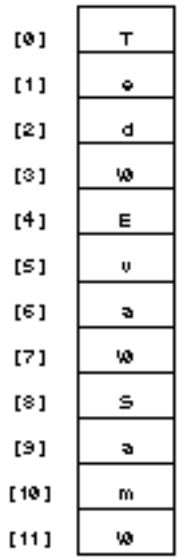


Figure 24.4. Storing more than one string in a single character array.

You have yet to see a character array that holds more than one string, but C++ allows it. The problem with such an array is how you reference, and especially how you print, the second and third strings. If you were to print this array using cout:

```
cout << names;
```

C++ would print the following:

```
Ted
```

cout prints strings in arrays starting at the array's address and continuing until it reaches the null zero.

Because cout requires a starting address, you can print the three strings with the following couts:

```
cout << names;       // Prints Ted
cout << (names+4);   // Prints Eva
cout << (names+8);   // Prints Sam
```

To test your understanding, what do the following couts print?

```
cout << (names+1);
cout << (names+6);
```

The first cout prints ed. The characters ed begin at (names+1) and the cout stops printing when it reaches the null zero. The second cout prints a. Adding six to the address at names produces the address where the a is located. The "string" is only one character long because the null zero appears in the array immediately after the a.

To sum up character arrays, the following refer to individual array elements (single characters):

```
names[2] and (names+1)[1]
```

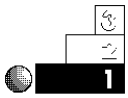
The following refer to addresses only, and as such, you can print the full strings with cout:

```
names and (names+4)
```

CAUTION: Never use the printf()'s %c control code to print an address reference, even if that address contains a character. Print strings by specifying an address with %s, and single characters by specifying the character element with %c.

The following examples are a little different from most you have seen. They do not perform "real-world" work, but were designed as study examples for you to familiarize yourself with this new method of array referencing. The next few chapters expand on these methods.

Examples



1. The following program stores the numbers from 100 to 600 in an array, then prints elements using the new method of array subscripting.

```
// Filename: C24REF1.CPP
// Print elements of an integer array in different ways.
#include <iostream.h>
void main()
{
   int num[6] = {100, 200, 300, 400, 500, 600};

   cout << "num[0] is \t" << num[0] << "\n";
   cout << "(num+0)[0] is \t" << (num+0)[0] << "\n";
   cout << "(num-2)[2] is \t" << (num-2)[2] << "\n\n";

   cout << "num[1] is \t" << num[1] << "\n";
   cout << "(num+1)[0] is \t" << (num+1)[0] << "\n\n";

   cout << "num[5] is \t" << num[5] << "\n";
   cout << "(num+5)[0] is \t" << (num+5)[0] << "\n";
   cout << "(num+2)[3] is \t" << (num+2)[3] << "\n\n";

   cout << "(3+num)[1] is \t" << (3+num)[1] << "\n";
   cout << "3+num[1] is \t" << 3+num[1] << "\n";
   return;
}
```

Here is the output of this program:

```
num[0] is       100
(num+0)[0] is   100
(num-2)[2] is   100
```

```
num[1] is       200
(num+1)[0] is   200

num[5] is       600
(num+5)[0] is   600
(num+2)[3] is   600

(3+num)[1] is   500
3+num[1] is     203
```

2. The following program prints strings and characters from a character array. The `cout`s all print properly.

```
// Filename: C24REF2.CPP
// Prints elements and strings from an array.
#include <iostream.h>
void main()
{
   char names[]={'T','e','d','\0','E','v','a','\0',
                 'S', 'a','m','\0'};

   // Must use extra percent (%) to print %s and %c.
   cout << "names " << names << "\n";
   cout << "names+0 " << names+0 << "\n";
   cout << "names+1 " << names+1 << "\n";
   cout << "names+2 " << names+2 << "\n";
   cout << "names+3 " << names+3 << "\n";
   cout << "names+5 " << names+5 << "\n";
   cout << "names+8 " << names+8 << "\n\n";

   cout << "(names+0)[0] " << (names+0)[0] << "\n";
   cout << "(names+0)[1] " << (names+0)[1] << "\n";
   cout << "(names+0)[2] " << (names+0)[2] << "\n";
   cout << "(names+0)[3] " << (names+0)[3] << "\n";
   cout << "(names+0)[4] " << (names+0)[4] << "\n";
   cout << "(names+0)[5] " << (names+0)[5] << "\n\n";

   cout << "(names+2)[0] " << (names+2)[0] << "\n";
   cout << "(names+2)[1] " << (names+2)[1] << "\n";
   cout << "(names+1)[4] " << (names+1)[4] << "\n\n";

   return;
}
```

Study the output shown below by comparing it to the program. You will learn more about strings, characters, and character array referencing from studying this one example than from 20 pages of textual description.

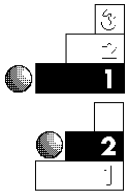
```
names Ted
names+0 Ted
names+1 ed
names+2 d
names+3
names+5 va
names+8 Sam

(names+0)[0] T
(names+0)[1] e
(names+0)[2] d
(names+0)[3]
(names+0)[4] E
(names+0)[5] v

(names+2)[0] d
(names+2)[1]
(names+1)[4] v
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: You must access an array in the same order you initialized it.
2. Where did the bubble sort get its name?
3. Are the following values sorted in ascending or descending order?

   33 55 78 78 90 102 435 859 976 4092

4. How does C++ use the name of an array?

5. Given the following array definition:

```
char teams[] = {'E','a','g','l','e','s','\0',
                'R', 'a','m','s','\0'};
```

What is printed with each of these statements? (Answer "invalid" if the `cout` is illegal.)

a. `cout << teams;`

b. `cout << teams+7;`

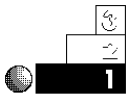
c. `cout << (teams+3);`

d. `cout << teams[0];`

e. `cout << (teams+0)[0];`

f. `cout << (teams+5);`

Review Exercises



1. Write a program to store six of your friends' ages in a single array. Assign the ages in random order. Print the ages, from low to high, on-screen.

2. Modify the program in Exercise 1 to print the ages in descending order.

3. Using the new approach of subscripting arrays, rewrite the programs in Exercises 1 and 2. Always put a `0` in the subscript brackets, modifying the address instead (use `(ages+3)[0]` rather than `ages[3]`).

4. Sometimes *parallel arrays* are used in programs that must track more than one list of values that are related. For instance, suppose you had to maintain an inventory, tracking the integer part numbers, prices, and quantities of each item. This would require three arrays: an integer part number array, a floating-point price array, and an integer quantity array. Each array would have the same number of elements (the total number of parts in the inventory). Write a program to maintain such an inventory, and reserve enough elements

for 100 parts in the inventory. Present the user with an input screen. When the user enters a part number, search the part number array. When you locate the position of the part, print the corresponding price and quantity. If the part does not exist, enable the user to add it to the inventory, along with the matching price and quantity.

Summary

You are beginning to see the true power of programming languages. Arrays give you the ability to search and sort lists of values. Sorting and searching are what computers do best; computers can quickly scan through hundreds and even thousands of values, looking for a match. Scanning through files of paper by hand, looking for just the right number, takes much more time. By stepping through arrays, your program can quickly scan, print, sort, and calculate a list of values. You now have the tools to sort lists of numbers, as well as search for values in a list.

You will use the concepts learned here for sorting and searching lists of character string data as well, when you learn a little more about the way C++ manipulates strings and pointers. To help build a solid foundation for this and more advanced material, you now know how to reference array elements without using conventional subscripts.

Now that you have mastered this chapter, the next one will be easy. Chapter 25, “Multidimensional Arrays,” shows you how you can keep track of arrays in a different format called a *matrix*. Not all lists of data lend themselves to matrices, but you should be prepared for when you need them.