Assignment 5
Calculator

by Jess Srinivas and Ben Grant
CSE 13S, Spring 2024
Document version 2 (changes in Section 4)

Due Wednesday May 15th, 2024, at 11:59 pm
Draft Due Monday, May 13th, 2024, at 11:59 pm

1 Introduction

It is common knowledge that computer scientists have very few hobbies. One of the hobbies that almost every computer science major has is making worse versions of software that already exists. While there are many existing calculator applications, in this assignment, not only will you be making a basic scientific calculator, you will also be making a subset of the functions in `<math.h>`. You will also be learning more about how computers work, and how they approximate mathematical values. Finally, you will get a chance to test your debugging and testing skills.

2 Helpful information

Reverse Polish Notation

In your academic career, you have probably used a lot of calculators, and most of them have probably had different ways to input an expression. Because it is hard to interpret and implement a calculator that inputs an expression the way that one might write it, we will use the standard Reverse Polish Notation. You can read more about it here, or in your C textbook. RPN always takes numeric values first, followed by an operator to apply. For example, the expression $2 + 2 =$ would be entered as 2 2 + [enter].

We can also chain these to make more complicated expressions. For example, `2 2 + 1 -` is the expression $(2+2)-1$. You'll notice that there is no need for parentheses in RPN, as it is always explicit which numbers are provided to which operator. With enough operators and a big enough buffer to hold previous results, it is possible to create a working, useful calculator.

RPN calculators are often implemented using a Stack ADT, which we will discuss more in the next section.

The stack ADT

To understand a Stack, we must first understand what an ADT is. ADT stands for "abstract data type". ADT's are simple instructions that describe the input and output of a data type, without giving any rules for how they will be implemented internally. For example, a stack ADT could be implemented with a singly linked list, a doubly linked list, an array, or many other things. Depending on the use case, each of these has its advantages, but they are all stacks. Instead of describing implementation, ADT's only describe functionality.

A good example of a stack is a stack of pancakes on a plate. There are only two things you can do to the stack: add one pancake to the top of the stack or remove a pancake from the top of the stack. Notice that there is no way to see or modify elements in the middle of the stack without removing all elements above it first.

A stack in its most basic form has 2 operations: Push and pop. Many versions of the stack ADT extend this functionality. To push to the stack is to add an element to the top of the stack. To pop from the stack is to remove the top element and crucially, return it to the user. Obviously, popping from an empty stack will return an error.

For simplicity, our stack implementation has a fixed capacity. This means that we decide the maximum number of elements it can contain at compile time, its `capacity`. This is not to be confused with its `size`, which is the number of elements in the stack. If a stack is at is capacity, it is not able to push an element, but it can still pop an element.

Some extensions of the stack include having a function that checks if the stack is empty (has a size of zero elements) or full (has a size of `capacity` elements if it is memory restricted), Printing the elements in the stack, emptying the stack, or a "peek" function, that returns a copy of the top element of the stack. The reason that `peek` is not part of the base stack ADT is that a peek can be performed by popping an element, returning a copy to the user, and pushing the same element back to the stack.

Libraries

Please read Ben's guide to compilation for more info.

Command Line Options and Arguments

Please read Ben's Guide to Options for more info.

Math

We have already covered the fact that the numeric data types in C are imprecise. You can verify this by adding `.1` and `.2`. While the output may look normal and correct at 16 digits, we are quickly reminded of the limitations of technology at 20 digits: the result is `0.30000000000000004`! This occurs because floating-point numbers can only precisely represent values that can be written as fractions with a denominator that is a power of 2. Just like how 1/3 cannot be written exactly in decimal, the binary representation of fractions like 1/10 have patterns that should carry on infinitely, but are limited by the precision of numeric types.

This limitation is combined with the fact that one can not store the exact value of irrational numbers on a computer (or anywhere, really). Because we are already so limited, computers use approximations of standard math functions as well. One well known method to approximate a function is the Taylor Series. This method approximates any function as a polynomial. As we increase the degree of the polynomial, the function becomes more accurate. The Taylor series uses a starting point, and repeatedly finds the derivatives, adding a new term with higher degree each time. This process increases the precision.

We can generalize the Taylor Series further. The Maclaurin Expansion is the Taylor Series centered around $x = 0$. Because we are only approximating functions between $x = 0$ and $x = 2\pi$, we can use the Maclaurin expansion. To understand how the Maclaurin expansion of $\sin x$ works, see this link.

The formula for the Taylor Series can be written as

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Our first natural question should be "How can a computer take a sum to ∞? Wouldn't that take forever?". This is where we must approximate. As we sum more terms, we get an increasingly precise answer. The next natural question is "How do we know where to stop?" We can pick a very small number, ϵ ("epsilon"), and continue our summation until our the absolute value of next term is no longer greater than epsilon. All the series that we use add smaller and smaller terms in order to refine a more and more accurate answer, so when the absolute value of the term is very small we know that our approximation is very accurate. For this assignment we set $\epsilon = 10^{-14}$, which is defined in `mathlib.h`.

Our next simplification is turning this Taylor series into a Maclaurin Series. We can do that because the a in $x - a$ is where we center our approximation, and that will always be $x = 0$. Now, we get the formula

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(0)x^n}{n!}$$

It is possible to simplify this further. You can find a pattern in $f^{(n)}(0)$ which will allow you to simplify this significantly. [1] That pattern is alternating as follows: 0, 1, 0, −1, 0, 1, ..., which can be simplified further by removing the zeroes. This makes the final function for an approximation of $\sin x$ this summation[2]:

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

And the approximation of $cosx$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

Computing $\tan x$ is much simpler. We know that the $\tan x = \frac{\sin x}{\cos x}$. We can use this fact to find $\tan x$ without using a series. You should use this in your code. If you are concerned about dividing by zero, you should remember that our functions will create an *approximation*, and therefore will never actually equal zero.

Function Pointers

You already know that your C program can have pointers to data variables. When you declare `char *s; double *d;` then `s` is a pointer to a `char`, and `d` is a pointer to a `double`. Once your program sets these pointer variables to valid addresses, your program can "dereference" them using the `*` prefix operator. So `*d` can be used as a `double` value.

The C programming language also lets your program declare a *pointer to a function*. Such a pointer can be set to the address of function, optionally passed as a parameter, and then called, just as the function would be called.

An example is using the function `apply_unary_operator()`. You can call this function like `apply_unary_operator(sqrt);`. You may recognize the parameter `sqrt` as is the name of a math function. In this context (with no parentheses after it), `sqrt` indicates a pointer to the `sqrt()` function.

Your program uses a function pointer as if it is a function name. That is, follow a function pointer with `()`, containing any necessary parameters, and whatever function the pointer is pointing to will be called.

In `operators.h` you will see two `typedef` lines. We use these because the programming-language syntax of function pointers is, let's say, complicated. It is easier to declare two types `binary_operator_fn` and `unary_operator_fn` and then use these two types to declare function pointers.

So, looking at this declaration, `bool apply_unary_operator(unary_operator_fn op);`, the parameter `op` is of type `unary_operator_fn`. The `typedef` of `unary_operator_fn` says that `op` is a pointer to a function that accepts one `double` parameter and returns a `double`.

In `operators.h` you will see three curious array definitions. These are each 256 bytes long, so you can index them with a character. Each one has a few function pointers at various indices that you can use to run a function given a character input. You can read more about this in the operators section.

[1] You may notice the use of $n!$ and x^n. These numbers grow very quickly, and calculating them quickly becomes time intensive. It is possible to store a value to avoid recomputing this every iteration. Consider how if $n = 5$, and you need to compute 5!, you must have already computed 4! on the previous loop iteration. Is there a simple way to convert 4! into 5!? You **MUST** use this technique. You are forbidden from using (or writing your own) `pow()` function, or writing a factorial function.

[2] You see $2n$ in the summation in two places because of the removal of the cases where $f(x) = 0$. This means that we don't add those cases, but because it is not possible to add 2 to n each summation, like we can in something like a `for` loop, we must do this instead.

3 Your task

Workflow

1. Complete and submit your `design.pdf` draft by Monday, May 13th.
2. Implement and test the functions in the files `operators.c`, `mathlib.c`, and `stack.c`
3. Complete your `main` function in `calc.c` and test your code
4. Fix any issues with your design draft, and finish the results section
5. Submit your final commit ID by Wednesday May 15th

Starter Files

The files we will provide to you to help you with this are as follows:

- `asgn5.pdf`: This file
- `mathlib.h`: A header file for implementing the math functions.
- `messages.h`: Contains the error and help messages you will need to print out.
- `operators.h`: Contains declarations for wrapper functions for most operators.
- `stack.h`: Contains the stack ADT function declarations
- `calc`: The reference program for this assignment.

Required functions

`mathlib`

In `mathlib.c`, you will be implementing the sin, cos and tan, and absolute value functions. Square root will be given to you.

`double Abs(double x)`

This function returns the absolute value of a double. You will not need to use `math.h` to complete this.

`double Sqrt(double x)`

This function returns the square root of a double.

```
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;
}
```

For all of the following functions, you should normalize the input (which comes to you in radians) to something in the range $(0, 2\pi)$

`double Sin(double x)`

This function returns the sine of a double. The angle passed in to this function will be in radians. For the sake of accuracy, you should normalize the angle to something between zero and 2π You will not use `sin` from `math.h` to complete this.

`double Cos(double x)`

This function returns the cosine of a double. The angle passed in to this function will be in radians. You will not use `cos` from `math.h` to complete this.

`double Tan(double x)`

This function returns the tangent of a double. The angle passed in to this function will be in radians. You will not use `sin`, `cos`, or `tan` from `math.h` to complete this.

`operators`

In `operators.c` you will be implementing `apply_binary_operator`, along with a few small wrapper functions. `parse_double` and `apply_unary_operator` will be given to you.

`operators.h` contains three arrays of function pointers: for binary operators like addition, unary operators like sine using your own functions, and unary operators using the standard `math.h` functions. These all have size 256 which means that you can index them using an ASCII character that the user input. For instance, `binary_operators` contains the entry `['+'] = operator_add`. This means that the index `'+'` (remember, characters are numbers!) of this array is set to `operator_add`. So if the user inputs +, you could look it up in this array and find which function you should use to perform that operator. Since these arrays are global, the unspecified elements are set to zero or `NULL`, so you can use that to check if some character is a valid operator.

`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.

There is no case in which this will make the stack too big. It is possible that there are not two elements to pop. If that is the case, this function will return an `false`, and not print anything (error handling should be done in `calc.c`). It will return `true` on success.

If, for example, the stack contains (bottom to top) "2, 4, 1", and we apply the "+" operator, the stack will end with "2, 5".

`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.

There is no case in which this will make the stack too big. It is possible that there are not enough elements to pop. If that is the case, this function will return an error, and not print anything.

```
bool apply_unary_operator(unary_operator_fn op) {
   // Make sure that the stack is not empty
   if (stack_size < 1) {
      return false;
```

```
    }

    double x;
    // It should be impossible for stack_pop to fail here, since we have checked
    // above that the stack is not empty. The assert() function causes your
    // program to exit if the condition passed is false. It is often useful for
    // "sanity checks" like this: if stack_pop fails, either our assumption that
    // it can't fail was false, or the implementation of stack_pop has a bug.
    // Either way, we'd like to know immediately instead of silently ignoring an
    // error.
    assert(stack_pop(&x));
    // Calculate the value we should push to the stack
    double result = op(x);
    // Similar to above: stack_push should never fail here, because we just
    // popped an element, so we are not increasing the stack size past the size
    // it had originally.
    assert(stack_push(result));
    return true;
}
```

`double operator_add(double lhs, double rhs)`

This function takes the sum of the doubles left hand side (`lhs`) and right hand side (`rhs`) and returns it.

Yes, this function is as simple as it sounds (as are the next three). We need these basic math operators to be functions so that we can store them in arrays of function pointers.

`double operator_sub(double lhs, double rhs)`

This function takes the difference of the doubles left hand side (`lhs`) and right hand side (`rhs`) and returns it. (lhs − rhs)

`double operator_mul(double lhs, double rhs)`

This function multiplies the doubles left hand side (`lhs`) and right hand side (`rhs`) and returns it.

`double operator_div(double lhs, double rhs)`

This function divides the double left hand side (`lhs`) by right hand side (`rhs`) and returns it. ($\frac{\text{lhs}}{\text{rhs}}$)

`bool parse_double(const char *s, double *d)`

This function attempts to parse a double-precision floating point number from the string `s`. If successful, it stores the number in the location pointed to by `d` and returns `true`. If the string is not a valid number, it returns `false`.

Here is the implementation of this function. It uses the library function `strtod`. `endptr` is set by `strtod` to point to the end of the number that was parsed from the input string. We check if `endptr` is the same as `s` to see if parsing failed; if they are the same, it means there was not a valid number in the string `s` therefore the number ends immediately at the start of the string.

```
bool parse_double(const char *s, double *d) {
    char *endptr;
    double result = strtod(s, &endptr);
    if (endptr != s) {
        *d = result;
        return true;
    } else {
        return false;
```

```
    }
}
```

stack

In `stack.c`, you will be implementing the stack ADT functions.

The stack that you will be building contains the following data:

- an `int` for storing the size (not capacity!)
- an array of `STACK_CAPACITY` doubles, which you should initialize to zero.

These variables are defined in `stack.h` as `extern`, meaning that you can access them in any file that includes `stack.h`, but must only define and initialize them once in `stack.c`. It's very similar to how functions are *declared* in header files and *defined* in the corresponding C file. You may modify them as you see fit. As much as possible, you should avoid touching the stack outside of `stack.c`. You will be implementing functions to push, pull, peek, pop, and clear the stack. We have provided a function to print the stack.

`bool stack_push(double item)`

Pushes `item` to the top of the stack. Updates `stack_size`. Returns `true` if a push was possible, and `false` if the stack is at capacity. If the stack is at capacity, the stack and size should not be modified.

`bool stack_peek(double *item)`

Without modifying the stack, copies the first item of the stack to the address pointed to by `item`. Returns false, and does not modify `*item` if the stack is empty, and true otherwise.

Why do this function and `stack_pop` take pointer arguments? We'd really like these functions to return two values: whether a value was popped/peeked successfully, and the value itself. But C functions can only return one value. To solve this, we make the success state the actual return value, and handle the numeric value through a pointer. Whoever calls this function should set up a variable in which they want the peeked value to go, and then pass the address of that variable into `stack_peek`. This will allow `stack_peek` to change the contents of that variable to the value that was peeked. See our implementation of `apply_unary_operator` for an example of how this looks in C code.

`bool stack_pop(double *item)`

Stores the first item of the stack in the memory location pointed to by `item`. Returns false, and does not modify `item` if the stack is empty, and true otherwise. Updates `stack_size`. Note that you do not need to change the element in the `stack` array, as it will get overridden when `stack_push` is called.

`void stack_clear(void)`

Sets the size of the stack to zero. Values in the stack do not need to be modified. See `stack_pop` for more info.

`void stack_print(void)`

This function prints every element in the stack to `stdout`, separated by spaces. It does not print a newline after. We give you the code for this function to ensure that your results are printed the same way as ours:

```
void stack_print(void) {
   // make sure we don't print stack[0] when it is empty
   if (stack_size == 0) {
      return
   }
   // print the first element with 10 decimal places
```

```
   printf("%.10f", stack[0]);
   // print the remaining elements (if any), with a space before each one
   for (int i = 1; i < stack_size; i++) {
      printf(" %.10f", stack[i]);
   }
}
```

Assignment specifics

CLI options

In this assignment (and future ones), we will be using a tool called `getopt`. It will help us process command line arguments from `argc` and `argv`. This program will support the following arguments:

- `m`: Uses trig functions from `libm`.
- `h`: Prints the help message,

Errors

This is the first assignment in which we will be using `stderr`. When we use `printf`, everything we print is sent to standard out, aka `stdio`. We can use the `fprintf` function to choose where we print anything (see the `man` pages for more). All errors must be printed to `stderr`. If the user enters an invalid option, the help message will be printed to `stderr`, and the program will quit.

While `stdout` and `stderr` are both printed to your terminal by default, they are different. You can tell them apart using your shell's redirection operators: `>` redirects stdout to a file and `2>` redirects stderr to a file. Combined with `<` to prove stdin from an existing file, you could run a command like `./calc < input.txt > out.txt 2> err.txt`, which redirects stdout and stderr to different files so that you can make sure they are split correctly.

You must use all errors in `messages.h` correctly. If you encounter an error with starting the program (like an invalid option), you must return a non zero exit code from `main` (you can run `echo $?` immediately after your program to check what the exit code was). If you encounter any other error (like empty stack or invalid operator), you must print the respective error message and continue running the program. If your program quits normally due to EOF (which you can accomplish by pressing Ctrl+D), it should exit with code 0 (even if some of the expressions that the user input were not valid).

Using `math.h`

In your `mathlib.c` file, you can only use 3 things from `math.h`. Those are: `M_PI` (The value of π), `fmod`, which allows you to use the modulo operator for floating point values, and `nan`, which the provided `Sqrt` implementation uses for its return value when the input is negative. **All other usage of `math.h` in `mathlib.c` will be considered an attempt to cheat.** Obviously, you may use `math.h` in `calc.c` or any other file.

Parsing user input

When you start the calculator, you will print a `>` character to standard err (`stderr`). You need to read one line of input at a time, and then split it into words separated by the space character and process each word (which may be a number, like `5.3`, or an operator like `+` or `r`) one by one. Reading a line of input may be done with the function `fgets` on a suitably sized buffer (1024 bytes is plenty). Keep in mind that `fgets` will store a newline character at the end of your buffer which you'll have to remove.

Once you've parsed a line of input into `expr`, you should use the `strtok_r` function according to the following template to split it into words:

```
// saveptr is a variable that strtok_r will use to track its state as we
// call it multiple times
char *saveptr;
// you can set error to true to stop trying to parse the rest of the expression
```

```
bool error = false;
// strtok_r splits the input string (expr) on any delimiter character in a
// sequence (for us, only spaces)
// it stores its own state in saveptr
// it returns a pointer to the next token, or NULL if we have processed the entire string
const char *token = strtok_r(expr, " ", &saveptr);

// loop until we finish parsing the string or we encounter an error
while (token != NULL && !error) {
   // process token
   // possibly set error to true


   // then, at the very end of your loop...
   // (note that we pass NULL instead of expr to indicate we are still processing the same string)
   token = strtok_r(NULL, " ", &saveptr);
}
```

You will then apply the operator specified, and use `stack_print` to print all the contents of the stack to `stdout`.

Submission

These are the files we require from you:

- `Makefile`: This should have rules for `all`, `calc`, and all `.o` files with the following compiler flags: `-Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic`. You may use other flags for the purposes of debugging, but you must remove them for your final submission. `make all` should make the `calc` binary. You must also have targets for `make clean` and `make format`.
- `design.pdf`: See the design template.
- `mathlib.h`: This file is given. It contains function declarations for the math functions you will be implementing.
- `messages.h`: This file is given. It contains some error messages you need to print.
- `operators.h`: This file contains some function declarations. You should not need to modify this file.
- `stack.h`: This file contains function declarations for the Stack ADT
- `calc.c`: This file should contain your main program.
- `mathlib.c`: This file should implement all required math functions using their approximations.
- `operators.c`: This file should contain the basic operators in their wrapper functions.
- `stack.c`: This file will contain your implementation of the stack ADT.

You must also submit any additional code you wrote to create the graphs. If any of this code is written in C, we recommend adding a rule to your Makefile to compile this code.

All header files may be modified to add more functions or variables, but existing function declarations may not be modified.

All header files and source files must be clang-formatted by running `make format` before you submit them.

Remember that you may use code from lecture, the textbook, this pdf, my guide, or sections, but they must be cited (at least) as a comment in your code and in your `design.pdf`. We do not require any particular citation style, but you must include the name of the author, and the resource location (page, slide, file... location). If you chose to use code from any of these resources, you should also understand how this code works.

Results

Your design will have a new section in this assignment: results. Some extras on testing are also described here.

Testing your code

This is required, and you must show your procedure and results in your `design.pdf`. You must also have a good idea of how you're going to test your code before you begin writing it, and when you submit your report draft. You must do complete all of these sections before you submit the report draft. Almost all of this testing should be done using the `diff` command and the provided binary. You must not rely on the pipeline for testing your code. For each of the following sections, please show any code or commands that you will use to test your code, as well as the inputs, expected outputs, and what cases or edge cases they cover.

Your code must also pass `valgrind` cleanly for any reasonable input (including inputs that cause caught errors) with no errors or memory leaks. Finally, running `scan-build make` must not produce any errors. You should ensure that your code has no undefined behavior to make sure that it passes all tests when we run it.

Testing your functions

You must describe how each of your functions was tested. You should do this by writing another c file, and creating a make target called `test`. You must describe it in your report. You must show testing of a diverse range of inputs. You should test every function that you code, making and record what inputs you give it. If the function is expected to error, you should test that as well. You may also use this to create graphs for the resutls section.

Testing your RPN framework

The first thing you will test is the rpn calculator itself. In the "Testing" section of your report draft, create a few diverse test cases that prove that your calculator works for the operators that you didn't write. You should make sure that all functions and command line options work as expected. You should be sure to test against edge cases and things that may produce errors. You must print all errors in `messages.h`. You should test how your program handles other edge cases, as it should never crash given a reasonable input.

Testing math

Obviously, using an approximation of trigonometric functions will create an approximate result. There are also other sources of error in this assignment. You should test your program to confirm that your program's output closely matches the `sin` function given in `math.h`.

Results

In the results section of your report, compare some of your results to the results given in `math.h`. You may test your functions outside of `calc.c` (for instance, you could write a separate program which simply calls your functions across a large domain of inputs, and prints the results in a format convenient for graphing like CSV, instead of acting as a calculator). **At a minimum, we expect a graph of the difference between your functions and the `math.h` functions for all 3 trig functions**. The graph can be created with any tool of your choosing, but should have a wide range of values on the x axis, and the *difference* between your output and math.h on the y axis. This is not the same as having two sets of data on the same graph. Make sure your axes are labelled clearly. **Please submit any code you used to create this graph**.

Another very important thing you must include is an explanation of where any error you may have found came from. You should describe as many sources as possible in detail. You may also include any other graphs that you believe we may find useful. If you use any optimizations for accuracy, please describe what you did, and why.

4 Revisions

Version 1 Original Draft.

Version 2 Removed requirement to make or submit a `tests` directory or a `tests.c` program. You still should test your program! It's just that the directory name was the same as the binary name, which won't work. Moved footnote 2 to avoid inadvertently creating $\sin^2 x$ when I really meant $\sin x$.