

ENSF 694 – Summer 2025
Lab Assignment 3

Written by: M. Moussavi, PhD, PEng.

Objective:

The objective of this lab is, first to introduce some the basic constructs of object-oriented programming in C++, with a focus in developing data structures such as Queues and Stacks. Also, you will be introduced to C++ STL data structure classes vectors and string.

Due Dates: Thursday July 24, before 12:00 PM. Please notice that the due date in this lab is different than previous lab.

Marking scheme:

- Exercise A 4 marks
- Exercise B 8 marks
- Exercise C 16 marks
- Exercise D 16 marks
- Exercise E 16 marks
- Exercise F 4 marks

Total: 64 marks

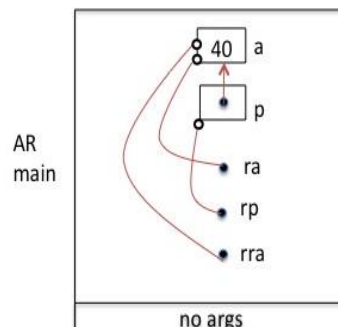
Exercise A: AR Diagram with C++ Reference Type

Read This First:

The AR notations that we use to show C++ references, are different from ordinary types such as: int, double, and pointer notations. This is because, when we declare a reference, we just provide an alias name for another memory spaces. Therefore, references in C++ don't have their own memory spaces, and we show them as a link (a line) between the reference-identifier and the actual allocated memory spaces. There are two little circles on both ends of these links. On one end there is a solid-black circle that represents the reference, and on the other end there is an open circle that represents the actual allocated memory space.

Here is an example:

```
int main(void) {  
    int a = 40;  
    int*p = &a;  
    int& ra = a;    // ra is referred to integer a  
    int*& rp = p;    // rp is referred to integer pointer p  
    int& rra = ra;   // rra is also referred to a  
    return 0;  
}
```



Notice that all references ra, rp, and rra **must** be initialized with an expression that represents an actual memory space or another reference.

What to Do:

Download the file `lab3exe_A.cpp` from D2L. Then, draw AR diagrams for points **one**, and **two**.

Submit your diagrams as part of your lab report in PDF.

Exercise B: Objects on the Computer Memory in C++

The objective of this exercise is to help you in understanding how C++ class objects are shown on memory diagram, and to find out how C++ class objects are associated with their member functions via a pointer 'this' pointer.

Read This First:

Drawing rules for AR diagrams in a C++ program is like C. However, in addition to the reference notation that was mentioned in exercise A, you need understand the concept of **this** pointer. Every member function of a class in C++ has a hidden argument as its first argument that is called **this**. The purpose of this hidden argument is to allow the compiler to know which object is invoking the function. Here is a simple example AR diagram when the constructor of class Point is called for the second time.

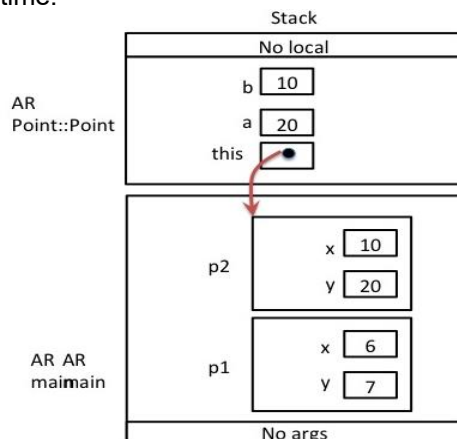
```
class Point {
private:
    double x, y;
public:
    Point(double a, double b); // prototype of constructor of class Point
    ...
}; // end of the definition of class Point

// implementation of constructor for class Point. Notice that implementation is
// outside the class definition.

Point::Point(double a, double b) // Point:: indicates that constructor
                                // belongs to class point
{
    x = a;    // this is in fact: this -> x = a;
    y = b;    // this is in fact: this -> y = a;
    // POINT ONE
}

int main() {
    Point p1(6, 7); // first call to the constructor of class Point
    Point p2(10, 20); // second call to the constructor of class Point
    ...
    return 0;
}
```

Here is the AR diagram for POINT ONE inside the constructor of class Point, when it is called for the second time:



What to Do: Download files `cplx_number.cpp`, `cplx_number.h`, and `lab3exe_B.cpp` from the D2L, and draw AR diagrams for points: one, two, and three. For this exercise you only need to read the given files carefully and draw the diagrams. You don't need to compile or run the program. However, if you want to compile and run it from command line on our lab computers you should have all the given files in the same directory and from that directory you should use the following command to compile and create the executable, `a.exe`:

```
g++ -Wall cplx_number.cpp lab3exe_B.cpp
```

Please notice that you shouldn't have any header file name(s) in this command -- only the .cpp files.

Please **submit your diagrams as part of your lab-report**.

Exercise C (16 marks): Writing a Class Definition and Its Implementation:

Read This First – What is a Helper Function?

One of the important elements of good software design is the concept of code-reuse. The idea is that if any part of the code is repeatedly being used, we should wrap it into a function, and then reuse it by calling the function as many times as needed. In the past labs in this course and the previous programming course, we have seen how we can develop global function to reuse them as needed. A similar approach can be applied within a C++ class by implementing **helper-functions**. These are the functions that are declared as private member functions and **are only available to the member functions of the class** -- Not available to the global functions such as main or member functions of the other classes.

If you pay close attention to the given instruction in the following "What to Do" section, you will find that there are some class member functions that need to implement a similar algorithm. They all need to change the value of data members of the class in a more or less similar fashion. Then, it can be useful if you write one or more **private helper-function**, that can be called by any of the other member functions of the class, as needed.

Read This Second – Instructions to Design Class - Clock

In this exercise you are going to design and implement a C++ class called, `Clock` that represents a 24-hour clock. This class should have three private integer data members called: `hour`, `minute`, and `second`. The minimum value of these data members is zero and their maximum values should be based on the following rules:

- The values of `minute`, and `second` in the objects of class `Clock` **cannot** be less than 0 or more than 59.
- The value of `hour` in the objects of class `Clock` cannot be less than 0 or more than 23.
- As an example any of the following values of `hour`, `minute`, and `second` is acceptable for an object of class `Clock` (format is `hours:minutes:seconds`): `00:00:59`, `00:59:59`, `23:59:59`, `00:00:00`. And, all of the following examples are **unacceptable**:
 - `24:00:00` (hour cannot exceed 23)
 - `00:90:00` (minute of second cannot exceed 59)
 - `23:-1:05` (none of the data members of class `Clock` can be negative)

Class `Clock` should have three constructors:

A default constructor, that sets the values of the data-members `hour`, `minute`, and `second` to zeros.

A second constructor, that receives an integer argument in seconds, and initializes the `Clock` data members with the values for `hour`, `minute`, and `second` in this argument. For example, if the argument value is 4205, the values of data members `hour`, `minute` and `second` should be: 1, 10, and 5 respectively. If the given argument value is negative the constructor should simply initialize the data members all to zeros.

The third constructor receives three integer arguments and initializes the data members `hour`, `minute`, and `second` with the values of these arguments. If any of the following conditions are true this constructor should simply initialize the data members of the `Clock` object all to zeros:

- If the given values for `second` or `minute` are greater than 59 or less than zero.
- If the given value for `hour` is greater than 23 or less than zero.

Class `Clock` should also provide a group of access member functions (getters, and setters) that allow the users of the class to retrieve values of each data member, or to modify the entire value of time. As a convention, let's have the name of the getter functions started with the word `get`, and the setter functions started with word `set`, both followed by an underscore, and then followed by the name of data member. For example, the getter for the data member `hour` should be called `get_hour`, and the setter for the data member `hour` should be called `set_hour`. Remember that getter functions must be declared as a `const` member function to make them read-only functions.

All setter functions must check the argument of the function not to exceed the minimum and maximum limits of the data member. If the value of the argument is below or above the limit the functions are supposed to do nothing.

In addition to the above-mentioned constructors and access functions, class `Clock` should also have a group of functions for additional functionalities (lets call them implementer functions) as follows:

1. A member function called `increment` that increments the value of the clock's time by one.
Example: If the current value of time is `23:59:59`, this function will change it to: `00:00:00` (which is midnight sharp). Or, if the value of the time is `00:00:00` a call to this function increments it by one and makes it: `00:00:01` (one second past midnight – the next day)
2. A member function called `decrement` that decrements the value of the clock's time by one.
Example: If the current value of time is `00:00:00`, this function will change it to: `23:59:59`. Or, if the value of current time is `00:00:01`, this function will change it to: `00:00:00`

A member function called `add_seconds` that **REQUIRES** to receive a positive integer argument in seconds, and adds the value of given seconds to the value of the current time. For example, if the clock's time is `23:00:00`, and the given argument is `3601` seconds, the time should change to: `00:00:01`.

3. Two helper functions. These functions should be called to help the implementation of the other member functions, as needed. Most of the above-mentioned constructors and implementer function should be able to use these functions:
 - A **private** function called `hms_to_sec`: that returns the total value of data members in a `Clock` object, in seconds. For example, if the time value of a `Clock` object is `01:10:10`, returns `4210` seconds.
 - A **private** function called `sec_to_hms`, which works in an opposite way. It receives an argument (say, `n`), in seconds, and sets the values for the `Clock` data members, `second`, `minute`, and `hour`, based on this argument. For example, if `n` is `4210` seconds, the data members values should be: `1`, `10` and `10`, respectively for `hour`, `minute`, and `second`.

What To Do:

If you haven't already read the “**Read This First**” and “**Read This Second**”, in the above sections, read them first. The recommended concept of helper function can help you to reduce the size of repeated code in your program. Then, download file `lab2exe_C.cpp` from D2L. This file contains the code to be used for testing your class `Clock`.

Now, take the following steps to write the definition and implementation of your class `Clock` as instructed in the above “**Read This Second**” section.

1. Create a header file called `lab3Clock.h` and write the definition of your class `Clock` in this file. Make sure to use the appropriate preprocessor directives (`#ifndef`, `#define`, and `#endif`), to prevent the compiler from duplication of the content of this header file during the compilation process. Marks will be deducted if appropriate style of creating header files is not followed.
2. Create another file called `lab3Clock.cpp` and write the implementation of the member functions of class `Clock` in this file (remember to include “`lab3Clock.h`”).
3. Compile files `lab3exe_C.cpp` (that contain the given main functions) and `lab3Clock.cpp` to create your executable file. Note that when compiling your code, use `g++` command and not `gcc` and moreover only compile the `.cpp` files (`lab3exe_C.cpp` and `lab3Clock.cpp`). header file `lab3Clock.h`, shouldn't appear on the command line.
4. If your program shows any compilation or runtime errors fix them until your program produces the expected output as mentioned in the given main function.
5. Now you are done!

What to Submit:

1. Copy and paste `lab3Clock.h`, and `lab3Clock.cpp`, and the program's output as part of your report.

2. Create a zip file that contains all your actual source codes (.cpp and .h file). Save you zip file using the following name-format: lab3exe_C_yourLastName.zip.
3. Then, submit your zip file along with your lab report on the D2L Dropbox.

Exercise D: Developing a Data Structure Called Circular Queue

Read this First:

In this exercise we develop a data structure thar represents a circular queue. Queues are one of the important data structures that is used in many, engineering, and computer science practices.

A circular queue, also known as a ring buffer, is a type of data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

In a circular queue, the last element points to the first element making a circular link. It operates on the FIFO (First In First Out) principle. then, the last position is connected back to the first position to make a circle. It is also called a 'Ring Buffer'.

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

As we discussed during the lecture a sample run of a circular queue is presented in the following figures:

Initially the queue is empty and two variable head and tail are set to 1. In general in this method of manipulating a circular queue, if values of head and tail are equal means that queue is empty and the operation of removing a value from queue (dequeuing) will fail. On the other hand when the value of head is equal to tail + 1, means the queue is full and the operation of adding (enqueueing) will fail.

Queue is empty:

OFFSET	0	1	2	3
DATA				

Head Pointer
1

Tail Pointer
1

Enqueue A : Fails if Head equals Tail+1

OFFSET	0	1	2	3
DATA			A	

Head Pointer
1

Tail Pointer
2

Enqueue B : Fails if Head equals Tail+1

OFFSET	0	1	2	3
DATA			A	B

Head Pointer
1

Tail Pointer
3

Enqueue C: Fails if Head equals Tail+1

OFFSET	0	1	2	3
DATA	C		A	B

Head Pointer
1

Tail Pointer
0

Dequeue: Fails if Head equals Tail

OFFSET	0	1	2	3
DATA	C			B

Head Pointer
2

Tail Pointer
0

Enqueue D: Fails if Head equals Tail+1

OFFSET	0	1	2	3
DATA	C	D		B

Head Pointer
2

Tail Pointer
1

Enqueue E: Failed because Head was equal to Tail+1

OFFSET	0	1	2	3
DATA	C	D		B

Head Pointer
2

Tail Pointer
1

Dequeue: Fails if Head equals Tail

OFFSET	0	1	2	3	Head Pointer	Tail Pointer
DATA	C	D			3	1

Dequeue: Fails if Head equals Tail

OFFSET	0	1	2	3	Head Pointer	Tail Pointer
DATA		D			0	1

Dequeue: Fails if Head equals Tail

OFFSET	0	1	2	3	Head Pointer	Tail Pointer
DATA					1	1

Dequeue: Failed, because Head was equal to Tail

OFFSET	0	1	2	3	Head Pointer	Tail Pointer
DATA					1	1

What to Do:

Download files `CircularQueue.cpp`, `CircularQueue.h`, `CircularQueue_tester.cpp`, and `data1.txt`, `data2.txt`. If you study the given source files (.cpp and .h files), you notice that there is a partial implementation of a class called `CircularQueue`. Your job in this exercise is to complete the program in a way that it follows the process defined in the figures, above.

Once you completed the missing implementation of member functions in class `CircularQueue`, you can run it with the two given input files, `data1.txt`, and `data2.txt`. Please submit your work after testing and working with both input file.

Please notice that a tester file, similar to one in one of the exercises in lab1, is designed for this lab to read the data from input files and send the program output to output file, using redirection operators, `<` and `>`.

Here is what you should do: Using the following command, compile your code:

```
g++ -Wall CircularQueue.cpp CircularQueue_tester.cpp -o myCircularQueue
```

Once your program is compiled with no errors, you should be able to run it, using:

```
myCircularQueue <data1.txt >output1.txt
```

IMPORTANT NOT: To run you program with the `data1.txt`, as input file the defined constant, `SIZE`, in the `CircularQueue.h` must have the value of 4 and with the file `data2.txt`, the value of `SIZE` must be 10.

What to Submit:

1. Copy and paste the content of your .cpp as part of your report (in pdf format).
2. Create a zip file that contains all your actual source codes (.cpp and .h file). Save you zip file using the following name-format: `lab3exe_D_yourLastName.zip`.
3. Then, submit your zip file along with your lab report on the D2L Dropbox.

Exercise E: Developing a Data Structure, Called Dynamic Stack

In this exercise you are going to complete the incomplete implementation of a C++ class called `DynamicStack`. A stack is a data structure that stores data list and allows insertions and deletions at one end of the list. A stack is also known as Last In First Out (LIFO).

Your dynamic stack should use a dynamic array, where the size of the array may be changed depending on the number of elements currently stored in the array, according to the following two rules:

If a new data is being added into a stack and the array is full, the capacity of the array is doubled. On the hand, when popping a value (removing a data from a stack), if the number of data is one-fourth the capacity

of the array, then the capacity of the array is halved. The size of the array may not be reduced below the initially specified size.

What to Do:

Download files `DynamicStack.cpp`, `DynamicStack.h`, `DynaicStack_tester.cpp`, from D2L. If you compile this program and try to run the program, you will notice that there is something wrong with the implementation of the `DynamicStack`. The problem is that most of the implementation of member functions of class `DynamicStack` are missing. Your task in this exercise is to complete and correct the implementation of those member functions in the file `DynamicStack.cpp`.

What to Submit:

1. Copy and paste the content of your .cpp as part of your report (in pdf format).
2. Create a zip file that contains all your actual source codes (.cpp and .h file). Save you zip file using the following name-format: `lab3exe_E_yourLastName.zip`.
3. Then, submit your zip file along with your lab report on the D2L Dropbox.

Exercise F: C++ STL Classes for Linear Data Structures, `vector` and `string`

The objective of this exercise is to gain some experience in understanding the C++ library classes, `vector`, and `string`.

What to Do:

Download the files `lab3exe_F.cpp` from D2L. In this file there is a declaration of `vector<string>`. If you compile and run this program it creates the following output:

```
ABCD
EFGH
IJKL
MNOP
QRST
```

Let's visualize this output as a matrix of letters (5 rows and 4 columns):

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P
Q	R	S	T

Your job is to complete the definition of the function called `transpose` that creates a new object of `vector<string>` where its strings are the transpose of the original vector:

A	E	I	M	Q
B	F	J	N	R
C	G	K	O	S
D	H	L	P	T

To test your program, you can change the values of the constants `ROWS` and `COLS`, in the `main` function to make sure your function works with other sizes of the `String_Vector`.

What to Submit:

Submit the definition of your function `transpose` and the program's output.