CS 2110 Final Exam: C

Your 2110 TAs <3

Summer 2022

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 timed lab portion of the exam period begins until your individual period ends. You have 75 minutes to complete *both of the timed lab (C and LC3-Assembly)* 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. *Note:* The Assembly and C coding sections are independent of each other so you may complete these in any order, therefore there is no enforced time limit to complete either section. Just be mindful of the time you have left!

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.

The timed lab 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

You have been given two C files - `stack.c` and `stack.h`. For `stack.c`, there is one function that you need to complete according to the comments and decriptions given below.

3 Instructions

For this section of the final exam, you will **manipulate a stack of strings**. You will be writing a function that pushes some data onto a stack: `push()`. At this point, it would be recommended to open both `stack.c` and `stack.h` to preview the files, and better understand the following sections.

Remember: 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 written in `stack.c` and pass the autograder. You will not be submitting `stack.h`, so any modifications in `stack.h` will not be reflected in the Gradescope autograder.

3.1 `push()`

```
int push(struct my_stack *stack, char *data);
```

This function takes in a pointer to `struct my_stack * stack` and `char *data` that comprises the information in a `struct data_t` element.

The function pushes `data` onto the top of `stack`. Our implementation of the stack uses the provided `struct my_stack` declared in `stack.h` to represent our array-backed stack data structure. For reference, the top of the stack is the *furthest back* of the array– with the highest (used) index.

The function also updates `stack`'s `numElements` variable upon a successful push.

3.2 Success or Failure

- If the passed `struct my_stack * stack`, its pointer to `elements` is `NULL`, or the passed in `char *data` is `NULL`, return FAILURE.
- If the stack would exceed its capacity by adding another element, return `FAILURE`.
- If dynamic memory allocation fails at any point, you should return FAILURE.
- If dynamic memory allocation is a success you should create the data struct and push the data onto the stack (this will be the new top of the stack) and `numElements` should be incremented. *Note:* the stack has an array of actual `struct data_t` elements not pointers to `struct data_t` elements, as further detailed in section 3.4.
- If the function is successful, return `SUCCESS`.

3.3 Useful Structs

A `struct data_t` contains `int length`, the length of the string **(not including the null terminator)**; and `char *data`, a string **whose characters are on the heap**.

A `struct my_stack` contains `struct data_t *elements`, a pointer to **element zero** of the stack, which is considered to be **the bottom of the stack**; `int numElements`, the number of elements currently in our stack; and `int capacity`, the **statically allocated** size of our `elements` array. (*more on this in the next section*)

3.4 Dynamic memory allocation?

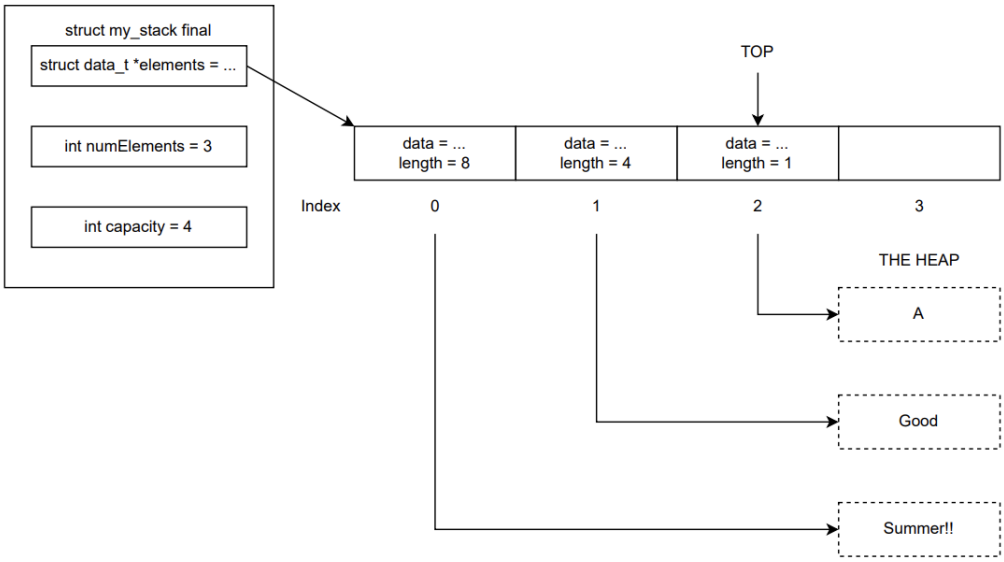
For this implementation of the stack, space for the backing array will be statically allocated by the autograder. If you would like to see how, you may reference the provided test file `stack_suite.c`. That means there are *two important implications*: **(1)** the correct space for the stack is statically allocated during compile time **(2)** you should not re-allocate / modify the backing array's allocation each time you push an element. For example, if the capacity of the stack is 3, the stack's elements array will be allocated enough space for `sizeof(3 * struct data_t)`. We should use that space as-is and not modify it. If the stack doesn't have enough capacity to add another element, the function should return `FAILURE`.

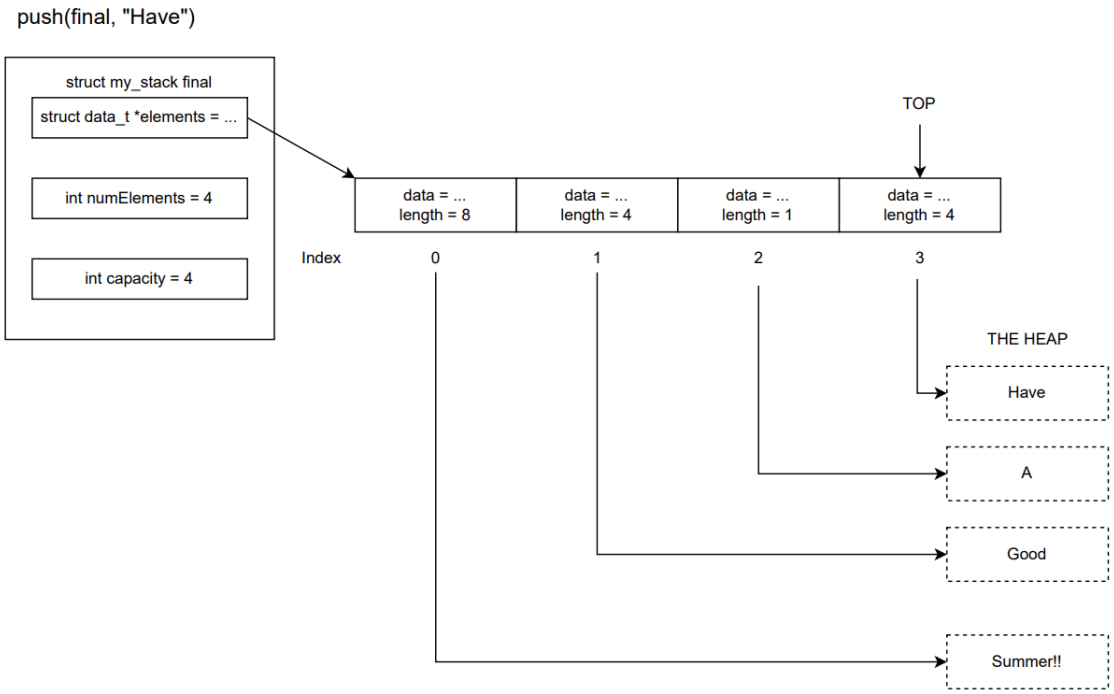
However, you **will** need dynamic memory allocation for some aspect of this assignment. Refer back to section 3.3: Useful Structs, if you are unsure. If dynamic memory allocation fails at any point, you should return `FAILURE`.

3.5 Diagram

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, the data will contain the string `"Have"`, and the length of the data which is 4, and push it onto index 3 of the stack. In the second diagram, you can see the results of `push(final, "Have")`. *Reminder*: `elements` points to the first element of the array-backed stack.

After ensuring that the stack and the data are not `NULL`, we had check to see if adding another element would make us go over the capacity. We know that adding the new element will not make us go over the capacity, so we use the `struct data_t` at the 'end' (labelled `TOP`) of the array to store the the newly allocated string as well as update the length appropriately.





4 Grading

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

- `push` (25 points):
  - If the passed in stack or its elements array is `NULL`, or the passed in data is `NULL`, you should return FAILURE.
  - If the stack would exceed its capacity by adding another element, you should return `FAILURE`.
  - If dynamic memory allocation fails at any point, you should return `FAILURE`.
  - If dynamic memory allocation does not fail, then the data should be pushed onto the stack and `numElements` should be incremented.
  - 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 the function according to the pdf and file comments to ensure that you are meeting all the requirements before moving onto testing and debugging. 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 Signal 6 Aborted message.
- 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!)