Course: CpSc 212: Algorithms and Data Structures Spring 2014
Lab 02: Infix to Postfix conversion using a stack

1 The StringStack class

You are to build the StringStack class which represents a stack of strings. The stack is implemented using an array. We have provided a lot of helper code and functions. So the first step is to download and extract these files into your working directory.

Please first download the *lab02.tar.gz* file from the following webpage:

```
http://people.cs.clemson.edu/~rmohan/course/cpsc2120-sp14/
```

Next, please extract the files using the following command.

```
tar -zxvf lab02.tar.gz
```

This command will extract all the files in the directory named *lab02*. Please change your directory using the command:

```
cd lab02
```

Now please open *stringStack.h* to see the specifications of the class. This class contains three private data members: *stack* - an array of strings which holds the contents of the stack, *capacity* that gives the capacity of the stack and *top*, an index that points to the top of the stack. Technically, *stack[top-1]* is the string on top of the stack.

Note the public interface to the class. Open *stringStack.cpp* and implement the following functions.

1. ***push(x):*** This function takes in a string x and pushes it into the stack. The function should print an error message if we try to push into a full capacity stack.
2. ***pop():*** This function pops and returns the top of stack. The function should print an error message if we pop from an empty stack.
3. ***topOfStack():*** This function returns the top of stack. A useful message should be printed if the stack is empty.
4. ***isEmpty():*** This function returns *true* if the stack is empty. Otherwise returns *false*.

The constructors and destructors are already provided. Please go through them and make sure you understand.

1.1 Testing your StringStack class

You can use *testStringStack.cpp* or create your own test program to test your stringStack. The file *testStringInput* contains a list of strings that can be used to push into the stack. Please compile your test program as:

```
g++ testStringStack.cpp -o testStringStack stringStack.cpp
```

You can then execute using the command:

```
./testStringStack < testStringInput
```

Also, uncomment certain lines to check if your stack operates correctly on full and nil capacities. Make sure to recompile your program every time you make changes to *testStringStack.cpp*.

2 Infix to Postfix conversion

An infix expression is an expression of the form $operand_1$ *operator* $operand_2$. A postfix expression is of the form $operand_1$ $operand_2$ *operator*. An example of an infix expression is:

*2 + ((5 * 3) / 1) * 2 + 1*

and its corresponding postfix expression is:

*2 5 3 * 1 / 2 * + 1 +*

Please open the file *main.cpp* and complete the function *string infixToPostfix(string infix)* which takes in an infix expression as a string and returns the corresponding postfix expression.

2.1 The algorithm

The algorithm to convert an infix expression to postfix is as follows. The input is a string in infix expression.

1. Scan each token in the infix expression from left to right.
2. if the token is an operand, write it to the postfix expression.
3. else if the token is an operator:
   (a) if the stack is empty, push the operator into stack.
   (b) else if the operator is a (, then push it into the stack
   (c) else if the operator is a), then pop all operators from the stack and write to postfix expression until a (. Pop (from the stack.
   (d) else:
      i. while the top of stack has higher or equal precedence than the input token.
         A. pop the top of stack and write it to the postfix expression.

ii. push token into stack.

4. Pop all the remaining items in stack and write to postfix expression.

Note that opertors (and) have higher precedence to *, / and %, which has higher precedence to + and -.

2.2 Input

The input to the program is an infix expression in one line. Each token is separated by a space.

2.3 Output

The output is the postfix expression where each token in the expression is seperated by a space.

Sample I/O

Test case#	Input	Output
1.	(2 + 3) * 5	2 3 + 5 *
2.	2 + 3 * 5	2 3 5 * +
3.	1 - (2 + 3) * 4	1 2 3 + 4 * -
4.	2 + ((5 * 3) / 1) * 2 + 1	2 5 3 * 1 / 2 * + 1 +

2.4 Tokenizer

The first task of the program is to retrieve tokens from the input expression. We have provided the function *getToken(.)*. This function takes as argument a string and an integer *index*, and returns the token formed from *string[index]*. This function returns the next token in the string. This function is called until the tokens are exhausted from the string.

2.5 Other helper functions

We have provided several other functions to aid you in your task for converting an infix expression to postfix. The function *isOperand(token)* takes in a token and returns *true* if it is an operand and the function *isOperator(token)* returns *true* if the token is an operator. The function *precedence(x,y)* takes as input two operators and returns the precedance relation. In particular, a value > 0, < 0 and 0 is returned if x has higher, lower and equal precedence respectively to y. Please go through and understand code that is already provided for you.

2.6 The std::string class

We use the c++ string class for storing token values and expressions. Note that we can use the overloaded operators to assign and concatenate strings.

```
#include <string>
using std::string;
```

```
string a = "infix";      // assignment of string a to the value "infix"
string b = "postfix";
string c = a + b;        // concatenates two strings together, holds the
                         // value "infixpostfix"
```

2.7 Extra Credit: Detecting syntax errors

For bonus credit (3 points), include checks in your program to handle gracefully syntax errors in the arithmetic infix expression. In particular, report errors for missing left/ right paranthesis and unexpected token. Examples of syntax errors are given below.

2 + 3 * 5)	missing left paranthesis
2 + (3 * 5	missing right paranthesis
2 + 3 *	invalid expression

2.8 Compiling and Running

Please compile your code using the following command.

```
g++ main.cpp -o main stringStack.cpp
```

Execute your code using:

```
./main
```

You can also enter input in a file named *in* and redirect your input using the following command:

```
./main < in
```

2.9 Testing

You are required to test for all cases that meet the specifications above. Please thoroughly test your program with all boundary cases. You will receive full credit only if the program works for all cases.

3 Submission

Please submit your solutions through handin.cs.clemson.edu. The due date for this assignment is Friday January 24th at 11:59pm.

4 Grading

Each section in this assignment is graded out of ten points. No points will be awarded to a program that does not compile. You will recieve 1 point for successful compilation and 1 point for successful execution (no seg faults and infinite loops). 3 points is awarded for correctness and efficiency each and 2 points awarded for good code: readability, comments, simplicity and elegance, etc.