LAB: ONE-DIMENSIONAL ARRAYS IN C++

Module 2
Introduction

This lab will provide students with an introduction to the fundamentals of one-dimensional arrays in the C++ programming language. The focus of this lab is on how to solve complex problems using one-dimensional arrays. At the end of this module, students will…

- Be able to solve simple problems using one-dimensional arrays
- Understand the structure of one-dimensional arrays in C++
- Understand the use of one-dimensional arrays as function parameters

For those exercises which ask you to write code, structure your storage drive in the following manner. There should be a palms folder, and within that folder there should be a LabArrays folder. All coding exercises (and related folders) for this lesson should be placed in the LabArrays folder.

As always, be sure to properly document your code. Consult the C++ Coding Guidelines document for proper coding standards. Use good design principles and design the solution before attempting to write code. All programs should make use of multiple functions besides main() (use the input/process/output cycle as your guide). Think about how each problem actually involves more than one subproblem. At a minimum, you must construct functions aligned with the input-process-output steps. Let’s begin!

Part #1: One-Dimensional Arrays in C++

Many problems in Computer Science involve manipulating one or more lists of information. In programming terms, these lists are called arrays. An array is a block of memory storage that consists of multiple elements, all of the same data type. Arrays are very much like a list or collection of variables - an array has one name, but the individual elements can be accessed on their own. For example, the following C++ definition creates an array of 16 integer variables:

```
int my_list[16];
```

This variable declaration works differently than normal variables. A normal variable of type int typically takes up four bytes of memory; The array definition above asks the computer to allocate 16 contiguous four-byte segments of memory (64 bytes total). We have said before that a variable name is simply a symbolic identifier for a memory address; in the case of an array, the name refers to the address of the first element in the array only.

The [16] in the aforementioned array declaration indicates the number of elements (size) of the array. The individual elements of the array can be accessed by means of a computable index or subscript; this allows us to refer to each integer variable in the array separately. Array indices
start at zero and end at one less than the size of the array. In our earlier example, the valid indices are [0...15]. We can use **subscripts** to access the elements in the following manner:

```cpp
int my_list[16];
my_list[5] = 94;
my_list[7] = 41;
my_list[0] = 300;
```

We use an integer as the subscript/index. This integer can be a literal, a variable containing an integer value, or an expression that evaluates to an integer value. We say the index is computed because the computer uses the subscript given to compute the location in memory of the requested array element; for instance, if the code calls for `my_list[5]` then the computer will compute the memory address of the 6th element in the array according to the formula:

\[(\text{Address of 0th element}) + (5 \times (\text{sizeof int}))\]

Where `(sizeof int)` is four bytes (on 32-bit computers). The use arrays means that there will be a need to process each element in an array with some repeated operation. For example, suppose we wanted to output the entire contents of an array, one element at a time. Since the indices are sequential (0, 1, 2, 3, …) and the process involves repetition, we can use a **for** loop:

```cpp
for (int i = 0; i < 16; i++)
{
    cout << my_list[i] << endl;
}
```

At this point you should watch and listen to the [One-Dimensional Arrays in C++](https://www.youtube.com/watch?v=example) podcast:

```cpp
double squares[10]; // array of 10 elements
for (int i = 0; i < 10; i++)
{
    // calculate the square and store it
    squares[i] = i * i;
}
```
Arrays as Parameters

Like other variables, arrays can be passed to functions as arguments. However, doing so requires special syntax. Remember that an argument is a value used in a function call. An argument can be either a literal value (e.g. 10, “James Robert”, 24.22) or a variable. Each argument corresponds to a function parameter, a “placeholder” that needs to be “filled” in a function call. Parameters are used by a function to complete its work.

Most of the parameters we have used are value parameters. This means that the arguments we provide have their value passed to the function; changes to the parameters inside the function have no effect on the actual argument. Array arguments are treated as reference parameters. When an array is passed as a parameter, the memory address of the element at index 0 is passed rather than a copy of the value stored at that memory address. Because the memory address of the array is passed as the argument to a function, any changes made to the contents of the array parameter are actually made to the array argument in the calling function.

For example, suppose we had a function called fillList which took an array as a parameter. We could specify an array parameter in the function header by using the [ ] notation to indicate the parameter is an array:

```cpp
int fillList(double list[], int size)
{
    // code for the function body...
}
```

This identifies list as an array parameter. Inside the function, the variable list can be accessed just like any other array. When the function is called, the corresponding argument of an array type is specified by using the name of the array variable, just like other arguments:

```cpp
double my_list[10];
int num_items = fillList(my_list, 10);
```

Note that the [ ] are not used when specifying an argument to a function call. It is important to remember that the called function has no knowledge of the actual size of the array. In this example, the second argument (“10”) is used to signify the size of the array. This argument corresponds to an integer parameter (“size”) that the function uses to help it do its work.

If we want to pass an array by value, simply precede the data type in the parameter list by the keyword const. A const parameter is constant, i.e. it cannot be altered by the function; a const parameter can have its contents fetched and examined, but not changed. This restriction is enforced by the C++ compiler. For example:
void computeSum(const double list[], int size)
{
    // code for the function body; changes cannot be made to the
    // contents of the parameter ”list” without a compile error.
}

At this point you should watch and listen to the Reference and Value Parameters in C++ podcast to learn more about these types of parameters:

---

Part #2: Applications of Arrays

The following exercises represent just a sample of the types of problems which use arrays. Any problem which needs to process large amounts of data is well-suited to the use of arrays.

Arrays and Grade Calculators

The GradeCalculator program from the “For Loops” lab is a great example of a problem well-suited to arrays. The problem involves a repeated set of instructions (the instructions to provide input) which differ only in terms of the prompt: each user prompt includes a number (#0, #1, etc) which increases by 1 each time. The repeated prompting of the user has a defined starting point (Exam Score #0) and a defined ending point (Exam Score #10). Consider the input/output dialog:
The succession of prompts is an example of counter-driven repetition, and the set of 11 scores could easily be stored in an array.

**Exercise 1:** Create a new C++ Project called `GradeCalculator2`. Add a new C++ code file called `GradeCalculator2.cpp` to the project, and re-construct the `GradeCalculator` program using arrays. A few notes as you construct the program:

- Your input function must place the input values into an array.
- Your function which computes the average must iterate through the array as part of its calculation. Use for loops!

Compile and Build your program to verify it works as expected. Enter in several different tests cases to verify your program’s logic.

**Arrays and Bacterial Growth**

An array-based computer program can be useful in modeling the projected growth of bacteria. Uninhibited bacterial growth is said to follow the exponential law given by

\[ N(t) = N_0 e^{kt} \]

Where

- \( N(t) \) is the number of bacteria present at time \( t \).
- \( N_0 \) is the initial number of bacteria present.
- \( k \) is a positive constant of proportionality.
- \( t \) is the number of hours elapsed.
Suppose we wanted to write a C++ program to compute the number of bacteria present at increasing time steps 0, 1, 2, 3, 4, 5, ..., 10 hours. We could use an array of double values to store the number of bacteria at each time step. Once the computations were complete, the program could iterate through the array and output the values to the screen.

**Exercise 2:** Create a new C++ Project called **BacterialGrowth**. Add a new C++ code file called **BacterialGrowth.cpp** to the project and construct the program described above. User input should consist of the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Data Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant ( k )</td>
<td>double</td>
<td>Must be in the range [0-1)</td>
</tr>
<tr>
<td>Initial population ( N_0 )</td>
<td>int</td>
<td>Must be in the range [1-1000]</td>
</tr>
</tbody>
</table>

Your user inputs should be validated according to the rules above. Your program must use arrays to store your computations. Once the processing is complete, your program must loop through the array and produce a table in the following format:

```
Bacterial Growth Summary

<table>
<thead>
<tr>
<th>Hour</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td>NNNNN.NNN</td>
</tr>
<tr>
<td>NN</td>
<td>NNNNN.NNN</td>
</tr>
<tr>
<td>.</td>
<td>. . .</td>
</tr>
</tbody>
</table>
```

Please note that C++ provides the following function to compute the exponential value (\( e \)):

```
double exp(double power);
```

The following test cases are provided for this exercise:
Test Case #1:

Enter the constant of proportionality [0.0-1.0]: 0.2
Enter the size of the initial population [1-1000]: 10

Bacterial Growth Summary

<table>
<thead>
<tr>
<th>Hour</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10.000</td>
</tr>
<tr>
<td>1</td>
<td>12.214</td>
</tr>
<tr>
<td>2</td>
<td>14.918</td>
</tr>
<tr>
<td>3</td>
<td>18.221</td>
</tr>
<tr>
<td>4</td>
<td>22.255</td>
</tr>
<tr>
<td>5</td>
<td>27.183</td>
</tr>
<tr>
<td>6</td>
<td>33.201</td>
</tr>
<tr>
<td>7</td>
<td>40.552</td>
</tr>
<tr>
<td>8</td>
<td>49.530</td>
</tr>
<tr>
<td>9</td>
<td>60.496</td>
</tr>
<tr>
<td>10</td>
<td>73.891</td>
</tr>
</tbody>
</table>

Test Case #2:

Enter the constant of proportionality [0.0-1.0]: 0.5
Enter the size of the initial population [1-1000]: 1000

Bacterial Growth Summary

<table>
<thead>
<tr>
<th>Hour</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000.000</td>
</tr>
<tr>
<td>1</td>
<td>1648.721</td>
</tr>
<tr>
<td>2</td>
<td>2718.282</td>
</tr>
<tr>
<td>3</td>
<td>4481.689</td>
</tr>
<tr>
<td>4</td>
<td>7389.056</td>
</tr>
<tr>
<td>5</td>
<td>12182.494</td>
</tr>
<tr>
<td>6</td>
<td>20085.537</td>
</tr>
<tr>
<td>7</td>
<td>33115.452</td>
</tr>
<tr>
<td>8</td>
<td>54598.150</td>
</tr>
<tr>
<td>9</td>
<td>90017.131</td>
</tr>
<tr>
<td>10</td>
<td>148413.159</td>
</tr>
</tbody>
</table>
Test Case #3:

Enter the constant of proportionality [0.0-1.0]: 1.0
*** Invalid input; the value must be in the range [0.0...1.0). Try again.

Enter the constant of proportionality [0.0-1.0]: -1
*** Invalid input; the value must be in the range [0.0...1.0). Try again.

Enter the constant of proportionality [0.0-1.0]: 0.456

Enter the size of the initial population [1-1000]: 0
*** Invalid input; the value must be in the range [1...1000]. Try again.

Enter the size of the initial population [1-1000]: 1001
*** Invalid input; the value must be in the range [1...1000]. Try again.

Enter the size of the initial population [1-1000]: 2

Bacterial Growth Summary

<table>
<thead>
<tr>
<th>Hour</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.000</td>
</tr>
<tr>
<td>1</td>
<td>3.156</td>
</tr>
<tr>
<td>2</td>
<td>4.979</td>
</tr>
<tr>
<td>3</td>
<td>7.855</td>
</tr>
<tr>
<td>4</td>
<td>12.393</td>
</tr>
<tr>
<td>5</td>
<td>19.553</td>
</tr>
<tr>
<td>6</td>
<td>30.850</td>
</tr>
<tr>
<td>7</td>
<td>48.674</td>
</tr>
<tr>
<td>8</td>
<td>76.796</td>
</tr>
<tr>
<td>9</td>
<td>121.164</td>
</tr>
<tr>
<td>10</td>
<td>191.167</td>
</tr>
</tbody>
</table>

Scoring a Gymnastic Event

Each of the nine judges in a diving contest scores a gymnastic routine from 0.0 thru 10.0. For example, one gymnast may obtain a score of 8.0 while another may score a 5.5. To determine the final score, the average of the scores is multiplied by a difficulty factor ranging from 1.0 thru 5.0. Remember than an average is the sum of the values divided by the number of values.

Exercise 3: Write a C++ program that inputs seven scores for a gymnastic routine and its difficulty factor from the user. The scores must be stored into an array of double values. Your program should also input the difficulty factor from the user (a double value). **Validate all inputs.** To validate the gymnast score, use the following function which checks for a value in the specified range and at 0.5 intervals. Examine the code closely to make sure that you understand the logic.
bool validateGymnastScore ( double gymnast_score )
{
    bool is_valid = false;  // return value
    double fractional = 0.0;  // stores the fractional part

    // check to see if the score is within the range...
    if (((gymnast_score >= 0.0) && (gymnast_score <= 10.0))
    {
        // computes the fractional portion...
        fractional = gymnast_score - (int) gymnast_score;

        // check to see if we were at the 0.0 or 0.5 "step"...
        if ((fractional == 0.50) || (fractional == 0.0))
        {
            is_valid = true;
        }
    }

    return is_valid;  // return the dive score status
}

After the input is complete, display the scores (one per line, with one digit of precision) and the difficulty factor with an appropriate message. An example of the output would look like this:

The gymnastic routine scores entered were:
  8.5
  10.0
  7.5
  6.5
  6.0
  5.5
  9.5
  7.5
  8.5

The difficulty factor was 4.0.

Once you have outputted the scores and the difficulty factor, compute the final score and display it to the screen with a descriptive message (using two digits of precision for the final score).

Sieve of Eratosthenes

A prime number is a number that is only divisible by itself and one (1). One of the earliest known algorithms for determining a set of prime numbers up to a given limit comes from Eratosthenes of Cyrene, who is said to have come up with the algorithm in the third century B.C. This algorithm is called a sieve because it distinguishes the desired facts (i.e. prime numbers) from the undesired facts (i.e. non-primes).
**Exercise 4:** Construct a C++ program to print all of the prime numbers in the range [1…100]. The algorithm you should use is as follows:

- Create an array to hold a boolean "flag" value for all the numbers in [1…100].
  - **Note:** Remember that array indices always start at 0; it is recommend that your array declaration create an array which holds 101 values with indices [0…100]. The value at index 0 will be unused.
- The value in the list at index $i$ should be true if number $i$ is prime, and it should be false otherwise. Initially, set all the values in the list to be true.
- Determine if each number $i$ is prime. If it is, set the list value for the number $i$ to true, and set the list value for all multiples of $i$ to be false, since if a number is a multiple of another number it cannot be prime.
- Go back through the list and show all the prime numbers, indicated by a true value at the list position.

No user input is needed for this program. Think about what type of control structure is needed to solve the problem; given that you know the bounds of your list, the best solution should be obvious. Test your program thoroughly when finished.

**Election Results**

Congratulations! The U.S. Election Commission has ordered you to write a computer program to help it determine who will be the next President.

**Exercise 5:** Create a C++ program which asks the user for the following input values:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidate’s First Name</td>
<td>string</td>
</tr>
<tr>
<td>Candidate’s Vote Total</td>
<td>int</td>
</tr>
</tbody>
</table>

Your program should repeatedly prompt the user for these two pieces of information until the user has entered three (3) names. Your program should validate each vote total to make sure that the number of votes is greater than or equal to 0. Your input values for each candidate should be stored in two parallel arrays (one of type string, one of type int).

Parallel arrays mean that the first candidates’ name is stored in index 0 of the “names” array while his vote total is stored at index 0 of the “votes” array, the second candidates’ name is stored in index 1 of the “names” array while his vote total is stored at index 1 of the “votes” array and so on. **You will always have exactly three (3) sets of input values.**
Once the input is complete, process your arrays by displaying a three-column table for the user listing the candidate’s first name, the total votes they received, and the percentage of the total vote they received. For example, the following output displays the results of the 2004 election:

<table>
<thead>
<tr>
<th>Candidate First Name</th>
<th>Vote Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>62040606</td>
<td>51.07</td>
</tr>
<tr>
<td>John</td>
<td>59028109</td>
<td>48.59</td>
</tr>
<tr>
<td>Ralph</td>
<td>411304</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Remember that the percentage of the vote received is the candidates’ vote total divided by the total number of votes for all candidates. You will need to compute this “total vote” before calculating any of the individual percentages. Your prompts should be descriptive messages, and the numerical output for percentages should use two (2) digits of precision. Beware of integer division! Remember that you can use typecasting to avoid this issue (e.g. `(double)x / (double)y`). Here are few sample tests:

**Test #1: 2004 Presidential Election**

Enter the Candidate's first name: George
Enter the Candidate's vote total: 62040606
Enter the Candidate's first name: John
Enter the Candidate's vote total: 59028109
Enter the Candidate's first name: Ralph
Enter the Candidate's vote total: 411304

<table>
<thead>
<tr>
<th>Candidate First Name</th>
<th>Vote Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>62040606</td>
<td>51.07</td>
</tr>
<tr>
<td>John</td>
<td>59028109</td>
<td>48.59</td>
</tr>
<tr>
<td>Ralph</td>
<td>411304</td>
<td>0.34</td>
</tr>
</tbody>
</table>

**Test #2: 1980 Presidential Election**

Enter the Candidate's first name: Ronald
Enter the Candidate's vote total: 43903230
Enter the Candidate's first name: Jimmy
Enter the Candidate's vote total: 35480115
Enter the Candidate's first name: John
Enter the Candidate's vote total: 5719850

<table>
<thead>
<tr>
<th>Candidate First Name</th>
<th>Vote Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ronald</td>
<td>43903230</td>
<td>51.59</td>
</tr>
<tr>
<td>Jimmy</td>
<td>35480115</td>
<td>41.69</td>
</tr>
<tr>
<td>John</td>
<td>5719850</td>
<td>6.72</td>
</tr>
</tbody>
</table>
Test #3: Fictional Presidential Election

Enter the Candidate's first name: Manny
Enter the Candidate's vote total: 500
Enter the Candidate's first name: Moe
Enter the Candidate's vote total: 100
Enter the Candidate's first name: Jack
Enter the Candidate's vote total: 400

<table>
<thead>
<tr>
<th>Candidate First Name</th>
<th>Vote Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manny</td>
<td>500</td>
<td>50.00</td>
</tr>
<tr>
<td>Moe</td>
<td>100</td>
<td>10.00</td>
</tr>
<tr>
<td>Jack</td>
<td>400</td>
<td>40.00</td>
</tr>
</tbody>
</table>

Paying for Computer Time

Computers were a novelty in the not-too-distant past. There was once a time when you were charged a fee for using a computer; the computers were large monolithic things called mainframes, and everyone accessed the same computer through so-called “dumb” terminals, simple monitor-keyboard combinations with no CPU (Note: there were no PCs until the 1970s).

Exercise 6: Suppose you have been asked by a local State University to help them build their billing system. They have asked you to create a program which allows State University personnel to compute the monthly computer charge for users based on their usage time (a real number, in hours) and their priority level (an integer). The prices and criteria for the user charges are described below:
The user’s fee is therefore the sum of two separate fees: one fee based on usage time and one fee based on priority level. This program should input a series of input values for a group of users. For each user, your program should input the user account identifier (a string), a priority code (an integer), and the number of hours of use (a double). The output for each user should consist of the following items:

<table>
<thead>
<tr>
<th>Output</th>
<th>Data Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>User account identifier</td>
<td>string</td>
<td>Four Characters</td>
</tr>
<tr>
<td>User monthly charge</td>
<td>double</td>
<td>Monthly fee based on time and priority</td>
</tr>
</tbody>
</table>

The output should be precise to two decimal places and include a $ where appropriate. The input should stop when the user enters 9999 for the user account identifier. No lines of output should be displayed for 9999 (the “sentinel” value).

You need to use two arrays in your solution: one array to store the user IDs and one to store the calculated monthly charges.

Here are few sample tests:

**Test #1: No Inputs**

Enter the Account ID: 9999

State University Computer Lab

<table>
<thead>
<tr>
<th>User Account ID</th>
<th>Monthly Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>================</td>
<td>===============</td>
</tr>
</tbody>
</table>

PALMS for CS1 v4.0
Test #2: Bad Inputs

Enter the Account ID: 123456
ERROR! The Account ID must be exactly four digits.
Enter the Account ID: AABB
ERROR! The Account ID must be exactly four digits.
Enter the Account ID: 1234
Enter the hours of use (0.00–99.99): 100.0
ERROR! The hours must be in the range [0.00...99.99].
Enter the hours of use (0.00–99.99): -100.0
ERROR! The hours must be in the range [0.00...99.99].
Enter the hours of use (0.00–99.99): 0.01
Enter Priority Code (0, 1, 2): 3
ERROR! The priority code must be 0, 1, or 2.
Enter Priority Code (0, 1, 2): -1
ERROR! The priority code must be 0, 1, or 2.
Enter Priority Code (0, 1, 2): 1

Enter the Account ID: 9999

State University Computer Lab

<table>
<thead>
<tr>
<th>User Account ID</th>
<th>Monthly Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>$150.00</td>
</tr>
</tbody>
</table>

Test #3: Good Inputs

Enter the Account ID: 1111
Enter the hours of use (0.00–99.99): 23.5
Enter Priority Code (0, 1, 2): 2

Enter the Account ID: 2222
Enter the hours of use (0.00–99.99): 5.4
Enter Priority Code (0, 1, 2): 0

Enter the Account ID: 3333
Enter the hours of use (0.00–99.99): 10.9
Enter Priority Code (0, 1, 2): 2

Enter the Account ID: 4444
Enter the hours of use (0.00–99.99): 44.87
Enter Priority Code (0, 1, 2): 1

Enter the Account ID: 9999

Pepsi State University Computer Lab

<table>
<thead>
<tr>
<th>User Account ID</th>
<th>Monthly Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>$452.50</td>
</tr>
<tr>
<td>2222</td>
<td>$110.00</td>
</tr>
<tr>
<td>3333</td>
<td>$347.50</td>
</tr>
<tr>
<td>4444</td>
<td>$723.05</td>
</tr>
</tbody>
</table>
Test #4: Good Inputs

Enter the Account ID: 2345
Enter the hours of use (0.00–99.99): 99.99
Enter Priority Code (0, 1, 2): 1

Enter the Account ID: 9876
Enter the hours of use (0.00–99.99): 23.75
Enter Priority Code (0, 1, 2): 2

Enter the Account ID: 2985
Enter the hours of use (0.00–99.99): 0.05
Enter Priority Code (0, 1, 2): 1

Enter the Account ID: 9999

<table>
<thead>
<tr>
<th>User Account ID</th>
<th>Monthly Charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>2345</td>
<td>$1549.85</td>
</tr>
<tr>
<td>9876</td>
<td>$456.25</td>
</tr>
<tr>
<td>2985</td>
<td>$150.00</td>
</tr>
</tbody>
</table>

Part 3: Vocabulary Words

You will be assessed on your knowledge of the following terms and concepts as well as the C++ concepts discussed in this document. As a result, you are encouraged to review these terms in the preceding document.

Array – An array is a block of memory storage that consists of multiple elements, all of the same data type. An array has one name, but the individual elements can be accessed on their own.

Index – An integer which allows us to refer to each element of an array separately. Array indices start at zero and end at one less than the size of the array.

Reference Parameters – function parameters for which the arguments provided have their memory address passed to the function; changes to the parameters inside the function are made to the actual argument(s).

Subscript – see the definition for Index.

Value Parameters – function parameters for which the arguments provided have their value passed to the function; changes to the parameters inside the function have no effect on the actual argument.