Computer Science II – Data Structures and Abstraction (CS23001)

Midterm-1: Stack (100 pts.)

Midterm-1 Submissions:

Please submit one Zip file containing all Source Code organized in separate folders along with evidence of working code, online through Blackboard by the end of the day on next Friday. No email submissions will be accepted. Name the Zip folder *yourname_midterm1* (i.e. *ychae_midterm1*)

Please ensure that your source code is properly organized. Create separate folders for each exercise and ensure that all needed files (headers, source, makefile) are included in each folder, Use exercise numbers as folder names. Run each exercise to test your code and save a screen shot showing the working code. Zip all folders together with screen shots before submitting it online.

Your code must have proper indentation, appropriate comments, pre and post conditions, and test cases. Test cases that you have used to test your program should be included as a comment on top of your driver file. Be sure you choose the **Test Cases** according to the "Testing" rules covered both in class/slides and in your book. Also be sure that you always separate the class declaration from its implementation and you create a Makefile for its compilation. This requirement MUST be satisfied in **midterm-1**.

Midterm-1 Deliverables:

- Complete working source code of each exercise (where it applies).
- Screenshot(s) of each exercise to show that the code works.

Midterm-1 Grading:

The grade for each midterm-1 question is based on working source code and the screenshots. Submission of Source Code along with Screen shots are mandatory for each lab. Grade distribution includes: Source code (75%), use of proper indentation and commenting of the code (5%), pre- conditions and post-conditions (5%), test cases (5%) and the evidence of working code in the form of screen shots (10%).

Good Luck!

Part 1. Stack Implementation (20 pts.)

Write a stack class. The stack class must include at least “push”, “pop”, “top”, “isEmpty”, “isFull” methods. You can implement any additional methods, as they needed. Write your class in stack.h and stack.cpp. The part 1 does not needed to be tested.

Part 2. Infix to Postfix (80 pts.)

Expressions

Expressions are usually written in a form called *infix* notation, for binary operators the operator is written between the operands.

With *postfix* notation operands are written first, then the operator.

With *prefix* notation the operator is written first, then the operands.

Expression evaluation may be done with a two-step procedure:

1. Convert the infix expression to postfix form.
2. Evaluate the postfix expression.

A stack is used in each step:

1. Infix to postfix conversion uses a stack of operators.
2. Postfix expression evaluation uses a stack of operands.

Prefix, Postfix Examples

infix	**postfix**	**prefix**
2 + 3	2 3 +	+ 2 3
2 + 3 * 4	2 3 4 * +	+ * 3 4 2
(2 + 3) * 4	2 3 + 4 *	* + 2 3 4

Order of evaluation in infix expressions must be observed.

In the second example, an infix expression with only a + and a * the * is evaluated first.

In the postfix version of the second example * must be done first to provide the second operand for the +.

Evaluation of postfix expressions is straight forward, we will look at it first.

Postfix Expression Evaluation

To evaluate a postfix expression process the expression left to right and do the following:

```
while more input
       get next item
       if the item is an operand
              push it in the operand stack
       if the item is an operator
              get the right operand from the operand stack (top, pop)
              get the left operand from the operand stack (top, pop)
              perform operation
       push the result onto the operand stack
```

Infix to Postfix Conversion

To perform infix to postfix expression conversion process the expression left to right and obey the following rules:

Current Input	**Operation**
Operand	Append the item to the postfix output.

Operator or **Left Parenthesis**	`while the stack precedence >= input precedence`
	`  append the top of stack to the postfix output`
	`  pop the stack`
	`push the input item on to the stack`

Right Parentheses	`while top of stack is not a left parentheses`
	`  append the top of stack to the postfix output`
	`  pop the stack`
	`pop the left parentheses from the stack`

No more input	Pop each remaining operator from the stack and append it to the postfix output.

Note:

- **stack precedence** means the precedence of the item on top of the stack from the *stack precedence* column in the table below.
- **input precedence** means the precedence of the current input item from the *input precedence* column in the table below.

Infix to Postfix Conversion Precedence

Please review the **postfix.pdf**.

Create the *postfix.cpp* file the converts an infix expression into a postfix expression. You must use the *stack.h* and *stack.cpp* implemented in **Part 1**. The program must allow the user inputs, and the inputs should not include any alphabets. You need to handle '(', ')', '+', '-', '/', '*' and '%' operators. '^' will not be considered in this program. Also, please consider only *one digit numbers*.

The following links can be helpful for the conversions from string to integer or double types.

http://www.cplusplus.com/reference/cstdlib/atoi/

http://www.cplusplus.com/reference/cstdlib/atof/

or you can try

```
char myChar = '5';
float myFloat = myChar - '0';
```

Test your program carefully. Here are some data tests for your program and output expected.

Output

Enter an infix expression: 2 + 3 * 4
The postfix form is 2 3 4 * +

Enter an infix expression: (1 + 2) * 7
The postfix form is 1 2 + 7 *

Enter an infix expression: a * b + c
I cannot create a postfix form - Error

Enter an infix expression: (7 + 8 * 7
I cannot create a postfix form - Error

Enter an infix expression: (9 + 7) 4
I cannot create a postfix form - Error

Enter an infix expression: 2 * 4 * 8 /
I cannot create a postfix form - Error