CS 211 – Programming Practicum   Fall 2017

Programming Project 5

Due: Monday, 11/6/17 at 11:59 pm

Infix Expression Evaluator

For this lab, write a C++ program that will evaluate an infix expression. The algorithm
REQUIRED for this program will use two stacks, an operator stack and a value stack. Both
stacks MUST be implemented using a **dynamic array class (not struct) that you write**.

The commands used by this system are listed below and are to come from standard input. Your
program is to prompt the user for input and display error messages for unknown commands. The
code in proj5Base.cpp already does this for you.   You are expected to use that code as the base
for your project.

Command	Description
q	Quit the program.
?	List the commands used by this program and a brief description of how to use each one.
<infixExpression>	Evaluate the given infix expression and display its result.

The **<infixExpression>** will only use the operators of addition, subtraction, multiplication,
division and the parentheses. It will also only contain unsigned (i.e. positive) integer values. We
obviously could allow for more operators and more types of values, but the purpose of this
assignment is to focus on writing classes and not on complicated arithmetic expressions.

Infix Expression are when the operators of addition, subtraction, multiplication and division are
listed IN between the values. Infix Expressions require the use of parentheses to express non-
standard precedence.   Examples of Infix Expressions are:

        42 + 64
        60 + 43 * 18 + 57
        (60 + 43) * (18 + 57)
        18 – 12 – 3
        18 – (12 – 3)

Postfix Expressions are when the operators of addition, subtraction, multiplication and division
are list AFTER (i.e. "post") the values. Postfix Expression never require the use of parentheses
to express non-standard precedence. The fact that postfix expressions do not need parentheses
makes them much easier to evaluate than infix expressions.   In fact, most computers first
convert infix expression to a postfix expression and then evaluate that postfix expression.

A postfix expression is evaluated using a stack. When a value is encountered, it is pushed onto
the stack.   When an operator is encountered, two values are popped from the stack. Those
values are evaluated using the operation implied by the operator. Then the result is pushed back
onto the stack.   When the end of a postfix expression is encountered, the value on the top of the
stack is the result of the expression. There should only be one value on the stack when the end
of the postfix expression is encountered.

Examples of Postfix Expressions are:

    42 64 +
    60 43 18 * + 57 +
    60 43 + 18 57 + *
    18 12 – 3 –
    18 12 3 – –

Both the algorithm to convert an infix expression to a postfix expression and the algorithm to evaluate a postfix expression require the use of stacks. Note that the conversion algorithm requires a stack of operators, while the evaluation algorithm requires a stack of values.

For this program, you are to write the following **methods** for the dynamic array stack class:

    void init( );   // initialize the stack for use THIS MAY BE A CONSTRUCTOR
    boolean isEmpty ( ); // return **true** if the stack has no members
    void push (data);   // add the data to the top of stack; grow dynamic array if needed
    data top ( );   // return the data value on the top of the stack
    void pop ( );   // remove the data value from the top of the stack
    void reset ( );   // "clear" the stack to remove any values it contains

For this program, your dynamic array MUST start with 2 positions in the array. When the array needs to grow, it size MUST grow by 2 additional positions each time (note the array to grow in size from 2 to 4 to 6 to 8 to 10 to ...). Note the init( ) method to set up the stack should really be a constructor (but could be a normal method). You can also write a "topPop( )" method that combine both the top and pop operations in a single method. Also, note that since the values from the infix expressions will all be integer values (we will only do integer division), and since the operators can all be represented as characters (which is a sub-class of integers), you are more than welcome to write a single stack class which store integers to be used for both stacks.

The instance/variable containing for each stack MUST be declared as a local variable to some method (it may be as a local variable to a method other than main( )). **It may NOT be global.** Each operation performed on the stack MUST be done in its own method. ALL data members of the stack class MUST use access modifier of private.

Algorithm for Infix to Postfix Conversation and Postfix Evaluation

First we will define the algorithm use for popAndEval ( ). This algorithm plays a critical role in the overall Evaluation of our expressions.

    op = top (OperatorStack)
    pop (OperatorStack)
    v2 = top (ValueStack)
    pop (ValueStack)
    v1 = top (ValueStack)
    pop (ValueStack)
    v3 = eval (v1, op, v2)
    push (ValueStack, v3)

To keep some of the error handling/recovery simple for this project, if you are trying to perform a top operation on an empty ValueStack, print out an error message and return the value of -999. If the operator op for eval ( ) is not one of *, /, + or -, print out and error message and return the value of -999.

When getting the input for an infix expression, the program **proj5Base.cpp** breaks everything down into "tokens". There are 3 types of tokens we are mostly concerned with: an OPERATOR token, a VALUE token and the EndOfLine token. The code in the function processExpression() already checks for these token but you will need to add the code how to interact with the stacks.

First step:
    Empty both the OperatorStack and the ValueStack

Second Step:
    While (the current token is not the EndOfLine Token)
        if (the current token is a VALUE)
                push the value onto the ValueStack
        if (the current token is an OPERATOR)
                if (the current operator is an Open Parenthesis)
                        push the Open Parenthesis onto the OperatorStack
                if (the current operator is + or -)
                        while (the OperatorStack is not Empty   &&
                                the top of the OperatorStack is +, -, * or /)
                                        popAndEval ( )
                        push the current operator on the OperatorStack
                if (the current operator is * or /)
                        while (the OperatorStack is not Empty &&
                                the top of the OperatorStack is * or /)
                                        popAndEval ( )
                        push the current operator on the OperatorStack
                if (the current operator is a Closing Parenthesis)
                        while (the Operator Stack is not Empty &&
                                the top of the OperatorStack is not an Open Parenthesis)
                                        popAndEval ( )
                        if (the OperatorStack is Empty)
                                        print an error message
                        else
                                        pop the Open Parenthesis from the OperatorStack
        get the next token from the input

Third Step:
    Once the EndOfLine token is encountered
        while (the OperatorStack is not Empty)
                popAndEval ( )
        Print out the top of the ValueStack at the result of the expression
        Popping the ValueStack should make it empty, print error if not empty

The error statements in the above algorithm should only occur if an invalid infix expression is given as input. You are not required to check for any other errors in the input.

Command Line Argument: Debug Mode

Your program is to be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program should display the operators and values as they are encountered in the input. The printf statements for these are already included in the program proj5Base.cpp, so you just need to make them print only if the –d flag is given.

When the flag is not given, this debugging information should not be displayed. One simple way to set up a "debugging" mode is to use a boolean variable which is set to true when debugging mode is turned on but false otherwise. This variable may be a global variable. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
        printf (" Debugging Information \n");
```

Provided Code for the User Inteface

The code given in proj5Base.cpp should properly provide for the user interface for this program including all command error checking. This program has no code for the stacks. It is your job to write the methods for the specified operations and make the appropriate calls. Most of the changes to the existing program need to be made in the **processExpression ( )** method. Look for the comments of:

```
// add code to perform this operation here
```

Note: the instances containing the operator stack and value stack are required to be a local variable in a function. The function processExpression() is strongly suggested as the function in which to declared the instances of these stacks.

Use of Existing C++ Libraries

You are responsible to write the code for the dynamic array stacks yourself! This is the primary purpose of this project. Since the C++ Standard Template Libraries already contain a class called **Stack**, you should name your class something like **MyStack** to avoid confusion.

You are not allowed to use any of the classes from the C++ Standard Template Libraries in this program. These classes include ArrayList, Vector, LinkedList, List, Set, Stack, HashMap, etc. **If you need such a class, you are to write it yourself.** These are sometimes called the C++ Standard Container Library. A full listing of the C++ Standard Template Libraries can be found at:
  http://www.cplusplus.com/reference/stl/

MULTIPLE SOURCE CODE FILES

Your program is to be written using at least two source code files. All of the storage structure code (the stack code) is to be in one source code file. While the other file (or files if you wish to have more than two source code files) would contain the remainder of the program. You are to include a makefile that creates an executable called **proj5**.

Program Submission

You are to submit the programs for this lab via the Assignments Page in Blackboard.

To help the TA, name the file containing the main( ) method that starts your program with your net-id and the assignment name, like:

- ptroy1Proj5.cpp

Also, make sure you replace "ptroy1" with your own Net-ID!

Since we are to have multiple source code files, please zip all of your java code files together into a file name with your net-id and the assignment name, like:

- ptroy1Proj5.zip

Again, please replace "Ptroy1" with your own Net-ID!