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

CSE013E: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 4 - Reverse Polish Notation
18 Points

Introduction:

In this lab you will be writing another calculator, but this one can take in long expressions in reverse Polish notation (also known as postfix notation). You'll be implementing another two libraries. One library will implement a data structure known as a *stack*. The other library will utilize your stack to parse Reverse Polish Notation strings. Finally, you will write a main file that uses your string-parsing library to make an interactive calculator.

Reading:

- **K&R** – Sections 5.1-5.3, 6.1-6.2

Concepts:

- String manipulation
- Structs
- Stacks
- Error Handling

Required Files:

- stack.c
- stack_test.c
- rpn.c
- rpn_test.c
- Lab04_main.c
- README.md

Lab Files:

- **DO NOT edit** these files:

  - stack.h — Contains the spec and 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.
  - rpn.h – Contains the spec and prototypes for functions to parse Reverse Polish Notation strings. Also includes a list of errors that the RPN parser can return.
  - 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.
  - GNUmakefile — used for compiling this lab on linux.
- **Edit** these files:
  - stack_test.c, rpn_test.c – starter code for your test harnesses. Build them up to produce thorough test harnesses. (they are included as stack_test_template.c and rpn_test_template.c, so rename them).
  - Lab04_main.c – starter code for your main calculator. Expand on the code here to make a calculator that meets the assignment requirements.
- **Create** these files:
  - rpn.c, stack.c – Create these files to implement the functions specified in their respective .h files.

Grading:

There are 19 points that can be awarded in this assignment. It is scored out of 18, so there is 1 point of "cushion."

- 9 points: Automated components:
  - 4 points – Correctly implementing all functions declared in Stack.h
    - 0.5 per function that works when called correctly
    - 1.0 – all functions return errors as appropriate
  - 5 points – Correctly implementing all functions in rpn.c
    - 4 point: Parser behavior
      - 1 – handles minimal RPN expressions
      - 1 – handles complex RPN expressions
      - 1 – handles all 6 rpn errors correctly
      - 1 – Passes all tests (tough!)
    - 1 point: ProcessBackspaces works correctly
- 10 points: Human components:
  - 2 points (1 for each test harness):
    - does thorough tests of each function and gives readable output
  - 3 points: Calculator functionality
    - 2 – User interface (greets, prompts, prints, loops)
    - 2 – Handles all 5 error messages correctly and gracefully
  - 3 points: Code style

        - 2 points: Readability and style guidelines
        - 1 point: Program structure
    - 2 points: README
        - 1 point: Contains a full lab report
        - 1 point: Clear communication and insightful ideas

- Deductions:
    - **NO CREDIT** for sections where required files don't compile
    - -2 points: at least one compilation warning
    - -1 printing in any functions other than main()
    - -1 using magic numbers
    - At grader discretion, other deductions may be made for things not covered in this rubric, such as bad programming practice, extremely unreadable code, etc.
    - It must compile within MPLABX. Code that does not compile within MPLABX will receive no credit.

Assignment requirements:

In general:

- No magic numbers in this lab (not even for Booleans)!
    - Use the constants that are #defined or `enum{}`'d in BOARD.h, stack.h, and rpn.h where appropriate
    - #define or declare your own constants for your `main()` files.
        - One exception: You may use magic numbers for expected return values in your test harnesses.
- All code should be well-formatted and follow the CSE013E_StyleGuidelines doc.
- Even if you don't finish part of a library, at least make sure each file compiles. Make bare-minimum definitions of functions if needed (ie, "`return;`" or "`return 0;`").
- While this lab can be done without the lab kit all parts must compile within MPLABX.

stack.c:

- Implement all of the functions whose prototypes and specifications are in the header file stack.h.
- Use the 'stack' struct defined in stack.h.
- No stack functions should print, scan, or any other UART activity.

stack_test.c:

- Test each function thoroughly.
    - Be sure to test correct-use cases *and* all error cases.

        - Make *at least* two meaningfully different tests per function.
    - Print a clear, readable, concise record of which tests passed and which failed

rpn.c:

- Implement both of the functions whose prototypes and specifications are in the header file stack.h.
    - RPN_Evaluate must parse the user input into a sequence of string tokens. We recommend using `strtok()` from string.h to perform tokenization, though other approaches are possible.
    - RPN_Evaluate should not print to or read from the UART.
    - RPN_Evaluate should use the functions in stack.h
- Note that ProcessBackspaces is only worth 1 point, so you may want to save it until the end.

rpn_test.c:

- Test each function thoroughly.
    - RPN_Evaluate() will require *many* tests to test thoroughly. Tests should grow in complexity, with each test string testing something new.
    - Return `NO_ERROR` when the string is successfully parsed. If called with an invalid RPN string, return the appropriate error as `enum{}`'d in rpn.h.
- Print a clear, readable, concise record of which tests passed and which failed

Lab04_main.c:

- 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, not including a trailing newline or null.
    - Make sure that your calculator handles doubles properly and that all calculations are done with values of type "double". This includes 0.0 and negative numbers.
        - You may use `scanf()`, `fgets()`, or `fgetc()`, according to your preference.
- Return to prompting the user for another RPN string to calculate.
- If an error is encountered, print an appropriate error message and handle it gracefully.
    - There are 5 possible errors: rpn.h defines 6 errors. Additionally, handle the error when the user enters more than 60 characters.

    - Error messages should be more human-readable than the `enum{}`'d constants, helping a naive user to understand the issue.

README.md:

- 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 you 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.

Doing this Lab on Linux:

This lab has multiple parts and instead of calling gcc multiple times we are instead going to start using the make system instead. We will go over how it works in class but the commands needed are below. For these to work the file 'GNUmakefile' needs to be in the same directory as the rest of your files, it is only used when compiling on Linux.

- `$ make`
    - This command will create the final executable named Lab4
- `$ make stack_test`
    - This command will compile your stack test and stack library files into the executable stack_test
- `$ make rpn_test`
    - This command will do the same thing but for your rpn test making an executable named rpn_test

Additionally there is no quit command for this lab, you will need to use 'ctrl-c' to exit your final calculator.

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 an overview in pseudo-code:

```
Main():

Output greeting to the user
while (TRUE):
      Read in characters from stdin until a newline is received
      ---
      ProcessBackspaces()
      RPN_Evaluate()
      ---
```

```
RPN_Evaluate():

Split incoming string into tokens
For each token:
      ---
      if operator:
            pop two elements and push result
            ---
      else if number:
            ---
            push number
---
output result
```

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. Each "`---`" above is a place where it would be useful to check for an error.

String Handling:

In this lab, you will have to work with strings using a level of detail that we have not yet explored in this class. In particular, you will have to perform a few operations that may be new to you.

Tokenization

The central string operation in this lab is called *tokenization*. This means taking a long string that is separated by space characters (or some other *delimiter*) and breaking it into substrings. For example, the string:

`“Hello World, + !5403”`

Is a string with 4 tokens:

- `Hello`
- `World,`
- `+`
- `!5403`

There are various ways to do this, but the string.h function `strtok()` is probably the best.

`strtok()` is called in 2 ways.

- Initialization: To use `strtok()` on a new string, pass it a pointer to the string, along with a delimiter. It will modify the string in place, replacing each delimiter with a null character. It also returns a pointer to the first token, which you already knew (it’s the same pointer that you passed in!)
- Subsequent tokens: To get the next token from a previous string, pass it a null pointer. `strtok()` will return a pointer to the next token. If it already returned the last token in the string, it will return null, signifying that the string has been consumed.

Other useful functions

Here are some other useful string.h functions. 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.

`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.

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. You should implement this functionality yourself within the function `ProcessBackspaces()`.

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, and shift all the remaining characters as appropriate. 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!

WARNING: When compiling this lab on Linux the terminal is in buffered mode and will most likely handle backspaces for you. To test this module you will need to use hard coded strings that include the backspace character.

Unit testing

As you are starting to see, good testing is an essential, component of good code, even though it is invisible to users.

It is challenging to come up with good tests, even for experienced programmers. There are a nearly infinite number of possible inputs to your code, and we mortals cannot imagine the surprises the real world can throw at our programs. In the end, it requires experience, careful thought, lots of work, and some imagination and creativity.

Here are a few pointers:

Edge cases most often occur when limits are reached, such as when things get full, are empty, or when 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.

Try to create each possible error in at least two different ways.

Functions often have side-effects that you won't notice from a single test Try filling up a stack, emptying it out, and seeing if it behaves the same way. Try filling a stack by pushing 20 times, and try filling a stack by setting its index directly.

Try to imagine different *kinds* of RPN strings. Use different orderings. "`1 1 1 1 + + +`" is a different test than "`1 1 + 1 1 + +`". Use variety. "`5 10 /`" is a different test than "`-5.325 10.0 /`". Write the longest RPN expression you can.

Structs

C has a four built-in data types that you should be familiar with at this point: char, int, double, float. 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.[2]

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, although accessing them requires a new notation.

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
```

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

Sometimes, though (particularly in the context of a function) you'll have a pointer to a struct. A different syntax is used in this case:

```
struct Stack *stackPointer = &myStack;

stackPointer->initialized = TRUE;
```

An ampersand (`&`) is used to get the address that is pointing to 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.[3] 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.

[3] Remember from CSE-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

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.

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
		1
+	Operator: pop, pop, calculate, push	5
6	Number: push	6
		5
4	Number: push	4
		6
		5
-	Operator: pop, pop, calculate, push	2
		5
*	Operator: pop, pop, calculate, push	10
8	Number: push	8
		10
/	Operator: pop, pop, calculate, push	1.25

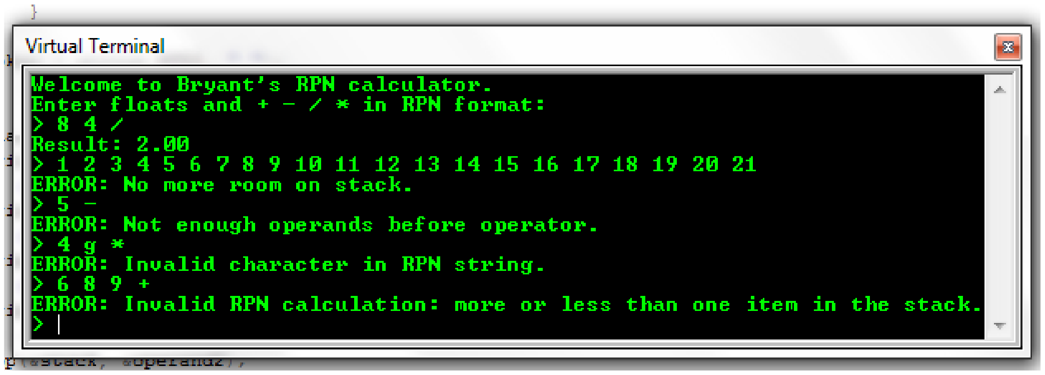
As number tokens are encountered they are pushed onto the stack. Operations pop elements off the stack and push back the result of their calculation. The last element on the stack after all the tokens are handled is the result of the entire expression. There must only be one element on the stack at the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

Example Output:

Here is an example of what it might look like to interact with an RPN calculator. Your interface does not have to match this one exactly:



```
Welcome to Bryant's RPN calculator.
Enter floats and + - / * in RPN format:
> 8 4 /
Result: 2.00
> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
ERROR: No more room on stack.
> 5 -
ERROR: Not enough operands before operator.
> 4 g *
ERROR: Invalid character in RPN string.
> 6 8 9 +
ERROR: Invalid RPN calculation: more or less than one item in the stack.
> |
```

Error handling

In C there is no error handling built-in to the language (unlike, for example, Java, Python, 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.

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 above 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.[4] 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 their value is important, but its importance is not obvious. Using such numbers is inevitable, 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 `SUCCESS` 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.

[4] 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) {
      ;
}
```