CSC 2111 **Spring 2013**

Name(s): _____

Lab 6: Stacks

Reference: *C++ Plus Data Structures*
Chapter 5.1 - "Stacks"

Objective: To work with the concepts of an abstract representation of a stack.

Overview:

In this lab you will work with an array implementation of the stack ADT. Using stacks, you will implement a calculator for handling postfix notation.

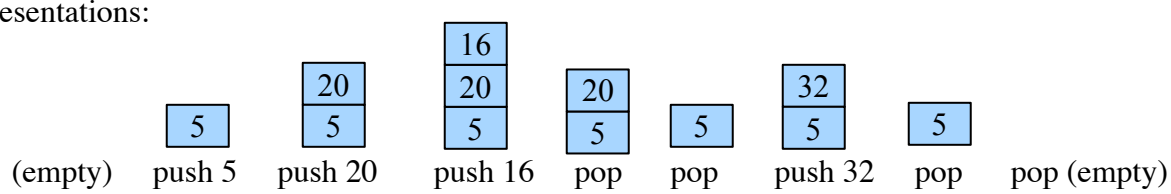
Files: *Stack.h, Stack.cpp, lab6.cpp*

Introduction

The objective of this lab is to introduce the stack data type and practice its use by evaluating expressions in postfix notation.

A stack is a basic data structure similar in use to a physical stack of papers. You can add to the top (push) and take from the top (pop), but you are not allowed to touch the middle or bottom. A stack adheres to the Last-In-First-Out, or LIFO, property. Often it internally uses a linked list, array, vector, or a doubly-linked list to contain the elements.

For example, consider the following sequence of operations, and the corresponding stack representations:



Given the following sequence of stack operations in the following table, fill in the corresponding <u>top-of-stack value</u> after each operation:

Operation	Top-of-stack
push 40	
push 55	
pop	
push 10	

push 15	
push 30	
push 35	
pop	
push 20	
pop	
pop	
pop	
pop	

In general, a stack needs to implement the following interface and functionality:

```
void push(StackElement e)
```

This adds the element to the top of the stack. (If there is some maximum capacity for your implementation, then somehow you will need to indicate an error.)

```
StackElement top() const
```

This returns the element on the top of the stack. It does not remove that element from the top. (If the stack is empty, then somehow an error must be indicated.)

```
void pop()
```

This removes the element on the top of the stack, but does not return it. (If the stack is empty, then somehow an error must be indicated.)

```
bool empty() const
```

This tells whether there are any elements left in the stack (false) or not (true).

In this lab, you will:

- Implement a class stack that implements a stack of integer numbers.
- Write a program that uses this class to implement a postfix calculator that accepts a valid postfix expression of an arithmetic calculation and evaluates that expression.

For this lab, you will use a stack of integer values. You will need to accept negative numbers, like -5 (for example), and numbers with multiple digits, like "34" (an integer 34, for example). Assume that the input, i.e. the postfix expression, is entered in on one line and that all numbers and operators are separated by white-space (spaces or tabs). You can assume that users will enter the proper number of operands/operators. Your calculator must implement +, -, *, and /.

Postfix

Postfix notation (also known as reverse Polish notation) involves writing the operators after the operands. Notice how parentheses are unnecessary in postfix notation:

(3 + 6) - (8 / 4)	==>	3 6 + 8 4 / -
Infix		Postfix

Here is another example:

(3 + 2) * (1 + (2 * 5)) ==> 3 2 + 1 2 5 * + *

Postfix expressions, also known as Reverse Polish expressions, can be evaluated very efficiently using a stack. The basic algorithm traverses the expression, looking at each token (integer or operator), and performs the following:

```
1. If the token is an operator (it is a +, -, *, or /)
2.      two items are popped off the stack
3.      the operator is applied to them
4.      the result is pushed back onto the stack
5. If the token is an integer (not a +, -, *, or /)
6.      it is pushed onto a stack
```

After the entire expression has been traversed, the value of the expression will be sitting on the top of the stack.

Apply the above algorithm to the following postfix expression and give the answer below:

10 6 + 6 3 - * is ☐

In order to evaluate an expression, you will first have to parse the expression into *tokens*. Because you can take for granted that each token is separated on each side by white space (as per the instructions above), the following C++ code will work:

```
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
   string line;
   getline(cin, line);
   // Create a string stream that will read from "line"
   stringstream ss(line,stringstream::in);

   // While there are more tokens...
   while (!ss.eof())
   {
      string token;
      ss >> token;

       //display token for testing

       // Put your code for evaluating the expression here!

   }; //end while

   // The should be one and only one item left on the stack
   // print the item at the top of the stack

   return 0;
}
```

The above code is given to you, and can be found in the file *lab6.cpp*.

Now, you will need to implement the following line in the above code, placing it after the comment `//display token for testing`:

```
cout << token << endl;
```

Does *lab6.cpp* compile without warnings or errors? Y/N

What does the program output when using the following input string:? 10 6 + 6 3 - *