Fall 2022

CPSC2620 Assignment 5

Due: December 6 (Tuesday), 2022 11:55pm

(15 marks)

Please note that due to the potential COVID situation, the requirements of this assignment might be changed and also the compilation and submission procedure might be changed. You will be notified should a situation arise.

(*) Please note that no collaboration is allowed for this assignment. Should any plagiarism be identified, all the involved students will get zero for the assignment and will be reported to the Chair of the department for further action. The marker has been asked to check your submissions carefully.
(*) Your submission will be marked on the correctness and readability (including comments) of your program.
(*) Please follow the guidelines to name your files (see below).
(*) Late submission policy: 1 day late: 5% off, 2 days late: 10% off, more than 2 days late: you will get 0 for the assignment.

NEW NEWNEW

Many Thanks to Nicole, who created and tested the Makefile and submission script for our assignments.

Please be advised that you have a new procedure to compile and submit your source code for your assignments. The details are provided in the README.txt file, which you have used for Assignments 3 and 4 already.

For this Assignment, replace N with 5 in the README.txt file.

Please use the provided Makefile to compile your program. Please see the README.txt. You don't submit your own Makefile.

Please follow the submission procedure in README.txt to submit your source code.

Please follow the requirement on the names of the source code files. They should have suffix .cc and .h. Other files are not accepted.

Notes:

(*) There is only one question in this assignment.

(*) The file to be created for this assignment is: *useStack.cc*. There is no header file. The executable for this question is called *useStack*. Please see below for more details.

Question

In many computer science applications, such as evaluating an arithmetic expressions, like (2 + 3) * (5 * 6 * (3 + 4)), parentheses must appear in a balanced fashion. *Balanced parentheses* mean that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. For example, each of the following parentheses sequences is balanced.

```
( ( ) ( ) ( ) ( ) )
( ( ( ( ) ) ) )
( ( ) ( ( ( ) ) ( ) ) )
```

On the other hand, each of the following parentheses sequences is not balanced.

```
( ( ( ( ( ( ( ) )
( ) ) )
( ( ) ( ) ( (
```

Create a function called *bool isBalanced*(*const string& str*) to check whether the variable *str* contains balanced parentheses. Assume that *str*contains some spaces (ignored), '(', and ')', but not any other characters. For example, "( )()" is balanced. You need to use container *stack* to do this work.

Arithmetic expressions can be converted into something called "Reverse Polish Notation" (RPN). A RPN expression is a sequence of operands (integers in this case) and operators. A good example is: 2 1 2 + 3 4 + * - . The expression is read from left to right. When an operand is seen, the operand is pushed onto a *stack*. If a binary operator is seen, the top 2 operands are removed from the stack, the operator is applied to them (second top-most element is the first operand), and the result of the operation is then pushed onto the stack.

The stack is empty at the beginning of the evaluation. At the end, there should be exactly one element on the stack which is the result of the expression evaluation. In the example above, the stack contents (left most is the bottom of the stack) before and after the application of each operator are:

2, 1, 2 (see +) → 2, 3

2, 3, 3, 4 (see +) → 2, 3, 7

2, 3, 7 (see *) → 2, 21

2, 21 (see -) → −19

Therefore, the final value of the expression is −19 and is the only element in the stack.

A valid RPN expression is the one that will never attempt to apply a binary operator when there are fewer than two elements on the stack. In addition, a valid expression will leave exactly one element on the stack at the end. Each expression will be given on a single line. There are only three different types of operators: +, -, and *.

Create a function called *bool evalRPN*(*conststring&str*, *int& res*). It, upon finishing evaluating a valid RPN expression in *str*, will put the evaluation result into variable *res* and return *true*, while it, upon finding an invalid RPN expression in *str*, will just return *false*.

In your main program, you read inputs one line after another from the following file.

/home/lib2620/Data/A5test.txt

The file contains 10 lines. The first 5 lines are used to test the function *isBalanced* and the second 5 lines are used to test *evalPRN*.

For the first 5 lines, you read one line and print it out. Then you call *isBalanced* on it. If the parentheses are balanced, you print out "Balanced parentheses". Otherwise, you print out "Imbalanced parenthesis".

For the second 5 lines, you read one line and print it out. Then you call *evalRPN* 5 on it to check whether it is a valid RPN expression. If yes, you print out something like "Valid RPN expression. Result is -19.". Otherwise you print out "Invalid RPN expression".

Your program file is called *useStack.cc* and your executable file is called *useStack.cc*. You execute your program like this:

%useStack < /home/lib2620/Data/A5test.txt

Note that the marker might create their own test file, which should have the same format as the one in A5test.txt.