ENSF 337: Programming Fundamentals for Software and Computer
Department of Electrical & Software Engineering
University of Calgary
Lab 7 - Fall 2022

M. Moussavi, PhD, P.Eng

Notice: As we mentioned at the beginning of the term, our labs by default are individual assignment and you cannot work with a group, unless that the document states it is a group lab. **This is NOT a group assignment, and you CANNOT work with a partner.**

Objective:

Here is the summary of the some of the topics that are covered in this lab:

- C++ objects on the computer memory
- Designing a C++ class
- Understanding the details of copying objects in C++

Marking Scheme:

Exercise A: 4 marks
Exercise B: not marked
Exercise C: 22 marks
Exercise D: 8 marks
Exercise E: 12 marks

Total: 42 marks

Important Notes:

- Exercise B is not marked, but it is an important exercise that helps you to understand how some of the elements of object-oriented programs, including: constructor, default constructor, and destructor work, and when and how they will be called.

- If you are compiling and running your program from command line, make sure to use C++ compilation command `g++` instead of `gcc`.

Due Dates:

Because of upcoming midterm exam on Thursday Nov 3rd and term break (Nov 7 to Nov 11), the due date is longer than usual, and both sections will have the same due date on:

Friday Nov 18, 9:00 AM

Late Due Date:

20% marks will be deducted from the assignments handed in up to 24 hours after each due date. That means if your mark is X out of Y, you will only gain 0.8 times X. There will be no credit for assignments turned in later than 24 hours after the due dates; they will be returned unmarked.

Exercise A:

The objective of this exercise is to help you in understanding how C++ objects are shown on a memory diagram, and how a member function of a class uses the `'this'` pointer to access an object associated with a particular function call. For further details please refer to your lecture notes and slides.

What to Do:

Download files `cplx.cpp`, `cplx.h`, and `lab7ExA.cpp` from the D2L and draw AR diagrams for: **point one** and **point two.**

For this exercise you just need to draw the diagrams. However, if you want to compile and run it from command line, 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 an executable:

```
g++ -Wall cplx.cpp lab7ExA.cpp
```

Please notice that you shouldn't have the header file name(s) in this command.

Exercise B: Construction and Destruction of Objects

Although this exercise is not marked, it a very important exercise. You are strongly recommended not to skip this exercise and try to understand its details.

What to Do:

Compile the files `lab7ExB.cpp`, and `lab7String.cpp`, run the program and fill out the following table to indicate which object or which pointer (a, p, d, b, c, or g) is associated with the call to a constructor or destructor. As an example, the answer for the first row is given – which indicates that the first call to the constructor (abbreviated as `ctor`) is associated with the `Lab7String` object, `a`.
Notes:

- Also, many programmer use `dtor` as an abbreviation for destructor.
- When compiling this program, compiler may give you warning for unused variables. Although warnings are always important to be considered and to be fixed, but in this exercise, we ignored those warnings to keep the program short and simple.

Program output in order that they appear on the screen	Call is associated with object(s):
`ctor called...`	`a`
`default ctor called...`	**p**
`default ctor called...`	**d**
`ctor called...`	**b**
`ctor called...`	**c**
`ctor called...`	**g**
`The first four calls to dtor`	**d, b, c, g**
`The last two calls to dtor`	**a, p**

Exercise C: Designing a C++ Class:

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 – Complete Design and Implementation 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 number of `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`.
3. 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`.
4. 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 `lab7ExC.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 `lab7Clock.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.

2. Create another file called `lab7Clock.cpp` and write the implementation of the member functions of class `Clock` in this file (remember to include "`lab7Clock.h`").
3. Compile files `lab7ExC.cpp` (that contain the given `main` functions) and `lab7Clock.cpp` to create your executable file.
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:

Submit your source code, `lab7Clock.h`, and `lab7Clock.cpp`, and the program's output as part of your lab report in pdf on the D2L Dropbox.

Exercise D: The DynString class

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.

Make a memory diagram for points-one, labeled in the function `truncate`.

Also answer the following questions:

- At point two in the main function, how many times the constructor and how many times the destructor of the class `DynString` is called?
- At the end of the main (just before main function terminates), how many times the constructor and how many times the destructor of the class `DynString` is called?

What to Submit:

Submit your diagram and the answer to the question s.

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.

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 append function, and the program output.

Exercise E: A Simple Class Vector and Copying Object

The objective of this exercise is to practice more code-level design concepts such as dynamic allocation and de-allocation of memory for class data members and to understand the concepts of copying objects.

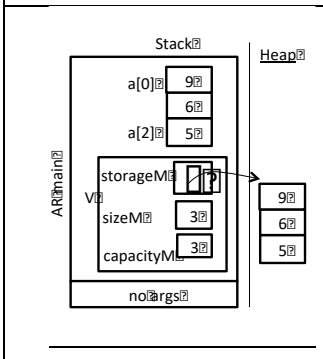
What to Do:

Download files `simpleVector.h`, `simpleVector.cpp`, and `lab7ExE.cpp` from D2L. Open the given files and read their contents carefully to understand the details of implementation of some of the given members. If you compile and run this program, you will notice that some of the expected test results are incorrect. This is because the `copy constructor`, the `assignment operator` (which is also known as `copy assignment operator`), and one of the member functions called `push_back` is missing.

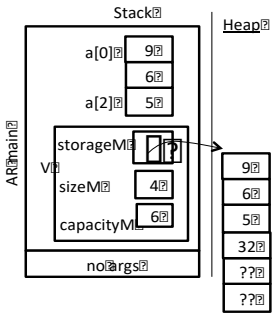
In this exercise your job is to complete the definition of these functions. In the given main function, part of the code that tests the copy assignment operator or copy constructor is commented out by being confined between `#if 0` … `#endif`. When ready to test your copy assignment operator and copy constructor, please change the `#if 0` to `#if 1`. You are also recommended to test one function at time; once one function works with no errors move to the next one. For this purpose you can move the location of the conditional compilation directives (`#if 0`) to an appropriate lower parts of the main function, as you progress. To better understand how this class is supposed to work, first read carefully the content of the given files and the comments on memory policy. Also, see the following example for a main function that creates an object of class `SimpleVector`, and the AR diagrams at points one, two, and three. The diagrams show how the object `v` is initially created, and then what will happen to the object if the function `push_back` works as it is expected.

```
int main(void)
{
   int a [] = { 9, 6, 5 };
   SimpleVector v(a, 3);
   // point 1
   v.push_back(32);
  // point 2
  v.push_back(55);
  // point 3
  return 0;
}
```

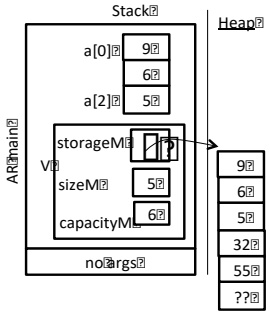
Point 1



Point 2



Point 3



What to Submit:
Submit your `simpleVector.cpp` and the output of the program, as part of your lab report in pdf format.