**Assignment #2**

***Stack Implementation***

***and***

***Postfix Expression Evaluation***

**Due Date: October 19th 11:59pm**

**Purpose:** In this assignment, you will explore data structures in C++ by implementing two different types of stacks: LinkedStack and DynamicArrayStack. These stack implementations will use a common interface defined in the Stack.h file. After creating these data structures, you will test your implementations by solving a computer science problem using stacks.

**Part I: Implement LinkedStack.h**

In this part of the assignment, you will:

1. Create a LinkedStack class that extends Stack class given in Stack.h.
2. Override necessary methods given in the Stack.h
3. Ensure the LinkedStack efficiently manages elements using a singly-linked list.

**Part II: Implement DynamicArray.h**

In this part, you will:

1. Create a class named DynamicArrayStack that extends Stack class given in Stack.h.
2. Override necessary methods given in the Stack.h
3. Ensure the DynamicArrayStack efficiently manages elements using a dynamic array
4. DynamicArrayStack has resize() method. When the stack becomes full, the resize() method doubles the stack's capacity to allow for more elements.
5. You are **not allowed** to use std::vector

**Part III: Solve a Computer Science Problem**

Now that you have created both LinkedStack and DynamicArrayStack, it's time to put these data structures to work. You are provided with the following computer science problem and you are expected to solve the problem using the data structures you’ve developed.

*Problem Description:* You are going to write C++ program to evaluate postfix expressions. Postfix expressions, also known as Reverse Polish Notation (RPN), are a mathematical notation in which operators follow their operands. In postfix notation, every operator is placed after its operands, which eliminates the need for parentheses to specify the order of operations. This notation is particularly useful for computer programs and calculators, as it allows for straightforward and unambiguous evaluation of mathematical expressions. Postfix expressions are often processed using stacks, where numbers and operators are pushed and popped in a way that ensures correct evaluation.

Example:

* Postfix Expression: **3 5 +**
* Evaluation:
  + Push 3 onto the stack.
  + Push 5 onto the stack.
  + Encountering the **+** operator, pop 3 and 5 from the stack, add them, and push the result (8) back onto the stack.
  + Result: The stack contains the resulting value 8.

**TODO**

You are provided with a starter code that includes the following key components:

1. **test.cpp:** This is the main testing program that you need to test your implementation. It reads postfix expressions and their expected results from an input file, evaluates the expressions using your implementations, and assigns points based on the correctness of your solution. You should not modify this file.
2. **postfix\_input.txt:** An input file where you will find a set of postfix expressions and their expected results. You should not modify this file; your code should correctly evaluate the expressions according to this input.
3. **Stack.h:** This file defines an abstract interface for the stack data structure. You need to inherit from this class in two concrete stack classes: **DynamicArrayStack** and **LinkedStack**.

**Assignment Workflow:**

1. Implement the **DynamicArrayStack** and **LinkedStack** classes according to the **Stack** interface in **Stack.h**. These stacks should provide the basic stack operations such as push, pop, peek, and checks for empty and full conditions.
2. Implement the **PostfixExpressionCalculator** class in the provided code. Ensure that it correctly evaluates postfix expressions using the stack implementations.
3. Run the provided **test.cpp** program to test your implementations with the postfix expressions and expected results provided in the **postfix\_input.txt** file.

**Exceptional Cases**

Your program should provide the following error outputs for the following exceptional cases:

1. Test Case: Division by Zero
   * Input Expression: “50/”, 0
   * Output: " Invalid postfix expression "
2. Test Case: Invalid Postfix Expression
   * Input Expression: "5+2\*", 0
   * Output: " Invalid postfix expression "
3. Test Case: Stack Underflow
   * Input Expression: "52+-", 0
   * Output: " Invalid postfix expression "

**Sample Output**

*Assuming the input file, postfix\_input.txt, contains the following:*

**34+,7**

**52/,2.5**

*Sample Run:*

34+

Testing LinkedStack:

Result: 7

Testing DynamicArrayStack:

Result: 7

52/

Testing LinkedStack:

Result: 2.5

Testing DynamicArrayStack:

Result: 2.5

Total Points Earned: 40 points

**Important Notes**

1. You are **not allowed to modify test.cpp** implementation. Please use it as it is.
2. Before you submit your work, please make sure the entire folder works. Here is a sample command that you can perform a sanity check:

*Windows Users:*

yourfolder> g++ -o myProgram.exe -std=c++14 \*.cpp  
  
yourfolder>myProgram.exe

*Mac Users:*

yourfolder$ g++ -o myProgram -std=c++14 \*.cpp  
  
yourfolder$ ./myProgram

**Submission**

You are asked to submit your work as a single zip file via CANVAS. Zip file will include all source codes including the following files:

* PostfixExpressionCalculator.h
* DynamicArrayStack.h
* LinkedStack.h

Please use the following file format while naming the zip file:

LastNameFirstnameX\_Y.zip where LastNameFirstname is your last name with the first letter in capital, followed by your first name with the first letter in capital; the X is the course code; the Y is the assignment #. (ex: SerceFatmaCS300\_2.zip)

**Rubric**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | 20 | 15 | 10 | 5 |
| PostfixExpressionCalculator | Passes all test cases without errors | Minor issues, errors | Significant issues, errors | Does not compile |
| LinkedStack | Passes all test cases without errors  No memory leak | Minor issues, errors | Significant issues, errors | Does not compile |
| DynamicArrayStack | Passes all test cases without errors  No memory leak | Minor issues, errors | Significant issues, errors | Does not compile |
| Error Handling and Exception Handling | Handles all the exceptions | Not cover all possible cases | Missing some essential exceptional cases | Not handle the exceptions correctly |
| Code Quality and Documentation | Well-organized, good coding practices, and includes comments and documentation | Lack consistent coding practices and through documentation | Disorganized and lacks documentation | Code is unstructured, unreadable, not well documented |