Assignment 3 – Calculator

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is to implement a calculator. The calculator takes input in Reverse Polish Notation. This means that the program takes numerical values first, followed by an operator.

How to Use the Program

To use this program, there are several files that must be included in a single directory: `Makefile`, `mathlib.h`, `messages.h`, `operators.h`, `stack.h`, `calc.c`, `mathlib.c`, `operators.c`, and `stack.c`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then, to compile the program, use command `make all`. After the program successfully compiles, use `./calc` to run the program using custom trig function implementations. The program should begin running and wait for user input. To use the calculator, enter an expression in Reverse Polish Notation and press enter. Valid operations are as followed: (s) sin, c (cos), t (tan), a (absolute value), r (square root), + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). The program will print the answer and then wait for input again. To end the program press `^c` or `^d`.

The program has three different command line argument options:

- `./calc -h`: This prints instructions to the screen. These instructions describe the usage of the program. It shows what the different command line arguments do.
- `./calc -m`: This makes the program use libm trig function implementations instead of custom ones.
- `./calc`: When given no arguments, the program uses custom trig function implementations.

The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.
- `-lm`: This flag links the math.h library. This allows the program to access and use the functions from the math.h library.

Program Design

The program begins by checking the command line arguments. The options are listed in the How To Use the Program section. If an in correct argument is received, the program prints an error message and usage message. The program sets a pointer to a constant variable of type `unary_operator_fn` to either `my_unary_operators` or to `libm_unary_operators`, depending on whether the command line argument `-m` was inputted or not. After this, the program prompts for input. Input is read by the program using `fgets` with a buffer size of 1024 bytes. This input is then spilt into single characters using `strtok_r()`. Each token is then sent to a function to check whether it is a double. If it is the function stores the double in a location given to it and returns true. This double is then pushed to a stack. If the function returns false, the program checks if the token is an operator. If it is an operator, the program checks whether it is a unary operator or binary operator. It accesses the correct operator using a constant array, then sends the operator function to another function which applies the operator and pushes the result to the stack. If the operation does not complete successfully, the function returns false and an error message is printed. If `EOF` is received, input stops being read and the stack is printed. If `EOF` is received again, the program prints the stack and exits. Once the input has been parsed, the program prompts for input again.

Algorithms

Sin

This program estimates the value of sin(x) using the Maclaurin Series.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ... \tag{1}$$

The program first normalizes x to within 0 to 2π. The summation is calculated using a while loop. Rather than going to infinity, the program continues summing until the term is less than a number epsilon (ϵ). It creates variable to represent different parts of the Maclaurin series. The variable `sign` represents $(-1)^n$. Every loop, it multiplies itself by -1 to switch between positive and negative as it does in the Maclaurin series. The variables `power` and `power_multiplier` represent x^{2n+1}. With each loop, `power` is multiplied with `power_multiplier` to get x^3, x^5, x^7, etc. The variable `previous_factorial` represents the factorial that is skipped between each term. For example between the terms $\frac{x^3}{3!}$ and $\frac{x^5}{5!}$, the factorial 4! is skipped. The skipped factorial is calculated by $2n$ each loop after n is incremented. The variable `factorial_value` represents what the current factorial is equal to. It uses itself and `previous_factorial` and $2n+1$ (from $2n+1$ in $(2n+1)!$ in the series) to calculate its value each loop. The variable `n` is the same as n in the series. The term is calculated in the same way it is in the summation: $\frac{(-1)^n}{(2n+1)!}x^{2n+1}$ or $\frac{\text{sign}*\text{power}}{\text{factorial\_value}}$. Term is initialized equal to x because the first term of the series is x. `n` is initialized to 1 because the loop starts with adding the first term (which is already known) to the result, then calculating the next term which is when $n = 1$.

```
double Sin(double x)
    x = x % 2pi
    if x < 0 then
        x = x + 2pi

    result = 0
    sign = 1
    term = x
    power = x
    power_multiplier = x * x
    factorial_value = 1
    previous_factorial = 2
    n = 1
```

```
    while |term| >= EPSILON
        result = result + (sign * term)
        sign = sign * -1
        factorial_value = factorial_value * previous_factorial * (2 * n + 1)
        power = power * power_multiplier
        term = power / factorial_value
        n = n + 1
        previous_factorial = 2 * n

    return sum
```

Cos

This program estimates the value of cos(x) using the Maclaurin Series.

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + ... \tag{2}$$

The program first normalizes x to within 0 to 2π. The summation is calculated using a while loop. Rather than going to infinity, the program continues summing until the term is less than a number epsilon (ϵ). It creates variable to represent different parts of the Maclaurin series. The variable `sign` represents $(-1)^n$. Every loop, it multiplies itself by -1 to switch between positive and negative as it does in the Maclaurin series. The variables `power` and `power_multiplier` represent x^{2n}. With each loop, `power` is multiplied with `power_multiplier` to get x^2, x^4, x^6, etc. The variable `previous_factorial` represents the factorial that is skipped between each term. For example between the terms $\frac{x^2}{2!}$ and $\frac{x^4}{4!}$, the factorial 3! is skipped. The skipped factorial is calculated by $2n + 1$ each loop before `n` is incremented. The variable `factorial_value` represents what the current factorial is equal to. It uses itself and `previous_factorial` and $2n$ (from $2n$ in $(2n)!$ in the series) to calculate its value each loop. The variable `n` is the same as n in the series. The term is calculated in the same way it is in the summation: $\frac{(-1)^n}{(2n)!} x^{2n}$ or $\frac{\text{sign*power}}{\text{factorial\_value}}$. Term is initialized equal to 1 because the first term of the series is x. `n` is initialized to 1 because the loop starts with adding the first term (which is already known) to the result, then calculating the next term which is when $n = 1$.

```
double Cos(double x)
    x = x % 2pi
    if x < 0 then
        x = x + 2pi

    result = 0
    sign = 1
    term = 1
    power = x * x
    power_multiplier = x * x
    factorial_value = 1
    previous_factorial = 1
    n = 1

    while |term| >= EPSILON
        result = result + (sign * term)
        sign = sign * -1
        factorial_value = factorial_value * previous_factorial * 2 * n
        term = power / factorial_value
        previous_factorial = 2 * n + 1
        n = n + 1
        power = power * power_multiplier
```

```
    return result
```

Tan

To implement tan(x) the program uses the functions it contains for sin(x) and cos(x). This is possible because Tan($\frac{\pi}{2}$) will not be undefined. This is true because of two reasons: the IEEE 754 double precision standard cannot perfectly represent pi in binary and the Taylor-Maclaurin series used to approximate Cos($\frac{\pi}{2}$) will not converge to 0. Therefore, it will instead to converge to a very, very small number resulting in a very, very large value of Tan($\frac{\pi}{2}$). This reasoning is given in the assignment pdf. [1]

```
double Tan(double x)
    sin = Sin(x)
    cos = Cos(x)
    result = sin / cos
    return result
```

Sqrt

The program implements $\sqrt{x}$ by using the Babylonian method. In this method, you start with an initial guess. Then calculate the next guess using the formula: newguess = (oldguess + x/oldguess)/2. You then repeat the calculation using the new guess until the difference between the current guess and the previous guess is smaller than a specified tolerance value. In the program, it first checks if x is greater than 0. If it is not, then the function return `nan`. This is because you can only take the square root of positive numbers. The program sets initial guess as a variable `old` and which is initialized to 0. The new guess is a variable `new` and is also initialized to 0. The function loops and repeats the formula for the new guess until the absolute value of the `new` − `old` is less than Epsilon (ϵ) which is the tolerance value.

```
double Sqrt(double x)
    if x < 0
        return nan

    double old = 0.0
    double new = 0.0

    while (absolute value of (old - new) > EPSILON)
        old = new
        new = 0.5 * (old + (x / old))

    return new
```

Function Descriptions

stack

- `bool stack_push(double item)`: This function takes in a double and returns either true or false (1 or 0). It adds the parameter `item` to the top of the stack and increases the stack size.

```
bool stack_push(double item)
    if stack_size == STACK_CAPACITY then
        return false
    else
        stack[stack_size] = item
```

```
        increment stack_size
        return true
```

- `bool stack_peek(double *item)`: This program takes in a pointer `item`. It returns false if the stack is empty. Otherwise, it sets the pointer to the top of the stack and return true.

```
bool stack_peek(double *item)
    if stack is empty then
        return false
    else
        *item points to top of stack
        return true
```

- `bool stack_pop(double *item)`: This function takes in a pointer `item`. It returns false if the stack is empty. Otherwise, it sets the pointer to the top of the stack, decreases the stack size by 1, and returns true.

```
bool stack_pop(double *item)
    if stack is empty then
        return false
    else
        *item points to top of stack
        decrease stack_size by 1
        return true
```

- `void stack_clear(void)`: This function takes no parameters and does not return anything. It sets the stack size to 0.

```
void stack_clear(void)
    set stack_size to empty
```

- `void stack_print(void)`: This function takes no parameters and does not return anything. It prints the elements of the stack from bottom to top. This function is from Assignment 3: Calculator [1]

```
void stack_print(void)
    if stack is empty
        return

    print the first element with 10 decimal places
    for i, 1 to i < stack_size
        print stack[i] with 10 decimal places and a space before
```

mathlib

- `double Abs(double x)`: This function takes in a double `x` and returns a double `result`. Its purpose is to calculate the absolute value of x and return it.

```
double Abs(double x)
    if x >= 0 then
        result = x
        return result
    else
        result = -1 * x
        return result
```

- `double Sqrt(double x)`: This function takes in a double `x` and returns a double `result`. If `x` is not positive, the function returns `nan`. Otherwise, this function calculates the square root of x and returns the result. This function was given in the assignment instructions. [1] The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Sin(double x)`: This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates sin(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Cos(double x)`: This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates cos(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.
- `double Tan(double x)` This function takes in a double `x` and returns a double `result`. The double `x` is an angle in radians. It calculates tan(x) and returns the result. The explanation of this algorithm and its psuedocode is in the Algorithms section of this report.

operators

- `bool apply_binary_operator(binary_operator_fn op)`: This function 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 as its arguments (the first element popped is the right-hand side, and the next element popped is the left-hand side). Finally, it pushes the result to the stack. [1] If there are not two elements to pop, the function will return false. It will return true on success.

```
bool apply_binary_operator(binary_operator_fn op)
    idk doesnt seem to bad tho
```

- `bool apply_unary_operator(unary_operator_fn op)` This function takes in an operator, and accesses the global stack. It then applies the operator to the first element on the stack and pushes the result to the stack. If there are not enough elements to pop, it returns false or an error. This function is given in the assignment instructions. [1]

```
bool apply_unary_operator(unary_operator_fn op)
    if stack_size < 1 then
        return false

    double x
    assert stack_pop(x) true
    double result = op(x)
    assert stack_push(result) true
    return true
```

- `operator_add(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the sum of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_add(double lhs, double rhs)
    result = lhs + rhs
    return result
```

- `operator_sub(double lhs, double rhs)`: This function takes in two doubles: lhs and rhs. Its calculates the difference of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_sum(double lhs, double rhs)
    result = lhs - rhs
    return result
```

- operator_mul(double lhs, double rhs): This function takes in two doubles: lhs and rhs. Its calculates the product of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_add(double lhs, double rhs)
    result = lhs * rhs
    return result
```

- operator_div(double lhs, double rhs): This function takes in two doubles: lhs and rhs. Its calculates the quotient of the lhs (left hand side) and rhs (right hand side) and returns the result.

```
operator_div(double lhs, double rhs)
    result = lhs / rhs
    return result
```

- bool parse_double(const char *s, double *d): This function takes two parameters: a constant character pointer s and a double pointer d. It attempts to parse a double-precision floating point number from the string s. If the string is not a valid number it returns false. Otherwise, it stores the number in the location pointed to by d and returns true. This function is given in the assignment instructions. [1]

```
bool parse_double(const char *s, double *d)
    char *endptr;
    double result = convert string s to double
    if endptr != s then
        *d = result
        return true
    else
        return false
```

Psuedocode

```
main
    unary_operators = my_unary_operators

    while (opt = getopt) != -1
        switch opt
            case h:
                print usage message
                return 0
                break
            case m:
                unary_operators = my_unary_operators
            default:
                print usage message
                return 1

    while true
        x = 0
        take line from user input
        if input is EOF then
            return 0
        separate line into words
```

```
for each word
    if word is terminating null byte
        print stack
        clear stack
        loop for input
    else if parse_double(word, x) is true then
        if stack_push false
            print error message
            clear stack
            loop for input
    else if word in binary_operators then
        if apply_binary_operator(binary_operators[word]) is true then
            continue to next word
        else then
            print error message
            clear stack
            loop for input
    else if word in unary_operators then
        if apply_unary_operator(unary_operators[word]) is true then
            continue to next word
        else
            print error message
            clear stack
            loop for input
    else
        print error message
        clear stack
        loop for input
    print stack
    clear stack
    loop for input
```

Error Handling

- Unknown operation: If the program encounters an unknown operation such as 'l', it will print an error message and re-prompt the user for input. It will discard all previous input.
- Invalid command line argument: If the program receives and invalid command line argument, it will print an error message and usage instructions. It will also terminate the program.
- Invalid binary operation: If the program attempts to calculate a binary operation with only a single number, an error message will be printed, input will be cleared, and it will prompt for input again.
- Invalid unary operation: If the program attempts to calculate a unary operation with no number on the stack, it will print an error message, clear the stack, and re-prompt the user for input.
- Exceeding stack capacity: If the stack capacity is exceeded at anytime, the program will print an error message, clear the stack, and re-prompt the user for input.

Testing

The code is ran with valgrind to ensure there are no memory leaks. It is also compiled with scan-build to catch any errors.

Testing Functions

To test the functions, a `test.c` program is utilized. This c program contains asserts and function calls that confirm the output or result of each function is as expected. Some of the tests are as follows:

```
// testing stack
double a = 0.0;

stack_clear();
assert(stack_peek(&a) == false);
assert(stack_pop(&a) == false);

/*fill in stack*/
for (int i = 1; i <= STACK_CAPACITY; ++i) {
assert(stack_push(1.0 * i) == true);
}

assert(stack_push(65.0) == false);

// testing operators

assert(apply_binary_operator(operator_add));
assert(apply_binary_operator(operator_sub));
assert(apply_binary_operator(operator_mul));
assert(apply_binary_operator(operator_div));

assert(apply_unary_operator(my_unary_operators['s']));
assert(apply_unary_operator(my_unary_operators['c']));
assert(apply_unary_operator(my_unary_operators['t']));
assert(apply_unary_operator(my_unary_operators['r']));

stack_clear();
assert(apply_binary_operator(operator_add) == false);
assert(apply_binary_operator(operator_sub) == false);
assert(apply_binary_operator(operator_mul) == false);
assert(apply_binary_operator(operator_div) == false);

// testing mathlib

assert(Abs(-10) == 10);
assert(Abs(3.4) == 3.4);

assert(Abs(Sin(2) - sin(2)) < EPSILON);
assert(Abs(Sin(10*M_PI) - sin(10*M_PI)) < EPSILON);
assert(Abs(Cos(2) - cos(2)) < EPSILON);
assert(Abs(Sin(-3*M_PI) - sin(-3*M_PI)) < EPSILON);
assert(Abs(Tan(5*M_PI) - tan(5*M_PI)) < EPSILON);
```

More tests will be implemented as progress continues.

Testing RPN Framework

To test the overall functionality of `calc.c`, the `diff` command is used to check whether the output of the program matches the calc binary reference. To do this, an input file will be created containing a series of valid and invalid inputs. This input will be given to the calc binary and the output of stdout and stderr will be printed in two different files. This is done with the command:

```
./calc_ref < input.txt > expected_out.txt 2> expected_err.txt
```

where `calc_ref` is the binary, `expected_out.txt` contains stdout, and `expected_err.txt` contains stderr. The same is done with the programs executable:

```
./calc < input.txt > out.txt 2> err.txt
```

where `calc` is the executable, `out.txt` contains stdout, and `err.txt` contains stderr. Then diff is used as `diff expected_out.txt out.txt` and `diff expected_err.txt err.txt`. This will print the differences between the two files showing what needs to be changed or fixed.

Testing Math

The file `test.c` also contains asserts and function calls to test the math algorithms included in the program. These are listed in the Algorithms section. Additionally, graphs are made with a wide range of values on the x axis and the difference between the created algorithm and the math.h algorithm on the y axis. The graphs shows how accurate the algorithms created are. In order to get the points for the graphs, an additional program is created. This program prints the x values separated by commas and the y values separated by commas. This outputs can then be taken to a spreadsheet and a graph can be made.

Results

References

[1] Dr. Keery Veenestra and TAs. Assignment 2: Calculator. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.