ENSF 614 – Fall 2021
Lab 3 – September 28, 2021

Department of Electrical & Computer Engineering
University of Calgary
Written by: M. Moussavi, PhD, PEng.

In this lab, you are allowed to work with a partner (ONLY groups of two. groups of three or more are NOT allowed).

If you decided to work with a partner:

1. **please submit only one lab report with both names on the report.**
2. **This is a very important lab assignment, and it should help you to understand some of the fundamental principles of object-oriented programming in C++. To make sure you will gain a good understanding on the basics of new types such as references and classes in C++, you should not split the work between your partner and yourself. Please work together on ALL exercises. Every exercise is very important and questions similar to these exercises may appear in the upcoming quiz on Oct 8.**

Objective:

The objective of this lab is to help you in understanding of the subjects such as:

- C++ reference type,
- Drawing C++ objects on the memory
- Designing C++ classes
- Understanding the concepts of dynamic allocation of objects, and issues with the bad-copy of C++ objects.

Due Date: Tuesday October 5, before 11:59 PM.

Note: There will no new lab assignment on October 5th to let you study and get prepared for the first quiz that will be on Friday October 8, at 5 PM.

Marking scheme:

- Exercise A 4 marks
- Exercise B 8 marks
- Exercise C 16 marks
- Exercise D 12 marks

Total: 40 marks

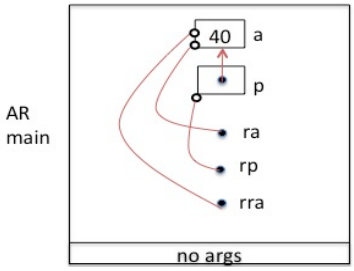
Exercise A: AR Diagram with C++ Reference Type

Read This First:

The AR notations that we use in ENSF 619, to show C++ references, are different from ordinary types such as: int, double, and pointer notations. This is due to the fact that, 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
```

```
        ...                 // assume more code here
        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**. You don't need to compile or run this program but if you want to do so, you should know that, this is a C++ program and you have to use the following command to compile it:

```
g++ -Wall lab3exe_A.cpp
```

Submit your diagrams.

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 similar to 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**.

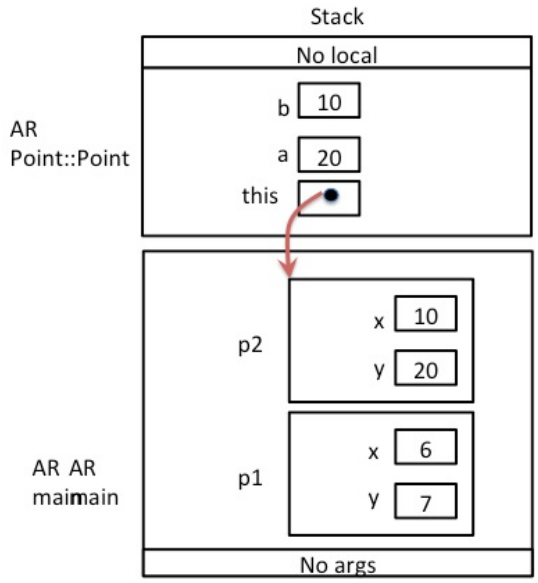
```
class Point {
  private:
    double x, y;
  public:
    Point(double a, double b); // prototype of constructor of class Point
    ...
 };  // end of the definition of class Point

// 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;
}
```

Now 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, three**, and **four**. 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 of 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_G.cpp
```

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

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, lets 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 he 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 `lab3exe_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 past `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 and your lab report on the D2L Dropbox.

Exercise D: The DynString class

Read This First:

To allocate memory dynamically, in C++ the operator `new` is used. Here is an example:

```
char *s;
s = new char[4];
```

now `s` will be pointing to the first byte of a memory space (an array of 4 characters) on the heap. If the memory to some reason is not available, the statement returns `nullptr,` which is technically equal to zero and in fact pointer s will be pointing to nowhere. If the allocation of the memory is successfully done, you can use the allocated memory for your programming purposes. For example, you can write:

```
s[0] = 'A';
s[1] = 'B';
s[2] = '\0';
cout << s; // should print AB
```

Then, once your job with this allocated space is done you can free the memory by using:

```
delete [] s;
```

Now, the allocated memory is not any more available to your program, and if by mistake you try to delete it again. For example, writing:

```
delete [] s;
```

on most of the platforms you will get a runtime error. On my Mac machine, an error similar to this will appear (you may see a different error message on your machine).

```
ABEXB Solutions(2477,0x10039b380) malloc: *** error for object 0x100708d90:
pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
```

Therefore, most of the compilers don't like to see that de-allocated memory space to be de-allocated again. One solution to this problem is that once you a memory space is deallocated any pointer pointing to that spot to be set to `nullptr`:

```
s = nullptr
```

Part one - What to do:

Download the files `DynString.cpp, DynString.h, part1.cpp` from D2L.

Read the file `part1.cpp`. Try to visualize what the program will do, including calls to constructors and the destructor. Build an executable using `part1.cpp` and `DynString.cpp` and run it to see what it does.
The take the flowing steps to complete your assignment:

Step 1: Draw AR diagrams for:

- Point one, when the diagram reaches at this point for the first time.
- Point one, when the program reaches at this point for the second time.
- Point three,
- Point 4, when the program reaches at this point for the first time.

Step 2: Answer the following questions:

1. At point four in the main function, how many times the constructor of class DynString is called?
2. At point four how many times the destructor of the class DynString is called?
3. Overall how many times in total the destructor of the class DynString will be called in this program?
4. Answer the question that is noted in the main function: What is going wrong after you press the **return key** in the main function?

What to Submit:

Submit your diagram and the answers for the questions.

Part two -What to do:

Before starting this part, make sure to complete part one. It will certainly help you to understand and solve this part of the exercise.

The definitions of the `append` member functions is missing from the file `DynString.cpp`. This function is supposed to change the length of the string, so the function requires the following approach:

- Allocate a new array of the right length.
- Copy whatever characters need to be copied into the new array.
- Deallocate the old array.
- Adjust the value of the `lengthM` variable.

Your task is to write the function's implementation. To check that it works download `part2.cpp`, change the `#if 0` to `#if 1` in this file, compile and run your program, and make sure the program output is as it is expected. For your information, the C/C++ preprocessor directives `#if 0` and `#if 1`, is used to include or exclude any code segment that is between `#if` and `#endif`. If `#if` follows by `1` the code will be included, and if it is `0` will be excluded.

Please pay attention to the following points:

- The memory management strategy for `DynString` says that the dynamic array should be exactly the right length to hold its contents. Make sure that your definitions of `append` is consistent with this strategy.
- The program in `part2.cpp` is not a very thorough test harness. It's not hard to get the correct output even if some of your code is defective. Read your function definition carefully to make sure that it properly handles dynamically allocated memory.

What to Submit:

Submit the copy of your function called `append`, and the program output, as part of your lab report in PDF format.