Lab #9: An Array of Numbers

Overview: In today's lab we will practice implementing one dimensional arrays of fundamental objects. Arrays are used extensively in many different applications, and it is important that you are comfortable with creating and manipulating the elements in an array structure.

Objectives: Before you leave lab today you should know how to create a one dimensional array of fundamental objects by initializing the array in the program or by reading values into the array from a file. You will also know how to write a loop that processes the elements of an array. You should understand how the insertion sort algorithm works for sorting an array, and you will be able to sequentially search an array for a value.

1. Setting up for the activity

Please perform the following activities as you wait for class to begin:

a. Boot up the machine and log on to the CS Department server.

b. Put your lab diskette in the *A*: drive and make a subdirectory *LAB09* in the *LABS* directory.

c. Download the files from the class Web site (http://s9000.furman.edu/chealy/cs11/lab09) to your new directory.

2. Create an array and initialize array elements

We will be using an array of integers throughout this activity. Remember that an array data structure allows you to store objects of the same type in memory. You can then access items of the array using an index into the array. In this activity, you will be using the array **squares** which should hold the squares of the integers 0-9.

a. Start up Borland C++. Open the program file *sq1.cpp*. This program file has a function that you can use to display the elements of an array on the screen. Look through main() and notice that you are to create and initialize an array and then display the array to the screen.

Initializing array elements when declaring the array

One technique that you can use to initialize the elements of an array is to use the following syntax when creating the array: type ArrayName [size] = {elem1,elem2,...};.

b. Using this technique, create the array **squares** as indicated in the comments and initialize the elements to {0,1,4,9,16,25,36,49,64,81}.

c. Build and run the program and be sure that your array is displayed properly.

Initializing array elements from a file

A more useful technique for initializing array elements, especially for large arrays, is to read the values from a file. The advantage of this approach over the first is that it is easier to change one of the initial values in the array. You don't need to change the program file and recompile to change an initial value in the array, but you simply need to change the data file holding the values.

d. Save *sq1.cpp* as *sq2.cpp* and modify *sq2.cpp* so that the array **squares** is not initialized in the declaration statement (i.e. remove the *={0,1,4,9,16,25,36,49,64,81}*). Add a function **set_array** which will read the values from the file *sq.dat*. You may assume that this file contains ten integers and nothing else. Functions cannot return arrays using the *return* statement, so you should send the array **squares** as a parameter. Whenever arrays are sent as parameters, they are always reference parameters (and you do not use &). Simply put empty [] for the size of the formal array parameter in the function interface. For example, *void set_array (int A[])* is the proper interface for a function which can change the values of array A. A proper function call would look like *set_array(squares).*
e. Build and run your program. Your output should be identical to the first approach. √

Initializing array elements by computing values in program

Another popular technique for initializing array elements is to compute the values in the program. Notice that in the array **squares**, that each element can be easily computed from the index for the element. For example, **squares[3]**=3*3 or 9.

f. Save *sq2.cpp* as *sq3.cpp*. Modify the **set_array** function in *sq3.cpp* so that the array elements in **squares** are initialized in a for loop which computes each value.

g. Build and run your program. Your output should be identical to the previous approaches. √

3. Processing an array

We will now practice processing an array in a program when you don't necessarily know how many elements are in the array. You must specify the maximum number of possible elements when you declare the array, but you do not have to actually use all of those elements. Two very common processing tasks are *sorting* array elements in ascending or descending order and *searching* for array elements once sorted. There are many different possible sorting and searching algorithms. You will explore these algorithms and their effectiveness in upcoming courses. For now, we will practice with a simple *insertion sort* algorithm, and you will write your own simple *sequential search* function.

The insertion sort algorithm processes the array elements sequentially (i.e. index 0 through index n). At any point in the algorithm the array elements to the left of (or above) the current element (say index c) are sorted. The objective is to shift sorted array elements to the right until the correct location for element c is found and then insert c into that position.

a. Close your open windows in Borland C++, and then open *sortex.cpp*. Read through the code carefully, concentrating on the **sort** function. Be sure you understand how this function is operating.

b. You have a file called *nums.dat* in your directory. Implement the **input_numbers** function so that it reads *nums.dat* and places each number read into the array. Your implementation should count the number of integers read and return that counter in the reference parameter **n**. Be sure that you return the number of integers read and not the index of the last number read. Do you understand the difference?

c. Build and run your program. You should see the sorted array displayed on your screen. Look at the file *nums.dat* to be sure that you have the correct number of integers in your array. If you are missing a value in your output, you have probably implemented **input_numbers** incorrectly. Try again!

You have successfully read an array from a file and sorted it using the insertion sort algorithm. Now let's see if you can write some code to search the sorted array for a particular number. Again, we will use a simple search algorithm called sequential search. This algorithm simply starts looking for the number at the beginning of the array and continues traversing the array until the number is found or a number larger than the number being looked for is encountered. Remember the array is sorted so once we find a larger number we know that we can stop searching.

d. Remove the slashes from *sortex.cpp* which comment out the module for searching. Implement the function **search** which can search an array of any size using the sequential search algorithm.

e. Build and run your program again. Test the program thoroughly. Can you find the first element in the array? the last element? What happens if you search for a number that is not in the array, or bigger than any other number in the array? √

f. Close all windows and exit Borland C++.