ENSF 694 – Summer 2024
Principles of Software Development II
Department of Electrical & Computer Engineering
University of Calgary

Lab Assignment 1

M. Moussavi, PhD, PEng

This is a group assignment, and you can work with a partner. Groups of 3 or 4 people are not allowed.

Objectives:

This lab assignment contains a few exercises to help you understanding:

- some of important basic constructs in C++
- Functions in C++
- Activation records and memory diagrams within running program
- Built-in array and C-string
  Introduction to problem solving and algorithm development using recursive methods
- memory models, and finally an introduction to use C++ built-in arrays as a simple and basic data structure.

Please heed this advice:

1. Some exercises in this lab and future labs will not be marked. **Please do not skip them**, as unmarked exercises are as important as other exercises.

2. Some students skip directly to the exercises that involve writing code, skipping the sections such as "**Read This First**", or postponing the diagram-drawing until later. That's a bad idea for several reasons:

   a. "**Read This First**" sections normally explain some of technical or syntax details that may help you to solve the problem or may provide you with some hints.
   b. Drawing diagrams is an important part of learning how to visualize memory used in a computer programs. If you do diagram-drawing exercises at the last minute, you won't learn the material very well. If you do the diagrams first, you may find it easier to understand the code-writing exercises, so you may be able to finish them more quickly.

Due Dates:

Your lab reports must be submitted electronically on the D2L, into the Lab 1 Dropbox, before **12:00 PM** on Wednesday July 3rd. All of your work should be in a single PDF.

- For instructions about how to provide your lab reports, study the posted document on the D2L called: `How to hand in your lab assignment.`

Important Notes:

- Some lab exercises may ask you to draw a diagram, and most of the students prefer to hand-draw them. In these cases, you need to **scan** your diagram with a scanner or an appropriate device such as your mobile phone and insert the scanned picture of your diagram into your PDF file. A possible mobile app to scan your documents is **Microsoft Lens** that you can install it on you mobile device for free. Please make sure your diagram is clear and readable, otherwise you may either lose marks, or it could be impossible for TAs to mark it at all.

Marking Scheme:

You shouldn't submit anything for the exercises that are not marked.

Exercise	Marks
A	8 marks
B	2 marks
C	2 marks
D	10 marks
E	4 marks
F	24 marks
Total:	**50 Marks**

Important Notes:

When submitting your source code (C++ code, files with the extensions: .cpp, or .h), make sure the following information appears at the top of your file:

- File Name
- Assignment and exercise number
- Lab section
- Your name
- Submission Date:

Here is an example:

```
/*
 *  File Name: for example: lab1exe_F.cpp
 *  Assignment: for example: Lab 1 Exercise F
 *  Completed by: Your Name (or your team name for group exercises)
 *  Submission Date: for example: May 20, 2024
 */
```

Exercise A: A C++ Program with User-Defined Functions (8 marks)

Read This First

In physics, assuming a flat Earth and no air resistance, a projectile launched with specific initial conditions will have a predictable range (maximum distance), and a predictable travel time.

The range or maximum horizontal distance traveled by the projectile can be approximately calculated by:

$$d = \frac{v^2}{g}\sin(2\theta)$$

Where:

*g is gravitation acceleration (*9.81 m/s^2)
θ: the angle at which the projectile is launched in degrees
v: the velocity at which the projectile is launched
d: the total horizontal distance travelled by the projectile

To calulate the projectile travel time (t), when reaches the maximum horizontal distace the followig formaula can be used :

$$t = \frac{2v\sin\theta}{g}$$

In this exercise you will complete a given C++ source file called `lab1exe_B.cpp` that prompt the user to enter a projectile's initial launch velocity (v), and displays the table of maximum horizontal distance and travel time for the trajectory angles of 0 to 90 degrees.

What to Do:

First, download file `lab1exe_A.cpp` from D2L. In this file the definition of function main and the function prototypes for four other functions are given. Your job is to complete the definition of missing functions as follows:

Function create_table: which is called by the main function, receives the projectile initial velocity and displays a table of projectile's maximum travel distance (d) and time (t), for trajectory angles of 0 to 90 (degrees), with increments of 5 degrees. Here is the sample of the required table:

```
Angle          t            d
(deg)        (sec)         (m)
0.000000     0.000000     0.000000
5.000000     1.778689     177.192018
10.000000    3.543840     349.000146
```

You don't have to worry about the format or the number of digits after the decimal point. The default format is acceptable.

Function projectile_travel_time: receives two double arguments, the trajectory angle (θ), and the initial velocity (v) and returns projectile travel time (t).

Function projectile_travel_distance: receives two double arguments, the trajectory angle (θ), and the initial velocity (v) and returns projectile maximum horizontal distance (d).

Function degree_to_radian: receives an angle in degrees and converts to radian. This function is needed, because C++ library function `sin` needs its argument value to be in radian.

Notes

- To use library function `sin`, you need to include header file `cmath`.
- Please pay attention to constant values of π, and gravitation acceleration, `g`, the following lines are already included in the given file:

```
const double PI 3.141592654
const double G 9.8
```

- While running your program, try a few times to enter a negative value or invalid input (for example, instead of a number enter xyz), for velocity, and observe how the program reacts.
- Study the set of slides called "Function Documentation", to understand what the requirements of the given functions are, also to follow the same principle to write function interface comment for each function that you will write in this exercise or future exercises.

How to compile and run your program:

- If you are using a text editor and Cygwin on your Windows OS, within Cygwin terminal navigate to your working directory (the same directory that you saved your `lab1exe_A.cpp`, then on the Cygwin command line enter:

```
g++ -Wall lab1exe_A.cpp
```

an executable file called `a.exe` will have been created. If the command fails -- which will be indicated by one or more error Messages--go back to your editor and fix the code, save the file again, try `g++` again.

- If you are using Mac OS, and Xcode IDE, press the run button to compile and run your program. Also, Mac users can use the Mac terminal and follow exactly the same steps mentioned above, for Cygwin, to

use the g++ command and compile the program. However, instead of executable file `a.exe` file will be called `a.out`.

- Once you have an executable, run it a few times by using the command

  `./a.exe (or ./a.out on the terminal of Mac machine)`

Hint 1: You don't need to type '`./a.exe` or `./a.out` over and over! You can use the up arrow on your keyboard to retrieve previously entered commands.

Hint 2: You can also compile your program using the g++ command in the following format to create an executable file name that you like. For example, you can name your executable file `exercise_A` by entering:

```
g++ -Wall lab1exe_A.cpp -o exercise_A
```

What to Submit:

Submit the copy of your program (your code and the program output) as part of your lab report in PDF format. You don't need to upload your actual source code. Only copy and past your program code and its output into your lab report.

Exercise B: Pointers as Function Arguments

What to do – Part I

First copy the file `lab1exe_B1.cpp` from D2L. Then carefully study the set pf slide called "Activation Records", and make the AR diagram for point one in this program, using "**Arrow Notation**". Then compare your solution with the posted solution on the D2L. You don't need to submit anything for this part.

What to do – Part II

Now download the file `lab1exe_B2.cpp` from D2L and draw AR diagram for point one in this file.

Submit the AR diagram for part II as part of your lab report.

Exercise C: Using Pointers to Get a Function Changing Variables

Read This First

Here is an important note about terminology:

- *Don't* say, ``Pointers can be used to make a function return more than one value.'' A function can never have more than one return value. A return value is transmitted by a return statement, not by a pointer.
- *Do* say, ``Pointer arguments can be used to simulate call by reference,'' or, ``Functions with pointer arguments can have the side effect of modifying the variables that the pointer arguments point to.''

What to Do:

Make a copy of the file `lab1exe_C.cpp` from D2L. If you try to compile and run this program it will give you some warning and displays meaningless output because the definition of function `time_convert` is missing.

Write the function definition for `time_convert` and add code to the main function to call `time_convert` before it prints answers.

Submit the copy of your source code and the screenshots of the program output, as part your lab report.

Exercise D – Simplest Form of Linear Data structure – A Built in Array

Read This First - A Few Facts About Built in Arrays:

A built-in array is a data structure that can store a fixed size of sequential collection of elements of the same type. It is part of the language and doesn't need to include or import any library or header file. Here is a quick overview of a few facts about arrays:

Fact 1: When you declare an array with n elements of type T, as a local variable, a chunk of memory equal to the size of T multiplied by n will be allocated. In the following examples the size of x is 80 bytes, which is 8 (size of double) multiplied by10 (number of elements):

```
double x [10];                          /* size of x is: 10 * 8 = 80 bytes */
double y[] = {2.3, 3.0, 4.0};           /* size of y is 24 bytes */
```

C++ also provides an operator called `sizeof` that can be used to find the size of a data object in bytes. It can be applied either to a type or an expression. Here are examples of using `sizeof` operator:

```
int n = (int) sizeof(x);           /* n == 80 */
int m = (int) sizeof(double) * 10; /* m == 80 */
```

The value produced by `sizeof` operator is of type `size_t`, which is some sort of integer type; exactly which type it is, depends on the particular implementation of compiler you are using. In this code segment we have used the type cast operator `(int)` to convert `size_t` type to the exact type of `int` on the left-hand side of the assignment operator.

The syntax `sizeof(`*`something`*`)` looks like a function call, but it isn't. When the compiler sees the expression `sizeof(x)`, it simply replaces the expression with the size of x in bytes.

Fact 2: Pointer and arrays are closely intertwined in C++. Most of the time, when we use the name of an array in an expression, that name is automatically treated like a pointer to the first element of the array:

```
int ia[] = { 4, 6, 9};
int *ip = ia;
```

Here is an exception to this fact, when passing the name of array ia to the sizeof **operator** it is not treated as a pointer. It is treated just like an array – the value of y in the following example will be 12.

```
int y = (int) sizeof(ia);
```

Similarly, for proper type-match when the name of an array is passed to a function, the corresponding argument of the function has to be a pointer. For example, a call to a function such as:

```
func(ia, 3);
```

Means the prototype of the function func should have two argument as follows::

```
void func(int*, int);
```

Fact 3: Arrays cannot be simply copied by using a single assignment statement that copies source array into an entire destination array. The following example produces a compilation error:

```
double x[3] = {7.5, 43.2, 0.3};
double y[3] ;
y = x;        /*ERROR */
```

Fact 4: Arrays cannot be resized. Therefore, they are often declared with a ``worst-case" size. In the following example we assumed the maximum number of data at some point may or may not reach to 100, but at this point we are using only the first four elements of the array data:

```
double data [100] = {120.40, 200.00, 34.56, 99.88} ;
```

Fact 5: When we pass a numeric array to a function, we should also pass an integer argument to the function indicating the actual number of elements to be used.

```
double x[10] = {2.50, 3.20, 33.0}; /* Note only first 3 elements of x are used */
double y[] = {5.00, 2.00};

my_function(x, 3);             /* my_function should use the first 3 elements */
my_fucntion(y, 2);             /* my_function should use entire array, 2 elements */
```

C-strings are exceptions: You don't have to worry about this exception in this lab -- we will discuss it during the lectures.

Fact 6: An array notation (square brackets) as a formal argument of a function is in fact a pointer. For example:
`int foo(int a[], int n);`

, is *exactly* the same as:

```
int foo(int *a, int n);
```

And, both are exactly same as:

```
int foo(int a[100], int n); // compiler ignore the number 100 between []
```

Read This Second – Multi-Dimensional Arrays:

A multi-dimensional array is an array of arrays, allowing you to create data structures like matrices or tables. These arrays can have two or more dimensions. The most common types are two-dimensional (2D). A 2D array is like a table with rows and columns. It is declared and initialized like this:
int matrix[3][4]; // A 3x4 array (3 rows and 4 columns)

You can also initialize it at the time of declaration:
`int matrix[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };`

Here's a simple example program demonstrating the use of a 2D array:

```
int main() {
    // Declare and initialize a 3x3 matrix
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
```

```
    // Print the matrix
    std::cout << "Matrix elements:" << std::endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Passing a 2-D array to a function in C++ involves specifying the array's type and dimensions in the function's parameter list. Here's how you can do it:

```
void printMatrix(int matrix[][3], int rows)
{
// code to display the values in each row/column
}
```

Please notice that the second dimension, which is the number of columns must be specified in the function argument.

Here is an example of a function call that the 2-D array, matrix and its number of columns are passed to the function:

```
printMatrix(matrix, 3);
```

What to Do:
Download the file `lab1exe_D.cpp` from D2L. Read the comment at the top of the file, and then try to predict the output of the program. Compile and run the program to check that your prediction of the output was correct.
Note: Some compilers may give warnings about size of pointers, but you still should be able to run the program.

Part I:
Draw memory diagrams for point one, two, and three.

Part II
When you run the program you will see the following output from last few lines of the program, which are either wrong or missing because two function called `good_copy` and `print_matrix` are missing.

```
The values in array x after call to good_copy are expected to be:
2.30, 1.20, 2.20, 4.10
And the values are:
2.1e-314  2.1e-314  3e-314  49
The values in matrix are:

Program Ends...
```

Please, read the function interface comment for these two functions and complete their implementations.

What to Submit:

Submit a properly scanned copy of your AR diagrams for point one, point two, and point three.
Copy and paste your source code, `lab1_exD.cpp` and the program output into your lab report.

Exercise E (6 marks): Pointer arithmetic

What to do:

Download the file `lab1exe_E.cpp` from D2L.Read the program and draw a memory diagram for the second time the program gets to point one, which occurs during the second call `main` makes to function `what` . If you have difficulty tracing the program you may want to add some calls to `cout` statements in `what` , then run the program to collect extra information.

Submit your diagram as part of your lab report, on the D2L.

Exercise F (24 marks): Manipulating Basic Data Structures (Arrays)

Read This First:

In this exercise you are expected to write a few functions that manipulates an array of integer as a data structure that is wrapped into a C++ `struct` data type called `MyArray`:

```
#define SIZE 5         // defines SIZE as constant. As an example, holding the value of 5
struct MyArray{
    int array[SIZE];   // A built in array with the maximum capacity of SIZE
    int list_size;     // A data member that keeps track of number of values
};                     // added into an element of the array.
```

The program also provides the prototype of the several functions that receive a pointer of type MyArray as their first argument (and other arguments as needed) and implements something on the wrapped array within and object of MyArray structure. Here is a few of the function prototypes stored in the file called MyArray.h:

```
void initialize(MyArray* myArray);
/* REQUIRES: pointer myArray points to an object of struct MyArray
 * PROMISES: initializes the member myArray->list_size to zero. In other words since
 * myArray->array is empty the list_size is set to zero.
 */
```

The function comment says: this function requires a pointer of type MyArray, called myArray that must be pointing to an object of MyArray. This means this is a requirement for this function to be able to do its job. Then the following part of the comment says: this function initializes the list_size to zero. Meaning, the array/list is empty.

Another function prototype is as follows:

```
void insert_at(MyArray* myArray, int pos, int val);
/*
 * REQUIRES: pos >= 0 and pos <= size(), and pointer myArray points to an object of struct
 * MyArray.
 * PROMISES: inserts the value of val in myArray->array[pos], after moving the values in the
 * myArray->array to the right of element pos. Then, increments that list_size by one.
 */
```

The function comment says: the user of the function is responsible not to pass value of pos less than zero, and greater than list_size. In other words, the function will not do any error checking in this regard. Also, function requires a pointer of type MyArray, called myArray pointing to an existing object of MyArray. The other arguments of the function are: pos is an integer indicating the position that new data, val must be inserted. As needed, the data in elements to the right of the element pos must be shifted to the right to open a space for inserting val. At the end if process of inserting val is successful the value of list_size must be incremented by one.

```
int search(const MyArray* myArray, int obj);
/*
 * REQUIRES: pointer myArray points to an object of struct MyArray.
 * PROMISES: returns the position of first occurance of obj in myArray->array. Returns -1
 * if there is no match for obj.
 */
```

```
void append( MyArray* myArray, int array[], int n );
/*
 * REQUIRES: pointer list points to an object of struct MyArray and array points to
 * an arrary of n integer numbers.
 * PROMISES: If (myArray->list_size + n), is less than or equal SIZE appends the numbers in
 * array to the end of the myArray->array. Otherwise, it does nothing.
 */

int retrieve_at(MyArray* myArray, int pos);
/*
 * REQUIRES: pos >= 0, and pos < size(), and pointer myArray points to an object of struct
 * MyArray.
 * PROMISES: returns the value of myArray->array at the position pos.
 */
```

In this file there are more functions prototypes as follows that you can read their function interface comment and understand what the function is supposed to do. Please don't hesitate to ask question(s) from course staff, if you need further explanation.

```
int remove_at(MyArray* myArray, int pos );
void display_all(MyArray* myArray);
bool is_full(MyArray* myArray);
bool isEmpty(MyArray* myArray);
int size(MyArray* myArray);
int count(MyArray* myArray, int obj );
```

What to do:

Download files: `MyArray.cpp, MyArray.h, MyArray_tester.cpp`, and the input file `data.txt`, and place them in the same directory.

If you compile the `.cpp` files, using the following command, an executable file called `myProgram` will be created in your working directory.

```
g++ -Wall MyArray.cpp MyArray_tester.cpp -o myProgram
```

Now, you can run the program using the following command:

```
./myProgram
```

If program logic is correct, you will see the following output, which shows all the function called from line 2 to line 19 are failed. The reason is that the functions in the file `MyArray.cpp` are not completely implemented and your job in this exercise is to complete them.

```
Starting Test Run. Using input file.
Line 1 >>  Passed
Line 2 >> Failed in insert_at
Line 3 >> Failed in insert_at
Line 4 >>  Failed in retrieve_at: expected value is 7, not 0
Line 5 >>  Failed in retrieve_at: expected value is 3, not 0
Line 6 >>  Failed in count(): expected value is 1, not 0
Line 7 >>  Passed
Line 8 >>  Failed in count(): expected value is 1, not 0
Line 9 >>  Failed in removed_at(): expected value is 7 not 0
Line 10 >>  Failed in retrieve_at: expected value is 3, not 0
Line 11 >>  Failed in removed_at(): expected value is 3 not 0
Line 12 >>  Passed
Line 13 >>  Failed in is_empty(): expected value is 1, not 0
Line 14 >> Failed in insert_at
Line 15 >> Failed in insert_at
Line 16 >> Failed in insert_at
Line 17 >> Failed in insert_at
Line 18 >>  Failed in retrieve_at: expected value is 101, not 0
Line 19 >>  Failed in retrieve_at: expected value is 500, not 0
Exiting...
Finishing Test Run
Showing Data in the List:
```

```
Program Ended ....
Program ended with exit code: 0
```

File `MyArray_tester.cpp` contains a listing that tests most of the functions implemented in `MyArray.cpp`. Although you don't need to know the details of code in this file, but here is a brief note about this file:
This file first uses the library function preopen to redirect the standard input to the file data.txt:

```
freopen("data.txt", "r", stdin);
```

Please note, you must have file `data.txt` in your working directory.

If you complete the implementation of the functions in `MyArray.cpp`, with no syntax and no logical error, your program should create the following output:

```
Starting Test Run. Using input file.
Line 1 >>  Passed
Line 2 >>  Passed
Line 3 >>  Passed
Line 4 >>  Passed
Line 5 >>  Passed
Line 6 >>  Passed
Line 7 >>  Passed
Line 8 >>  Passed
Line 9 >>  Passed
Line 10 >>  Passed
Line 11 >>  Passed
Line 12 >>  Passed
Line 13 >>  Passed
Line 14 >>  Passed
Line 15 >>  Passed
Line 16 >>  Passed
Line 17 >>  Passed
Line 18 >>  Passed
Line 19 >>  Passed
Exiting...
Finishing Test Run
Showing Data in the List:
101  200  100  500
Program Ended ....
```

To provide an output file from your program output and submit as part of your work, you can use the following command to produce the file `output.txt`:
`./myProgram > output.txt`

For your information: You can also comment out the line `freopen("data.txt", "r", stdin);` in the file `MyArray_teester.cpp`, and instead, use the following command to allow your program to read from file `data.txt` and send the output to the file `output.txt`:

```
./myProgram < data.txt > output.txt
```

The line, above, calls the executable file and uses the redirection operators (<) to read from data.txt and (>), write the program output to the file `output.txt`.

What to Submit:

Copy and paste your source code, `MyArray.cpp`, and the program output into your lab report.
Also upload your source code: `MyArray.h`, and `MyArray.cpp` into the Dropbox on the D2L.