CS 201, Spring 2021
Homework Assignment 4

Due: 23:55, May 16, 2021

In this homework assignment, you are supposed to implement a strange calculator class that, in addition to performing algebraic operations, converts between infix and prefix algebraic expressions.

As mentioned, you must implement the program using stacks. You must define your own pointer-based ADT stack implementation for the program, meaning that you cannot use available stack libraries or implementations. However, you can use the material and the codes available in the course textbook and the slides. Please remember that when the ADT stack is used to solve a problem, all standard stack functionality must be preserved. That is, you can only use standard stack operations which are defined in the course slides. For this you will define a StrangeCalculator class and implement your global functions in that class.

Your implementation must include the following global functions:

```
string infix2prefix( const string exp );
```

This function takes as an input an infix expression and converts it to a prefix expression.

```
double evaluatePrefix( const string exp );
```

This function takes as an input a prefix expression and calculates its result.

Note, for the implementation of all the above-mentioned functions you MUST use stacks, in addition, you cannot use a recursion-based solution. For this first set of functions, namely, *infix2prefix,* and *evaluatePrefix,* you can make the following assumptions:

1. You may assume that the inputs to the functions are always valid. For example, the input to the *infix2prefix* is always a valid infix format string.
2. Only ***+,-,*,/*** (addition, subtraction, multiplication, division) operands are used for all the expression types.
3. You can assume that the operands are single-digit numbers. In other words, only numbers from 1 to 9 are used.
4. You can assume that the parentheses are always balanced in infix expressions. Ex: “(1+5)*3” but no inputs like “((1+5)*3”.
5. And you may assume that no spaces will be included in between the characters. That is, no inputs in the form: “5 +3 -4” instead, the same expression will have the form “5+3-4”.

Please keep in mind, however, that the assumptions above do not necessarily apply to the next set of functions.

The next set of global functions to include in your implementation are:

bool isBalancedInfix(const string exp);

This checks if the parentheses in the input infix expression are balanced. For example, for the input '3*((5+2)-1' the function should return false. This function also MUST use stacks for implementation.

void evaluateInputPrefixExpression( );

This function (i) asks the user for an infix input, (ii) removes the possible spaces in the input, (iii) checks if the expression has a balanced number of parentheses using *isBalancedInfix* function (if not give a warning), (iv) if the input is of correct format, converts the input infix expression to prefix using *infix2prefix* (v) and computes the final result using *evaluatePrefix.* Note that you don't have to check the user input for any other type of errors. That is except for the unbalanced parentheses and spaces you may assume that the input is valid. Also, you don't have to watch for invalid computation input. (ex: 0/0).

For these two functions you can make the following assumptions:

1. Only ***+,-,*,/*** (addition, subtraction, multiplication, division) operands are used for all the expression types.
2. You can assume that the operands are single-digit numbers. In other words, only numbers from 0 to 9 are used.

Below is an example of a driver file that we will use to test your program. During grading we will use, similar but a different driver. In addition to testing your code with the provided driver, we encourage you to test it with additional driver files of your own. However, please do not submit your own test files. During your tests and implementation do not alter the top half of the provided driver file. That is, before the main clause, you may only have following statements:

```
#include <iostream>
using namespace std;
#include "StrangeCalculator.h"
```

This is important, because, if your implementation relies on additional "include" statements in the main file, it might not work with our test file. So please keep that in mind.

The driver(main) file:

```
#include <iostream>
using namespace std;
#include "StrangeCalculator.h"

int main(){

    cout << "prefix of: 5+6 is: " << infix2prefix("5+6") << endl;
```

```
    cout << "prefix of: 5+6*7 is: " << infix2prefix("5+6*7") << endl;

    cout << "prefix of: 5+6*(7/2) is: " << infix2prefix("5+6*(7/2)") << endl;

    cout << "prefix of: (5+6)*(7/2) is: " << infix2prefix("(5+6)*(7/2)") << endl;

    cout << "prefix of: (5+6)*(7/2)-5+7*(8*2) is: " << infix2prefix("(5+6)*(7/2)-5+7*(8*2)") << endl;

    cout<< "Checkpoint 1" << endl;

    cout << "result of: 5+6 is: " << evaluatePrefix( infix2prefix("5+6")) << endl;

    cout << "result of: 5+6*7 is: " << sc.evaluatePrefix( infix2prefix("5+6*7")) << endl;

    cout << "result of: 5+6*(7/2) is: " << evaluatePrefix(infix2prefix("5+6*(7/2)")) << endl;

    cout << "result of: (5+6)*(7/2) is: " << evaluatePrefix( infix2prefix("(5+6)*(7/2)")) << endl;

    cout << "result of: (5+6)*(7/2)-5+7*(8*2) is: " << evaluatePrefix(infix2prefix("(5+6)*(7/2)-5+7*(8*2)")) << endl;

    cout << "Checkpoint 2" << endl;

    cout << isBalancedInfix( "((5+7" ) << endl;
    cout << isBalancedInfix( "((5+7)))" ) << endl;
    cout << isBalancedInfix( "(5+4" ) << endl;
    cout << isBalancedInfix( "(5+4)*3-4*(9+8))" ) << endl;
    cout << isBalancedInfix( "(5+4)*3-(4*(9+8))" ) << endl;

    return 0;
}
```

The example output for the following program is:

```
prefix of: 5+6 is: +56
prefix of: 5+6*7 is: +5*67
prefix of: 5+6*(7/2) is: +5*6/72
prefix of: (5+6)*(7/2) is: *+56/72
prefix of: (5+6)*(7/2)-5+7*(8*2) is: +-*+56/725*7*82
Checkpoint 1
result of: 5+6 is: 11
result of: 5+6*7 is: 47
result of: 5+6*(7/2) is: 26
result of: (5+6)*(7/2) is: 38.5
result of: (5+6)*(7/2)-5+7*(8*2) is: 145.5
Checkpoint 2
0
0
0
0
1
```

NOTES ABOUT IMPLEMENTATION:

1. You ARE NOT ALLOWED to use any stack libraries, you must implement your own.
2. Your code must not have any memory leaks. You will lose points if you have memory leaks in your program even though the outputs of the operations are correct.
3. This time around you are allowed to have global variables, structures and functions. (But not in the main file)

NOTES ABOUT SUBMISSION:

This assignment is due by 23:55 on May 16 2021. **This homework will be graded by your TA Aydamir Mirzayev (aydamir.mirzayev@bilkent.edu.tr). Please direct all your homework-related questions to him.**

1. In this assignment, you must have the separate interface and implementation files (i.e., separate .h and .cpp files) for your class. The file names should be "StrangeCalculator.h" and "StrangeCalculator.cpp". You should also submit other .h and .cpp files if you implement additional classes. We will test your implementation by writing our own main function. Thus, you should not submit any function that contains the main function.

Although you are not going to submit it, we recommend you to write your own driver file to test each of your functions. However, you SHOULD NOT submit this test code (we will use our own test code).

2. The code (main function) given above is just an example. We will test your implementation also using different main functions, which may contain different function calls. Thus, do not test your implementation only by using this example code. Write your own main functions to make extra tests (however, do not submit these test codes).

3. You should put your "StrangeCalculator.h" and "StrangeCalculator.cpp" (and additional .h and .cpp files if you implement additional classes) into a folder and zip the folder (in this zip file, there should not be any file containing the main function). The name of this zip file should conform to the following name convention: Lastname_Firstname_StudentID.zip. For example, GündoğmuŞ_CerenEcem_21501578.zip . It is important that your surname comes first.

The submissions that do not obey these rules will not be graded.

4. **Then, before 23:55 on May 16, you need to upload this zipped file containing only your header and source codes (but not any file containing the main function) to Moodle.**

No hardcopy submission is needed. The standard rules about late homework submissions apply. Please see the course syllabus for further discussion of the late homework policy as well as academic integrity.

5. You are free to write your programs in any environment (you may use Linux, Mac OS X or Windows). On the other hand, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. If we could not get your program properly work on the dijkstra machine, you would lose a considerable amount of points. Therefore,

we recommend you to make sure that your program compiles and properly works on “dijkstra.ug.bcc.bilkent.edu.tr” before submitting your assignment.