PROGRAM 2 / CSC1310

INFIX TO POSTFIX CONVERTER / STACKS & QUEUES

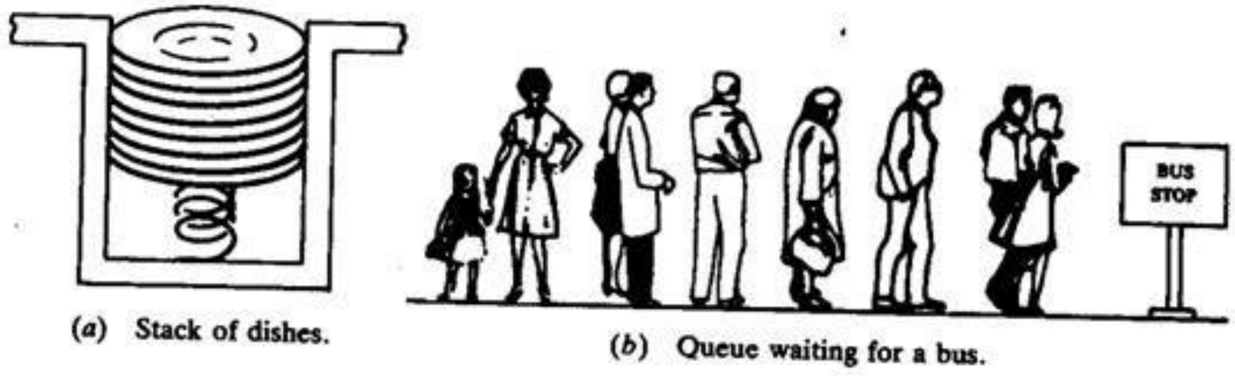
ASSIGNMENT & DUE DATE

Assignment Date: Thursday, September 27, 2018

Due Date: Thursday, October 18. 2018

WHAT SHOULD THIS PROGRAM DO?

Get a math expression from the user in ***Infix*** notation (read in as a string), convert the expression into ***Postfix*** notation using a stack & queue, and then calculate the expression using a stack.



(*a*) Stack of dishes. (*b*) Queue waiting for a bus.

WHAT IS INFIX NOTATION?

Infix notation is the traditional way we write arithmetic expressions, with each operator between two operands. For example,

```
5 + 4 * (9 – 3) / 2
```

The problem with infix notation is that in order to get the correct answer of 17, you have to know that multiplication and division have higher precedence than addition and subtraction, and that parentheses can subvert precedence. A more logical notation is postfix.

WHAT IS POSTFIX NOTATION?

Postfix notation is where an expression is read strictly from left to right, and each operator works with the two results immediately preceding it. In postfix notation the expression above becomes

```
5 4 9 3 – * 2 / +
```

Which may seem a bit strange, but is much easier for a computer to process.

WHAT IS AN ALGORITHM TO TRANSLATE INFIX TO POSTFIX?

Translating an infix expression (preferred by humans) into its postfix equivalent (preferred by machines) is reasonably straightforward. This algorithm uses a stack to store lower precedence operators until it is time to add them to the postfix version; each expression is represented as a queue of "tokens."

```
Input:  infix, the infix expression, a queue of tokens
Output:  postfix, the postfix expression, a queue of tokens
Require:  opstack, a stack of operator tokens
-------------------------------------------------------------------
while infix isn't empty do
      token <- next element from infix

      if token is a left parenthesis then
            push token on opstack

      else if token is a right parenthesis then

            while top element of opstack is not equal left parenthesis do
                  append top element of opstack to postfix

            remove left parenthesis from opstack

      else if token is an operator then

            while opstack isn't empty and priority(token) <= priority(opstack's top) do
                  append top element of opstack to postfix

            push token on opstack

      else //token is an operand
            append token to postfix

while opstack isn't empty do
      append top element of opstack to postfix
```

HOW TO EVALUATE A POSTFIX EXPRESSION

Stacks can be used for evaluating postfix expressions. The postfix expression

```
5 4 9 3 – * 2 / +
```

can be evaluated as follows:

```
push 5 on the stack
push 4 on the stack
push 9 on the stack
push 3 on the stack
pop top two values; subtract; push result on the stack
pop top two values; multiply; push result on the stack
push 2 on the stack
pop top two values; divide; push result on the stack
pop top two values; add; push result on the stack
```

The result (17 in this case) is what remains on top of the stack.

FILES THAT SHOULD BE INCLUDED IN YOUR SUBMISSION

All the following files you will have to create:

- **Program2.cpp** – this file will contain your main function as well as two other functions.
- **Stack.h** – a template class implementing a Stack (do not use the stack class provided in the standard template library – this must be one that you create and include)
- **Queue.h** – a template class implementing a Queue (do not use the stack class provided in the standard template library – this must be one that you create and include)

The following three files are provided for you in ilearn, but must be included in your submission.

- **RUN.bat** – this is the batch file shouldn't be changed
- **Makefile** – this is the Makefile used to compile your program – don't change this either.
- **TEST_CASE.txt** – this is the test case similar to one that I will use to grade your program – don't change this either.

PROGRAM SPECIFICATIONS (DIRECTIONS ON HOW TO WRITE THE PROGRAM)

STACK.H

Create a file (Stack.h) that declares a template class named Stack that implements the Stack ADT. This template class should be able to create a stack of any type of object. You should be able to look at the examples that I provided for you in ilearn to know what attributes you need and what the following required functions should do:

- Constructor
- Destructor
- Push
- Pop
- Peek
- IsEmpty

QUEUE.H

Create a file (Queue.h) that declares a template class named Queue that implements the Queue ADT. This template class should be able to create a queue of any type of object. You should be able to look at the examples provided for you in ilearn to know what attributes you need and what the following required functions should do:

- Constructor
- Destructor
- Push_back (or enqueue)
- Pop_front (or dequeue)
- IsEmpty
- Display

PROGRAM2.CPP

This is the driver. This file should have a main function and several other functions to organize or help the main function as described below:

- **main**
  - The main function will begin by printing out "Infix to Postfix Converter". Then, using a loop the program allows the user to convert & calculate as many arithmetic expressions as they want.

  - The program first asks the user for the infix expression and reads it in as a string. The program should remove any spaces in the string. **Note: you may assume that the user will only be entering in SINGLE DIGIT NUMBERS in their**

calculations.

- Then, the program should print out the expression to the screen, printing one space between each character for easier reading.

- Then, the function should implement the given algorithm that uses a stack to store lower precedence operators until it is time to add them to the postfix version and each expression is represented as a queue of "tokens." The "tokens" are characters from the infix notation string.
  - Make sure to call the **isLowerPriority** function in this algorithm to determine if the first character (operator) is lower priority then the second one.

- Then, call the Queue display function to print out the postfix notation of the expression.

- Then, pop all the characters from the postfix Queue and place them in a string.

- Call the **calculateExpression** function, sending this postfix notation string to this function. Then, print the result of the expression, which was returned from this calculateExpression function.

- Ask the user if they want to run the program again. If not, then print "Goodbye!" and end the program.

- **calculateExpression** – this function accepts a string as a parameter and will return a double, which holds the result of the calculation. This function will create a Stack<double> object. Then, it will parse through the string, one character at a time.

  - If the character is a digit, then convert the character to a double and then push it on the stack.

  - Otherwise, the character is an operator, so pop two nodes from the stack, saving their values in two different temporary variables. Then, if the character is a '+', then get the addition of these two values. If the character is a '-' then get the subtraction of the first one from the second one. Do this for addition, subtraction, multiplication, and division. No other operator is supported in this program. Last, push this result on the stack.

  Once you parse through the entire string, you can return the final result from this function.

- **isLowerPriority** – this function accepts two characters as parameters and returns true if the first character is an operator that is lower priority than the second character. The possible operators are (,), +, -, *, /. Priority zero is lowest priority and priority 3 is highest priority in the list below.

  - '(' is priority 0
  - ')' is priority 3
  - '*' & '/' is priority 2
  - '+' & '-' is priority 1

READABILITY OF OUTPUT & CODE DOCUMENTATION

- Make sure that your output looks similar to my sample output (below). When I run your program, it shouldn't make me want to scream. It should be extremely readable and user-friendly.
- **For this program, don't worry about comments except put your NAME at the top of all files!!!**

WHAT TO TURN IN

Zip ALL the files required to compile & run the program, (including the files I provided for you) in a single zipped file named whatever you want. Then, upload this zip file to the assignment folder in ilearn. **I will remove one point if you turn in unzipped files. Programs that do not include all the files listed in the "FILES" section above will not be graded.**

SAMPLE OUTPUT

```
Infix to Postfix Converter

Enter your infix expression:  5+4*(9-3)/2


Infix Expression:  5 + 4 * ( 9 - 3 ) / 2
Postfix Expression:  5 4 9 3 - * 2 / +
Result of the Expression:  17

Would you like to run the program again? (Y/N)  Y


Enter your infix expression:  4+6*5


Infix Expression:  4 + 6 * 5
Postfix Expression:  4 6 5 * +
Result of the Expression:  34

Would you like to run the program again? (Y/N)  y


Enter your infix expression:  (2+3)-7/9


Infix Expression:  ( 2 + 3 ) - 7 / 9
Postfix Expression:  2 3 + 7 9 / -
Result of the Expression:  4.22222

Would you like to run the program again? (Y/N)  n


Goodbye!
```