CS 302
Assignment 3
Postfix Evaluation

In this assignment you need to write a program that takes in a file of postfix expressions and outputs the result of each expression.

Postfix Notation:

Most of us have learned to evaluate arithmetic expressions in the following format:

2 + 2, x % y, etc.

This is known as *infix* format (for reasons we will see later on in the course.) Another way to write arithmetic expressions is in *postfix* format. In this format, the two operands are listed first, and then the operator is listed last:

22+, xy%, etc.

In the above, 22+ is equivalent to 2 + 2, and xy% is equivalent to x % y (x mod y). Postfix expressions, like their infix cousins, can also be nested, so you can have something like

2 2 + x y % -

which translates to

(2 + 2) - (x % y)

As you can see the final substraction operation requries usage of parenthesis in infix notation, but not in postfix notations. This is the most important advantage of postfix notation. There is no concept of order of operations. All operations are clearly defined by the order of the postfix notation. So the idea of parenthesis to define order of operations does not exist in postfix!

It turns out that many compilers convert arithmetic expressions to postfix format and then evaluate them using a stack. The general rules for evaluating a postfix expression using a stack are:

• If the next item in the expression is an operand, push it onto the stack
• If the next item in the expression is an operator, pop the top two items off of the stack and perform the operation on those values; then push the result onto the stack.
• When the end of the expression is reached, the stack should contain only one item: the result.

These rules imply that there are valid postfix expressions and invalid postfix expressions. For example,

3 3 3 +   or 3 3 + +

would be invalid.

The Program:

Your mission, should you chose to accept it (and you will get a zero if you don't), is to write a program that takes in a file of postfix expressions (using linux redirection) and then evaluate it using the mentioned algorithm. Some of the expressions will be valid, and some of them might be invalid. If an expression is valid, then you must print out the result of the expression. If the expression is invalid, you must print out an error message.

Each item in an expression will be separated by a single space. You may assume that the expressions will contain only integers and the arithmetic operators for addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). So no invalid characters.

Notes:
-Comment your source code appropriately.
-Make sure you name your program file in accordance with the syllabus stipulations
-Test your program by running it a few times with different input to make sure the output is correct.
-Don't forget your write-up!
-You must create your own stack class. Do not use the Standard Template Library(STL) or you will get a 0!

Test and execute your program by using data-in by Linux input redirection. If your executable code is Ast# and your data file is named data# execute as: `./a.out < data3.txt`

Sample Input File:

```
$ more data3.txt
4 3 * 2 2 + /
4 5 2 * + 10 -
2 + 8 2 /
```

Sample Output:

```
$ g++ cs302hw3.cpp
$ ./a.out < data3.txt
4 3 * 2 2 + / => 3
4 5 2 * + 10 - => 4
2 + 8 2 / => invalid expression
$
```

Part II, Your Post-Mission Report.:

When completed, create and submit a write-up (PDF Format) of no more than three pages (1 page is fine, half a page is not):
-Name, Assignment, Section
-Summary of the implemented stack data structure. Analyze and provide the Big-Oh Notation for each of the Stack operations as well as the overall run time for the expression evaluation algorithm.
-A discussion on the efficiency of the algorithm. Is there a way another way to write the algorithm to make the time complexity better? Are there any computations that are done more than once and that could be avoided?
-What modifications need to be made to your code in order to implement a use the square root operator? Suppose you are using the @ symbol for square root. Example: 45+@ would be √ (4+5) which equals 9. You don't need to implement the changes but explain them and show step-by-step execution of the following:

4 6 + 30 5 - @ *

As always, should any of your pointer, or linked list nodes be left dangling or leaking memory, the Secretary will disavow any knowledge of your actions. This paper will self-destruct in five seconds. Good luck Mr. Hunt.