# ENSF 337: Programming Fundamentals for Software and Computer Lab 8 Assignments

M. Moussavi
Department of Electrical & Computer Engineering
University of Calgary

## Notes:

In this lab you can work with a partner. If you chose to work with a partner:

- You partner must be in your lab section. In other words partner cannot be from different lab section, B01 and B02.
- You should work on all exercises with your partner.
- You have to submit one lab report with both names (your name and your partner's name).
- Groups of more that two are NOT allowed.

Due to upcoming midterm exam on Thursday Nov 1st, and with regard to reading-week period that we not have lectures and labs, the due dates and the format of submitting your reports for this lab is different from previous labs:

- There is **no in-lab assignment** for this lab.
- The due dates for different exercises are as follows:
  - For exercises A, B, and C: Thursday Nov 8, 2018 at 5:00 PM.
  - For exercises D, and E: Thursday Nov 22nd at 2:00 PM.

Material related to Linked list and C++ text-file manipulation will be discussed during the week of Nov 5th.

## Objective:

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

- Designing C++ classes and understanding of code-level design of C++ classes
- Understanding the concepts of "Law of Big 3" in design and development of C++ class and the low-level design issues when copying objects.
- Designing and writing a complete program that uses Linked List.

## Marking scheme:

The total mark for the exercises in this lab is: **80 marks**

| | |
|---|---|
| Exercise A | – 22 marks |
| Exercise B | – 12 marks |
| Exercise C | – 12 marks |
| Exercise D | – 12 marks |
| Exercise E | – 22 marks |

## Exercise A: 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 `lab8ExA.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 `lab8Clock.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 `lab8Clock.cpp` and write the implementation of the member functions of class `Clock` in this file (remember to include `"lab8Clock.h"`).

3. Compile files `lab8ExA.cpp` (that contain the given `main` functions) and `lab7Clock.cpp` to create your executable file. Reduce
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, `lab8Clock.h`, and `lab8Clock.cpp`, and the program's output as part of your lab report on the D2L Dropbox, before Thursday Nov 8 at 5 PM.

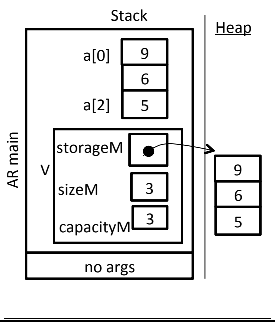
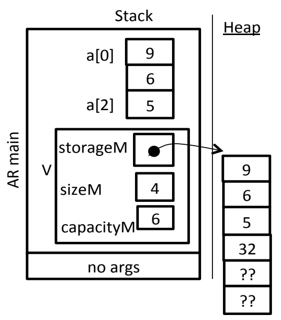
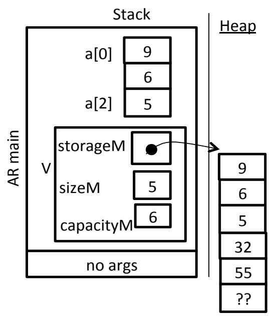
**Exercise B (12 marks): A Simple Class Vector and Copying Object**

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 `lab8ExB.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` are 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) | Point 1 | Point 2 | Point 3 |
|---|---|---|---|
| {<br>int a [] = { 9, 6, 5 };<br>SimpleVector v(a, 3);<br>// point 1<br>v.push_back(32);<br>// point 2<br>v.push_back(55);<br>// point 3<br>return 0;<br>} |  Stack<br>Heap<br>a[0]<br>9<br>6<br>a[2]<br>5<br>AR main<br>v<br>storageM<br>sizeM<br>3<br>capacityM<br>3<br>no args<br>9<br>6<br>5 |  Stack<br>Heap<br>a[0]<br>9<br>6<br>a[2]<br>5<br>AR main<br>v<br>storageM<br>sizeM<br>4<br>capacityM<br>6<br>no args<br>9<br>6<br>5<br>32<br>??<br>?? |  Stack<br>Heap<br>a[0]<br>9<br>6<br>a[2]<br>5<br>AR main<br>v<br>storageM<br>sizeM<br>5<br>capacityM<br>6<br>no args<br>9<br>6<br>5<br>32<br>55<br>?? |

**What To Submit:**

Submit your `simpleVector.cpp` and the output of the program, as part of your lab report.

## Exercise C (12 marks): Tracing linked list code

**What To Do**

Download the files `OOList.cpp, OLList.h` and `lab8ExC.cpp` from D2L. Read the files carefully. Points one and two have been marked in the definition of `OLList::insert` and point three in function `OLList::remove`. Draw memory diagram when program reached these points **for the first time**.

*Submit your diagrams. Please provide a digital copy of your diagram using a scanner, and post it as part of your lab report on the D2L. Also make sure the scanned copy of your diagram is clear and readable.*

## Exercise D (12 marks)

**Part I:**

If you read file downloaded for exercise C, you will notice the member function `OLList::copy` is a helper for both the copy constructor and the assignment operator. Its job is to generate a new chain of nodes with items identical to the items in an existing `OLList` object. In the files downloaded for exercise C, rewrite the definition of `OLList::copy` so that it does what it is supposed to do. Do not call `insert` from your `copy` function; instead allocate nodes using `new`, and manipulate pointer variables to create links. There are two reasons to avoid `insert`:

1. Use of `insert` in this situation is inefficient--each insertion causes an unnecessary list traversal.
2. You need practice manipulating pointers in linked lists.

Now download file `lab8ExD.cpp` from D2L. This file contain a main function to be used for testing your exercise D. Change the first preprocess directive from `#if 0` to `#if 1`. Then compile and run the program to test if your `OLList::copy` function works.

**Part Two:**

The given `OLList::remove` function also doesn't do the right job to remove some nodes (only removes the nodes located at the beginning of the list properly). There is a missing code segment that you should add.

Now, in the file l`ab8ExD.cpp` change the second preprocess directive from `#if 0` to `#if 1`. Then compile and run the program to test if your `OLList::remove` function works.

**Part Three:**

The `OLList::destroy` function is a helper for both the copy constructor and the assignment operator. Its job is to remove the entire set of nodes in a liked list. In other words it should delete the entire nodes in an existing linked list, one by one, using a loop.
The given `OLList::destroy` function doesn't do the right job to remove dynamically allocated nodes of a linked list. In fact the given code for the `OLList::destroy` function causes a memory leak. Your job is to rewrite the definition of `OLList::destroy` so that it does what it is supposed to do.

*What to submit: `OLList.cpp`, and the program output as part of your lab report.*

## Exercise E (22 marks): A Complete Program - Designing a Linked List Application

In this exercise you are going to analyze, design, and develop a program that reads river flow data stored in a text file, and creates a linked list that represent a database for the revivers annual records, and allows some statistic analysis.

## Read this First

Basically, the techniques for file input and output in C++ are virtually identical to the standard input /output when reading from keyboard and writing to the screen. The purpose of this section is to explain briefly the file I/O in C++ (reading from and writing into text files). This section only reviews the basics of file I/O that you need to complete this exercise.

To open a text file for input (reading data), you need to take the following steps:

First, you must include a header called *fstream*:
```
        #include <fstream>
        using namespace std;
```

Second, you have to create an object of a class called `ifstream`. Here is an example of creating an `ifstream` object:

```
      ifstream  inObj;
```

Third, to read data from a file called *mydata.txt*, you need to open the file. To open a file located in the current working directory, you write:

```
            inObj.open ("mydata.txt");
```

You can put a complete file path between double quotation marks.

Fourth step is to use *inObj* exactly like *cin* to read data from the text file *mydata.txt*. For example, to read two integer number and store them in integer variables a, and b, you can write:
```
      inObj >> a  >> b;
```

You can also use functions such as `inObj.eof()` to check for end of file. This function returns true if process of reading encounters an end of file.

You need to take similar steps to open a text file for output (writing data). First, you must have included the header file, fstream.h (the same file that is also required for input). Then, you need to create an object of a class called *ofstream*, and to open the output file. See the following example:

```
        ofstream    outObj;
        outObj.open("myoutput.txt);
```

Now, you can use *outObj*, similar to *cout*, to write any data into *myoutput.txt*. For example, you can write the values of two integers *a* and *b* into *myoutput.txt*.

```
        outObj   <<  setw (10) << a  << setw (15) << b << endl;
```

Although all opened files will be automatically closed, when the C++ programs terminate, it is always a good practice to close them manually, whenever you don't need them anymore:

```
inObj.close();
outObj.close();
```

Here is a complete example of a C++ program that opens a file for writing, writes two integers into that file and then closes the file.

```
#include <iostream.h>
#include <fstream.h>

int main() {
   char* infile = " /usr/mydirectory/myoutput.txt";
   int a = 2543,  b = 465;

   // Create an ofstream object named outObj and open the target file
   ofstream outObj ;
   outObj.open(infile)

   // Make sure if file was opened successfully
   if (! outObj)  {
        cout << "Error: cannot open the file << filename << endl;
        exit(1);
   }

   // store values of two integers,  a, and b, in the file
   outObj <<setw(15) << a << setw(15) << b << endl;

   outObj.close();
   return 0;
}
```

***Note:If you want to run this program under the Microsoft Windows environment, and for example your file is located in the directory: c:\mydirectory, you have to assign the file path to infile, as follows:***

```
char* infile = "c:\\mydirectory\\myoutput.txt"
```

**<u>Any backslash must be preceded with another backslash.</u>**

When you open a file for writing, there are two common file open modes, "truncate" and "append". By default, if no mode is specified <u>and the file exists</u>, it will be truncated (its data will be lost).

To append data (add the data to the end of file), you can open the file in append mode:

```
OutObj.open("/usr/mydirectory/myoutput.txt", ios::append);
```

Further details about different file open modes, and file types such as binary files are deferred to future lectures or lab instructions.

## What to Do

Assume engineering firm that works on hydropower plants has asked you to design and write a program in C++ to help them to study the variation of annual water flow in a river of their interest. There should be only one value of flow for each year (no duplications). Here is an example of the format of the records to be used:

```
Year     Flow (in billions of cubic meters)

1963      321.5
1961      322.7
```

Your program should be able to calculate the average and the median flow in the list, and also to be able to add new data to the list, or remove data from the list. An example of the sequence of operations that your program must perform is explained in the rest of this section:

The program starts with displaying a title and brief information about the program. For example:

```
Program: Flow Studies – Fall 2017
Version: 1.0
Lab section: B??
Produced by: Your name(s)

<<< Press Enter to Continue>>>>
```

Then it should read its input from a text file called *`flow.txt`* (posted on the D2L), and create a linked list of the data (year, flow). **<u>You are not allowed to use arrays to store these data</u>**. The program should be written in several modules (we call the collection of a `.cpp` and a `.h` file, a module)

## First Module:

The program should have a module that contains a header file called `list.h` and `list.cpp`. This module will contain two structures and one class as follows:

1. Structure called ListItem that maintains an annual flow record (year and flow):

```
struct  ListItem {
          int year;
          double flow;
   };
```

2. Structure called Node that contains the data (an instance of ListItem) and a pointer to Node:

```
struct Node {
          ListItem item;
          Node *next;
   };
```

3. A class called `FlowList` that holds and manages a set of nodes that contains `ListItems`. This class can be similar to class `OLList`, in the previous exercise. However you may need to add member function to get access to the item value of a node. For this purpose you can also add another data member of type `Node*`, to be able to set it to different nodes when traversing through the list and have access to the item in the node (this is just a suggestion and you can come up with different solutions if you like).

## Second Module:

Your program should have a second module that contains two files (`hydro.cpp, and hydro.h`). Some of the required functions to be defined in this module include:

- `main` that creates and uses the objects of `FlowList`.
- `displayHeader` - that display the introduction screen:

```
Program: Flow Studies – Fall 2017
Version: 1.0
Lab section: B??
Produced by: Your name(s)
<<< Press Enter to Continue>>>>
```

- Function `readData` - that reads records years and flows from input file (`flow.txt`), inserts them into the list, and returns the number of records in the file.
- Function `menu` - that displays a menu and returns the user's choice (an integer 1 to 5).
- Function `display` - that displays years and flows, and shows the **average** and the **median** of the flows in the list (calling functions `averge`, and `median)`.
- Function `addData` - that prompts the user to enter new data, inserts the data into the linked list, and updates the number of records.
- Function `removeData` – that prompts the user to indicate what year to be removed, removes a single record from the list, and updates the number of records.
- Function `average` – that returns the flow average in the given list
- Function `median` – that returns the median flow.
- Function `saveData` - that opens the flow.txt file for writing and writes the contents of the linked list (annual flow records) into the file.
- Function `pressEnter` - that displays `<<<Press Enter to Continue>>>`, and waits for the user to press the <Return Key> . You can implement this function by printing the message followed by a call to function cin.get() that works similar to functins fgetc() or getc()in C. Here is the statements needed:

```
cout << "\n<<< Press Enter to Continue>>>>\n";

cin.get();
```

Note: These function are all global functions like main(), not a class member function of a class.

## Samples run of the program:

To give you a better idea that how the program is supposed to work, here are some screenshots of the sample run of the program. The program starts with displaying a title and brief information about the program. For example:

```
Program: Flow Studies, Fall 2017
Version: 1.0
Lab section: Your lab section
Produced by: Your name(s)

<<< Press Enter to Continue>>>>
```

When user presses <Enter>, the program opens an input file called *flow.txt*. The program should give an error message and terminate, if for some reason it cannot open the file.

If the file exits it reads the data (years and flows) and insert the data into the linked list in ascending order, based on the flow data (**Not** year). **Then, closes the file**, and displays the following menu:

```
Please select on the following operations
   1. Display flow list, average and median
   2. Add data.
   3. Save data into the file
   4. Remove data
   5. Quit
   Enter your choice (1, 2, 3, 4, of 5):
```

If user enters 1: the program displays the content of the list on the screen in the ascending order based on the flow data. For example:

```
Year      Flow (in billions of cubic meters)
2003        99.5
2002       100.0
1964       222.7
1963       321.5
…
The annual average of the flow is: ... millions cubic meter
The median flow is:... millions cubic meter.

<<< Press Enter to Continue>>>>
```

When user presses <Enter> the menu will be displayed again:

```
   Please select on the following operations
  1. Display flow list, average and median
     2. Add data.
     3. Save data into the file
     4. Remove data
     5. Quit
     Enter your choice (1, 2, 3, 4, of 5):
```

When user selects 2, the program prompts the user to enter a year and then the flow. For example user can enter the following values (in bold):

```
Please enter a year: 1995
Please enter the flow: 102.99
```

If data for the same year doesn't exit, the program should insert the record in proper order (ascending order, based on flows) in the list, and display the following message:

```
New record inserted successfully.
<<< Press Enter to Continue>>>>
```

If the year already exists in the list, the program displays the following message:

```
Error: duplicate data.
<<< Press Enter to Continue>>>>
```

If user selects 3 from the menu options: the program opens `flow.txt` again, but this time for writing into the file, by creating an *`ofstream`* object. Then, it writes the data (years and flow) into the file, closes the file, and displays the following messages on the screen.

```
Data are saved into the file.
<<< Press Enter to Continue>>>>
```

When user selects 4, the program prompts the user to enter the year that he/she wants to be removed:

```
Pleas enter the year that you want to remove: 1995
```

If data exist, the program should remove the record from the list, then displays the following message:

```
Record was successfully removed.

<<< Press Enter to Continue>>>>
```

(The program at this point doesn't rewrite the records into the data file)
If the requested year doesn't exist in the list, the program displays the following message:

```
Error: No such a data.
<<< Press Enter to Continue>>>>
```

If user selects 5: the program terminates, showing the following message:

```
    Program terminated successfully.
```

Your main function should test and shows all the functionalities of your program.  As illustrated in the following example a good option would be to use a C/C++ `switch` statement.

```
int main(void) {
  FlowList x;
  int numRecords;
  displayHeader();
  numRecords = readData(x);
  int quit =0;

  while(1)
  {
    switch(menu()){
            case 1:
                  // call display function;
                       // call pressEnter;
                  break;
            case 2:
                  // call addData function
                  // call pressEnter;
                  break;
            case 3:
                  // call saveData function;
                       // call pressEnter;
                  break;
            case 4:
                  // call removeData
                  // call presenter;
                  break;
            case 5:
                  cout << "\nProgram terminated!\n\n";
                  quit = 1;
                  break;
            default:
                  cout <<  "\nNot a valid input.\n";
                  // pressEnter();
    }
   if(quit == 1) break;
  }
```

Normally you are expected to have learned about `switch` statement in previous courses. However if you didn't, please study this topic in one of your C or C++ textbooks.

*Submit all your program files, and the programs output that shows your program work. Be sure to show all of the possible actions from the menu, including attempts at duplicate year entries and removing a non-existent year.*