Assignment 5 – Calc Report Template

Piper Stickler
CSE 13S – Spring 24

Purpose

This program will perform basic scientific calculations. It is a calculator with commands for add, subtract, multiply, divide, sine, cosine, tangent, absolute value, and square roots. The user can enter their expression in reverse polish notation, and it will print out the result. Using reverse polish notation means that the user will not need to use parentheses or yield to order of operations rules.
Edit 05/15/2024: I didn't realize until starting my program, but calc also allows the user to choose between the standard library sin, cos, and tan functions or my own functions based on the Maclaurin approximation.

Questions

• Are there any cases where our sin or cosine formulae can't be used? How can we avoid this?
If the value of x in the MacLaurin series causes the terms to increase as n increases, there will be no way to reach a small epsilon absolute value in order to make an approximation. The solution to this is to normalize input to be between 0 and 2 pi so that the x value is always within a range that can be calculated. Because the sine function repeats its pattern over and over this would not affect the actual answer.

• What ways (other than changing epsilon) can we use to make our results more accurate?
Epsilon works well for simple computations where the input values generally fall within a certain range. In order to deal with more complex math and varying inputs, we can make use of the Taylor Series to approximate the terms needed. This will depend on how precise you want the number to be. The convergence of the Taylor series for the approximation of the sine or cosine being computed can be used to evaluate the precision with a given number of terms.
Edit 05/15/2004: this is essentially the same as using Epsilon, I realize now.

• What does it mean to normalize input? What would happen if you didn't?
Normalizing input means to make sure the user's input is in a consistent format that the program is set up to deal with, before actually performing the computations. This is done to ensure accuracy and consistency from the results of the functions' calculations. If you didn't normalize input you could end up with different results from the same function with the same inputs, which means that one of the two is incorrect. You don't want the user's input format to affect their results. By making sure the input is set up for the program to handle, you also prevent runtime errors and exceptions.

• How would you handle the expression 321+? What should the output be? How can we make this a more understandable RPN expression?
This expression is trying to add 3+2+1. The output will be 3 because the program pushes 3, then 2, then 1 to the stack and then pops the most recent two values, 1 and 2, to perform the operation. To make this a more readable RPN expression, you can write it in more detailed steps, like “3 2 + 1 +”.

• Does RPN need parenthesis? Why or why not?
RPN does not need parentheses because the operators are always assigned to the numbers directly preceding them. The program processes each expression one at a time, so there is no way to mix up values.

Testing

I need to test each of my individual functions and the main program.
Sine function
Cosine function:
Tangent function:
apply_unary_operator():
apply_binary_operator():
parse_double():

Calc.c:

Edit 05/15/2024: I left this blank initially because I wasn’t clear enough on my own program structure to know what tests to do. I tested things kind of in the order from the smallest pieces toward the overall program. First I tested the unary and binary functions that would actually do computations on two numbers. Then I tested the apply_operator functions by making sure they performed the correct computation for any given operator and returned what they should. Then I tested my calc.c program with a range of inputs, valid and invalid.

How to Use the Program

To use this program, the user must be familiar with reverse polish notation. This is where you enter the operands(the numbers) before the operator that is to be applied. For example, instead of typing in 1+2, you would type in 1 2 +. This format has a lot of benefits for the implementation of the program. The user only needs to know the order in which to type the numbers and the operators. The add, subtract, multiply, divide, and modulus operators are as usual +, -, *, /, and %. The other operators are as follows: sine - s, cosine - c, tangent - t, absolute value - a, and

square root - r. To run the program enter the command ./calc <your input operands and operators in RPN>.
Edit 05/15/2024: I realize reading this back it sounds like you are supposed to enter your input before you press enter to run the program. This is incorrect. You must start the program, which will cause an '<' to appear, prompting you to enter your input.

Program Design

This program is making use of the stack ADT to perform operations that are entered by the user in reverse polish notation. To normalize the input, we make sure that the user input is read one line at a time, and then each line is separated into words space-separated. There is a function for each of the operations that calc.c is able to perform. The file mathlib.c contains the functions for sine, cosine, tangent, absolute value, and square root. The file operators.c contains functions that handle the application of unary and binary operators, as well as a function that parses the string of input and separates it into floating point doubles that can be used in the operator functions. operators.c also contains the simple functions for add, subtract, multiply, and divide. They are in functions here so that they can be accessed via function pointers stored in an array. There are three arrays in operators.h for function pointers to unary operators using our functions, unary operators from mathlib.h, and binary operators. These can be used to directly access the functions using the operator provided as a way to index the operator array and call the correct operator function. The file stack.c contains functions that are used to perform stack operations. This includes stack_push, stack_peek, stack_pop, stack_clear, and stack_print.

Pseudocode

calc.c main program:
//include header files
//define the maximum input length of 1024 bytes
//define boolean variable error to use as a condition later

Main function:
    //Define input buffer with max input length
    //Print prompt char ">" to stdout
    //while loop to read and process input. Condition: if input is not NULL
        //remove trailing newline character
        //tokenize input(using strtok_r template given in asgn5 pdf)

        //while loop to loop through tokens. Condition: token is not NULL, and error is false
            //define variable num that will hold the location of the parsed number

//if statement: use return of parse_double(token, &num) to check for valid number input
//if valid, push the double float to the global stack
//else: the token is an operator
//if statement for add operation(use parse_double() to check). Call apply_binary_operator with the operator add function pointer as its argument
//else if statements for the remaining 7 operators. They will call the function that applies to that operation(unary ours, unary library, or binary) with the function pointer to that operation as the argument
//final else statement: if no other condition has been met than the token is unknown.
//print error message to stderr
//update error variable to be true
//outside of the outermost if statement, use strtok_r to get the next token
//outside of innermost while loop, after the input line has been processed call stack_print()
//call stack_clear to clear for the next input cycle
//print the prompt char “>” for the next input
//reset error to false
//outside of outermost while loop, return 0

Edit 05/15/2024: I changed my main program from this design a lot. I used argc and argv to check the command line and respond accordingly. First I checked for options that necessitated exiting the program, or invalid options in the command line and wrote conditional statements to handle these. Then, I used a while loop to continue taking in, processing, and returning results. Within this while loop I printed the prompt character < to stderr and used fgets to take in user input. Then I used strtok to parse the input and call the correct functions to do work on the numbers. This while loop will continue until strtok reaches a null character.

Function Descriptions

operators.c:

```
apply_unary_operator(unary_operator_fn op) {
    //from the asgn5 document
    //check that the stack is not empty, if yes return false;
    //declare a double int
    //use assert() and stack_pop() to make sure the stack is working before doing operation
    //perform op(x). Op called based on user input. Assign to a double int called result
    //assert(stack_push(result)). Adds result to stack and makes sure it is not overflowing
    //return true
```

```
}

apply_binary_operator(binary_operator_fn op); {
        //make sure stack isn't empty, else return false
        //declare two double ints
        //pop the top two numbers off the stack
        //assign the result of the op() to a variable result
        //push the result to the stack and return true
}
operator_add(double lhs, double rhs); {
        //return lhs+rhs
}
operator_sub(double lhs, double rhs); {
        //return lhs-rhs
}
operator_mul(double lhs, double rhs); {
        //return lhs*rhs
}
operator_div(double lhs, double rhs); {
        //return lhs/rhs
}
parse_double(const char *s, double *d); {
        //from asgn5 document
}
```
Edit 05/15/2024: My operator functions worked as intended for the most part. My actual program didn't vary much from these descriptions.

mathlib.c:

```
Double Abs(double x); {
        //if x is <0 return -x
        //if x is >0 return x
}

Double Sqrt(double x); {
        //from the asgn5 specification document
}
```
Edit 05/15/2024: I ended up having bugs with this simple implementation. I researched other ways and used the Babylonian method to compute Sqrt. I iterated through computing Abs(num_a

- num_b), starting with 0.0 and 1.0 respectively, and continued updating the value of these two until their difference was negligible.

```
Double Tan(double x); {
        //declare double int for the sum. Initialize to x(first term)
        //define int current term to be added to sum
        //double int numerator of the next term
        //double int denominator initialized to 1.0 of next term
        //double int sign of next term initialized to 1.0
}
```

Edit 05/15/2024: My implementation of tangent ended up being much simpler than this. I simply called the Sin and Cos functions, and applied them to the operand and saved these two values, then divided them as in the tangent trig equation.

```
Double Sin(double x); {
        //first normalize the input. If  x>2 pi subtract 2 pi until within range. If x<0 add 2 pi until x>0
        //define variables for sum, current term, and the next numerator, denominator, and sign
        //for loop to iterate through turns {
                //calculate numerator and denominator of next term
                //calculate the next term and assign value to term variable
                //add the completed turn to the sum and update sum variable
                //change the sign for the next term
        }
//Return the sum
}

Double cos(double x); {
        //normalize input same way as in sine function
        //declare variables same as in sine function
        //for loop iterates through calculations from maclaurin series approximation
        //return the sum
}
```

stack.c:

```
stack_push(double item); {
        //make sure the stack is not full. If it is return false
        //push the item to the top of the stack, use stack size to keep track
        //increment stack size
```

```
}
stack_pop(double *item); {
        //make sure the stack isn't empty, if yes return false
        //retrieve item from top of stack, account for null char
        //decrement stack size
        //return true
}
stack_peek(double *item); {
        //make sure stack isn't empty, if yes return false
        //put the top item of the stack into the location pointed to by item pointer
        //return true
}
stack_clear(void); {
        //set stack size to zero
}
stack_print(void); {
        //given in asgn5 document
}
```

Edit 05/15/2024: Similar to operators.c, all the functions in stack.c worked out as intended and detailed above.