**Programming Assignment #4 – PostFix Calculator**

**Problem: This assignment is an application of the stack abstract data type. Use stacks to design a calculator that can handle postfix or infix notation (if specified by the user) for calculations involving +,-,\* and /. The program should be capable of converting from infix to postfix and the other way around. It should be capable of evaluating infix and postfix. (Note that for infix, your program should convert to postfix first, then evaluate.) The user should be given an option to perform the calculations in floating point (doubles) or in integer arithmetic.**

**Step 1: Complete the code for the linked-node implementation of the templated Stack. Use the incomplete Stack.h and Stack.cpp found on the Z drive. Complete the functions copy, push and pop. The push function should check for enough memory. The pop function should first check to see if the stack is empty; if so, the function should output “Stack Empty” and exit with error code 1: exit(1). You can use PFCalculator.cpp to help test your Stack class.**

**Step 2: Complete the PFCalculator.cpp file – download the skeleton of this file from the Z drive. This program will compile assuming you have finished Step 1. You must complete the following functions:**

**a) checkParentheses: This function should take one argument of type string and return a Boolean value. This function should behave like the balParen.cpp example – it should return false if there are unbalanced parentheses (or an unmatched ‘)’ ) in the string, otherwise return true. You will use this function to check the input for the infix manipulations. Hint: You can use the “break” command for the loop to avoid popping an empty stack. This is all the error checking you need to do for text. (Math error checking is described below).**

**b) evaluatePostfix: This function should be a template function following the given prototype. “T” (the type parameter) will be either “double” or “int”. The function should take one argument of type string and a reference argument of type “T”, and return a Boolean value. The argument should be a string containing a postfix expression (you do not need to check this – assume that it will be a proper postfix expression). Here are the rules for the input:**

**(i) The expression may contain only nonnegative integers. Distinct integers are separated by spaces. If two (or more) digits are not separated, they should be considered part of the same integer.**

**(ii) The expression may contain the binary operators +, -, \*, /. (It is not necessary to assume the operators are separated by spaces).**

**(iii) You may assume the argument gives a valid postfix expression.**

**You will need a stack of type “T” for this function. When evaluating “/”, check to see if the right operand is 0, if so, issue a warning, and return false. In all other cases, the reference variable should contain the result of the calculation and the function should return true. Note that you will have to be careful when evaluating “/” and “-“ since they are not commutative. It might help to try to write this function as a nontemplate function first.**

**c) infixToPostfix – This function should take one argument of type string and return a type string. The argument should be a string containing a postfix expression (you do not need to check this). It would best to use a stack of chars for this. Here are the rules for input expression:**

**(i) The expression may contain only nonnegative integers. Distinct integers are separated by spaces.**

**(ii) The expression may contain the binary operators +, -, \*, /. (It is not necessary to assume the operators are separated by spaces).**

**(iii) The expression must be FULLY PARENTHESIZED, but not over parenthesized. This means that every instance of a binary operator is of the form “( lhs op rhs )” where “op” is the binary operator and lhs and rhs are the right and left operands (which can be integers or other expressions). These are valid: “(1+2)” “(7-(5\*2))” “((7-5)\*2)”. These are invalid: “(2+3)\*7” (not enough parentheses), “(((2+3)))” (too many parentheses). So it is only one ‘)’ per operator.**

**You don’t have to check for valid input in this function. The string returned should be a valid postfix expression.**

**d) postfixToInfix: This is the hardest function. The function should take one argument of type string and return a type string. The argument should be a string containing a postfix expression (you don’t need to check this). Here are the rules for the input expression:**

**(i) The expression may contain only nonnegative integers. Distinct integers are separated by spaces.**

**(ii) The expression may contain the binary operators +, -, \*, /. (It is not necessary to assume the operators are separated by spaces).**

**The string returned should be a valid infix expression and should follow the rules of input exression for the “infixToPostfix” above. This function can be done using a stack of strings and no other stacks.**

**HINTS: Some of the examples from class could be altered to help on some of the functions.**

**1) For the checkParentheses function, see the example “balParen.cpp”**

**2) For the evaluatePostfix function, see the example “evalpf.cpp”**

**3) For the infixToPostfix function, see the example “ifixpfix.cpp”, but note that the example only works for single digit numbers.**

**\*\* For this assignment, you must alter the given test program PFCalculator.cpp by adding the function definitions and filling in where indicated. Suggestion: you can always uncomment/comment out portions of the test program, so you can test your class function by function.**

**Submit: Stack.cpp and PFCalculater.cpp. You don’t need to change Stack.h**

**Extra Credit: Will be added later.**