CS 211 – Programming Practicum Spring 2021

Programming Project 5

Due: **Friday 4/2/21** at 11:59 pm

Infix Expression Evaluator

Writ 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 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 C++ classes and not on complicated arithmetic expressions.

Infix Expression are when the operators of addition, subtraction, multiplication, division and modulo 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)

The simplest algorithm to evaluate an Infix Expression includes a translation from an Infix expression to a Postfix Expression and then evaluating the Postfix Expression. The algorithm that we will use combines the translation and evaluation into a single algorithm. So, the following short discussion of Postfix expression may be useful in understanding our algorithm. Note that all of these algorithms require the use of stacks; hence, the reason for the stack class.

Postfix Expressions are when the operators of addition, subtraction, multiplication, division and modulo 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.

CS 211 – Programming Practicum Spring 2021

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 a dynamic array stack class. Your class should have proper **methods** supporting stack operations, such as push( ), pop () , top ( ), and isEmpty( ).

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 ...). 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 modulo), 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 stores 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.

CS 211 – Programming Practicum **Spring 2021**

Algorithm for Infix to Postfix Conversation and Postfix Evaluation

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.

The above algorithm calls another algorithm called popAndEval ( ) which is shown below. 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.

Print statements:

Invalid expressions:

```
printf("Invalid expression.");
```

Valid results:

```
printf("Expression result: %d\n",  );
```

Provided Code for the User Interface

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.

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 multiple source code files. All of the storage structure code (the stack code) is to be in one source code file. The token class should be in another source code file. The other file would contain the remainder of the program. The above implies that you will need to write any appropriate .h file(s) and a makefile.

Coding Style

Don't forget to use good coding style when writing your program. Good coding style makes your program easier to be read by other people as the compiler ignores these parts/differences in your code. Elements of good code style include (but may not be limited to):

- Meaningful variable names
- Use of functions/methods
- Proper indentation
- Use of blank lines between code sections
- In-line comments
- Function/method header comments
- File header comments

The Code Review Checklist also hints at other elements of good coding style.

Program Submission

You are to submit the files for this project as a single zip file on **Gradescope.**
In your makefile, please rename the final executable program to "Project5".
Create a directory called "Project5", then put all of your source code files, your header file and your makefile into this directory. Zip this directory and submit the zip file.

Note that the folder name and file name are case sensitive.

Comments on the Program and Given Algorithm

The algorithm provided will properly evaluate a properly written infix expression. The algorithm does SOME error checking but does not do ALL possible error checking. To perform all of the possible error checking, the algorithm would need to be much, much larger and more complex than provided here. So, your program is not expected to give "intelligent answers" for all possible inputs**. Again, your program MUST provide the proper result for any valid infix expression.** If the input given is not a valid infix expression, the algorithm should give you an error message.

As was stated above, **the purpose of this assignment is to focus on writing and using C++ classes** and not on complicated arithmetic expressions or on proper error recovery of invalid input. Such a program could be written, but that is beyond the scope of what this project is attempting to accomplish.

Also, since the purpose of this assignment is to focus on WRITING a C++ class, **you may NOT use C++ templates in this program**. There are too many sources of a template C++ stack class available on the web. Using such code only shows that you know how to copy and paste code from the internet, so you are NOT allow to use C++ templates for this program.