Programming Assignment 1:

Stack and Linked List Practice in C++

**Description:**

One popular feature of many applications is allowing multiple undo operations with the ability to redo those as well. This feature requires the use of stacks, so we’re going to practice with stacks by writing a program that features theoretically unlimited undo and redo.

Because we want to keep our focus on the undo and redo, we’re going to apply it in a simple text calculator application. You will first write the program using an array-based Stack class that I have provided to you. Then you will modify the Stack class to use a linked list representation, without changing the program.

**Provided Files:**

You will find the array based Stack class, an OperationData.h file, a tester for the Stack class named test\_Stack.cpp, and sample input and output files on the School’s Linux machines in /home/ad.ilstu.edu/mbhatt1/Public/it279/Program1

Create a directory in your space on the Linux machines and copy the files to it.

**The Program:**

Start by writing the program using the provided Stack class. You will have an opportunity to submit the program code to check functionality only for a homework grade several days before the assignment is due.

Your are writing a program that provides unlimited undo and redo capabilities for a simple text calculator. The calculator will be integer-only. The user will enter a command possibly followed by a positive integer. The calculator will perform the appropriate operation and report the current value of the calculation.

The possible commands will be +,-,\*,/,%,C,U, R and Q. The arithmetic operations will be followed by an integer (with or without a space between command and number) and represent the appropriate integer calculation. C means clear. It will set the current value to 0. U is undo. R is redo. Q is quit. Use “>” to prompt for the next command. Redo will print the operation that is being redone as well as the result. If there’s nothing to undo or redo, report that and then the unchanged current value.

At the beginning, provide a brief set of instructions to the user (be sure to match the sample data precisely).

Note that you will need two different stacks to accomplish the task. The stacks will hold OperationData structs, which have three fields: a character representing the operator, an int for the operand, and an int for the previous accumulator value (the value the calculator held before this operation was completed). Look over the OperatorData.h file before beginning work on the program.

To compile your program, you will compile the provided Stack.cpp file along with the .cpp files that you write. For example, if you write files Calculator.h, Calculator.cpp, and CalculatorApp.cpp, you would compile by typing:  
  
g++ Stack.cpp Calculator.cpp CalculatorApp.cpp

Make sure that you do NOT #include .cpp files in other .cpp files. That **will** result in a compile-time error when your program is tested.

Once you have the program working, you can test it interactively or by using input and output redirection. Before submitting, make sure that your output matches the sample output exactly. Don’t forget to use the diff tool in Linux (and possibly the VSCode file comparison).

To run the stack program with sample input and output (assuming you’ve copied the necessary files over and compiled) run the following commands:

./a.out < stackInputFile.txt > myStackOutputFile.txt

diff stackOutputFile.txt myStackOutputFile.txt

If your program works as expected, the diff should not output anything.

For more info on how to run and diff a program given sample input and output, see the Linux Information Handout.

See below for an example of running the program interactively and what it should look like.

**The Stack Class**

Once your program is working, you will modify the stack class to use a linked list representation. You will convert it from using a static array (and being of fixed maximum size) to using a singly linked list and handling any amount of data.

You must not modify the **public** interface of class (changing private data or methods in the .h file is not changing the public interface).

You must make the class a correct dynamic class with no memory leaks and a correct copy constructor and copy assignment operator. Note that adding these methods to the class does not make a change in the public interface. They already exist in the array version of the class; we’re just using the default versions. Make sure to use private helper methods to reduce code duplication.

Use the provided stack tester program to test your stack class. Also use valgrind to help ensure that your stack class does not contain memory leaks.

Make sure that your program works correctly with either the provided Stack class or your linked list program, testing using diff and also making sure valgrind reports no errors.

**General program requirements:**

Your stack class must be a good ADT implementation. Do not allow the specifics of the application to impact the design and implementation of the class. Make sure that it uses good dynamic memory management. You may wish to review the Dynamic Classes video.

Programs will be graded in accordance with the IT 279 Program Grading Criteria provided to you.

Your program will be tested on the School’s Linux machines. Make sure you have tested it well there. The tester you were provided will also be run against your Stack class, and all provided output scenarios should exactly match the input scenarios you were given as well. Failure to match the provided test data in any way will impact your grade. This counts as getting a C in the “Functionality” category of the Program Grading Criteria. Note that valgrind errors on your program or on the stack tester will also count as errors on provided sample data, resulting in no better than a C on the program. Test well.

Use diff. Differences in formatting are differences in functionality.

Make sure to test with additional input (I will).

**Submission requirements:**

Name your files with clearly meaningful names (no Main.cpp or StackMain.cpp, please). Zip your source code files into a single zip file (do not zip folders, just the files). Only include the files you modified or wrote (not OperationData.h). Attach that zip to the program 1 assignment on Canvas and submit.

Sample interactive program run (user input in bold):

$ **./a.out**

Welcome to the Undo-Redo Calculator!

This calculator handles the following commands:

+, -, \*, /, %, C, U, R, and Q

Arithmetic operators are followed by a single positive integer.

C will clear the accumulator, U will undo, R will redo, and Q will quit.

0

> **+ 300**

300

> **- 500**

-200

> **/ 100**

-2

> **R**

Nothing to redo

-2

> **U**

Undoing: / 100

-200

> **U**

Undoing: - 500

300

> **R**

Redoing: - 500

-200

> **C**

0

> **U**

Undoing: C

-200

> **R**

Redoing: C

0

> **U**

Undoing: C

-200

> **+ 3**

-197

> **R**

Nothing to redo

-197

> **Q**

Goodbye