Assignment 3 – Calculator

Johann Lee

CSE 13S – Fall 2023

Purpose

The purpose of this program is to emulate a scientific calculator that makes use of the ¡math.h¿. This calculator will use Reverse Polish Notation and will be able to calculate trigonometric functions such as cos, sin, and tangent.

How to Use the Program

This program will consist of a Makefile that was created by the writer and will require the command ”make calc” to compile and ”./calc” to open up the calculator program. Some command line prompts while typing ”./calc” are flags -m and -h. ”./calc -m” will swap the trigonometric functions to follow the ones in the math library instead of the custom-written trig functions. ”./calc -h” will show all pre-execute commands which for this project are only -m and -h as mentioned before. The calculator requires all inputs to be in the form of Reverse Polish Notation meaning that numbers must be given and followed by an operation and the program reads from left to right until it reaches an operation. An example of this would be 2 + 2 which in Reverse Polish Notation would be 2 2 +. The operations signs for this calculator are as follows, ”+ - * / s c t”. The operations are for the most point self- explanatory with the trig functions being represented by their first letter. Any incompatible inputs will return their respective error message. To exit the program Ctrl+D or Ctrl+C can be used.

Program Design

This program consists of 11 files, 5 of which should not be edited. The files that can be edited are as listed:

- Makefile: Contains all the rules to compile and create an executable
- calc.c: Contains the main program which holds all the other files together
- tests.c: Contains a main function to test other code
- mathlib.c: Contains all required math functions using approximations
- operators.c: Contain basic operations in their wrapper functions
- stack.c: Contains implementation of stack ADT

Functions

The functions within these files and their descriptions are as follows:

- double Abs(double x)
  - This function accepts a double and returns the absolute value of the double x.

```
double abs (double x)
    multiply x onto itself
    square root the product
    return square root
```

- double Sqrt(double x)
  - This function returns the square root of the double x.
  - The code for this is given to us in the asgn3.pdf and is as follows

```
double Sqrt(double x) {
    // Check domain.
    if (x < 0) {
        return nan("nan");
    }

    double old = 0.0;
    double new = 1.0;

    while (Abs(old - new) > EPSILON) {
        // Specifically, this is the Babylonian method--a simplification of
        // Newton's method possible only for Sqrt(x).
        old = new;
        new = 0.5 * (old + (x / old));
        }

    return new;

    }
```

- double Sin(double x)
  - This function returns the sine of the input double x which should be given in radians. This will not use the sin from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- double Cos(double x)
  - This function returns the cosine of the input double x which should be given in radians. This will not use the cos from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- double Tan(double x)
  - This function returns the tangent of the input double x which should be given in radians. This will not use the sin, cos, or tan from the math library and outputs a normalized angle between zero and 2pi.
  - The pseudo code for this function is located in the Algorithms section
- bool apply_binary_operator(binary_operator_fn op)
  - This functions takes in an operator and accesses the global stack. It then pops the first 2 elements on the stack, and calls the op function with those two (first element is right side, second is left). It then pushes the result of the op on these elements. If there are no two elements then it returns false.

```
bool apply_binary_op (binary_op_fn op)
    if stack has less than 2 elements
        return false
    else
        right hand = pop element
        left hand = pop element
        op with right hand and left hand
        push op result to stack
        return true
```

- bool apply_unary_operator(unary_operator_fn op)
  - This function takes in an operator and accesses the global stack. It pops element then applies the the operation before pushing back to the stack. If there are no elements to pop then returns false.
  - The code for this function is given in asgn3.pdf
- double operator_add(double lhs, double rhs)
  - Takes the sum of doubles lhs and rhs and returns it.

```
add function
    variable = lhs + rhs
    return variable
```

- double operator_sub(double lhs, double rhs)
  - Takes the difference of doubles lhs and rhs and returns it.

```
add function
    variable = lhs - rhs
    return variable
```

- double operator_mul(double lhs, double rhs)
  - Takes the product of doubles lhs and rhs and returns it.

```
add function
    variable = lhs * rhs
    return variable
```

- double operator_div(double lhs, double rhs)
  - Takes the quotient of doubles lhs and rhs and returns it.

```
add function
    variable = lhs / rhs
    return variable
```

- bool parse_double(const char *s, double *d)
  - This function attempts to parse a double-precision floating point number from string s. If it is successful, it stores the number in the location pointed by d and returns true. If the string is not a valid number, it returns false.
  - This function is given to us in asgn3.pdf
- bool stack_push(double item)

  - Pushes an item to the top of the stack and updates the stack_size. Returns true if it was possible else if the stack is at capacity it returns false.

```
stack push
    if stack size >= stack capacity
        return false
    else
        push item to top of stack
        add 1 to stack size
```

- stack_peek(double *item)
  - Copies the first item of stack to address pointed by item. Returns false if stack is empty.

```
stack peek
    if stack size is 0
        return false
    else
        pop stack and copy to item
        push back to stack
        return true
```

- stack_pop(double *item)
  - Stores the first item of the stack in memory pointed at item. If the stack is empty it does not modify item and returns false. Updates stack_size if pop goes through.

```
stack pop
    if stack size =0
        return false
    else
        remove top of stack and copy to item
        subtract 1 to stack size
        return true
```

- stack_clear(void)
  - Sets the size of the stack to zero

```
stack clear
    stack size = 0
```

- stack_print(void)
  - This function prints every element in stack to stdout and is separated by only spaces. There is no newline after. If the stack is 0 it returns a blank. If not, it prints each each element of the stack with a for loop. The code is given in asgn3.pdf.

Algorithms

The algorithms section are meant to describe the the functions for sin, cos, tan, and sqrt.

- Sine
  - Sine utilizes a method that uses the Maclaurin series in order to calculate itself. It can be best split into portions of the summation, above the fraction, below the fraction and the x variable. The given equation for sin can be described as sin(x)=$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$.

- The summation can be described as a while loop that goes on n amount of times until our next factor of the summation is less than the given epsilon. Everything else is placed within this while loop.
- The portion above the fraction is merely be replaced by a modulo operator placed on n and returns 1 if n mod 2 is 0 and -1 if n mod 2 is 1.
- The portion below the fraction is finding 2n+1 then creating a for loop that mimics a factorial multiplying and adding from 1 until it reaches 2n+1
- The final portion is once again using 2n+1 then creating a for loop where x multiplies itself until the for loop reaches 2n+1
- The summation of these are what represents sin using a Maclaurin series. The pseudocode is below:

```
sin (x)
    float z is final answer
    int n = 0


    while (will always run)
        float a is top of fraction
        float b is bottom of fraction = 1
        float c is the x variable = x
        float d is current

        if n mod 2 is 0
            a = 1
        if n mod 2 is 1
            a = -1

        for i until 2n + 1
            b = b * i

        for i until 2n + 1
            c = c * x

        d is (a/b) * c

        if d is less than epsilon
            break
        z = z + d
        n increases by 1

    return z
```

Cosine

- Sine utilizes a method that uses the Maclaurin series in order to calculate itself. It can be best split into portions of the summation, above the fraction, below the fraction and the x variable. The given equation for cos can be described as cos(x)=$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$.
    - The summation can be described as a while loop that goes on n amount of times until our next factor of the summation is less than the given epsilon. Everything else is placed within this while loop.

- The portion above the fraction is merely be replaced by a modulo operator placed on n and returns 1 if n mod 2 is 0 and -1 if n mod 2 is 1.
- The portion below the fraction is finding 2n+1 then creating a for loop that mimics a factorial multiplying and adding from 1 until it reaches 2n
- The final portion is once again using 2n then creating a for loop where x multiplies itself until the for loop reaches 2n
- The summation of these are what represents cos using a Maclaurin series. The pseudocode is below:

```
sin (x)
    float z is final answer
    int n = 0


    while (will always run)
        float a is top of fraction
        float b is bottom of fraction = 1
        float c is the x variable = x
        float d is current

        if n mod 2 is 0
            a = 1
        if n mod 2 is 1
            a = -1

        for i until 2n
            b = b * i

        for i until 2n
            c = c * x

        d is (a/b) * c

        if d is less than epsilon
            break
        z = z + d
        n increases by 1

    return z
```

Tangent

- Tangent is much more simpler as it just requires our sine function and cosine function from before and creating a quotient between them with sine being the numerator and cosine being the denominator.

```
Tan (x)
    variable = sin(x) / cos(x)
    return variable
```

Square root

- Square root is a given function. It is shown earlier and explaining it begins with the first if statement. This statement checks if the given input is less than 0 which it return nan or "undefined" which makes sense if square rooting a negative number. The next portion is running a reverse squared function through a while loop until it is the absolute value of the old value - new value is less than the given epsilon.

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- The inputs of every function (even if it's not a parameter)
- The outputs of every function (even if it's not the return value)
- The purpose of each function, a brief description about a sentence long.
- For more complicated functions, include psuedocode that describes how the function works
- For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge.

Results

Audience: Write this section for the graders. If you completed only part of the assignment, explain that here.

To write this section, use your code according to its intended purpose. Does it successfully achieve everything it should? Is anything lacking? Could anything be improved? Talk about all of that here, and use your code's output to prove it.

References