Lab 4 – Reverse Polish Notation Calculator

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

- FloatStack.h – Provides the function prototypes for the functions that you will implement in FloatStack.c along with brief descriptions of each function. Add this file to your project directly. You will not be modifying this file at all!
- lab4.c – This file contains main() and all the support code you’ll need. NOTE: User input is now echoed back to the terminal as you will be able to see what you are typing as you type it.

Assignment requirements

- For this lab you are required to create an iterative-design plan in order to receive help with your code. For a reference of what this looks like, refer to the lab manual for labs 0 and 1.
- Create a new file called FloatStack.c that implements all of the functions whose prototypes are in the header file FloatStack.h.
The return values from these functions should use the constants defined in FloatStack.h where appropriate.

Expand main() in lab4.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 doubles properly and that all calculations are done with values of type double. So this includes 0.0 and negative numbers.
  - Your calculator must also handle backspaces in the input string.
- 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 FloatStack”
- 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:

- 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.”

Add the following to the top of your FloatStack.c AND lab4.c files as comments:

- Your name
- The names of colleagues who you have collaborated with
  - WARNING: collaboration != copying

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 NO credit. Compilation warnings will result in lost points.

• Submit exactly FloatStack.c and lab4.c via eCommons before the due date.

**Grading**

This assignment again consists of 10 points:

• 3 points – Correctly implementing all functions declared in FloatStack.h

• 5 points – Correctly implementing RPN calculator functionality

• 1 point – Properly handles backspaces in input string

• 1 point – Correctly handles the four error messages

• Extra Credit: (1 point) – Implement `toString()`
  - This function does not need to be used within your calculator, but it will be tested for correct functionality.

You will lose points for the following:

• -10 points: code doesn’t compile (seriously)

• -2 points: any compilation warnings

• -2 points: the files you submit aren’t named as described in this document or you submit more or less than just the required documents

• -2 points: name and collaborators are not named at the top of all source files submitted

• -2 points: more than 3 lines with style errors

• -1 point: the error and Boolean constants from FloatStack.h weren’t used (TRUE, FALSE, STANDARD_ERROR, etc.)

**Program Flow**

This program follows a very similar outline to the calculator you did for lab 1, but takes a few more steps to get to the user input because we will be parsing a string. Here’s a high level overview.

Output greeting to the user
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. You will need to 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 a 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.

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 C30 Libraries->Standard C Libraries. These functions are always 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.

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. Now 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.

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

```c
struct FloatStack {
    float StackItems[STACK_SIZE];
    int currentItemIndex;
    int 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:

```c
struct FloatStack myStack;
```

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

```c
myStack.initialized = 1;
```

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.

```c
struct FloatStack *stackPointer = &myStack;
stackPointer->initialized = 1;
```

An ampersand (`&`) is used to get a pointer from a variable. You’ve used it a lot with `printf()` and `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.

So 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:

```c
struct FloatStack stackPointer = myStack;
Push(&myStack, 7.0);
```
Stacks

A stack is an abstract concept 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. (we will ignore other things you can do with cards like shuffle or rearrange them because 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 is only so big something can get and the same applies to the memory in a computer. Since it’s 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 

\[
(1 + 4) \times (6 - 4) \div 8
\]

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.

<table>
<thead>
<tr>
<th>Token</th>
<th>Operation</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number: push</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Number: push</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>Operator: pop, pop, calculate, push</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Number: push</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Number: push</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Operator: pop, pop, calculate, push</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>Operator: pop, pop, calculate, push</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>Number: push</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
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 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:

```
Welcome to Bryant’s RPN calculator.
Enter floats and + - / * in RPN format:
>> 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 like with 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.

Now 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. Now 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 doesn’t get it and so it has to shut down.

Now 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 return a value, but to implement error checking it will. 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.

Now this should remind you of the MatrixEquals() function you implemented in the MatrixMath lab, but it’s actually a little different. See functions like this return an int and can be used in Boolean expressions because a 0 evaluates to false and non-zero values evaluate to true (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. Now this may become more clear in the following example where the return values are not 0 or 1 for an error.

A lot of code in use relies on obtaining the size of something. This is done for arrays, abstract data types such as the stack you’re implementing, or many other data types (trees, queue, circular arrays, 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. Now the size of a queue corresponds to how many items are within it with the smallest value being a 0. This means that if an int is the return type of the function all of that negative range is 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).
So this example 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.

So we have a few different ways to return an error status from functions. But what do we do with them? Well this is where you need to think a little and ask if you actually care. Sometimes functions can fail and that checking on whether 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 for 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. So in this case you would care about the return value of scanf().

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

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

What the above code does is continually calls scanf() with the format string until the input would match (scanf() should return 2 if it was successful as we’ve specified that it should be expecting 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. 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 always be defined as constants, like TRUE and FALSE. This makes their meaning clear and also allows for the actual number behind the constant to change without all of its uses to change. A priority in writing code is readability and modularity, which this addresses. Besides TRUE and FALSE, an ERROR constant that is also defined as 0 (along with a corresponding SUCCESS value of 1) would make the code even more readable because it distinguishes between errors and normal Boolean return values for someone reading the code even though the underlying numbers are the same.

We have defined these all for you in FloatStack.h. We have also added an additional constant SIZE_ERROR that evaluates to -1. You are required to use these constants for the appropriate return values.