UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

# Lab 4 - Reverse Polish Notation

**Commit ID Form: https://goo.gl/forms/CzOtzohM6o23FkyF3**

**18 Points**

## Introduction

In this lab you will be writing another calculator, but this one can take in very long expressions in reverse Polish notation (also known as postfix notation). You'll be implementing another library, in this case to implement a data structure known as a stack, and you will then use it in your calculator. A stack is the fundamental paradigm on which reverse Polish notation is based, which will be obvious later.

## Reading

- **K&R** – Sections 5.1-5.3, 6.1-6.2

## Concepts

- String manipulation
- Structs
- Stacks
- Error Handling

## What we provide

- Stack.h — provides the function prototypes for the functions that you will implement in Stack.c along with brief descriptions of each function. Add this file to your project directly. **You will not be modifying this file at all!**
- StackTest.c — provides a place for the test harness for unit testing your Stack.c implementation. This contains a `main()` and starter code for the testing module.
- BOARD.c/h — contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values. **You will not be modifying this file at all!**
- rpn.c — this file contains `main()` and the starter code.

## Assignment requirements

- Create a new file called Stack.c that implements all of the functions whose prototypes are in the header file Stack.h.
    - The return values from these functions should use the constants defined in Stack.h and BOARD.h where appropriate.

- Expand main() in StackTest.c to test each of the functions within stack.h
    - Include StackTest.c in your project to run your test harness. StackTest.C and rpn.c cannot be run at the same time so you will have to remove it once you are done testing.
    - Test each function at least twice (just like in MatrixMath)
    - Keep a tally of which tests passed and which failed
    - Print back a record of which tests passed and which failed

- Expand `main()` in rpn.c to do the following
    - Greet the user once on startup
    - Prompt the user for an RPN string that includes doubles and the 4 arithmetic operators: + - / *
        - Your input string should be able to handle up to 60 chars
        - Make sure that your calculator handles floats properly and that all calculations are done with values of type floats. So this includes 0.0 and negative numbers.
        - HINT: using `fgets()` to read user input here will be helpful!
    - Parse the string into a sequence of string tokens using `strtok()` from string.h
    - Process each token and utilize the stack to perform operations as dictated by the RPN syntax
        - The stack must be properly declared as a variable of datatype "struct Stack"
    - Return the only element in the stack as the result otherwise alert the user that there was an error according to the sample output given below.
    - Return to prompting the user for another RPN string to calculate.

- Your calculator must be able to handle and return errors for the following situations. If more than one of these errors is appropriate for the given situation, only one needs to be displayed. After an error the calculator should prompt the user for a new string, much the same as if the calculation had been correct:

- RPN strings that don't return a single result (more or less than 1 element left on the stack)
    - Output: "ERROR: Invalid RPN calculation: more or less than one item in the stack."
- Invalid input characters/tokens (non-operator, non-numeric)
    - Output: "ERROR: Invalid character in RPN string."
- When the stack is empty and you are attempting to pop a value off
    - Output: "ERROR: Not enough operands before operator."
- When the stack is full and you are attempting to push another value on
    - Output: "ERROR: No more room on stack."

- Create a readme file named README.txt containing the following items. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.

    - First you should list your name & the names of colleagues who you have collaborated with.[1]
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? How did you work with other students in the class and what did you find helpful/unhelpful?
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understand this lab or would more teaching on the concepts in this lab help?

[1] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

- Format your code to match the style guidelines that have been provided.
- Make sure that your code triggers no errors or warnings when compiling. Compilation errors will result in a significant loss of points.[2] Compilation warnings will result in lost points.
- **Required Files**
    - Stack.c
    - StackTest.c
    - rpn.c
    - README.txt.

**Grading**

This assignment is worth 18 points:

- 5.5 points – Correctly implementing all functions declared in Stack.h
- 4.5 points – Correctly implementing RPN calculator functionality
- 3.0 points – Correctly tests all the functions in Stack.h
- 2 points – Correctly handles the four error messages
- 1 point – Provided a README.txt file
- 1 point – Followed proper GIT hygiene
- 1 point – Follows style guidelines (<= 5 errors in all code) and has appropriate commenting
- **Extra Credit:** 1 point for properly handling backspaces in the input string. This code should be implemented inside the helper function `ProcessBackspaces()` declared immediately below `main()`. This function follows from the function prototype defined above `main()`.

You will lose points for the following:

- NO CREDIT for sections where required files don't compile
- -2 points: **any** compilation warnings

**Program Flow**

This program follows a very similar outline to the calculator you did previously, but takes a few more steps to get to the user input because you will be parsing a string. Here's a high level overview.

```
Output greeting to the user
```

[2] There is NEVER any reason to submit code that does not compile. Make sure you take the 30 seconds to check this before submitting your code. We will not attempt to fix code that does not compile; you will simply get a zero.

```
while (TRUE)
Read in characters from stdin until a newline is received
Split incoming string into string tokens
Check that the tokens are valid operators or numbers
For each token
	if operator
		pop two elements and push result
	else if number
		push number
if only one element in stack
	output result
else
	output an error
```

For this program you will need to be more careful about handling unexpected input. For example the user may not enter a properly formatted RPN string. Or the final calculation could result in two elements in the stack. Both of those are errors and need to be dealt with reasonably.

**Backspace handling**

Backspaces are not natively handled by any of the string handling functions provided in the C standard library as they're just another ASCII character. For extra credit you may implement this functionality yourself within the function `ProcessBackspaces()`, whose prototype is declared above `main()`.

Backspaces are a special character within the ASCII character set and so appear in a string just as any normal character would. Your function should process any input string and whenever it encounters a backspace character, it should overwrite the preceding character with the following character. There are two edge cases here to think about: handling multiple backspaces in a row and strings with more backspaces than characters. Your function should be able to manage both situations, which could also occur in the same string!

You should write unit tests to test this function outside of your calculator functionality as it will be faster and easier. A simple example test follows:

```
char testString1[] = "1 2\b3 +"; // Should simplify to "1 3 +"
ProcessBackspaces(testString1); // testString1 should now have "1 3 +" in it.
puts(testString1); // Print the string. This will not show \b chars!
```

In running the above test code I suggest setting a breakpoint on the `puts()` and looking at `testString1` in the Variables window to confirm that it looks as expected. Now expand from here into other tests.

**Relevant functions**

You will want to use the following functions in your code. It should be fairly apparent where they would be useful. Documentation is available under Help->Topics->MPLAB XC32 Toolchain -> XC32 Standard Libraries->Standard C Libraries. These functions are also described online (see Wikipedia's string.h entry) and in K & R.

`strtok()` – Splits a string into tokens based on the delimiter you passed it. Everywhere the delimiter is seen in your string, it is replaced with the null character ('\0'). Be sure to read up on any examples because it's a slightly different function that what you're used to.

`strlen()` – Returns the number of characters in a string (the number of characters in a character array before the first null character).

`atof()` – converts a string to a double. Be careful how you detect errors in the conversion process.

`fgets()` – Reads in user input until a newline is reached. Remember to pass "`stdin`" as the third argument.

`sprintf()` – Creates a string based on another formatting string and some input variables. Works similar to `printf()` but stores the result in a string.

**Unit testing**

To complete this lab properly you will again need to perform your own unit testing on your functions to confirm that they operate according to spec. You should do this as you write your Stack library using the provided StackTest.c file and then exclude this file from your project once you move on to the RPN calculator portion of this lab.

Some sample code is provided here to illustrate the start of some basic testing, but it does not test any edge cases! Make sure you handle those in your code. Edge cases most often occur when limits are reached, such as when things get full, are empty, or numbers transition across the 0-boundary. For your stack code ensure that your initialization code is correct and emptying or overfilling a stack works as expected.

```
struct Stack testStack;
StackInit(&testStack);
StackPush(&testStack, 3.14159);
printf("Stack has %d elements!\n", StackGetSize(&testStack));
float testFloat;
StackPop(&testStack, &testFloat);
printf("%f = 3.14159!\n", testFloat);
printf("Stack is empty: %d!\n", StackIsEmpty(&testStack));
```

**Structs**

C has a few built-in data types that you should be familiar with at this point: char, int, double, etc. These are known as datatype primitives. As you may be able to guess from the name there can also be non-primitive data types. The most commonly used non-primitive is called a struct (short for structure).

A struct is very much like a physical structure in that it is built up out of smaller components. In the case of a C struct, these components are other data types, either primitives or non-primitives. Why use a structure? A structure is useful for collecting a bunch of related values together. You can then pass the entire structure around to different functions very easily as it's all nicely contained.[3]

A struct that you will be using looks like the following:

```
struct Stack {
	float stackItems[STACK_SIZE];
	int currentItemIndex;
	uint8_t initialized;
};
```

This struct contains three primitives, an array of floats of size `STACK_SIZE`, and two integers. These structure members work just as you would expect an array of floats or integers to work outside of a structure. The difference in using them is how to reference them.

But first we will need to declare an instance of the struct in a new variable. A struct is always referenced first by writing `struct` and then the `STRUCTNAME`, so you can think of the data type of a struct as `struct STRUCTNAME` and then declare it like you would any other variable:

```
struct Stack myStack;
```

Now that we've declared a struct, how do we reference its members? Use the syntax `STRUCTNAME.STRUCTMEMBER`.

```
myStack.initialized = TRUE; // TRUE is from GenericTypeDefs.h
```

Sometimes, though, you'll have a pointer to a struct. We haven't covered pointers yet, so you're not expected to fully understand them, but you should know both how to get a pointer from a variable and how to refer to members of a struct pointer.

```
struct Stack *stackPointer = &myStack;

stackPointer->initialized = TRUE;
```

[3] This is referred to as *encapsulation*, and is one of the key ideas in modern software design.

An ampersand (&) is used to get a pointer from a variable. You've used it before with `scanf()`. You'll be using it in this lab to pass a struct pointer to the functions you'll be implementing. And to refer to the members of a struct from its pointer you use a right-arrow (->) instead of the period (.) used before.

For this lab all of the functions take struct pointers. You'll probably end up writing code that declares a structure variable and then pass it's pointer to the functions using the ampersand syntax shown above or as follows:

```
struct Stack myStack;
StackInit(&myStack);
StackPush(&myStack, 7.0);
```

**Stacks**

A stack is an **a**bstract **d**ata **t**ype (ADT) that we are implementing on top of the array datatype in C. A stack works similar to a deck of cards sitting on a table: you can remove a card from the top of the stack or put a card on top of the stack. Those are the basic operations you can perform on a stack and they're referred to as a pop and a push.[4] A stack is only defined with popping and pushing.

A stack also has a couple of properties when it's implemented, namely its maximum size and its current size. The maximum size of a stack is how many entries it can contain. When working in the real world there are limits on size; the same applies to the memory in a computer. Since it is finite, a stack will only be able to get so large before it can't hold anymore. The current size is the number of items in the stack which will always be less than or equal to the maximum size.

To understand where a datatype with such limited operations may be useful, just think of the game Solitaire that comes with Microsoft Windows. The foundations in the upper-right corner that hold the cards in order are stacks. A stack is also used within C itself to keep track of variables that you declare.

**Reverse Polish notation**

Now that you understand what a stack is we can talk about reverse-Polish notation. Reverse polish notation is a way to describe a mathematical expression. For example you can write "(1 + 4) * (6 – 4) / 8" as "1 4 + 6 4 - * 8 /". The numbers and operators are referred to generically as tokens and are evaluated left-to-right.

[4] Remember from CMPE-12 that you used a stack for temporary memory storage of registers that you wanted to recover later, and also for subroutine parameters and data return. There you directly manipulated the $sp to push and pop.

Reverse-Polish notation (RPN) uses a stack for keeping track of what has already been evaluated. As we progress from left to right we will encounter numbers and operators. Numbers will be pushed onto the stack and an operator will pop two elements off of the stack and push the result back on top. We'll walk through the example above to demonstrate.

| Token | Operation | Stack |
|---|---|---|
| 1 | Number: push | 1 |
| 4 | Number: push | 4<br>1 |
| + | Operator: pop, pop, calculate, push | 5 |
| 6 | Number: push | 6<br>5 |
| 4 | Number: push | 4<br>6<br>5 |
| - | Operator: pop, pop, calculate, push | 2<br>5 |
| * | Operator: pop, pop, calculate, push | 10 |
| 8 | Number: push | 8<br>10 |
| / | Operator: pop, pop, calculate, push | 1.25 |

As number tokens are encountered they are pushed onto the stack. Operations pop elements off the stack and push back the result of their calculation. The last element on the stack after all the tokens are handled is the result of the entire expression. There must only be one element on the stack at the end of the calculation or the RPN sentence was incorrect.

Remember that when handling division or subtraction the order of operations matters and the calculation must be done in a specific order. The first operand that you pop off must be subtracted or divided from the second.

For example, "4 7 -" must result in -3 and not 3.

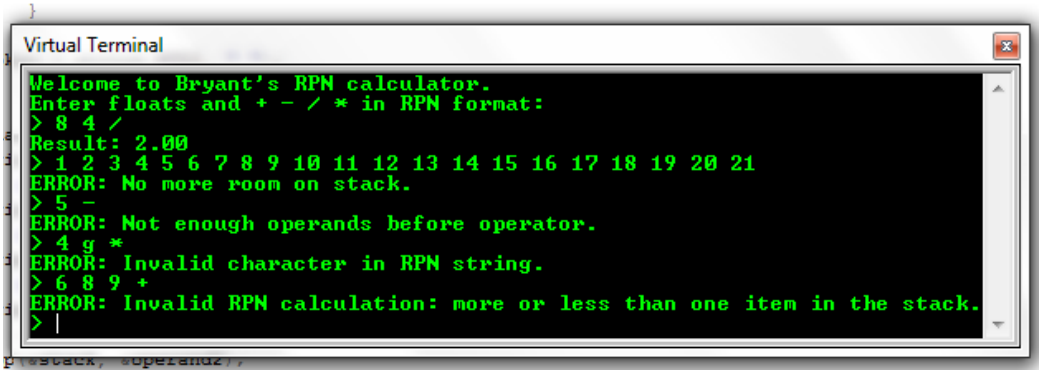
**Output**

There are five possible outputs to this program:

1. Calculation of the result
    - Valid RPN Strings should output a single number that is the result of the calculations.

2. Pushing too many elements onto the stack
    - If the user is trying to push more elements on the stack than the stack size allows, state that this error has occurred.
3. Attempting to pop from an empty stack.
    - If the user is trying to pop elements off an empty stack.
4. Invalid character in RPN string
    - If the user inputs an invalid character into their RPN string.
5. Invalid RPN calculation
    - If the user inputs a string that does not result in a single element left on the stack display this error.

Example output of these 5 scenarios in this order is given below:




**Error handling**

In C there is no error handling built-in to the language (unlike Java or C++). While this may lead you to think that there is no real need for it that is most definitely not the case. C just relies on you to develop your own strategy for managing errors. Now the great thing is that over the last 30 years since C's been around a lot of people have developed strategies for handling errors that we'll use in our own implementation.

You may be wondering what exactly I mean when I say error handling. Error handling is actually two things: code must have a way to signify that an error has occurred and other code must be able to respond to that error. Since I'm referring to errors generally they can be something small such as not being able to find a configuration file where the program would just load with defaults instead. Or it could be catastrophic such as the program requests more memory and it isn't available thus requiring a shut-down.

At the function level, errors are commonly handled by altering the return value. Sometimes a function has a return value solely for this purpose. An example of this is an initialization function (like the `Init()` you will implement). Generally an initialization function wouldn't need to return a value, but to implement error checking it would. So this function will return a 1 if it succeeded and 0 if it failed. Sometimes this is even done for functions that have no defined way of failing solely to adhere to this convention.

This should remind you of the `MatrixEquals()` function you implemented in the MatrixMath lab, but it's actually a little different. Functions like this return an `int` and can be used in Boolean expressions because 0 evaluates to false and any non-zero value evaluates to true (though we generally use 1). The difference is that `MatrixEquals()` doesn't alter its return value based on an error, it is only returning Boolean values for whether its two input matrices are the same. The following example clarifies where the return values are not 0 or 1 for an error.

A lot of code in use is concerned about the size of something. This is done for arrays, abstract data types (ADTs) such as the stack you're implementing, or many other data types (trees, queue, circular buffers, etc.). The size attribute is a very fundamental part of these more complicated data structures. But what if you have a `Size()` function that returns the size of something, let's say a queue, and it encounters an error (a queue is just an array where items come in one side and out the other, like a stack where you could only push in items on one end and pop them out the other). Since it's already supposed to return the size of the object how can it also return an error?

The easy answer is through another argument or some other external variable, but this is complicated and adds a lot more code. We can do something clever here if we think about what the size of a queue actually means. The size of a queue corresponds to how many items are within it (the smallest valid value being 0). This means that if an `int` is the return type of the function all negative values are unused. What we can do is return a negative value if there is an error. In fact we could return different negative values for different errors (but if there is only one error the convention is to return -1).

For this example, the `Size()` function for the queue would have a multitude of return values: -1 for an error, 0 for if its empty, and the actual number of elements in it otherwise.

We have a few different ways to return an error status from functions. But what do we do with them? This is where you will need to think a little and ask if you actually care about the error. Sometimes functions can fail and checking if it succeeded isn't worth it. For an example think about the `printf()` function. This function writes to standard output. If there's a problem with this function that is so fundamental to C, your program probably won't be able to handle it appropriately at runtime and so checking if an error occurs wouldn't be worth it. A counter-example to this would be `scanf()`. This function returns how many string tokens it successfully captured according to its format string passed. Now if your code is expecting two integers from the user and it doesn't parse two integers then that will most likely present a problem. This error could be easily handled by checking to see if `scanf()` did store two integers and prompting the user again until they input those integers correctly. In this case you would care about the return value of `scanf()`.

Continuing with the `scanf()` example, what would the code look like that could handle this error?

```
while (scanf("%d-%d", &int_1, &int_2) != 2);
```

The code does continually calls `scanf()` with the format string until the input matches (`scanf()` should return 2 if it was successful as we've specified that it expects two integers). This takes advantage of a while-loop to continually do this while our condition that the return value is two is not met. Notice that there is no code in the body of the while loop and it has been replaced with a semi-colon.[5] This happens sometimes when you can fit all of the code execution that you'd like to do into the control statements header. This is perfectly valid C (in fact you should have seen it before with the "`while (1);`" at the end of `main()` in all of the earlier labs).

There is one last thing about return values that can make the code more readable: using predefined constants instead of numbers. Numbers that are fixed and have a special meaning in certain contexts are called *magic numbers* because they have a meaning besides this standard numerical meaning. Using magic numbers is fine, but they should be defined as constants (e.g.: `STACK_SIZE` and `MAX_INPUT_LENGTH`). This makes their meaning clear and also allows for the actual number behind the constant to change without needing to change a lot of code. A priority in writing code is readability and modularity, which this addresses. We have defined some return values in the BOARD.h header to make the code even more readable; these include `STANDARD_ERROR` (0) and `SUCCESS` (1). Note that STANDARD_ERROR will evaluate to FALSE if used in a boolean context and SUCESS will evaluate to TRUE, making it quite easy to use them in conditional statements. We have also added SIZE_ERROR (-1). You are required to use these constants in this lab as the appropriate return values.

**Iterative design plan**

What follows is an iterative design plan for this lab. This is the last lab where one will be provided. In subsequent labs you will need to develop your own. This will be checked by the TA/tutors when you ask for help, so please have this with you for them to check.

The basic idea when approaching these labs is to start with the libraries and work towards the integration after the libraries are tested and done. So for this lab we start with the Stack library and then incrementally build the core integration functionality.

---

[5] You need to be careful about this, since if you decide later to add some code to the while loop, it isn't contained within it. A more verbose style, with the opening and closing curly brackets makes this less likely to occur:

```
while (scanf("%d-%d", &int_1, &int_2) != 2) {
        ;
}
```

1. Implement required functionality for the Stack library
    - Implement `StackInit()`
        - Implement your tests inside StackTest.c
        - Test it by calling it and checking that the proper values initialized.
    - Implement `StackIsEmpty()`
        - Implement your tests inside StackTest.c
        - Check it on your initialized stack from the previous step
        - Use `StackIsEmpty()` within `StackPop()` and try to pop from an empty stack.
    - Implement `StackPush()`
        - Implement your tests inside StackTest.c
        - Test it by calling it in `main()` with a value to be pushed onto your initialized stack, and use the debugger to make sure that the "struct Stack" variable that you're using in testing shows the correct value.
        - Also test pushing `STACK_SIZE+1` elements to confirm that it fails properly when trying to push onto a full stack.
    - Implement `StackIsFull()`
        - Implement your tests inside StackTest.c
        - Test it by using `StackPush()` until the `STACK_SIZE` is reached.
    - Implement `StackPop()`
        - Implement your tests inside StackTest.c
        - Test it by pushing a single value onto the stack and then popping that value off.
        - Attempt this with full and empty stacks as well.
    - Implement `StackGetSize()`
        - Implement your tests inside StackTest.c
        - Test it by pushing a specific number of elements onto you stack and then calling `StackGetSize()` to check if it returns the correct number.

At this point your entire Stack library should be completed and tested. You should try a multitude of tests for each function, especially focusing on the edge cases, like pushing onto a full stack, etc. You should commit, push, and submit the google form before

moving into the next stage of completing the integration portion of this lab.[6] Remove StackTest.c from the MPLABX project. Alternatively you could either exclude this file in your project. Regardless of how you do so you will not be able to run both rpn.c and StackTest.c at the same time. This is also an optimal time to fill in some of your README file that you'll be making for this lab. You should submit this and your library so that if anything goes wrong, you'll have at least something submitted for grading.

1. Customize the user greeting in rpn.c, confirming that printing to `stdout` works fine.
2. Read in an input string using `fgets()`
    - Test this by typing a string into a serial terminal and echo it back out using `puts()`. I also suggest using the debugger and breaking after that puts() call to look at the string you read in so that you're sure of all the non-printing characters (like carriage returns, newlines, etc.) that are in the string.
3. Parse the string into tokens using `strtok()`
    - Test by printing out every token to confirm that you're getting the expected tokens for your input strings.
4. Process each token one at a time, converting to a float or processing it as an operator.
    - Just print out the converted float or that an operator was encountered.
    - This is also when the proper handling of 0.0 should be added
5. When encountering a float, push it onto the stack, and when encountering an operator, pop a value off the stack.
    - Print out something like "Value pushed 7.9" and "Value popped 1.0" when that occurs. Verify that this is correct for your input string.
6. Add the computation code for doing each operation, outputting the result.
    - At this point the final output from your program after entering a string will be a sequence of result outputs, with the final one being the total result of your RPN string.
7. Now add error handling to the program as specified from the lab requirements.
8. Comment out test code in rpn.c that shouldn't be in the final submitted version.

[6] Of course, you should have been committing and pushing at every single function you implemented and successfully tested (or even before testing, then after fixing them). This would be considered good GIT hygiene.

9. Go through the lab requirements and point deductions section to make sure you didn't miss anything, excluding the extra credit.
10. Do the extra credit: Process any backspaces in the input string **(optional)**
    - Testing this should be done with the debugger by using a hardcoded string with backspaces and checking that your algorithm removes the appropriate characters. This will be much easier than having to type in a proper string every time and easily gives repeatable results.
11. Go through your code checking the style guidelines examples provided to you to make sure you don't miss any of these points.
12. Commit, push (again, you should have been doing this all along)
13. **Make sure you haven't modified your Stack.h file or we may have problems testing your code resulting in loss of points. Any helper functions within Stack.c need have function prototypes inside your Stack.c file. This means the ONLY functions you can use in rpn.c are the ones we have defined for you in Stack.h.**
    - **We do NOT use your Stack.h file when grading this lab.**
14. Submit all of your code (Stack.c, StackTest.c, rpn.c, README.txt)
15. Submit the Google form with your correct CommitID