OOP LAB 12

TOTAL MARKS: 90

THE OBJECTIVE OF THIS LAB IS TO:

1. Understand and practice polymorphic behavior on different levels.
2. Understand and practice function overriding and virtual functions/classes.
3. Practice good coding conventions e.g commenting, meaningful variable and functions names, properly indented and modular code.

INSTRUCTIONS!

1. This is a graded lab, you are strictly NOT allowed to discuss your solutions with your fellow colleagues, even not allowed asking how is he/she is doing, it may result in negative marking. You can ONLY discuss with your TAs or with me.
2. Strictly follow good coding conventions (commenting, meaningful variable and functions names, properly indented and modular code.
3. There should not be more than 5 lines of code in Main. 10 marks will be deducted in violation

> "LET THE WATERS SETTLE AND YOU WILL SEE THE MOON AND THE STARS MIRRORED IN YOUR OWN BEING."
>
> - RUMI

CONSIDER AND IMPLEMENT THE FOLLOWING FILE DEFINITIONS:

COMMAND.h

```
#ifndef ICOMMAND_H
#define ICOMMAND_H

class COMMAND
{
public:
    virtual bool canExecute(const CString & data) =0;
    virtual void Execute(const CString & data) =0;
    virtual CString getCommandName() =0;
}

#endif
```

Command is a pure virtual class that provides an interface for the above-mentioned functions. Since they are pure virtual functions, there implementation is a mandatory in every class that inherits them. CanExecute returns true if a command is executable and Execute is the actual implementation of the command. Similarly getCommandName tells you the which is the current command you are working in (helps you understand the flow)

ClearCommand.h

Here is a sample execution of the Clear Command.

```
class ClearCommand: public virtual COMMAND
{
private:
    CString commandName;
public:
    Clear(){
        commandName = "Clear";
    }
    bool canExecute(const CString & data){
        if(isEmpty())
            return false;
        else return true;
    }
    void Execute(const CString & data){
        data.remove(0, data.getLength());
    }
    CString getCommandName(){
        return commandName;
    }
}
```

Button.h

Button is another pure virtual class that serves as an abstraction between a user and a Command. There are three pure virtual methods mentioned in it. SetCommand sets a specific command in the command pointer according to its context, whereas getCommand returns the command name. ButtonExecute executes the command.

```
class Button
{
private:
    COMMAND * buttonCommand;
public:
    virtual void setCommand() = 0;
    virtual void getCommand() = 0; //returns the commandName
    virtual void buttonExecute() = 0;
}
```

ClearButton.h

Here is a sample implementation of the Clear Button.

```
class ClearButton: public virtual Button {
public:
    void setCommand()
    {
        buttonCommand = new ClearCommand();
    }
    void getCommand()
    {
        buttonCommand->getCommandName();
    }
    void buttonExecute()
    {
      if(buttonCommand->canExecute())
        buttonCommand->Execute();
    }
}
```

Window.h

Here is the class that will interact with the user and will allow him to insert or remove a button from the button array. Perform action will allow them to execute a particular button command in the index.

Note: you are only allowed to make extra functions in this class only. The functionality of main should also be defined here.

```
class Window
{
private:
    Button * buttons;
    CString data;
public:
    void insertButton();
    void removeButton();
    void performAction(int index);
}
```

Implement the Buttons for following Commands

- Clear

Clears the content of the data

- Add

Asks user to enter a string and appends it to the content of the data

- Remove

Asks user to enter a string and removes it (if present) from the content of the data

- ConvertToLower

Converts the entire data to lower case

- ConvertToUpper

Converts the entire data to upper case

- ConvertToInt

If possible, it converts the entire data to Integer or shows an error.

- ConvertToFloat

If possible, it converts the entire data to Float or shows an error.

- Display

Displays the data on the console

- Reverse

Reverses the entire content of data

Bonus: Collection API

MARKS: 30

The Collection API is the framework that provides an architecture to the stores and manipulate the group of projects and basically it is a package of data structures that includes Array lists, Linked-lists, Hash sets, etc.

A collection is simply an object that groups multiple elements into a single unit. It simply means a single unit of objects, collection API provides many interfaces (set, list etc) and classes. We can call it as container also, here collection allows some duplicates and others do not.

You have studied and implemented about three Data Structures up till now, which are Arrays, Matrices and Sets.

In your case, if you are implementing an object ADT, let's say Buttons, you must design it in a way that it must be able to implement the Collection APIs for the given Data Structures. And you must provide virtual functions that must be implemented by anyone who uses that particular API.

So, the classes become like this.

```
class ButtonArray
{
    Button * buttons;
    int capacity;
    bool isEmpty();
    bool isFull();
public:
    ButtonArray(int size);
    ~ButtonArray();
    ButtonArray(const ButtonArray & ref);
    void reSize(int newSize);
    void addButton(const Button & button);
}

class ButtonCollection()
{
public:
    virtual ButtonArray createbuttonCollection() = 0;
}
```

```
class CStringButtonCollection: public ButtonCollection
{
private:
    ButtonArray buttons;
public:
    void createButtonCollection()
    {
        //Implement it here
    }
    ButtonArray getButtons() { return buttons; }
}

class Window
{
private:
    ButtonCollection buttons;
    CString data;
public:
    void insertButton();
    void getCollection();
    void removeButton();
    void performAction();
}
```