University of Calgary
Department of Electrical and Computer Engineering
ENSF 614
Lab 6, Fall 2021

Instructor: M. Moussavi, PhD, Peng.

Notes:

- **This is an INDIVIDUAL lab assignment, and you are NOT allowed to work in a group.**
- The first two exercises are your last lab exercises in C++. The third exercise in this lab refers to the material discussed in the second chapter of the course, "Design Pattens". In this new part of the course some of the lab exercise will be in Java and some in C++.

Due Dates:

Tuesday Nov. 16, before 5:00 PM

Objectives:

The purpose of this lab is:

1. More practice to understand concepts data structures in C++
2. Understand the concepts of overloading operators in C++.
3. Understanding and developing a simple program in Java that uses one of the important design pattern models called "Strategy Pattern", and "Observer Pattern".

Marking scheme:

The total mark for the exercises in this lab is: **59 marks**

- Exercise A (14 marks)
- Exercise B (16 marks)
- Exercise C (12 marks)
- Exercise D (2 marks)
- Exercise E (15 mark)

Exercise A (14 marks):

Part I – Drawing an AR Diagram for a Dictionary Data Structure (5 marks):

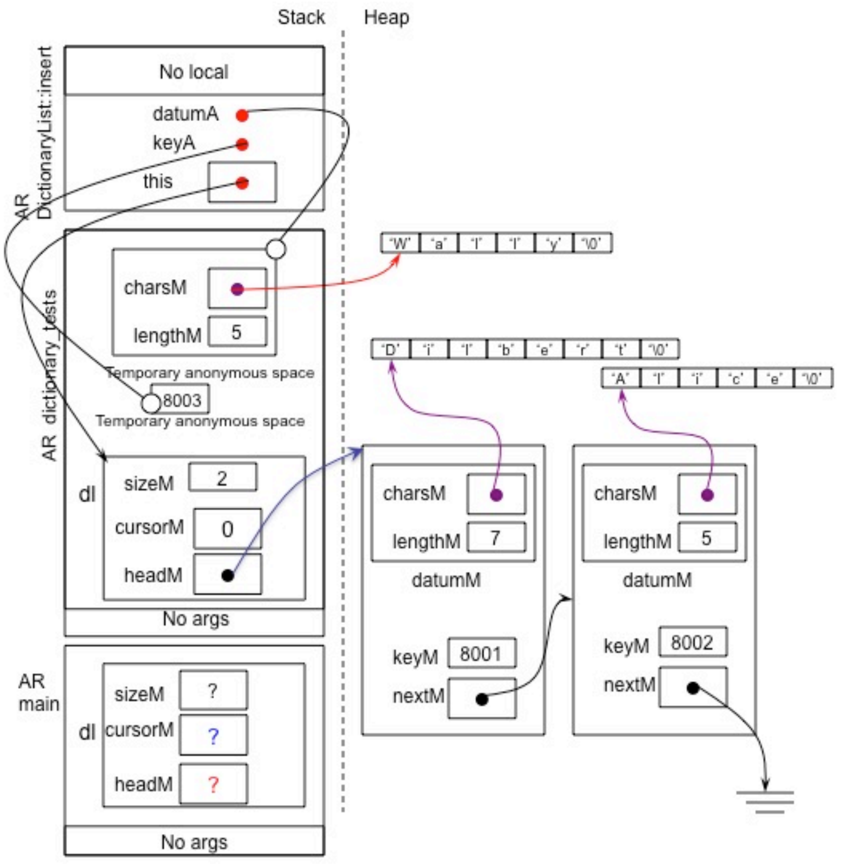
Dictionary is a data structure, which is generally an association of unique keys with some values. (Other common names for this type of data structures are Map and Lookup Table). *Dictionaries* are very useful abstract data types (ADT) that contain a collection of items called *keys*, which are typically strings or numbers. Associated with each key is another item that will be called a *datum* in this exercise. (``Datum" is singular form of the plural noun "data".)

Typical set of operations for a Dictionary include: inserting a key with an associated datum, removing a key/datum pair by specifying a key, and searching for a pair by specifying a key. Dictionaries can be implemented using different data structure such as arrays, vectors, or linked lists. In this exercise a linked list implementation, called `DictionaryList` class is introduced. Class `DictionaryList`, in addition to a node-pointer that usually points to the first node in the linked list has another node-pointer called `cursor` that is used for accesses to the individual key/datum pairs.

What to do:

Download files `dictionaryList.h`, `dictionaryList.cpp`, `mystring.h`, `mystring.cpp`, and `exAmain` from D2L. Read them carefully and try to understand what the functionalities of these classes are doing. Note also the definition of class `Node` that contains three private data members: `keyM`, `datumM`, and `nextM`, a pointer of type `Node`. Also, class `Node` declares class `DictionaryList` as a `friend`. For details about `friend` keyword in C++, please refer to your lecture notes.

The idea of implementing a dictionary as a linked list is simple: each node contains a key, a datum, a pointer to the next node, and another node pointer called cursor that can either point to any node or can be set to NULL. For better understanding of the details of `DictionaryList`, try to understand how the function `insert` works. Further details about the operation of inserting a node into the list, can be learned from following AR diagram, which represents **point one** in this function, when reaches this point for the second time:



Now that you know how class `DictionaryList` and its member function insert work, draw an AR diagram for **point TWO** in function remove, when the program reaches this point for the first time.

Important Note: If you read this exercise carefully, you will find that this exercise is an excellent example for learning more about concept of destroying and copying C++ objects. It tells you more about how de-allocate memory of nodes in a linked list, and how to make copies of a linked list object. I recommend you pay attention to these details and if you have any questions please do not hesitate to contact me.

What to Submit:

Submit you AR diagram.

Exercise B - Overloading Operators in C++ (16 marks)

In this exercise you are going to use the same files that you have used in exercise B, and you will overload several operators, for classes Mystring, and DictionaryList.

What to Do:

Open the file `exAmain.cpp` and uncomment the line that calls function `test_overloading`, and its prototype at the top of the file. Then, change the conditional compilation right above the implementation of `test_overloading` from `#if 0` to `#if 1` and try to compile or run the program. Now you will see a few errors. These errors are due to the fact that this function is trying to make some operations on Mystring or DictionaryList objects using common C++ relational operators such as, >, <, <=, !=, or other type of operators such as << or [] that are not by default defined by C++ compiler for objects of Mystring or DictionaryList. Your job in this assignment is to find out which operator is required to be overloaded and write the necessary code in `mystring_B.h, mystring_B.cpp, dictionalyList.h` and `dictionaryList.cpp`. For example, one of the lines in `exBmain.cpp` is:

```
if(dl.cursor_datum() >= (dl2.cursor_datum()))
    cout << endl << dl.cursor_datum() << " is greater than or equal " <<
dl2.cursor_datum();
```

In the above if-clause, the binary operator >= is used to compare two `Mystring` objects and find out which one is greater than the other one (lexicographically). Then it tries to print the value of string on the screen using C++ insertion operator <<.

Without overloading the operators >= , and << for class Mystring, this line will show errors such as: `Invalid operands to binary expression Mystring`

Since there are several operators to be overloaded, again the best strategy is to work incrementally. Means first comment out most of the lines in function terst_overloading, except the two lines that uses the operator >= and << . Then write the necessary code in the given files. If they work fine, then uncomment the next few lines to implement and test the next operator -- until all required operators are properly defined and tested.

What to Submit:

For exercises B following files electronically on the D2L:

- As part of your lab report (PDF file), submit the copy of all .cpp and .h files, and the output of your program.
- Submit a zipped file that contains your actual source code (all .cpp and .h files)

Exercise C - Strategy Desing Pattern

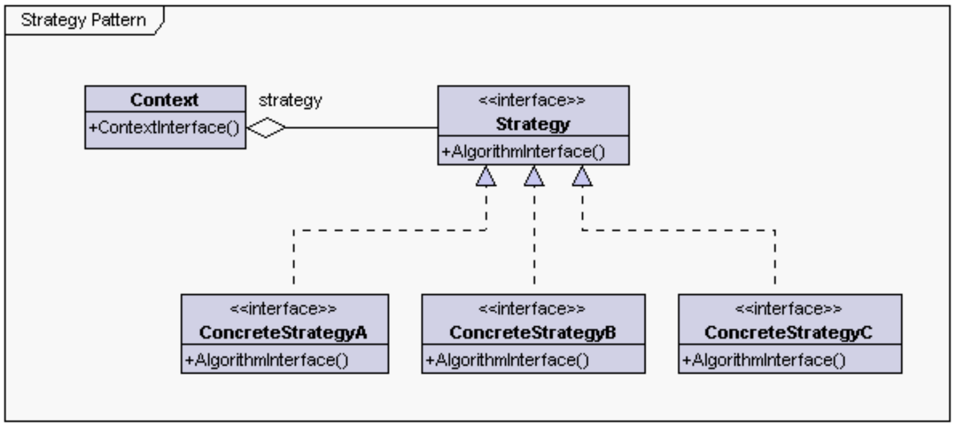
The purpose of this exercise is to give you an opportunity to practice using Strategy Design Pattern in you program.

Read This First – Why and Where to Use Strategy Pattern

The subject of applying different strategies at different points of time (for good reasons) is a real demand in many real-world projects or processes. This is also a true requirement for many software applications. One of the most common applications of the Strategy Pattern is where you want to choose and use an algorithm at the

runtime. A good example would be saving files in different formats, running various sorting algorithms, or file compression.
In summary, Strategy Pattern provides a method to define a family of algorithms, encapsulate each one as an object, and make them interchangeably used by a client.

Read This Second – A Quick Note on Strategy Pattern
The Strategy Pattern is known as a **behavioural** pattern - it's used to manage algorithms, relationships and responsibilities between objects. The definition of Strategy Pattern provided in the original Gang of Four book states:



The diagram shows that the objects of Context provide means to have access to objects that implement different strategy.

What to Do:

Since last Spring the subject of Java Generic was not covered. We have two versions of Exercise C and D. One exercise which is not using Java Generic, and related files are posted in subdirector EXC_D-V1. The second version is for those of your that seeking for more challenge. The files related to this version is stored under the subdirector EXC_D-V2. No matter which version you choose the rest of the instructions will be the same:

Assume as part of a team of the software designers you are working on an application that allows its clients to be able to use different sort methods for a class called `MyVector`. For the purpose of this exercise you just need to implement two sort methods: `bubble-sort`, and `insertion-sort`. And of course, your design must be very flexible for possible future changes, in a way that at anytime the client objects should be able to add a new sort technique without any changes to the class `MyVector` (for the generic version will be MyVector<E>).

Please follow these steps:

Step 1: Download file `DemoStrategyPattern.java` form D2L. This file provides a client class in Java that must be able to use any sort techniques at the runtime.

Step 2: Download file called `Item.java` This is a class that represents data object. It means its private data member `item` can be used and sorted within the body of `MyVector` objects.

Step 3: Now your program must have the following classes:

- Class `MyVector` (or MyVector <E> which is also Bound to only Java Number type and its decedents). This class should have a private data member called `storageM` of type `ArrayList<Item>` (or ArrayList< Item<E> > for the generic version), which provides space for an array of certain size, and more

data member as needed, and a second private data member called `sorter` that is a reference to an object of the Java interface Sorter (or interface Sorter <E> for the generic version).
Class `MyVector` should also have at least two constructors as follows:

- A constructor that receives only an integer argument, n, to allocate memory for an array with n elements.
- A constructor that receives only an ArrayList object, `arr`, and makes `storageM` an exact copy of `arr`.

Also must have at least the following methods, which are used in the client class `DemoStrategyPattern`:

`public void add(Item value)`: That allows to add a new Item value to `storageM`

`public void setSortStrategy(Sorter s)`: That allows its private data member register with a an object that implements `Sorter`.

`public void performSort()`:That allows sort method of any sorter object to be called.

`public void display()`:That displays data values stored in `storage` on the screen in one line. For example: `1.0  2.0  3.0  4.0  5.0`

- Two Concrete classes called `BubbleSorter` and `InsertionSorter` that one implements a bubble sort algorithm and the other one implements insertion sort algorithm.

Exercise D:

This is a smaller size exercise. The purpose of this exercise is to demonstrate how you can add a new algorithm to exercise A called `SelectionSorter` that uses selection-sort and can be used by the class client without making any changes to the class `MyVector`.

What to Submit for Exercises C and D?

1. Copy and paste all your source codes and your program output as part of your lab report and submit it in PDF format into the D2L Dropbox.
2. Create and submit a zip file that contains your source code file (.java files) and submit it on the D2L Dropbox.

Exercise E (15 marks):

The purpose of this exercise is to give you an opportunity to practice using Observer design pattern in a simple Java program.

Read This First – A Quick Note on Observer Pattern

The Observer pattern is also one of the **behavioural** patterns - This pattern is also used to form relationships between objects at the runtime.

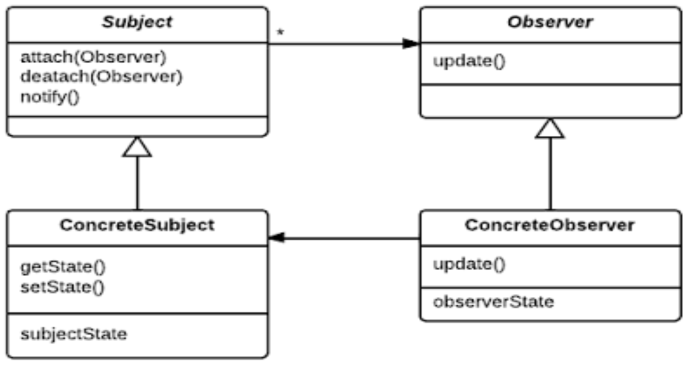


Figure 1

Figure 1: shows that the concrete subjects can add any observers, and when any changes happen to the data, all observers will be notified. The following figures may help you to better understand how this pattern works.

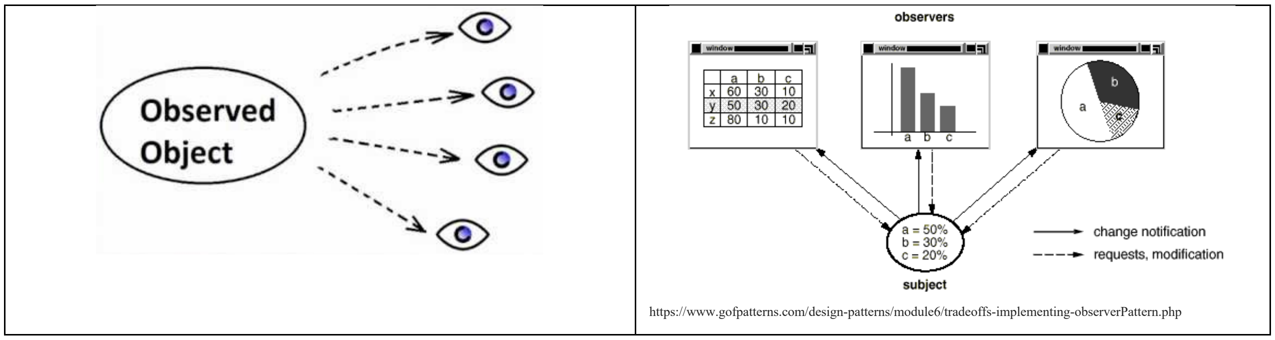
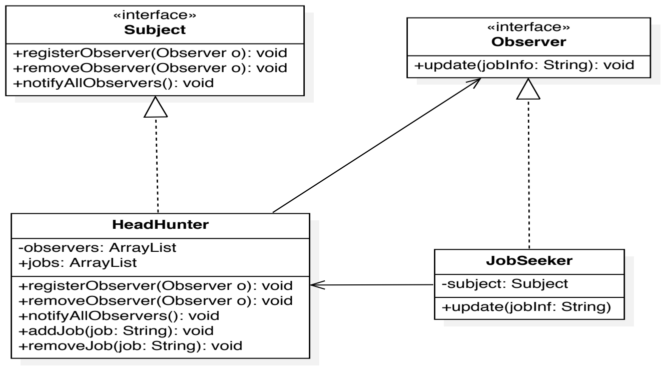


Figure 2

Figure 2 is a simple demonstration of the observers' notification concept and figure 3 illustrates the most common form of using this pattern. Any changes to the subject (data a, b or c) will be immediately translated in three observer views (tabular format, bar chart, or pie chart).
Using observer pattern is not limited to GUI presentation; it can be used for any notification system. Here is another example:


«interface»
Subject
+registerObserver(Observer o): void
+removeObserver(Observer o): void
+notifyAllObservers(): void
«interface»
Observer
+update(jobInfo: String): void
HeadHunter
-observers: ArrayList
+jobs: ArrayList
+registerObserver(Observer o): void
+removeObserver(Observer o): void
+notifyAllObservers(): void
+addJob(job: String): void
+removeJob(job: String): void
JobSeeker
-subject: Subject
+update(jobInf: String)

What to Do:

Download file `ObserverPatternController.java` form D2L. This file provides a client class that demonstrates how your observer pattern works. For the purpose of this exercise you just need to have three observers, and your design must be very flexible for change. In other words at anytime you should be able to add a new observer or remove an observer without any changes to the subject or observer classes. Your program must have the following interfaces and classes:

- Interface `Observer` with the method `update` that receive a parameter of type `ArrayList<Double>`
- Interface `Subject` with the required methods.
- Class `DoublArrayListSubject,` with a data list of type ArrayList<Double> , called `data` that is supposed to be visible to the observers. Consider other data members as shown in the Observer Pattern Design Model. This class should also have at least the following methods:
  - A default constructor that initializes its data members as needed. For example should create an empty list for its member called `data`.
  - Method `addData` that allows a new Double data to be added to the list
  - Method `setData` that allows changing the data at any element in the list
  - Method `populate` that populates the list with the data supplied by its argument of the function, which is an array of `double`.
  - Other methods as needed

- Three concrete Observer classes as follows:
  - Class `FiveRowsTable_Observer` This class should have a function display that shows the date in 5 rows as illustrated in following example (any number of columns, as needed):

```
10      30      11

20      60      23

33      70      34

44      80      55

50      10
```

    This class should also have a constructor to initialize it data member(s) as needed and to register the object as an observer.

  - Class `ThreeColumnTable_Observer` that displays the same list of data in tabular format as illustrated in the following example (3 columns and any number of rows as needed):

```
10     20     33

44     50     30

60     70     80

10     11     23

34     55
```

    This class should also have a constructor to initialize it data member(s) as needed and to register the object as an observer.

  - Class `OneRow_Observer` that displays the same vector of data in single line as follows:

```
10 20 33 44 50 30 60 70 80 10 11 23 34 55
```

    This class should also have a constructor to initialize its data member(s) as needed and to register the object as an observer.

If you have all the classes and methods defined properly, your program with the given client class `ObserverPatternController` should produce the following output:

```
Creating object mydata with an empty list -- no data:
Expected to print: Empty List ...
Empty List ...
mydata object is populated with: 10, 20, 33, 44, 50, 30, 60, 70, 80, 10, 11, 23, 34, 55
Now, creating three observer objects: ht, vt, and hl
which are immediately notified of existing data with different views.

Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 33.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
33.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 33.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0

Changing the third value from 33, to 66 -- (All views must show this change):

Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
66.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0

Adding a new value to the end of the list -- (All views must show this change)

Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0 1000.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
66.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0 1000.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0 1000.0

Now removing two observers from the list:
Only the remained observer (One Row ), is notified.

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0 1000.0 2000.0

Now removing the last observer from the list:

Adding a new value the end of the list:
Since there is no observer -- nothing is displayed ...
```

```
Now, creating a new Three-Column observer that will be notified of existing data:
Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0 1000.0
2000.0 3000.0
```

What to Submit for Exercise E?

3. Copy and paste all your source codes and your program output as part of your lab report and submit it in PDF format.
4. Create and submit a zip file that contains your source code (.java file(s))