Lab 3 – 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). You’ll be implementing another library, in this case for a stack, and you will then use it in your calculator. A stack is the fundamental paradigm on which reverse Polish notation is based and it will be very clear why it’s implemented this way.

Required reading

- K&R Chapter 2, 4.5, and 5.1-5.2
- Style guidelines discussed in Lab 0

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!
- lab3.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

- Create a new file called floatStack.c that implements all of the functions whose prototypes are in the header file floatStack.h.
  o The return values from these functions should use the constants defined in floatStack.h where appropriate.
- Expand main() in lab3.c to do the following
  o Great the user once on startup
  o Prompt the user for a long RPN string that includes floats (included 0.0) and the 4 arithmetic operators: + - / *
  ▪ 1 POINT EXTRA CREDIT: Handle backspaces in the input string
  o Process the string into a sequence of string tokens
  o Read in each token and utilize the stack to perform operations as dictated by the RPN syntax
o Return the last element in the stack as the result otherwise alert the user that there was an error as there wasn’t only one item left on the stack after computation
o Return to prompting the user for another RPN string to calculate

- Your calculator must be able to handle and return errors for:
  o RPN strings that don’t return a single result
  o Invalid input characters/tokens (non-operator, non-numeric)
  o When the stack if full or empty and it shouldn’t be
- Add the following to the top of your floatStack.c file as comments:
  o Your name
  o The names of colleagues who you have collaborated with
- 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. Two or more compilation warnings will result in an additional lost point.
- Submit both floatStack.c and lab3.c via eCommons before the due date.

Grading

This assignment again consists of 10 points:
- 3 points - Implementing correctly all functions declared in floatStack.h
- 5 points – Implementing correctly an RPN calculator
- 2 points – Adhering to the style guidelines and variable naming guidelines (this means both variable capitalization and naming)

You will lose points for the following:
- -10 points: code doesn’t compile (seriously)
- -1 point: 2 or more compiler warnings
- -1 point: the files you submit aren’t named as described in this document or you submit more or less than just the required documents
- -1 point: the error and Boolean constants from floatStack.h weren’t used (TRUE, FALSE, STANDARD_ERROR, etc.)
- -1 point: program can’t handle 0.0 as an input

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.

Relevant functions

You will want to use the following functions in your code. It should be fairly apparent where they would be useful.

strtok() – Splits a string into tokens based on the delimiter you passed it. Be sure to read up on any examples because it’s a slightly different function that what you’re used to.

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, it’s a combination of primitive and non-primitive types (but we’re just interested in the primitive types for now). A struct that you will be using looks like the following:

```c
struct floatStack {
    float StackItems[STACK_SIZE];
    int currentItemIndex;
    int initialized;
};
```
Now first off, 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.

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 a struct. 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; // Declared myStack to be a floatStack struct
```

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

```c
myStack.initialized = 1; // Sets the initialized member of myStack to 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; // A pointer to myStack.
stackPointer->initialized = 1; // Sets initialized to 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 (->).

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; // A new stack struct
Push(&myStack, 7.0); // Push 7.0 onto myStack
```

---

**Stacks**

A stack is a datatype that 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’ll 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 big it can get. When working in the real world there is always something 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. It will also have a certain size that it is currently at.

Now that is the basic of a stack. 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.

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>Operator: pop, pop, calculate, push</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Number: push</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Number: push</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Operator: pop, pop, calculate, push</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Number: push</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Operator: pop, pop, calculate, push</td>
<td>1.25</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.

**Error handling**
In C there is no error handling built-in to the language like Java or C++. While this may lead you to think there’s 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 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 that other code then has a chance 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 handled by altering the return value. Sometimes a function only has a return value because it sets it depending on if it encountered an error. An example of this is an initialization function (like the Init() you will implement). Generally an initialization function would return any real value, but it can fail. 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 just 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 statements because a 0 evaluates to false and non-zero values evaluate to true (generally chosen as 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 or abstract data types such as the stack you’re implementing or the various others (trees, queue, circular arrays, etc.). A size attribute is a very fundamental part of a lot of these more complicated data types. But what if you have a Size() function that returns the size of something, let’s say a queue and it can encounter 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 that it could set, but those are getting complicated and adding a lot more code and code complexity. 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 for 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 our sample \texttt{Size()} function that we'd use on the queue we're imagining 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 \texttt{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 fix it or handle it appropriately and so checking for if an error occurs wouldn't be worth it. A counter-example to this would be \texttt{scanf()}. This function returns how many string tokens it successfully captured according to the format string passed to it. Now if your code is expecting two integers from the user and doesn’t get them then that will most likely present a problem later on. And this error could be easily handled by checking to see if \texttt{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 \texttt{scanf()}.

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

\begin{verbatim}
while (scanf("%d-%d", &int_1, &int_2) != 2);
\end{verbatim}

What the above code does is continually keeps calling \texttt{scanf()} with the format string until the input would match it (\texttt{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 “\texttt{while (1);}” at the end of \texttt{main()} in all of the earlier labs).

There is one last thing about return values that can make the code more readable: using \texttt{TRUE} and \texttt{FALSE}. Now since you’re familiar with the variable naming scheme we’ve adopted in this class (and is common for C in general) you should recognize these as being defined constants. What you can do is define \texttt{TRUE} as 1 and \texttt{FALSE} as 0 and then use those for your return values. You could even add an \texttt{ERROR} constant that is also defined as 0 (along with a corresponding \texttt{SUCCESS} value of 1) that would make the code even more readable because it distinguishes between errors and normal Boolean return values for someone reading the code.

We have defined these all for you in \texttt{floatStack.h}. We have also added an additional constant \texttt{SIZE\_ERROR} that evaluates to -1. You are required to use these constants for the appropriate return values.