A Simple Calculator using Stacks

Programming Assignment# 2

- **Objective:** Learn about the Stack ADT (Abstract Data Type). Implement Stack ADT as an extendable array. Learn about postfix notation. Use Stack to evaluate a postfix expression. Design an object oriented implementation in C++.
- **Points**: 50
- **Team-based**? You have the option to either work individually or work in pair. It is highly suggested that you work in pair.
- **Due**: Sunday, 2015-April-26 @ 11pm
- **What to hand in:** Refer to the section on "what to hand in" of this document.
- **Note:** Late assignment submitted within 24 hrs. from the deadline shall be penalized 10%. No late assignment shall be accepted after 24 hours from the deadline.

You may lose points if you don't pay special attention to the following in your program:

- *Your code should be properly indented. Separate various sections of your code with a blank line, like variable declaration, taking input, computation, printing, etc.*
- *Use appropriate data types, e.g., use integer data types where there will be no need to store floating point numbers.*
- *Avoid magic numbers, i.e., use constants instead of numeric/string literals and write a short comment that explains its purpose (it is okay to have string literals in cout for displaying messages or numeric/string literals when initializing variables).*

  E.g., `const double PI = 3.14; //value of pi`

  `area = PI * radius * radius;` *instead of* `area = 3.14 * radius * radius;`
- *Variables and constants should be named according to their purpose. E.g., if you need to store the number of cars then* `numCars` *is more descriptive then saying* `x`*.*
- *Pay attention to naming convention for variables and constants. Constants are as a practice declared in upper case; multiple words are joined by underscore. Variables are declared in lower case, multiple words are joined by underscore or first letter is capitalized for each word except for the first word.*

- ***Document your code by writing comments where appropriate to explain what the various pieces of your code might do. Refer "additional notes" section of this document for more details.***
- *Each line should not have too much code, as a matter of practice any line longer than 80 columns should be broken down into multiple lines of code.*

Introduction

Stacks have many uses. They can be used to design a calculator to process numbers and math operators. You are accustomed to seeing math expressions such as 3 + 4 in infix format where the operator (+) is in between the operands (3 and 4). This format is easy to compute for the human user: for example, if you see an expression 3 + 4 * 5, you know that first 4 should be multiplied by 5, and then 3 should be added to the result. It is harder for a computer that reads input from left to right to follow the same steps. It first reads 3, then reads "+" sign. When it encounters 4, it does not know whether any operation of higher precedence is coming, so it may attempt to add 3 and 4 right away. The solution is to use a special format called postfix notation where the operator comes after the operands so 3 + 4 becomes 3 4 +.

In this assignment, you will implement a Stack ADT as an extendable array to compute the answer to arithmetic expressions in the postfix notation.

Converting Arithmetic Expression from Infix to Postfix Notation

For the purposes of this assignment you do not need to write a program to convert infix expression to postfix. Your program will take the postfix expression as input, which you will accept from the user. However, learning to convert from infix to postfix will be useful to you. First thing that you need to understand is operator precedence and associativity. Precedence tells us which operator is more important and gets computed first. In 3 + 4 * 5, the * has higher precedence than + so 4*5 = 20 gets computed and then 3 + 20 = 23 gets computed. Similarly, 3 * 4 + 5 means 3 * 4 = 12 gets computed and then 12 + 5 = 17 gets computed. Associativity tells us which operator is more important when both operators have the same precedence. In 3 + 4 – 5, both operators have the same precedence so we compute the answer from left to right (left associativity). The expression 3 + 4 – 5 means 3 + 4 = 7 and

then 7 – 5 = 2. The precedence and associativity of the operators is preserved when we convert infix to postfix.

Arithmetic expressions in infix notation can be easily converted by hand to postfix notation.

Consider the infix expression: 5 * 7 + 9 / 3

- Step 1. Fully parenthesize the expression according to the order of precedence and associativity.

  ((5 * 7) + (9 / 3))
- Step 2. For each set of ( ), move operator to the end of the closing parenthesis.

  ((5 7 *) (9 3 /) +)
- Step 3. Remove all of the parentheses, which results in the below postfix version.

  5 7 * 9 3 / +

Getting started

Download the file Assignment2.cpp from Titanium, in this file you will find the code for your Stack class with some of the function prototypes that you will implement. For your help there is a slide set on "evaluation of arithmetic expressions" posted on Titanium. Note that there are a lot of details to this assignment; too many to simply start programming without any forethought. For your own benefit you would want to plan your solution in pseudo code before you start to write your C++ code.

Operations to implement

(Note: To improve readability of your C++ code, you should define your member functions outside the class using scope resolution operator.)

- You will be coding your own Stack class implemented as an extendable array. Do **NOT** use the STL Stack. Below are the functions to write, additionally you are free to write helper functions that you may need.
  - `OperandStack(int capacity); Constructor to your stack class.`
  - `~OperandStack(); Destructor to your stack class.`
  - `int size() const; returns the number of elements in the stack.`
  - `bool isFull() const; checks for the full stack.`

- `bool isEmpty() const;` checks for the empty stack.
- `double top() const;` returns the element at the top of the stack without removing it from the stack.
- `void push(double x);` push a new element x on the stack. If the stack is full then call the growStack function to increase the capacity of the stack and then push the new element.
- `void pop();` removes the top element from the stack. Note that this function does not return the removed element.
- `void growStack(int newCapacity);` increases the capacity of the stack. Note that this will require a deep copy of the current stack into a bigger one.

❖ **Solving a postfix expression.**

Algorithm to compute the solution to a postfix expression:

- Examine each number and operator in the input.
  - If it's a number, push it onto the stack
  - Otherwise (it must be an operator)
  - Pop the stack and store the number as the right operand
  - Pop the stack and store the number as the left operand
  - Compute the value of "left operand OPERATOR right operand"
  - Push the result onto the stack
- The final result will be the last value on the stack. Pop the stack and return the result. Check for the errors: If the stack has more than one element left and there are no more operators OR the stack has one element left and there is at least one operator then signal an error.

Example: Solving postfix expression 3 5 1 - * [infix equivalent: (3 * (5 - 1))].

Postfix (input) **3 5 1 - ***	**Stack afterwards**
3	3
5	3 5
1	3 5 1
–	3 4
*	12

Lookup the slide set "evaluation of arithmetic expressions" on Titanium for additional examples and possible error cases.

For the sake of simplicity, your program should perform only the following binary arithmetic operations: Addition +, subtraction -, multiplication *, and division /. Optionally you are free to enhance your program by adding more arithmetic operations like power, etc.

Your program only needs to work for an input postfix expression with operands that are single digit integers in the range of 0 to 9. However, optionally you are free to enhance your program such that it works for operands having multiple digits (lookup the narrative and the code on pages 1101-1103 of your C++ Early Objects 8th edition for ideas on working with multi digit numbers. The same code has been posted on Titanium under the "Code" section as postfixToInfix.cpp).

C++ function to write: `void solvePostfix(string postfixStr);` computes the value of an arithmetic expression given in postfix notation as an input string postfixStr. This function will use the aforementioned algorithm to solve the postfix expression and print out the final result. From your main function you may take the postifix expression as input from the user and pass it to the solvePostfix as argument.

Below are some sample postfix input expressions that you could use to test your program:

Infix	**Postfix (input)**	**Result**
((8 * (2 + 3)) / 4)	8 2 3 + * 4 /	10
((8 / 2) + (3 * 4))	8 2 / 3 4 * +	16
((8 + 5) / (9 - 7))	8 5 + 9 7 - /	6.5
	8 2 * 3 4 +	error
	8 2 * + 3 /	error

You should include in your assignment report:

- Pseudo code for your function: `void solvePostfix(string postfixStr);`
- Include screen shots for below input and output.

```
Enter a postfix expression: 8 2 3 + * 4 /
Result: 10
```

```
Enter a postfix expression: 8 2 / 3 4 * +
Result: 16

Enter a postfix expression: 8 5 + 9 7 - /
Result: 6.5

Enter a postfix expression: 8 2 * 3 4 +
Result: error – malformed expression

Enter a postfix expression: 8 2 * + 3 /
Result: error – malformed expression
```

What to hand in

A) Submit your C++ source code (cpp file). If you choose to work in pair, **both students must individually** upload the cpp file via Titanium. Write *yours and your partner's name, section number, and "Assignment 2"* at the top of your ***cpp*** file as comments.

B) Submit a written report electronically as a PDF or MS Word doc file through Titanium. Again, if you worked in pair then each student uploads the report individually. Write *yours and your partner's name, section number, and "Assignment 2" on the first page of your report*. **Your report should include:**

1. Give pseudo code for each of those functions as previously asked in this document. (Refer "additional notes" section of this document for tips on how to write good pseudo code).
2. Screen shots of your input and output for each of those functions as previously asked in this document. **Note that typed or hand written input/output will not be considered. You must provide screen shots of your actual program run.**
3. Do not include any C++ code in the report. It is to be uploaded separately as a .cpp file as asked in A).

Additional Notes

- **Pseudo code**

  Word of caution: It may be tempting to jump into C++ coding and later convert it to some sort of pseudo code. This is contrary to the purpose of writing pseudo code. If you find yourself in such a situation then you are missing out on the benefits that result from detailed planning.

  The idea behind pseudo code is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of your solution.

  Another advantage is that pseudo code is a useful tool for planning your program design. It allows you to sketch out the structure of your program and perform stepwise refinement before the actual coding takes place.

  Below is a sample pseudo code for the game of fizz buzz. It contains some programming language elements augmented with high level English description. The goal is to have enough useful details while leaving out elements not essential for human understanding of your program, like, variable declaration and system specific code.

```
fizzbuzz()
Input: none
Return: none
   for i <- 1 to 100 do
       print_number <- true
       if i is divisible by 3
           print "Fizz"
           print_number <- false
       if i is divisible by 5
           print "Buzz"
           print_number <- false
       if print_number == true
           print i
       print a newline
```

  Pseudo code style that you follow for your program does not have to be exactly as above. You may choose the style described in your textbook. In addition, there are numerous resources available on the internet that describe various popular styles, feel free to look up and choose one.

How detailed? Check for balance. If the pseudo code is hard for a person to read (too close to a particular programming language's syntax) or difficult to translate into working code, i.e., too vague, (or worse yet, both!), then something is wrong with the level of detail you have chosen to use.

- **Code Writing:** A bad approach is to write entire code all inside one function and later perform a massive refactoring, i.e., split your code into multiple cohesive functions. It leads to increased effort in integration, testing, and debugging. Good approach is to plan ahead and start by writing small cohesive functions (up to a maximum of 30 lines or what might easily fit on one screen) with reasonable refactoring of code as you go along.

- **Choice of loops:** Choose appropriate loop type (for, while or do while) in a way that it makes your program logic simpler and code more readable.

- **Testing:** You should test each function as you code it. Use of big bang approach to testing, i.e., postponing to test after you have written all functions would make it very difficult to isolate errors in your code.

- **Commenting your code**

1. **File comments**: Start your cpp file with a description of your program, names of authors, date authored, and version. Add other comments as asked in the "what to hand in" section of this document.

2. **Commenting function header:** Right above the header of each of your function you should have a comment. The comment will have three pieces to it:

   Describe what the function accomplishes.

   Inputs: Indicates what the required parameters are for the function. "nothing" for no parameters.

   Return: Clearly states what the function computes and returns. "nothing" for void functions.

   Here is an example:

```
/*******************************************************
 * greetings will print a welcome message, specifically *
 * addressing the name supplied                         *
 * Inputs:                                              *
 *      name, a string, is the name of the person being *
 *      addressed                                       *
 * Return:                                              *
 *      nothing                                         *
 *******************************************************/

void greetings(string name)
{
    string message = "Hello, " + name + ". How are you?";
    cout << message << endl;
}
```

3. **Section comments:** Add comments for each section of your code within your function like variable declaration, taking input, computation, display output, etc.

4. **Explanatory comments** should be added for tricky or complicated or important code blocks.

5. **Line comments:** Lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by 2 spaces. Break long comments into multiple lines.