The George Washington University

CSCI 3410 Systems Programming

Homework 4 - Stack and File I/O

Due Date: Monday Mar 25, 2019 at 18:00 PM

This is an *individual assignment*, you may not collaborate or share code with other students.

Background Information

tl;dr Use `gdb` when you see 'Segmentation Fault'. Get a Linux VM.

First and foremost, you should configure a Linux virtual machine to compile C code for this assignment (and in the rest of the course). The primary platform is VMWare (the free version should be sufficient, but you can contact SEAS for a license). There are numerous guides on the Internet (like this one) that offer a step-by-step approach to configuring a VM. We recommend that you install Ubuntu.

You may remember having studied the stack in a prior course. A stack operates on the premise that the last element in is the first element out. And vice versa, the first element in is the last element out.

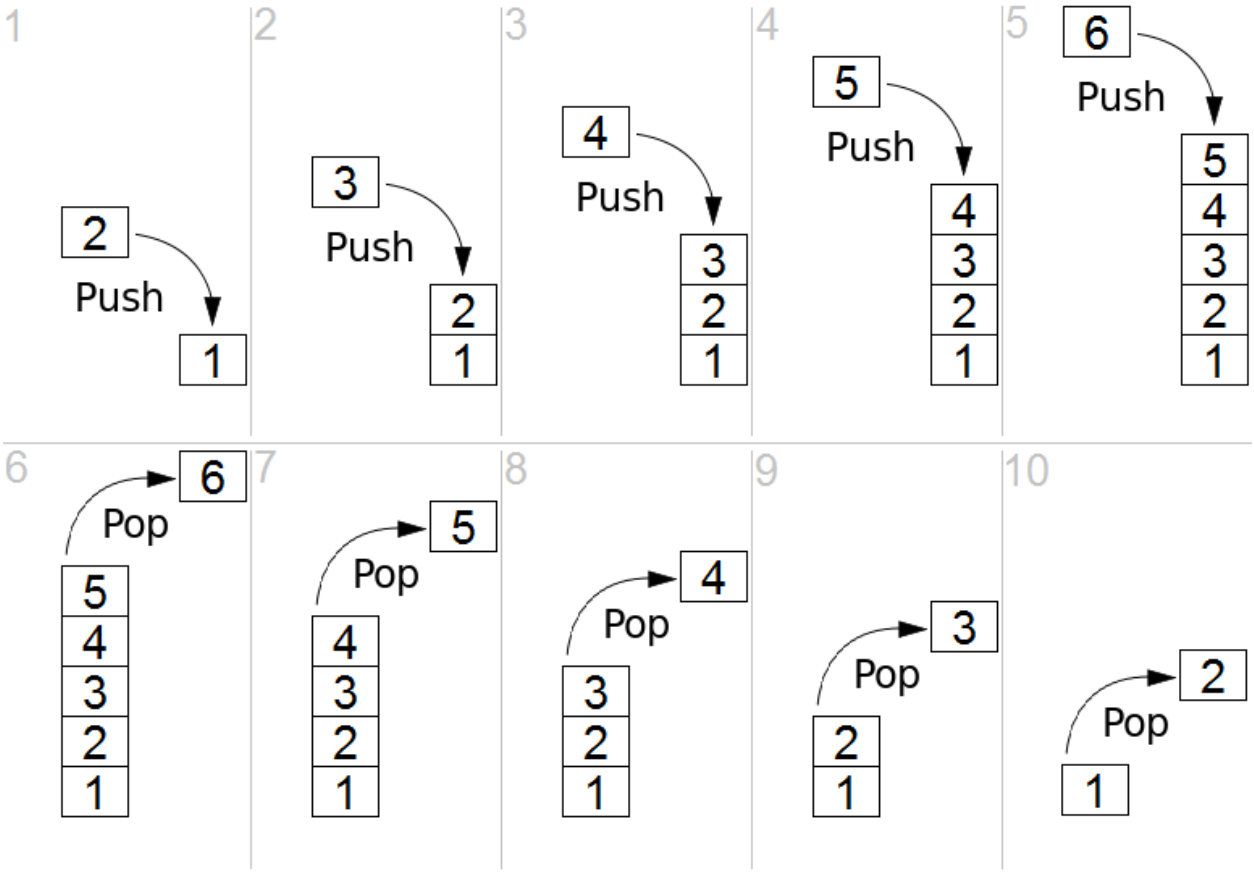

Figure 1: A Standard Stack

The stack which will be used in this assignment and can be found in stack.h is defined as:

```
struct stack {
    double* stack_nums;  // The structure that holds the actual values within the stack
    int max_size;        // The maximum capacity of the stack
    int current_size;    // The current amount of values that the stack holds
}
```

Next, some notes about dynamic memory allocation. You should be familiar with using both the `malloc` and `free` functions from the fall. You must use dynamic memory allocation in this assignment, so be sure to read up if you need to. For more information, run `man malloc` on any UNIX-like system.

And finally, debugging. Do not forget the old adage: *The last 20% of the work takes 80% of the time.* Therefore, be sure to start early and ensure you have plenty of time to debug your code, especially when pointers are involved. You should use the programs `gdb` and `valgrind` to help you debug your code. Do not ask the TAs for help without having first attempted to debug your issues yourself with `gdb`.

System Specification

We will be implementing a stack that will compute reverse polish notation (RPN) equations. If you are not familiar with RPN then look here.

The equations will be input with a text file labeled 'input.txt'. This file will have a RPN equation on each line. The file reading code is already implemented and ready to go. Your job is to interpret the read values, use your stack to compute the result, and write the results to a new file titled 'output.txt'.

RPN is a little bit difficult for the human mind to interpret, but can easily be computed with a stack. If the equation '3 4 + 6 -' is input. Then the system will push 3 and then push 4. Once the + sign is observed, the last two values (in this case 4 and 3) will be popped and added. The result (7) will be pushed back onto the stack. From here, 6 will be pushed onto the stack and then a minus sign will be read. The last two values will be popped (6 and 7). And the operation will be computed, resulting in an answer of 7-6 = 1.

1. Your first task is to implement the following functions for the stack, as defined in stack.h:

- `create_stack`: This function will malloc the memory needed for the stack using max_size as the maximum amount of values that stack_nums can hold. Return a pointer to the stack upon success and 'NULL' otherwise. Remember: only (a) certain element/elements within the stack need to be malloc'd. What is it/are they?
- `destroy_stack`: This function will destroy the stack, freeing the elements which were initially malloc'd. Return 1 upon success and 0 otherwise.
- `peek`: This function will peek the value from the stack if the stack isn't empty. Return the number peeking if successful and -1 otherwise (even though -1 could be in the stack, we are keeping things simple)
- `pop`: This function will pop the value from the stack if the stack isn't empty. Return the number popped if successful and -1 otherwise (even though -1 could be in the stack, we are keeping things simple)
- `push`: This function will push the value onto the stack if there is room for it. Return 1 upon success and 0 upon failure.
- `clear`: This function will clear the elements of the stack, setting them all to -1. Return 1 upon success and 0 upon failure. Don't forget to reset the current_size var!
- `is_full`: This function returns 1 if the stack is full and 0 otherwise
- `is_empty`: This function returns 1 if the stack is empty and 0 otherwise
- `has_atleast_two_nums`: This function returns 1 if there are 2 or more numbers within the stack and 0 otherwise. This function helps with checking for a valid equation. For instance, if only one or no numbers are on the stack and an operand is seen, then the equation is invalid.

2. Your second task is to implement the code at the TODO in main.c. This code is responsible for the pushing and popping onto your stack as well as computing the result if an operator is seen.
3. Your last task is to write the result to a new file 'output.txt'. If the input file's contents are:

   9 3 +

   4 5 + 8 *

   Then your output file's contents should be:

   9 3 + = 12

   4 5 + 8 * = 72

You will have to find the locations to place your file writing code as it will be variable to your implementation in main.c.

Note that an example 'input.txt' is provided; however, we expect you to test your own equations to confirm the robustness of your stack and file I/O code.

Testing

We have provided a test file `test-stack.c` in this repository containing some test cases.

BONUS (10 pts): You can write your own test file, `test-stack-<your netid>.c`, which contains 10 additional cases. Note that you need to put your netid in the Makefile for your test cases to run. You will need to also delete the comments below the TODO in the Makefile that comment out your own personal tests from running. These test cases should ensure your code works as intended for all possible system configurations. You should include `assert.h` to validate system behavior. This page outlines some good testing practices. Your tests should not duplicate the instructor-provided test file.

To run your code, simply type 'make stack'. To run tests, type 'make test'. Note that you will need to delete the comment pound signs (or hashtags) in the Makefile under make test if you want to run your own personal tests. And last but not least, type 'make clean' to clean up the files that make created.

It might be helpful to copy the existing test file to your own personal test file and modify the assertions from there. The functions that have still yet to be tested are clear(), is_full(), is_empty(), and has_atleast_two_nums(). For the bonus points, your test cases should test all of the previous functions and other edge cases such as trying to push to the stack when it is already full.

Implementation Tips

Here are some tips on how to implement this system:

- I recommend you implement your system in the following order:
  1. Write the implementation of the stack
  2. Run the test file and ensure the basic functionality of your stack. If you choose to write extra tests for bonus, write your test file to ensure the robustness of your implemented stack (meaning test edge cases)
  3. Implement the specified functionality in main.c to calculate the result of the RPN equations
  4. Add in the file writing code to output the result from the equations to a new file 'output.txt'

Deliverables and Grading

- Push your code to GitHub (a link will be posted on Blackboard) before the deadline.
- All of your code (including your tests) should compile with the following `gcc` flags: `-Wall -Wextra -Werror -pedantic`. You will lose credit if your code compiles, but only without those extra flags.
- Be sure to write clear and concise commit messages outlining what has been done.
- Write clean and simple code, using comments to explain what is not intuitive. If the grader cannot understand your code, you will lose credit on the assignment.
- Be sure your code compiles! If your program does not compile, you will receive **no credit**. It is better to submit a working program that only does a subset of the requirements than a broken one that attempts to do them all.

Table 1: Grading Rubric

Category	Percentage
Correct RPN Results	30%
Instructor Tests Pass	30%
RPN Results Written to File	30%
Compilation with All Flags in Makefile	10%
BONUS: Student Tests Pass (10 cases minimum)	5%
BONUS: Student Tests have Good Quality	5%