Object Oriented Software Development

Arrays

**Synopsis**

This laboratory session is intended enable you to:

* Learn to declare, initialize, and use arrays
* Learn to send array elements to a function
* Learn to use partially filled arrays
* Learn to use arrays in a class
* Learn to define array of objects, and
* Learn about two dimensional arrays

By now you know how to write a C++ program to find the minimum and the maximum of a given set of numbers that are either read from a file or are entered from the keyboard.  In this case, you do not need to remember all the numbers and you can read the numbers one-by-one and compare each one with the max or min and replace the min and/or max with the number in hand, if needed, until the last number is read.  But there are cases where you need to keep track of numbers that are entered or read.  For example, think of these questions.

How do you write a set of 4 numbers in the reverse order?   
How will you write a set of 4 numbers in ascending or descending order?   
How do you find out how many occurrences of a number you have had in a list?

As you may immediately realize, now you must remember all the numbers to be able to write them in a specific order.

Suppose you were asked to write a C++ program to read 4 integer values from the keyboard then find the largest (max) and to display the list of all numbers on one column and their difference with the max on the second column.

Here is an example:

Input: 3 4 5 8   
Output:   
Max = 8

Num    Diff\_from\_Max   
3            5   
4            4   
5            3   
8            0

How would you do this?

Suppose I tell you today that you can declare 4 integers x[0], x[1], x[2], and x[3] using:

int x[4];

The above statement will create x[0], x[1], x[2], and x[3].  Use these instead of a,b,c, and d (or any other variables that you have used in earlier problems) to solve the problem.

Take a guess, how do we define a set of 10 characters using the same method?

**Introduction to Arrays**

So far all our variables have could hold only one value at any one point in time. Such variables are called *scalar* variables. Now it is time for our first non-scalar variable, an *array*.

An array is a variable capable of storing multiple values. When we declare an array we tell the compiler how many values we want the array to hold. We also tell the compiler what type of values the array can store. All of the values in an array must be of the same type.

Here is a declaration of an array called *numlist* that will be used to store 8 integers:

int numlist[8];  // declaring an integer array that can store 8 values

Each of the integers in the array is stored in the same number of bytes as a scalar integer, which on most machines is 4 bytes. Thus, the entire array will occupy 32 bytes of memory. The compiler always stores an array in contiguous memory locations (all of the elements of the array are stored in one chunk of memory with no gaps).  Here is one way you may visualize the above array numlist when it stores the following 8 integers: 12,  8, 10, 123, 1000, 23, 4, 10

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Imaginary**  **Memory**  **Address** | 1023 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 |
| **Array Index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Indexed**  ***numlist* Variable** | numlist  [0] | numlist  [1] | numlist  [2] | numlist  [3] | numlist  [4] | numlist  [5] | numlist  [6] | numlist  [7] |
| **Array**  **Content** | 12 | 8 | 10 | 123 | 1000 | 23 | 4 | 10 |

The individual values stored in an array are called the *elements* of the array. You will also hear them called *indexed variables* or *subscripted variables*. Each of the elements of an array is assigned an *index*. An index is a natural number in the range {0,1,2,...}.  Note that the array index started from 0.

As it is shown in the above table, to access one of the elements of an array, you put the index of that element in square brackets after the name of the array. The 0th element in the array called numlist is numlist[0], the next one is numlist[1], and so forth. Since we start the numbering with 0, the last element in numlist is numlist[7].

To put the value of 12 into the 0th element of numlist, we will use:

numlist[0] = 12;

If we wanted to store a value that is entered from the keyboard into element numlist[1] we use:

cin >> numlist[1];

An array element like numlist[4] can be used in any way that a scalar variable can be used. All of the following statements are legal:

if(numlist[2] > numlist[1]) // Compares the third element of

// the array with the second

// element of the array

cout << numlist[5];         // Displays the sixth element of

// the array

sum = sum + numlist[7];     // Adds the 8th element to sum

The index inside the square brackets does not have to be an integer constant such as 3 or 4.  It can be any integral expression that evaluates to an integer within the permitted range of the array's index. So, an expression such as this:

for(i = 0; i < 3; i++)   
     numlist[2\*i+1] = 0;  // set the odd elements of the array

// to 0

If you wish to fill the array numlist with the integers typed from the keyboard, you can use a *for loop* too. Here is a *for loop* that will allow you to enter 8 values from the keyboard and will store them in the array numlist. Notice that we have used the variable i as an index for array numlist.

for (i=0; i<8; ++i)

{

        cout << "Enter value #" << i+1 << ": ";

        cin >> numlist[i];

}

By asking for value 1, then value 2, etc., we are allowing our user to count in a more natural way than C++ forces us to count.  That is the most confusing part of working with arrays.  It is natural to think that an array of size 8 will keep 8 values, thus, assuming that the indices would be 1 through 8.  For an array of size 8, index 1 is a valid index, but index 8 is invalid and will cause a run-time error if it is used.

The following program allows you to enter 8 integers from the keyboard and will store those values in array numlist.

**Questions**

1. Locate c7e1-1 and type the program into an editor. Save this as c7e1.cpp then compile and run.

**Array Index Out of Range**

A common error when working with an array is attempting to access an element of the array using an index that is out of range. In the above program, array numlist has 8 elements. The final value is called numlist[7]. If we try to access numlist[8], most C++ compilers will not give us an error message at run time. C++ does not verify that your index is within the proper range when you compile the program. If you print the value of numlist[8], the compiler will grab the 4 bytes following numlist[7], interpret whatever is stored there as an integer and print the value of that integer. If you store a value at numlist[8], the compiler will place that value into the first 4 bytes following numlist[7], even if those bytes happen to be storing a different variable! That is what happens in the following program.

#include <iostream>   
using namespace std;   
int main(void)   
{   
  int numlist[8], i;

  cout << " \t i \t numlist[i] \n";   
  cout << " \t ===== \t ======== \n";

  for (i = 0; i <= 8; i++)   
  {   
     numlist[i] = i\*2;   
     cout << " \t " << i << "\t " << numlist[i] << endl;   
  }   
 return 0;   
}

Here is the output of this program:

i       numlist[i]   
        =====    ========   
         0             0   
         1             2   
         2             4   
         3             6   
         4             8   
         5            10   
         6            12   
         7            14   
         16          1      // Do you see any thing wrong

// here?

**Initializing Arrays**

A scalar variable can be initialized when it is declared, like this:

int num = 4;

An array can also be initialized when it is declared. Here we put the value 12 into numlist[0], the value 8 into numlist[1], etc.:

int numlist[8] = {12,  8, 10, 123, 1000, 23, 4, 10};

If you list fewer values within the braces ‘{‘ and ‘}’ than the **declared size** (8 in the above example) of the array, our C++ compiler will initialize all the rest of the elements to 0. However, not all C++ compilers will do this. If you initialize an array when it is declared, you can omit the size of the array. C++ will use the number of initializers in your list as the size of the array. Here is an example:

char vowels[] = {'a', 'e', 'i', 'o', 'u'};

This declared a character array of size 5 which stores the lowercase vowels, *a, e, i, o,* and *u*.

**Creating Arrays of Flexible Size**

One way to create an array with a particular size is to use a global variable to define the size of that array. Thus, every time one can change that number to increase or decrease the size of an array.  This requires you to recompile the program for the new size to take effect.

**Questions**

1. Locate c7e2-1 and type the program into an editor. Save this as c7e2.cpp then compile and run.

This produces the same result as c7e1.cpp.  Now, you are limited to array of 8 integers.  By changing the value for SIZE, you can read as many numbers as you wish.

1. Modify program c7e2.cpp such that it reads some number of integers as defined by SIZE, stored them in array numlist, displays the array numlist, then reverses the contents of the array, and at last displays the contents of that array again.

Make your program as general as possible.  Thus, your program should be able to reverse the contents of an array of any size defined by SIZE.  Note that we didn't ask you to display the array in reverse.  We want you to reverse the contents of the array, then display the array itself.  Example:

int numlist = {1, 2, 4, 5, 8, 2, 0, 9};

After you reverse the contents of array numlist, that array would become: {9, 0, 2, 8, 5, 4, 2, 1}. So, you will display the array numlist before you reverse the content and after you reversed the contents.

**Arrays in Functions**

You can use both the array index variables and the entire array itself as arguments to functions.  The following is perfectly valid:

double x[7] = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5};   
double y[7];   
int i;

for (i = 0; i<7; ++i)   
        y[i] = sin(x[i]);

Here we send the value of x[i] (as one single value) to the sine function.  The function's parameter, in this case, is a call-by-value parameter.  When you deal with a single variable from the array, everything is seen as a single variable.  Thus, if one wants to update a value for the 5th element of an array in a different function, he/she has to use the call\_by\_reference.

**Questions**

1. Locate c7e3-1 and type the program into an editor. Save this as c7e3.cpp then compile and run.

Here, we update the elements of array grade, one-by-one. Thus, we have used a call-by-reference mechanism to update the values.  Note that there is a major difference between the grade in the function call and the one in the function definition.  In the statement:

 get\_grade(grades[i]);

grades is the arrays grades[] and we are referring to its ith element, while in statement:

void get\_grade(int& grade)

grade is a single integer.

We could use the entire array as an argument too. An example is given next.

**Questions**

1. Locate c7e4-1 and type the program into an editor. Save this as c7e4.cpp then compile and run.

Note that we no longer need to pass the array as call-by-reference, no need for &, because once you call the function as:

get\_grade(grades); // can read, modify all elements

You have passed the address of the first element of the array, i.e., the address of grade[0],  to the function get\_grade.  The declaration of the array in the get\_grade definition is:

void get\_grade(int grade[])

which means, an array of some size that starts from the address that was passed to it from the function call, i.e. an array of some consecutive addresses starting from the address of grade [0].  Thus, if in function get\_grade you use an array of 5 elements, the memory address of that array will be the same as the array grades that was defined in the main.

**Using const with an Array Parameter**

If you place the reserved word **const** in front of an array parameter, the function cannot change the value of any of the elements of the array. An attempt to do so, will result in a syntax error.  This is a good idea if you are writing a function that will be used by other programmers and you want to make it clear to them that your function will not make any changes to the array that is passed to it. On the other hand, sometimes we want to make changes in the array.  For example, in the above program if we had used:

void get\_grade(const int grade[])

Then, we would be unable to update the elements of the array.

**Partially Filled Arrays**

Sometimes we don't know in advance how many values we will store in an array. In such a case, we declare the array to be of a size large enough for storing the maximum number of values we may have. Then, as we fill the array, we count the number of elements that get filled and use that count as the used size of the array.  Soon you will see an example of a partially filled array. First, let's describe a program that requires using this example.

Suppose we want to write a program to add very large integers. Our C++ compiler integer data type is stored in 4 bytes. This means that the largest integer we can store in an integer variable is approximately 2 billion. If we need larger integers, we will have to develop a new way of storing them.  One way to store a large integer is to store the digits of the integer in a character array. Let's say that we will never need integers with more than 20 digits. Then we can declare a character array of size 20 and be sure that any of our large integers will fit into it. As we fill the array with digit characters, we will count the digits so that we know how many elements are currently filled. Here is the example:

// An example for partially filled arrays   
// Saving very large integers as array of characters

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

const int MAXSIZE = 20;

int main(void)   
{   
        char digit\_array[MAXSIZE], digit;   
        int size, i;

        size = 0;   
        cout<< "Enter an integer with no more than 20 digits:

";   
        do {   
                cin.get(digit);   
                if (isdigit(digit))   
                {   
                        digit\_array[size] = digit;   
                        ++size;   
                }   
        } while (size < 20 && isdigit(digit));

        cout<< "The integer you entered is: ";   
        for (i = 0; i<size; ++i)   
                cout << digit\_array[i];   
        cout<< endl;   
        return 0;   
}

**Partially Filled Arrays as Function Argument**

Sometimes you have a partially filled array and you want to pass that to a function as an argument.  In such a case, it is best to pass the used size of the array as an argument as well.  For example, in the above program suppose we want to use a function that does the reading and another one that does the writing.  So we will have the read\_array and write\_array functions with two arguments.  Here is the modified version of the program.

// An example for partially filled arrays   
// Saving very large integers as array of characters   
#include <iostream>   
#include <ctype>   
using namespace std;

const int MAXSIZE = 20;   
void read\_array(char d\_array[], int& size); // array will be modified   
void write\_array(const char d\_array[], int size); // array will not be modified

int main(void)   
{   
        char digit\_array[MAXSIZE];   
        int size, i;

        size = 0;   
        cout<< "Enter an integer with no more than 20 digits:

";   
        read\_array(digit\_array, size);   
        write\_array(digit\_array, size);

        return 0;   
}

void read\_array(char d\_array[], int& size)   
{   
       char digit;   
       do {   
                cin.get(digit);   
                if (isdigit(digit))   
                {   
                        d\_array[size] = digit;   
                        ++size;   
                }   
        } while (size < 20 && isdigit(digit));   
}

void write\_array(const char d\_array[], int size)   
{   
        cout<< "The integers you entered are: ";   
        for (int i = 0; i<size; ++i)   
                cout << d\_array[i];   
        cout<< endl;   
}

**Questions**

1. Create a new program called c7e5.cpp that asks users to input up to 20 integers, and then finds the maximum, minimum, average, and median of the numbers that were entered.

Median is the number that appears in the middle of the sorted list of numbers.  If the array has an odd number of elements, median is a single number in the middle of the list (array).  If the array has an even number of elements, then median is the average of the two numbers in the middle.

Example:

Odd number of elements:   1 4 6 3 8, first sort the numbers: 1 3 4 6 8, then find the median,

i.e, 4.

Even number of elements:   1 4 8 3, first sort the numbers:  1 3 4 8, then find the median as

the average of 3 and 4, i.e., 3.5

To find the smallest element in an array, you only need to find the index of the smallest number.  You can use the following function to do this.  This function is also used in the *sort\_array* function.

int index\_of\_smallest(const int a[], int start\_index, int user\_size)   
{   
      int min = a[start\_index], index\_of\_min = start\_index;   
     for(int i = start\_index+1; i < used\_size; i++)   
          if(a[i] < min )   
          {   
                min = a[i];   
                index\_of\_min = i;   
           }   
           return index\_of\_min;   
}

To sort an array that has *used\_size* elements, use the following function:

void sort\_array(int a[], int used\_size)   
{   
      int index\_of\_next\_smallest;   
      int temp;

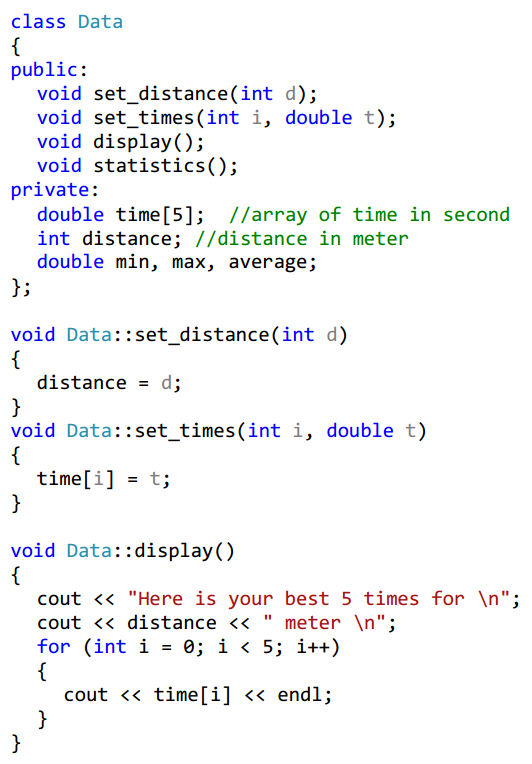
      for(int i = 0; i < used\_size-1; i++)   
      {   
              index\_of\_next\_smallest = index\_of\_smallest(a,

index, used\_size);   
              // swap two elements   
              temp = a[i];   
              a[i] = a[index\_of\_next\_smallest];   
              a[index\_of\_next\_smallest] = a[i];   
       }   
}

Note that you can write three more functions; 1) one similar to the one that finds the index of smallest number for finding the index of the largest number, 2) one that computes the average and returns it to the main, 3) and the last one that takes the sorted array and will return the median.

**Arrays as Class Members**

You can use arrays with classes and structures in almost an identical way.  Here is an example:



**Questions**

1. Create a new program called c7e6.cpp that uses class *Data.* Your program first asks a runner for a distance and 5 of his/her best times in that event, and it will store the times in the array time.  Then it finds the best, the worst, and the average time for that runner.  Finally, it displays all five times, followed by the worst, the best, and the average.

You need to write the member function *statistics.* Please note that we assume we are only dealing with 5 times.  So the *statistics function* is a simple function.  This program finds the fastest time as the best time, the slowest time as the worst time, and computes the average of 5 times for each event.

**Array of Objects**

It is possible to create an array of objects.  A program that uses the class AltMoney to create the objects of type money for two different people, then added two amounts of money, we could use an array to define the two objects as:

AltMoney m[2];

In this case, the first object is m[0] and the second object is m[1].  Note that previously, we would have created the two money objects using:

AltMoney m1, m2;

This does not seem to be much different from the one we defined using an array.  But wait, what if you wanted to create 100 objects of type AltMoney?  Which method is more convenient to use, then?

Once you used an array to create objects, you can perform computation on each one of the objects using the array index.  So, if you want to use add two values of money you might do something like this:

sum.add(m[0], m[1]); // uses array called m. two objects.

Rather than this:

sum.add(m1, m2); // uses two scaler variables.

**Two Dimensional Arrays**

Two dimensional arrays are defined in a similar way as the 1-D arrays.  Imagine a container with two columns of partitions that one can use to store pills.  My grandmother used to have one of those.  She had to take two pills a day.  So, we would load the container for her every Sunday night and she was good to go for the rest of the week.  She only had to know where the pills for each day were and which one of the pills was a day pill and which one was night pill. We would tell her that the left ones were day pills and the right ones were night pills.  The container had 7 rows for 7 days of the week and two columns one for the day and the other for the night. If we would view the container as a 2-D array, we would define it as:  pill container[7][2].

Here we use two brackets; one to define the number of rows, and the other to define the number of columns.  Similarly, a 2-D array of integers with 3 rows and 4 columns would be defined as:

int x[3][4];

Now, let me ask you a question.  Suppose my grandmother would use the container with 7 rows and 2 columns to keep her pills. Assume that the bottom row is used for Sunday, i.e. row 0. Can you tell me, in terms of rows and columns, where she could get the pill for Tuesday night?

Row: \_\_\_\_\_\_   Column: \_\_\_\_\_\_ OR in an array form:   container [\_\_\_] [\_\_\_]?

Now let's write a program that uses a 2-D arrays.

**Questions**

1. Locate c7e7-1 and c7e7-2 and type the program into an editor. Save this as c7e7.cpp then compile and run.

In the above program, we can access the 3rd time of the 4th event (400 m) in:

data[3][2];  // note that the 3rd time is stored in column

// with index 2 and the 4th event is stored in

// row with index 3

The 4th event was 400 meter.

To access the 5 times for 150 m event, we can use:

data[1][0], data[1][1], data[1][2], data[1][3], data[1][4]

Or use a *for* loop to access them:

for(i = 0; i < 5; i++)   
    data[1][i];

**Questions**

1. Modify the above program such that it finds the best, the worst, and the average time for each of the six events. The program should display, for each event, all five times, the worst, the best, and the average.

**Submission Instructions**

1. Create a zipped folder with the following files:
   * c7e1.cpp
   * c7e2.cpp
   * c7e3.cpp
   * c7e4.cpp
   * c7e5.cpp
   * c7e6.cpp
   * c7e7.cpp
2. Upload to the Moodle Link: Chapter 7 Lab Exercises