CS 2110 Final Exam: C

Your 2110 TAs

Fall 2021

Contents

Please take the time to read the entire document before starting the assignment. It is your responsibility to follow the instructions and rules.

1 Rules - Please Read

You are allowed to submit this portion of the final exam starting from the moment your exam period begins until your individual period ends. You have 2 hours and 50 minutes to complete *all* portions of the exam, unless you have accommodations that have already been discussed with your professor. Gradescope submissions will close precisely at the end of your allotted exam period. **You are responsible for watching your own time.**

If you have questions during the exam, you may ask the TAs for clarification, though you are ultimately responsible for what you submit. The information provided in this document takes precedence. If you notice any conflicting information, please indicate it to your TAs.

All sections of the final exam are open-resource. You may reference your previous homeworks, class notes, etc., but your work must be your own. Contact in any form with any other person besides a TA is absolutely forbidden. **No collaboration is allowed.**

2 Overview

2.1 Description

Please read this entire document before starting.

You have been given three C files - **stack.c, stack.h, and main.c**. (`main.c` is only there if you wish to use it for testing; the autograder does not read it). For `stack.c`, you should implement the `create_stack()`, `stack_push()`, and `stack_pop()` functions according to the comments. Optionally, if you want to write your own manual tests for your program, you can modify `main()` in `main.c`. The entire assignment must be done in C. Please read all the directions to avoid confusion.

THERE ARE NO CHECKPOINTS; you can implement the functions in any order you want. Each function can be implemented independently, so you can get full credit for any function without getting credit for any other function. If you think a function is difficult to implement, you can save it for later and work on a different function.

3 Instructions

For this section of the final exam, you shall be writing a function that creates a stack data structure using dynamic memory allocation: `create_stack()`; a function that pushes a value to a stack: `stack_push()`; and a function that pops a value from a stack: `stack_pop()`.

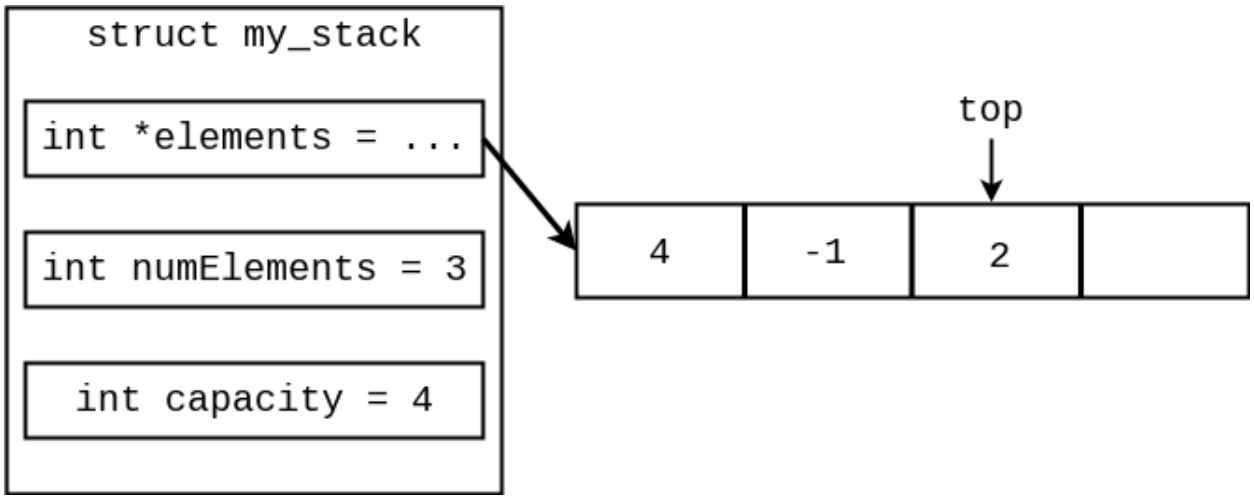
You should **not** modify any other files. Doing so may result in point deductions. You should also **not** modify the `#include` statements nor add any more. You are also not allowed to add any global variables. You may however add macros and helper functions as long as they are in `stack.c` and pass the autograder.

3.1 The `my_stack` struct

The three functions you will implement use the provided `my_stack` struct in `stack.h` to represent our stack data structure.

A `struct my_stack` contains `int *elements`, a dynamically allocated array of ints that stores the elements currently in our stack; `int numElements`, the number of elements currently in our stack; and `int capacity`, the current allocated size of our `elements` array.

Refer to the following for a visual representation of the data structure, where "..." denotes an unknown pointer address (pointing to some memory on the heap):



In this example, `stack_pop()` would pop the element `2` from the stack, and `stack_push()` would push the new element into index 3 (0-based) on the stack.

3.2 `create_stack()`

This function dynamically allocates space on the heap for the `my_stack` struct and its `elements` array. The function also initializes the `numElements` and `capacity` member variables and returns the allocated `my_stack` struct.

```
struct my_stack *create_stack(void);
```

- `elements` should be dynamically allocated to store 1 element.
- `numElements` should be initialized to 0 and `capacity` should be initialized to 1.

- If dynamic memory allocation fails at any point, free the `my_stack` struct and all dynamically allocated variables within the struct and return NULL.
- If the function is successful, return `SUCCESS`.

3.3 `stack_push()`

This function takes in a pointer to the `my_stack` struct and the integer value to push onto the given `my_stack`. If the stack doesn't have enough capacity to add another element, the function doubles the capacity of the given `my_stack`. The function also updates the `numElements` and capacity variables.

```
int stack_push(struct my_stack *stack, int data);
```

- If the passed in stack or its element array is `NULL`, do not free any dynamically allocated memory and return FAILURE.
- If the stack would exceed its capacity by adding another element, double the capacity of the stack, and update the `capacity` variable to match.
- `numElements` should be incremented.
- If dynamic memory allocation fails at any point, free the `my_stack` struct and all dynamically allocated variables within the struct and return FAILURE.
- If the function is successful, return `SUCCESS`.

3.4 `stack_pop()`

This function takes in a pointer to the `my_stack` struct and a pointer to variable where you should store the popped element's value. The function also updates `numElements`.

```
int stack_pop(struct my_stack *stack, int *dataOut);
```

- If the passed in stack, its elements, or `dataOut` is `NULL`, do not free any dynamically allocated memory and return FAILURE.
- `numElements` should be decremented.
- If the stack is empty, free the `my_stack` struct and all dynamically allocated variables within the struct and return FAILURE.
- If the function is successful, return `SUCCESS`.

4 Grading

Point distribution for this portion of the final exam is broken down as follows:

- `create_stack` (4 points):
  - `elements` should be dynamically allocated to store 1 element.
  - `numElements` should be initialized to 0 and `capacity` should be initialized to 1.
  - If dynamic memory allocation fails at any point, you should free the `my_stack` struct and all dynamically allocated variables within the struct and return NULL.
  - If the function is successful, you should return `SUCCESS`.
- `stack_push` (8 points):
  - If the passed in stack or its element array is `NULL`, you should free any dynamically allocated memory and return FAILURE.
  - If the stack would exceed its capacity by adding another element, you should double the capacity of the stack, and update the `capacity` variable to match.
  - `numElements` should be incremented.
  - If dynamic memory allocation fails at any point, you should free the `my_stack` struct and all dynamically allocated variables within the struct and return FAILURE.
  - If the function is successful, you should return `SUCCESS`.
- `stack_pop` (8 points):
  - If the passed in stack, its elements, or `dataOut` is `NULL`, you should not free any dynamically allocated memory and return FAILURE.
  - `numElements` should be decremented.
  - If the stack is empty, you should free the `my_stack` struct and all dynamically allocated variables within the struct and return FAILURE.
  - If the function is successful, you should return `SUCCESS`.

5 Deliverables

Turn in the following files on Gradescope during your assigned final exam period.

1. stack.c

Your file must compile with our Makefile, which means it must compile with the following gcc flags:

```
-std=c99 -pedantic -Wall -Werror -Wextra -Wstrict-prototypes -Wold-style-definition
```

All non-compiling final exam submissions will receive a zero. If you want to avoid this, do not run gcc manually; use the Makefile as described below.

6 Autograder and Debugging

6.1 Makefile

We have provided a Makefile for this final exam section that will build your project.
Here are the commands you should be using with this Makefile:

1. To clean your working directory (use this command instead of manually deleting the .o files): `make clean`
2. To compile the code in `main.c`: `make stack`
3. To compile the tests: `make tests`
4. To run all tests at once: `make run-tests`
   - To run a specific test: `make run-tests TEST=test_name`
5. To run all tests at once with Valgrind enabled: `make run-valgrind`
   - To run a specific test with Valgrind enabled: `make run-valgrind TEST=test_name`
6. To debug a specific test using gdb: `make run-gdb TEST=test_name`

   Then, at the `(gdb)` prompt:

   (a) Set some breakpoints (if you need to—for stepping through your code you would, but you wouldn't if you just want to see where your code is segfaulting) with `b suites/stack_suite.c:43`, or `b stack.c:39`, or wherever you want to set a breakpoint

   (b) Run the test with `run`

   (c) If you set breakpoints: you can step line-by-line (including into function calls) with `s` or step over function calls with `n`

   (d) If your code segfaults, you can run `bt` to see a stack trace

To get an individual test name, you can look at the output produced by the tester. For example, the following failed test is `test_create_stack_normal`:

```
suites/stack_suite.c:45:F:test_create_stack_normal:test_create_stack_normal:0:
                         ^^^^^^^^^^^^^^^^^^^^^^^^
```

Beware that segfaulting tests will show the line number of the last test assertion made before the segfault, not the segfaulting line number itself. This is a limitation of the testing library we use. To see what line in your code (or in the tests) is segfaulting, follow the "To debug a specific test using gdb" instructions above.

6.2 Debugging with GDB - List of Commands

Debug a specific test:

```
$ make run-gdb TEST=test_name
```

Basic Commands:

- `b <function>` **break point** at a specific function
- `b <file>:<line>` **break point** at a specific line number in a file
- `r` **run** your code (be sure to set a break point first)
- `n` **step over** code
- `s` **step into** code
- `p <variable>` **print** variable in current scope (use `p/x` for hexadecimal)
- `bt` **back trace** displays the stack trace (useful for segfaults)

6.3 Autograder

We have provided you with a test suite to check your work. You can run these using the Makefile.

Note: There is a file called `test_utils.o` that contains some functions that the test suite needs. We are not providing you the source code for this, so make sure not to accidentally delete this file as you will need to redownload the assignment. This file is not compiled with debugging symbols, so you will not be able to step into it with gdb (which will be discussed shortly).

We recommend that you write one function at a time and make sure all of the tests pass before moving on the next function. Then, you can make sure that you do not have any memory leaks using Valgrind. It doesn't pay to run Valgrind on tests that you haven't passed yet. Below, there are instructions for running Valgrind on an individual test under the Makefile section, as well as how to run it on all of your tests.

The given test cases are the same as the ones on Gradescope. We formally reserve the right to change test cases or weighting after the lab period is over. However, if you pass all the tests and have no memory leaks according to Valgrind, you can be confident that you will get 100% as long as you did not cheat or hard code in values.

Printing out the contents of your structures can't catch all logical and memory errors, which is why we also require you run your code through Valgrind. You will not receive credit for any tests you pass where Valgrind detects memory leaks or memory errors. Gradescope will run Valgrind on your submission, but you may also run the tester locally with Valgrind for ease of use.

We certainly will be checking for memory leaks by using Valgrind, so if you learn how to use it, you'll catch any memory errors before we do.

Your code must not crash, run infinitely, nor generate memory leaks/errors.

Any test we run for which Valgrind reports a memory leak or memory error will receive no credit.

If you need help with debugging, there is a C debugger called gdb that will help point out problems. See instructions in the Makefile section for running an individual test with gdb.

6.4 Valgrind Errors

If you mishandling memory in C, chances are you will lose half or all of a test's credit due to a Valgrind error. You can find a comprehensive guide to Valgrind errors here: `https://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs`

For your convenience, here is a list of common Valgrind errors:

- Illegal read/write: this happens when you read or write to memory that was not allocated using malloc/calloc/realloc. This can happen if you write to memory that is outside a buffer's bounds, or if

you try to use a recently freed pointer. If you have an illegal read/write of 1 byte, then there is likely a string involved; you should make sure that you allocated enough space for all your strings, including the null terminator.

- Conditional jump or move depends on uninitialized value: this usually happens if you use malloc or realloc to allocate memory and forget to intialize the memory. Since malloc and realloc do not manually clear out memory, you cannot assume that it is full of zeros.
- Invalid free: this happens if you free a pointer twice or try to free something that is not heap-allocated. Usually, you won't actually see this error, since it will often cause the program to halt with an Aborted signal.
- Memory leak: this happens if you forget to free something. The memory leak printout should tell you the location where the leaked data is allocated, so that hopefully gives you an idea of where it was created. Remember that you must free memory if it is not being returned from a function. (Think about what you had to do for empty_list in HW9!)