CS 2110 Timed Lab 4: C and Dynamic Memory Allocation

Sophie Imhof, Adrien Cao, Eric Stuhr, Richard Zhang

Version 1.0

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 Timed Lab Rules - Please Read

You are allowed to submit this timed lab starting from the moment your assignment is released until your individual period is over. You have 75 minutes to complete the lab, unless you have accommodations that have already been discussed with your professor. Gradescope submissions will remain open for several days, but you are not allowed to submit after the lab period is over. **You are responsible for watching your own time. Submitting or resubmitting after your due date may constitute an honor code violation.**

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

The timed lab is 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 for timed labs.**

2 Overview

2.1 Description

Office hours for CS 2110 are often very crowded. To manage this complexity, the TAs have tasked you with building a queueing system. When a student arrives at office hours, they are added to the end of the line. When a TA is ready, they remove the person at the front of the line and service them. Thus, the office hours queue is a first-come-first-served structure, or alternatively a first-in-first-out structure.

To implement this queue, you will use a singly-linked list. That means each node in the list contains: *(1)* data for a student, and *(2)* a pointer to the next node in the list (or `NULL` if it's the last node).

In order to help you manage the data structure, we have provided two global pointers. There is `queue_head` for the 'head' (frontmost) node and `queue_tail` for the 'tail' (backmost) node.

Note that dynamic memory allocation is required for this implementation. It's unknown a priori how many students will be in the queue at any given time. You have to be able to allocate nodes on the fly.

For this queue, you shall be writing two functions that will aid TAs with keeping track of students on the queue: `queue_add()` and `queue_remove()`.

Now that we have our OH queue, we still have one more issue. We are not allowed to display student's real name on the queue due to privacy reasons. The 2110 TA's have decided they need you to implement two more functions: `encryptName()` and `decryptName()`. This encryption will be done using an XOR encryption and decryption method (See instructions for more details).

After testing the office hour queue, the TA's now want the option to append a student's last name to a queue node. To do this, please implement the `add_last_name()` function.

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

You have been given three C files - **tl04.c, tl04.h, and main.c** (`main.c` is only there if you wish to use it for testing; the autograder does not read it). For `tl04.c`, you should implement the `queue_add()`, `queue_remove()`, `encryptName()`, `decryptName()`, and `add_last_name()` functions according to the comments. Optionally, if you want to write your own manual tests for your program, you can implement `main()` in `main.c`. You are allowed to use standard string functions in `string.h` (e.g. strlen(), strncpy())

You should **not** modify tl04.h. 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.

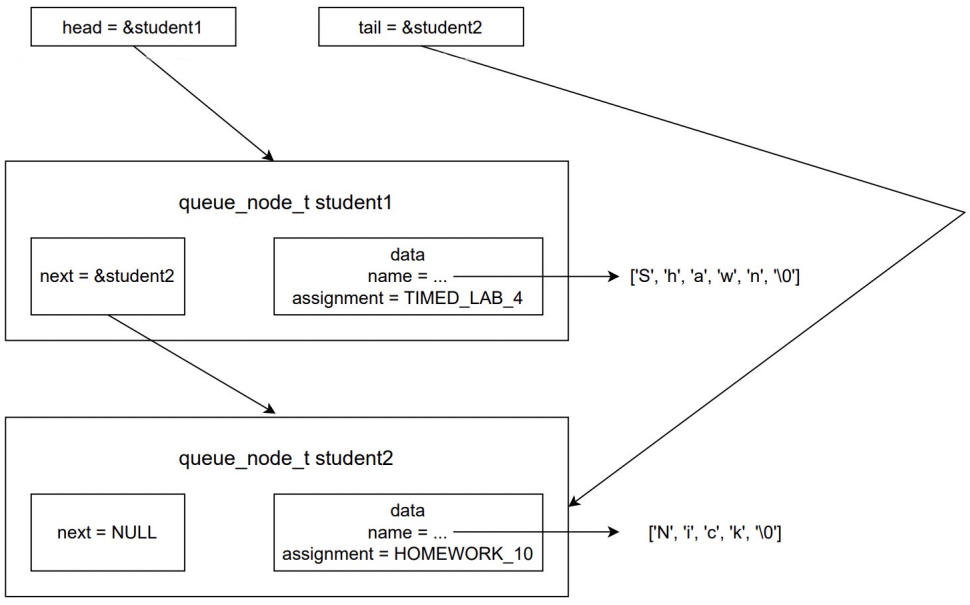
3.1 Structs and Global Variables

This timed lab involves two structs. The first struct that you will work with is `struct student_t`. This type represents individual student's data. Each student has a string for their name (`char* name`), as well as the assignment (`enum assignment_t`) that they need help on. For example, a student might be named `"George P. Burdell"` and need help on `"HOMEWORK_1"`. **Note: The student's name is stored somewhere else in memory, as indicated in the diagram.**

The other struct that you will work with in this assignment is `struct queue_node_t`. It is a *wrapper* which represents the actual node in the queue. That means we need to combine the `struct student_t` along with a pointer to the next node in the list into a single `struct queue_node_t`.

There are two global variables that act as pointers to ends of the singly-linked list. These pointers keep track of where the singly-linked list is in memory. The [queue_head] pointer points to the first node of the list, and [queue_tail] likewise points to the last. **Intially, both of these pointers are 'NULL'. The list is initially empty, so there is no first or last node.**

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



3.2 Writing `queue_add()`

This function will be called by client code to add a student to the end of the queue. (*Think about what this means for our next node pointer*) The caller will supply the data of the student to add. This function should allocate a `queue_node_t` on the heap, and deep-copy all the other data. **Any pointers in the `queue_node_t` or `student_t` will require their own dedicated memory allocation**. This function should return `SUCCESS` if the student was added successfully. If it fails, it should return `FAILURE` and leave the list unchanged. It should fail if and only if:

- `malloc` fails,
- the student's name is `NULL`, or
- the student's name is an empty string.

Remember to set all relevant fields of `struct queue_node_t` and properly insert the student in the queue with the help of the `queue_head` and/or `queue_tail` pointers. Refer back to the diagram for specific details about how to link the nodes.

3.3 Writing `queue_remove()`

This function will be called by client code to remove a student from the front of the queue. It will return whether a student was removed successfully, and the data removed in that case.

The way this function returns the student data is by using the data out technique. This is to get around the limitation that functions may only have one return value. As such, the caller will pass in a pointer where the student's data should be stored. Then this function will store the returned data at that pointer. Independently, it returns whether it succeeded via the normal path.

Finally, remove the student in the queue with the help of the `queue_head` and/or `queue_tail` pointers. Refer back to the PDF/diagram for specific details about how to link the nodes, and consider any edge cases.

Remember to free the node when you're done. A copy of the data is being returned, the node containing the data is not. The node will no longer be accessible when the function returns, so it should be freed.

If this function succeeds, it should return `SUCCESS` and modify `'*data'` to be the data of the student removed. If it fails, it should return `FAILURE` and leave both the list and `'*data'` unchanged. It should fail if and only if:

- data is `NULL`, or
- the list is empty.

Remember to free any unused data.

3.4 XOR Encryption

The XOR Encryption algorithm is based on applying an XOR mask using the plain text and a key. Reapplying the same XOR mask (using the same key) to the cipher text outputs the original plain text.

Example:

This example demonstrates how to encrypt an integer with an XOR key and decrypt the result back to the same integer by using the same XOR key.

1. Encrypt an integer:
    - plain text = 45 (binary: 0b101101)

- key = 11 (binary: 0b001011)
- plain text ^ key = 0b101101 ^ 0b001011 = 0b100110
- cipher text = 0b100110 = 38

2. Decrypt an integer:
   - cipher text = 38 (binary: 0b100110)
   - key = 11 (binary: 0b001011)
   - cipher text ^ key = 0b100110 ^ 0b001011 = 0b101101
   - plain text = 0b101101 = 45

3.5 Writing `encryptName()`

This function will be called by client code to encrypt a student's name for the purpose of anonymity. The function should encrypt every character in the student's real name to an encrypted integer value.

To encrypt each character in a students' name:

- convert the character to its integer ASCII value
- XOR the ASCII value with the `xor_key` passed into the function
- store the result in the respective position the int array (`ciperName`)

The function should return `SUCCESS` if the name was successfully encrypted. If the function fails, it should return `FAILURE`. It should fail if and only if:

- the `plainName` is `NULL`, or
- the `cipherName` is `NULL`

3.6 Writing `decryptName()`

This function will be called by client code to decrypt a student's name so a TA can identify the student who needs help. The function should decrypt every integer in the cipher name to it's decrypted character value. The number of integers to decrypt is given by the `plainNameLength` parameter.

To decrypt each integer in a students' cipher name:

- XOR the integer with the `xor_key` passed into the function
- convert the integer to its ASCII character
- store the resulting character in the respective position the string (`plainName`)

The function should return `SUCCESS` if the name was successfully encrypted. If the function fails, it should return `FAILURE`. It should fail if and only if:

- the `plainName` is `NULL`, or
- the `cipherName` is `NULL`

3.7 Writing `add_last_name()`

This function appends a last name to the student's name in a queue node. The function takes in two parameters: a pointer to a `queue_node_t` holding a specific student's data and a pointer to the `lastName` to be appended. The `queue_node_t` passed into the function only has enough space for the student's first name. You must first append a space character before appending the last name (See function comments for an example). *Think about what function we can use to change the size of a previously reserved memory block.* The function should return `SUCCESS` if the last name was successfully appended. Please assume that the queue node parameter is not null. If the function fails, it should return `FAILURE`. It should fail if and only if:

- `malloc` or `realloc` fails,
- `last_name` is `NULL`, or
- `last_name` is an empty string.

3.8 Testing your program with `main()`

`main()` can be used to test all of the functions that you've written so far. You can set up your own test cases and check that everything is working.

4 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

5 Rubric and Grading

5.1 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 half or no credit(depending on the test).

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.

5.2 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, or if it is not attached to a valid ascii_image struct. (Think about what you had to do for empty_list in HW9!)

5.3 Makefile

We have provided a Makefile for this timed lab 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 tests: `make tests`
3. To run all tests at once: `make run-tests`
   - To run a specific test: `make run-tests TEST=test_name`
4. 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`
5. 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/tl4_suite.c:420`, or `b tl04.c:69`, 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
6. To compile the code in `main.c`: `make tl04`

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

```
suites/tl4_suite.c:50:F:test_set_character_basic:test_set_character_basic: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 Deliverables

Please upload the following files to Gradescope:

1. tl04.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 timed labs will receive a zero. If you want to avoid this, do not run gcc manually; use the Makefile as described below.

Download and test your submission to make sure you submitted the right files!