Calculator Design

Purpose

The purpose of this program is to create a simple version of a calculator in C. Some calculations will be easy to calculate, while others will require approximations. This will allow addition, subtraction, multiplication, and division, as well as sin, cos, tan, and absolute value.

Questions:

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

Are there any cases where our sin or cosine formulae can't be used? How can we avoid this?

We can’t use sin or cosine if there aren’t any elements in the stack. We can avoid this by checking to make sure that the stack size is at least 1 before calling sin or cosine.

What ways (other than changing epsilon) can we use to make our results more accurate?

Because of trig identities, we know that tan = sin/cos, so we don’t have to do a separate taylor series for tan, but rather simply do sin/cos because those functions are already designed.

What does it mean to normalize input? What would happen if you didn't?

Normalizing input means to rewrite it in a way that is easier to handle. For sin and cos, we can normalize it to the range (0, 2pi). If we didn't do this, our taylor series approximations would not be as accurate because we centered them around 0, and so taking the sin of 100pi would not be as accurate.

How would you handle the expression 321+? What should the output be? How can we make this a more understandable RPN expression?

It looks like there's no space in that expression, so the output should just be 321. To make it more understandable, we just need to add spaces between the numbers and operators.

Does RPN need parenthesis? Why or why not?

RPN does not need parentheses because it operates on a stack. The first operator to be used is the first one it is given, and it will always use the top or top two items in the stack. Instead of using PEMDAS, the order of operations depends on the order of our expression.

Testing

I will have to test my code in two different ways. First, I will test individual functionality of the helper functions in tests.c. This begins with including all of my own .h files. To test mathlib.c, I can #include <math.h> and compare my results with those from math.h with varying inputs for absolute value, sin, cos, and tan. Next, I will test stack.c. I will push some elements to the stack, and print them out. Then I will peek the top one and make sure it works. Then I will

pop an item off the stack and make sure it gets removed and put into the proper location. Then I will make sure that clearing it actually clears it by printing it again after. Lastly I must test operators.c. I will begin by testing parse_double by creating a variety of strings that may contain doubles, then seeing if it actually parses them into a double variable through the pointer to that variable that I pass in. Next, I will add a few numbers to the stack, and test the apply_binary_operator function by passing in a +, -, *, or /, and checking if it matches the expected output. I will then do the same for apply_unary_operator with sin, cos, tan, absolute value, and square root, and checking them with what is given by math.h. If all that is good, I will return 0.

The next thing to test is my actual calc program. I will create a tests directory and make many .sh test scripts. These will test many cases, such as the command-line flags, invalid input characters, not enough numbers in the stack to perform operations, as well as all the binary and unary operations. I will compare them with the binary reference given, and save each into an output file, then check to make sure there is no difference between them.

How to Use the Program

To use this program, you first need to have calc.c on your computer, as well as all of the helper and header files in this assignment. Use the make command on your terminal to compile and link it, then run calc using ./calc, followed by any optional flags. The optional flags are -h for help, or -m to use the trig functions from the C library instead of my own code. After that, you will be greeted with a “> “ character and you can begin typing what you would like to calculate. You will be using Reverse Polish Notation, meaning that if you want the equation 1+1, then you

would type 1 1 + into the terminal. Each time you enter a number, it gets added to the stack, and each time you enter an operator, it takes either the top or the top 2 numbers to perform that operation. For sin, use s, for cos, use c, for tan, use t, and for absolute value, use a. After you push enter, the program will print all the numbers remaining in the stack. Once you do a calculation, you will be able to do another, and as many as you want. If you want to exit the program, simply push control d.

Program Design

This program is split up into many different files to help it run more smoothly. There is a file called stack.c, with stack.h being its header, which contains information and functions relating to the stack. The variables tracking stack size and capacity are there, and so are the functions to add to the stack, pop from it, peek at the first element, clear it, and print out all the elements in it.

There is another file called operators.c, with operators.h being its header, which is responsible for accessing the stack and performing some given operations to it. Binary operations require two parameters, which will be accessed from the top of the stack. Apply_binary_operator takes in a pointer to a function that takes in two doubles and returns a double. Apply_binary_operator will perform this given function with the top two elements in the stack if they exist. These functions include add, subtract, multiply, divide, and mod. Apply_unary_operator takes in a pointer to a function that takes in a double and returns a double. It will do this given function with the top element in the stack, if it exists. The functions are in mathlib.c, and include absolute value, square root, sin, cos, and tan. These functions use

taylor series to approximate the answer unless the user wants to use the functions in the standard library, which can be done using the -m flag during the program call. The calc.c file will be responsible for putting everything together, and is the file that contains main. It uses getopt() to check for command-line flags, which include -m to use the trig functions of the standard library, and -h for a help menu. It then enters an infinite loop, and uses fgets() to get an expression from the user. Then it uses strtok_r to parse it with " " character as a delimiter. Until we reach the end of the expression, it evaluates each term given. If it is an integer, it adds it to the stack, assuming it is not at capacity. If it is a binary operator, it calls the apply_binary_operator function with that input, assuming there are at least two elements in the stack. If it is a unary operator, it calls the apply_unary_operator function with that input, assuming there is at least one element in the stack. If the term is not a number or an operator, an error message is displayed. All the error messages are kept in messages.h. Finally, if no errors are found, we print the stack. Then, we clear it. This repeats until the user hits Control+D.

Function Descriptions

Stack_push takes in a double called item and returns a boolean of whether adding it to the stack was successful or not. If the stack size is already at capacity, it returns false. If not, then it adds the double to the stack, increases the stack size by 1, and returns true. Stack_peek takes in a pointer to a double called item and returns a boolean. If the stack is empty, it returns false. Else, it sets the double that we want to store the peeked value into to the value of the top of stack, then returns true. Stack_pop takes in a pointer to a double and returns a boolean. The

implementation is the same as stack_peek, except that it decreases the size of the stack by 1. Stack_clear has no parameters and a void return type, and simply sets the stack size to 0. Stack_print has no parameters and a void return type. It returns void if the stack size is 0. Otherwise, it prints the first item in the stack, then for every subsequent item until stack_size is reached, it prints a space then the item. These are all printed to 10 decimal places.

Moving onto mathlib, the Abs function takes in a double and returns a double, which is the absolute value of it. If the number is greater than 0, it returns that number. Else, it returns 0 - that number. The square root function takes in a double, and returns a double, which is the square root of the number. It first checks if the given double is less than 0, in which case it will return nan. Otherwise, it initializes two local variables old and new, with values 0.0 and 1.0 respectively. Then, while the absolute value of the difference between the two is still greater than epsilon, we enter a loop. Old gets set to new, and new is set to 0.5 * (old + (x/old)). This keeps getting closer and closer to the square root of the given double, and once the new - old is within the epsilon value, we return the new. The Sin function takes in a double x and returns a double. It first normalizes x by adding or subtracting 2pi until it is in the range of 0 to 2pi. It sets two local variables, new and old, to 0.0 and x, respectively. While the absolute value of the difference between the two doubles is less than epsilon, we enter a loop. Old gets set to new. There are a few other variables to help with the approximation. The taylor series of sin(x) is x - x^3/3! + x^5/5! - x^7/7!..... The variable exp keeps track of the term with the exponent. It is initialized to x, and each iteration, it is multiplied by x^2. The variable fact keeps track of the factorial term in the denominator. It is initialized to 1.0 and multiplied by the next two numbers because the factorial term keeps increasing by 2. Finally, we have a counter to keep track of

whether we are adding or subtracting from old. Counter is incremented by one at the end of each loop, and if it is even, new = old - (exp/fact). If it is odd, new = old + (exp/fact). Once we exit the loop, we return new. Cos is very similar in that it takes in a double x and returns a double. Again, we must normalize x to be in the 0 to 2pi range. The local variables old and new get set to 0.0 and 1.0 respectively. Old once again gets set to new. The taylor series for cos(x) is 1 - x^2/2! + x^4/4! + x^6/6!. It is similar to sin(x). Exp gets set to 1 to begin with, and each loop is multiplied by x^2, so that the degree value remains even. Fact begins at 1, aka 0!, and each time is multiplied by the next two numbers to get the proper factorial. Once again, we will have a counter that is incremented at the end of each loop. If it is even, new = old - (exp/fact). If it is odd, new = old + (exp/fact). Finally, after the loop, return new. Tan is the easiest one yet. It takes in a double x and returns a double. Since we already made Sin and Cos, Tan will simply be Sin(x)/Cos(x).

In operators.c, there are 4 basic arithmetic functions. Operator_add takes in two doubles and returns their sum. Operator_sub takes in two doubles and returns their difference, the first minus the second. Operator_mul takes in two doubles and returns their product. Operator_div takes in two doubles and returns their quotient, the first divided by the second. Parse_double takes in a string and a pointer to a double variable, and returns a bool of whether it was able to parse a double from the expression or not. It creates a local pointer to a char called endptr. It then uses the strtod function to attempt to parse a double from s, and save it the memory address of endptr. If the endptr address is not the same as s, aka it successfully found a double to parse, then it sets the double to the return result of the strtod function call and returns true. Otherwise, it returns false. The apply_binary_operator function takes in a function that takes in

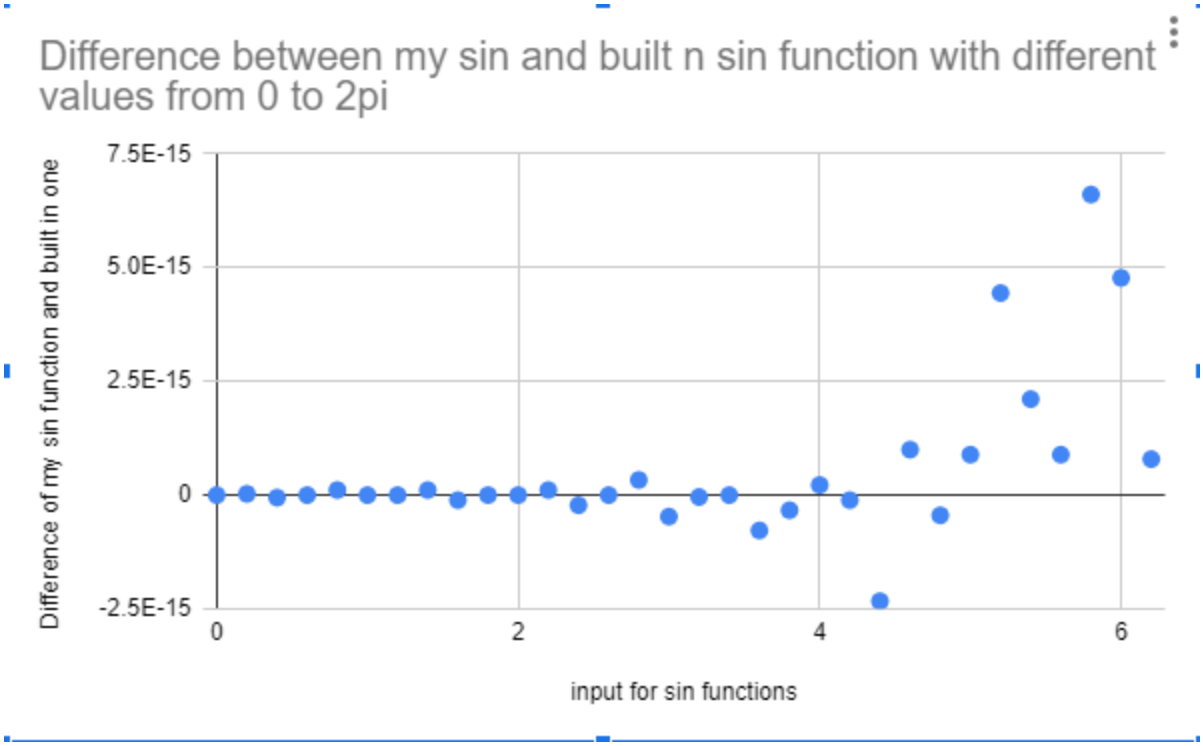
two doubles and returns a double. It first makes sure that the stack size is at least two, and if it is not, it prints the ERROR_BINARY_OPERATOR message found in messages.h and returns false. Otherwise, it creates two doubles a and b. It pops two elements in the stack and saves them to the memory address of a and b. It then calls the function given as a parameter with the values b and a respectively, and pushes the result back into the stack, then returns true. Apply_unary_operator is very similar. It takes in a pointer to a function that takes in a double and returns a double. If the stack is empty, it prints the ERROR_UNARY_OPERATOR found in messages.h, and returns false. Otherwise, it creates a double x, and pops an element from the stack and saves it into the memory address of x. It then does the given operation on the double x and pushes it back into the stack. Finally, it returns true.

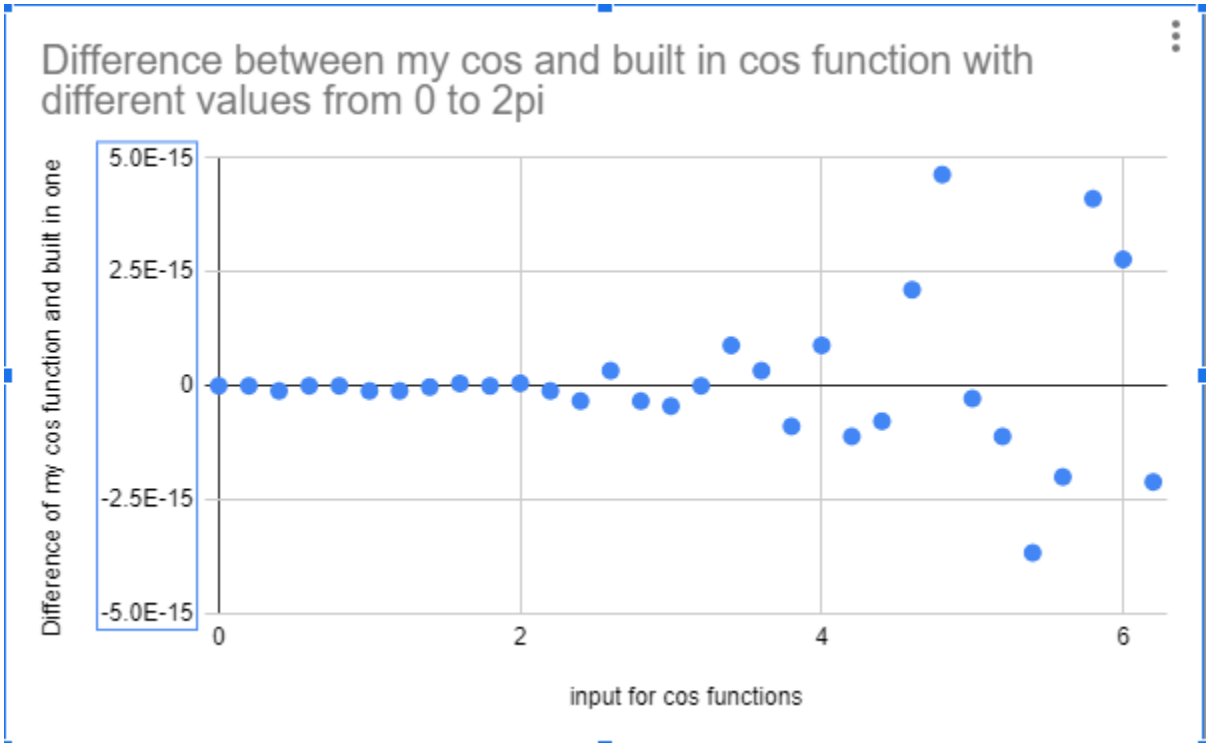
The calc program is what contains the main function, and begins by using the getopt() function to check for command-line flags. There is a variable called trigf that is initialized to 0 to begin with. This keeps track of whether we should use my own trig functions or the ones contained in the C Standard Library. If a -m is given, then trigf=1. If -h is given, then the USAGE message from messages.h is printed, and 0 is returned. If a flag is given but it is neither -m or -h, then we print an error message to stderr telling the user that an invalid option was given, and the program returns a nonzero exit code. Then we can enter an infinite while loop, and use fgets to read from stdin and save the result to a string called expr. The '\0' must be removed at the end. Then we create a string called token that is equal to the result of the strtok_r function, which parses the string expr with " " as a delimiter, and saves the result to the memory address of a string called saveptr so that when we call strtok_r again, we can continue where we left off. Then,

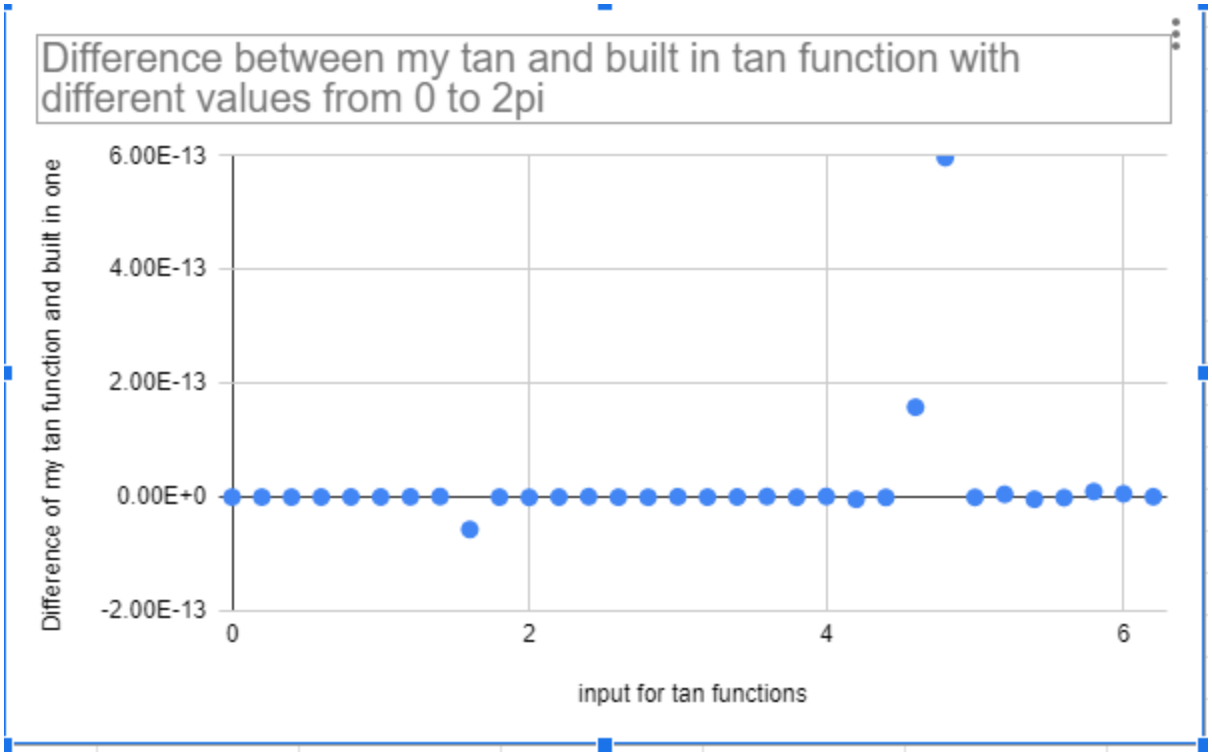
while the token exists, we can use the parse_double function to check if it is a double by passing in a token and the memory address of a double variable called parse_double_slot. If the stack is already full, it prints an error message. Otherwise, it pushes parse_double_slot into the stack. If the token is not a double, and is one character long, the program checks if it is an operator. If it is in binary operators, it calls the apply_binary_operator function with the given token. If it is in unary operators, it calls the apply_unary_operators function with the given token. For unary operators, we must check the value of trigf to know whether to use my own trig functions or that of the standard C Library. If none of these is the case but the token still exists, we inform the user that they have inputted a bad char or a bad string by using the error messages in messages.h, and clear the stack. If there were no errors, we can print the stack, clear it, and print a newline character. This should repeat until the user exits with Control+D.

Results

My program works as intended. My outputs match the binary reference, and the code passes the pipeline entirely. Now of course, there is a small error from the taylor approximations when compared to the actual results of trig functions. I have made 3 graphs to display this. On the x-axis, there are certain values between 0 and 2pi, and I plot a point every 0.2 units. The y-axis contains the difference between my trig functions and the ones from the C standard library. Check them out!


Difference between my sin and built n sin function with different values from 0 to 2pi
Difference of my sin function and built in one
7.5E-15
5.0E-15
2.5E-15
0
-2.5E-15
0
2
4
6
input for sin functions


Difference between my cos and built in cos function with different values from 0 to 2pi
Difference of my cos function and built in one
5.0E-15
2.5E-15
0
-2.5E-15
-5.0E-15
0
2
4
6
input for cos functions



The difference tends to increase the further away we get from 0, since that is what we centered the taylor approximations around, but is very small and negligible for most applications.

Thank you for taking the time to read this.

Happy coding!