Assignment 3: Postfix Calculator

CSE30: Computer Organization and Systems Fall 2021

Instructors: Bryan Chin and Keith Muller

Updated: Monday, October 12th, 2021 @ 2:00AM

Due: Sunday, October 17th, 2021 @ 5:00PM

Please read over the entire assignment before starting to get a sense of what you will need to get done in the next week. REMEMBER: Everyone procrastinates but it is important to know that you are procrastinating and still leave yourself enough time to finish. Start early. **You MUST run the assignment on the pi cluster. You HAVE to SSH: You will not be able to compile or run the assignment otherwise.**

NOTE: Please read the FAQ and search the existing questions on Edstem before asking for help. This reduces the load on the teaching staff, and clutter/duplicate questions on EdStem.

Table of Contents

Learning Goals

- Learn to implement a multi-file C program
    - Indexing and manipulating arrays
    - Process input from stdin and interpreting the input
    - Perform operations based on user input
    - Use static variables that are hidden to a file
    - Use static variables that are hidden to a function
- Export an interface to a `.c` file using a `.h` file

Background: Postfix Notation

What is a Postfix Notation?

A postfix notation is a mathematical notation in which the operands are followed by the operator. For example, if 8 and 9 are to be added, the notation would be 8 9 +. If there are multiple operations that need to be done, for example, 5 x 3 + 9, is written as 5 3 x 9 +. The main advantage of using postfix notation is the elimination of parentheses, which leads to lesser processing for evaluating the expression. Most modern systems use the postfix notation internally to evaluate expressions. Many of the classic HP calculators worked this way https://en.wikipedia.org/wiki/HP_calculators.[1]

How are Postfix Notations Processed?

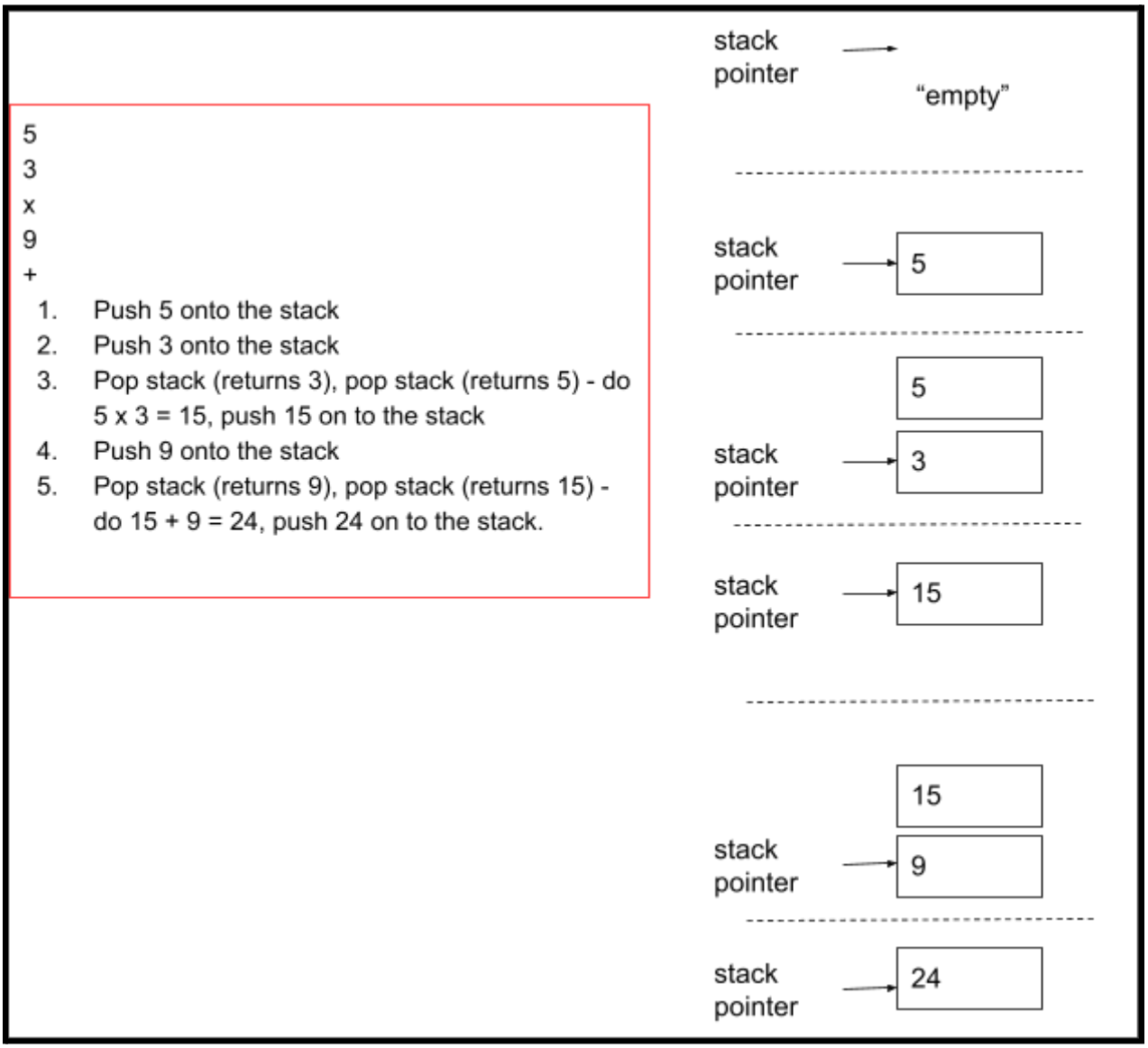
A data structure called stack (last in, first out) is used to evaluate postfix expressions. The stack data structure has two basic operations associated with it - push and pop. Push inserts an element to the top of the stack, whereas pop removes the top most element from the stack (imagine a stack of plates - push operation is when you add a plate onto the stack, pop operation is when you take a plate off the top of the stack). Stacks can be represented as an array.

The processing of a postfix expression can be described by the following steps:

1. If a number is encountered, push onto the stack
2. If an operator is encountered, fetch the operands by popping the stack once or twice, evaluate the result of the operation on the operand(s), push the result onto the stack. If the stack does not contain enough operands to perform the operation, ignore the operation.

[1] Many computers have been designed around a stack based architecture (although this particular kind of hardware architecture has fallen out of favor) - https://en.wikipedia.org/wiki/Stack_machine.
The Java Virtual Machine (JVM) is designed around a stack execution model. The UCSD p-system https://en.wikipedia.org/wiki/UCSD_Pascal was also based on a stack architecture.

Stack Example

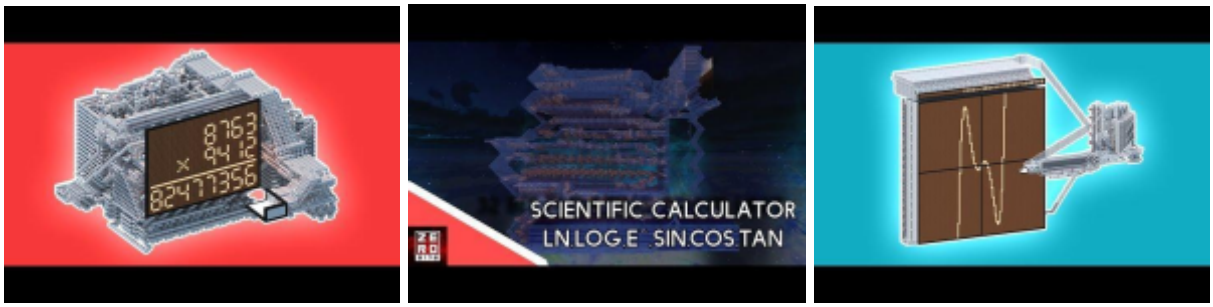


Program: Postfix Calculator

Assignment Overview

One of the most complex and interesting things to play with in Minecraft is redstone. Added to the game in the first update to the Alpha version over 11 years ago, the material was first simply intended to be used to open doors. However, the potential uses of redstone were much greater, as the way the item mechanically functioned essentially replicates real life circuitry. Specifically, when placed on other blocks, redstone simulates real life wiring, and when crafted into redstone torches, it simulates transistors. Don't worry if you don't understand electric engineering, since this is a computer science class (though you may learn things related to this in 140/141/142!!). The main takeaway is that this leads to the ability to create logic gates and, most importantly, circuits -- the building blocks of all electronics, including computers.

With that being said... You've recently been thinking of creating a really cool and complex redstone contraption. But you also want to make something that no one has ever made before. Yet, this seems to be extremely difficult since it seems like every single thing has been made, from something simpler like a calculator to a full-blown computer has full length tutorials. And, all different kinds of calculators have been made (normal, scientific, graphing):



However, there are no postfix calculators in the entire Minecraft community! (Seriously, go Google it.) What an opportunity! These redstone calculators are pretty slow, so imagine the processing time that could be saved by allowing for postfix evaluation instead of infix!

But, before you get to building in Minecraft, you decide to try to figure out the logic to how a postfix calculator works. What better way to do so than writing a program?

Getting Started

Developing Your Program

For this class, you **MUST** compile and run your programs on the **pi-cluster**. To access the server, you will need your `cs30fa21cxx` student account, and know how to SSH into the cluster.

Need help or instructions? See Edstem FAQ or watch CSE30 Development Tools Tutorial
(Do NOT wait until Friday to try this. There will be limited or no ETS support on the weekends!)

We've provided you with the starter code at the following GitHub repository.

1. Clone/download the GitHub repository.
2. Fill out the fields in the `README` before turning in.
3. Open your favorite text editor or IDE of choice and begin working.

Running Your Program

We've provided you with a `Makefile` so compiling your program should be easy!

Additionally, we've placed the reference solution binary at:
`/home/linux/ieng6/cs30fa21c/public/bin/pfcalc-a3-ref`.
You can directly use `pfcalc-a3-ref` as a command.

Makefile: The `Makefile` provided will create a `pfcalc` executable from the source files provided. Compile it by typing `make` into the terminal. By default, it will run with warnings turned on. To compile without warnings, run `make WARN=` instead. Run `make clean` to remove files generated by `make`. For additional make instructions specific to this assignment, click here.
Have a look at the Makefile and see that you understand how it works.

How the Program Works

Your program will be run, and provided the commands as listed in the table. The program must implement the behaviors described in the table.

`./pfcalc`

The commands are::

Command	Description
h	Print a helpful message summarizing all the commands.
[#] (number)	Push the number onto the stack. The number may be any floating point number and does not necessarily contain a decimal point.
Any of the operators: +, -, *, /, T	Pop the top two stack elements (top one if square root) and operate on them. T denotes the square root operation. Print the top of the stack or 0.0 if the stack is empty.
c	Clear the value of the top of the stack entry (set it to 0.0), pop the entry, and then print the new top of the stack. Print the top of the stack or 0.0 if the stack is empty.
s	Swap the two top stack elements. Print the top of the stack or 0.0 if the stack is empty.
r	Rotate the stack - the oldest entry becomes the youngest, the youngest becomes the second youngest, etc. Print the top of the stack or 0.0 if the stack is empty.
p	Print the stack - starting from the oldest stack entry to the top of the stack.
q	Quit the program.

- All valid commands (besides help, numbers, print, and quit), print the entry at top of the stack (or 0.0 if the stack is empty) after executing.
- If the user attempts to push a number onto the stack but the stack is full, an error message is printed and the input is ignored - the error message is "`Stack is full, # ignored\n`" where `#` denotes the number that was entered and `\n` is the newline character.
- If an operation is requested that requires more operands than are currently in the stack,, then the operation is ignored and the top of stack is printed (or 0.0 if the stack is empty). For example, if the stack contains only one number, and the "+" operation is requested, the "+" operation is ignored and the top of stack is printed. If the stack is empty and the "T" operation is requested, the sqrt is ignored and 0.000000 is printed.
- If an invalid command is entered, print the error message - "`Unknown command 'x'\n`" where x denotes the command entered, and `\n` is the newline character. Note the single quotes around the illegal command x.

Example Usage

```
$ ./pfcalc
>>> h
Commands:
        h   Print this help message
        [#] Push number onto the stack
        +   Add youngest stack element and prior stack element
        -   Subtract youngest stack element from prior stack element
        *   Multiply youngest stack element and prior stack element
        /   Divide youngest stack element into prior stack element
        T   Square root of the youngest stack element
        c   Clear the value of the youngest stack element
        s   Swap the top two stack elements
        r   Rotate the stack (oldest becomes youngest, ...)
        p   Print the stack, oldest to youngest
        q   Quit
>>> 1
>>> 2
>>> 3
>>> 4
>>> 5
Stack is full, 5 ignored
>>> +
7.000000
>>> -
-5.000000
>>> *
-5.000000
>>> 0
>>> /
-inf
>>> p
-inf
>>> c
0.000000
```

```
>>> 3
>>> 5
>>> 7
>>> 9
>>> s
7.000000
>>> p
3.000000
5.000000
9.000000
7.000000
>>> r
3.000000
>>> p
5.000000
9.000000
7.000000
3.000000
>>> z
Unknown command 'z'
>>> q
```

Note: Floating point numbers in computers are represented as an approximate value rather than an exact value. There are certain values that represent specific values, like 0, infinity and not-a-number, for which floating point numbers represent negative infinity (-inf), positive infinity (inf) and not-a-number (NaN) - which is seen as one of the outputs above when we end up dividing -5 by 0. NaN occurs from illegal math operations such as dividing a 0 by 0.

Program Implementation

The program is split into two files: `main.c` and `calc.c`. The file `main.c` contains only the command interpreter logic and none of the stack or calculation operations. The file `calc.c` has the functions listed below which you need to implement according to the description given. These functions are **declared** in the header file `calc.h` - which needs to be included in both `main.c` and `calc.c` - **do NOT change any declarations that we have provided you with.** You will need to add some declarations to `calc.h` to allow the functions in `calc.c` to be called from `main.c` based on the descriptions given below.

The functions to be implemented in `calc.c` are:

Function Name	Return Value	Argument(s)	Description
`int push`	1 if successful, 0 if stack is full	`double number`	Pushes a number to the stack. Does nothing if the stack is full.
`void pop`	none	none	Pops the topmost element from the stack. Does not return a value.
`double get_entry`	Element from the stack	`int flag`	If the flag > 0, returns the entry on the bottom of the stack. else, returns the entry above the last entry returned by this method. (Note: If you call this method with a number <= 0 after previously returning the topmost entry, or before calling it with a number > 0, the functionality is undefined.)
`int get_size`	# of elements in the stack, 0 if empty	none	Returns the number of elements currently in the stack. Returns 0 if the stack is empty.
`double get_tos`	Element on the top of stack	none	Returns the element on the top of the stack. If the stack is empty, return 0.0.
`void do_oper`	none	`int operation` (use operation constants in `calc.h`)	Performs the operation on either the top two stack entries or the top stack entry.
`void swap_entry`	none	none	Swaps the top two entries of the stack. Does not return a value. If the stack has fewer than 2 entries, do nothing.
`void rotate_stack`	none	none	Rotates the stack such that the bottom element becomes the top, topmost element becomes the second topmost and so on. If the stack has one or fewer entries, do nothing.

NOTE: YOUR FUNCTIONS MUST HAVE THE EXACT SAME DATA TYPES AS DESCRIBED ABOVE OR YOUR FUNCTIONS WILL **NOT** WORK WITH THE TESTS.

The file `main.c` needs to implement the command interpreter described above and call the appropriate functions defined in calc.c to perform the operations. The starter code has the boiler plate for implementing a command interpreter and you need to complete the implementation. Follow the instructions mentioned in the starter code - we have mentioned the areas of code you are supposed to modify and not to modify. The following are the details with regard to the `main.c` file:

- The while loop in the `main` function is responsible for accepting commands from the user until the user gives the `'q'` command (or pressing `CTRL+D`, which sends EOF).
- We read the input provided by the user into `buf` and get rid of the newline character from `buf`.
- After we read some input into buf, you need to determine if the input was a number or a command, and perform the operation based on the table of commands described above.
- If the input was a number, it would be available in the variable `number`, else the command would be available in the variable `cmd`.

Additional Instructions

There are other restrictions while implementing this assignment:

- You are **NOT** allowed to have any `printf(), puts(), fprintf()` or any other function to print to standard output in the file `calc.c` - **ALL input/output operations** have to be done in the `main.c` file.
- Your stack related variables (like the stack itself, top of stack) **MUST NOT** be visible in `main.c` scope. Any variable added to `main.c` for iteration purposes must be an integer only.
- Stack implementation **MUST BE** done with an array variable (you figure out what data type!) named `stack`. You should have index 0 represent the bottom of the stack at all times, and the top of stack be represented by the highest stack index. This is important because almost all function tests will look for this. For stack size, look at calc.h for a hint!
- The default stack size is 4, however, it can be changed via compile time flags. The way to do this is described below. You should test your program with different stack sizes before submitting. The maximum stack size is 512.
- As described in the `do_oper` arguments, you **MUST** use the operation constants in `calc.h` when calling `do_oper`. Do not call `do_oper` with any other values.

All of the above will be enforced in the autograder and any submission found to be not adhering to these set of guidelines will be penalized.

The size of the stack is determined by the pre-defined value `STACKSIZE`. If not specified via compile time, the value will be 4. However, if you want to increase the size of the stack you need to compile your program with the following command:

```
% make STACKSIZE=10
```

This would make the value of STACKSIZE 10.

Submission and Grading

Submitting

1. Submit your files to Gradescope under the assignment titled “A3: Postfix Calculator”. You will submit the following files:

   ```
   calc.c
   calc.h
   main.c
   README.md
   ```

   You can upload multiple files to Gradescope by holding CTRL (⌘ on a Mac) while you are clicking the files. You can also hold SHIFT to select all files between a start point and an endpoint.

   Alternatively, you can place all files in a folder and upload the folder to the assignment. Gradescope will upload all files in the folder. You can also zip all of the files and upload the `.zip` to the assignment. Ensure that the files you submit are not in a nested folder.

2. After submitting, the autograder will run a few tests:
   a. Check that all required files were submitted.
   b. Check that `main.c` and `calc.c` compiles.
   c. Runs some tests on the resulting `pfcalc` executable.

Grading Breakdown [50 pts]

Make sure to check the autograder output after submitting! We will be running additional tests after the deadline passes to determine your final grade. Also, throughout this course, make sure to write your own test cases. ***It is bad practice to rely on the minimal public autograder tests.***

The assignment will be graded out of 50 points and will be allocated as follows:

Function Unit Tests [25 pts]

- `push()` - 3 points (1 public test, 2 hidden tests, 1 pt each)
- `pop()` - 3 points (1 public test, 2 hidden tests, 1 pt each)
- `get_entry()` - 4.5 points (1 public test, 2 hidden tests, 1.5 pt each)
- `get_size()` - 2 points (1 public test, 1 hidden test, 1 pt each)
- `get_tos()` - 2 points (1 public test, 1 hidden test, 1 pt each)
- `do_oper()` - 4.5 points (1 public test, 2 hidden tests, 1.5 pt each)
- `swap_entry()` - 3 points (1 public test, 1 hidden test, 1.5 pt each)
- `rotate_stack()` - 3 points (1 public test, 1 hidden test, 1.5 pt each)

Main Tests [25 pts]

- 2 public tests, 3 hidden tests (5 pts each)
- Penalties
    - I/O operations are found outside of main.c (-5 pts)
    - Stack variables are found or accessible outside of calc.c (-5 pts)
    - `do_oper` is called with values other than constants defined in `calc.h` (-2.5 pts)

Make sure your assignment compiles correctly through the provided `Makefile` on the pi-cluster. **Any assignment that does not compile will receive 0 credit.**

Writeup Version

1.0 Initial release.

1.1 Changed `get_entry()` description and added stack implementation requirements.
Some wording changes and formatting changes also made for clarity.
Added grading breakdown test cases.

1.2 Clarified requirements on `do_oper()` and added a penalty for it in grading breakdown.
Clarified requirements for all functions.
Filled Table of Contents.