Note: This is an ***OPTIONAL*** project that will replace the grade of a previous project if you had a lower grade on projects 1-5.

Project 6: Post Fix Calculator

Background:

One of the first projects a computer science student learns to make is a basic calculator. Here you will the *stack* data type to make a calculator that is slightly more advanced than the ones you made in csci 127. This calculator will take into acount the order of operations (or PEMDAS) and it will also check or illegal characters and operators.

Before we get into making the calculator we should understand the different type of expressions you will be dealing with this project.

The classic expression that normal people understand and write are called ***in-fix***:
An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another infix string.

```
A
A + B
(A + B) + (C – D)
```

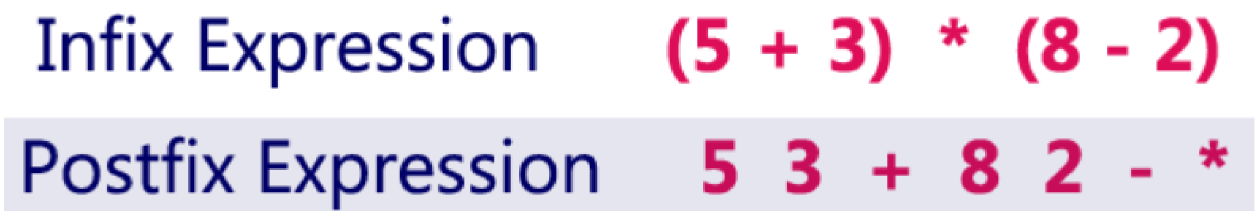
Here we know that we must do what is in the parenthesis first so we do the (A + B) first but then we do the (C - D) since it's also in parenthesis. We can add the 2 together due to the plus in the middle.

You will convert the in-fix expressions mentioned above to what we call ***post-fix***:
A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

```
A
A B +
A B + C D –
```

There is also a type of expression knows as *pre-fix* but for this project we will not need to know about that.
I will go more into depth on what post-fix is since this is probably a newer notation for a lot of you.

Let's consider the following expression:



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Now the top two are equivalent to each other. Let's see why:
In our normal infix expression we would use PEMDAS since infix allows us to use parentheses. Following the order of operations we do what's in the parentheses first so 5+3 is first. To convert that to post fix we have:
5 3 +
This means we take the number 5 and we take the number 3 and we add them together, so the operator itself will always be the THIRD string/ character, as opposed to being in the middle with infix: 5+3
Now we still have parentheses left so we ignore the multiplication for now, we have to do 8-2 first, so in our postfix expression we concatenate 8 2 - so that means our overall postfix expression now looks like:
5 3 + 8 2 -
Now at the end we slap on the * which means to multiply together which means we now have:
5 3 + 8 2 - *
Why is it the case we can just slap it on at the end? Well let's see how post fix is calculated, we don't use PEMDAS, there are no parenthesis or order of operations for post fix, we simple just read left to right.
So what 5 3 + 8 2 - * translates to is:
We take 5 and 3 and add it, We take 8 and 2 and subtract them and then we multiply the results we have left:
So our steps look like
5 3 + 8 2 - *
8 8 2 - *
8 6 *
48
Remember the operator always proceeds 2 numbers (the operands)
That's why in step two when we have:
8 8 2 - *
The - applies to the 2nd 8 and the 2 and we don't do anything with the first 8 yet nor do we do anything with the * since the we read left to right and the * comes after another operator, meaning we apply it last.

This might seem a bit confusing to read as text but there are plenty of YouTube tutorials on these notations, and even more tutorials on how to convert from one to another, which is what you will be doing for this project.

Your objective for this project is to a class called ***PostFixCalculator.cpp*** In order to successfully complete this project, you must understand the concept of the regular **Stack** ADT. This differs significantly from the queue ADT you worked on for the last project since a stack is now a **LAST IN FIRST OUT** data structure (LIFO).

Implementation:

- ***You will also be given the cpp file for PostFixCalculator***
- You will need to implement each method that resides in the cpp file and follow the instructions in the comments.

Additional resources:

• ***Stack:*** ○ GeeksForGeeks ○ Wikipedia	• ***YouTube links to help you get through this:*** ○ Infix to Postfix ○ Evaluation of Postfix ○ Infix to Postfix Part 2

You are *NOT* limited to these resources, Google as much as you need.

Task: Implementing the `PostFixCalculator` functions!

Some of the functions on here are going to be very easy and only a few lines of code. As long as you read the instructions you will be fine.

Your first "challenging" function might be the one that determines unbalanced parenthesis. This function must use a stack, specifically the #include <stack> data type. If you use other methods you may get points off.

Here is the documentation for the <stack> data type:
https://www.geeksforgeeks.org/stack-in-cpp-stl/

For `convertToPostfix` you will be taking in an infix expression as a string and return the post fix one as a string, once again youtube and google will help you a lot with this but here is a tip:
You can first use the unbalanced parentheses, along with illegalOperator to automatically disqualify any bad expressions.
Then you should loop through the string and check whether or not each character is an integer, an operator or a left parenthesis or a right parenthesis. Each of these cases will do different things clearly.

One thing to note however is for postfix we want to add spaces in between so:
(A + B)
will be:
A B +
and not:
AB+

For `calculatePostfix` it's just a simple matter of calculating the post fix expression.
Due to the way we tested this make sure the first line of your code is:

```
if(postfix_expression == "Not a valid expression!"){
	return "Not a valid expression!";
}
```

Also note that we are returning a string so if you are doing your calculations on a double data type make sure to use:

```
std::to_string(your double variable here)
```

And return that string as your final result.

Testing
How to compile:

```
g++ <test main file> -std=c++17
```

You must always implement and test your programs **INCREMENTALLY!!!** What does this mean? Implement and test one method at a time.

- Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable).
- Implement the next function/method and test in the same fashion. **How do you do this?** Write your own `main()` function to test your class. In this course you will never submit your test program, but you must always write one to test your classes. Choose the order in which you implement your methods so that you can test incrementally: i.e. implement mutator functions before accessor functions. Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use stubs: a dummy implementation that always returns a single value for testing. Don't forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Grading Rubric

Correctness 80% (distributed across unit testing of your submission) **Documentation 10% Style and Design 10%** (proper naming, modularity, and organization)

Important: You must start working on the projects as soon as they are assigned to detect any problems with submitting your code and to address them with us **well before** the deadline so that we have time to get back to you **before** the deadline. This means that you must submit and resubmit your project code **early** and **often** in order to resolve any issues that might come up **before** the project deadline.

There will be no negotiation about project grades after the submission deadline.

Submission:

You will submit the following file(s)**:** **`PostFixCalculator.cpp`**

Your project must be submitted on Gradescope. Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You **MUST** test and debug your program locally. Before submitting to Gradescope you **MUST** ensure that your program compiles (with g++) and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the "Programming Rules" document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you don't have through Gradescope. "But it ran on my machine!" is not a valid argument for a submission that does not compile. Once you have done all the above you submit it to Gradescope.