CS 2110 Homework 9
Dynamic Memory

Jason Ng, Udit Subramanya, John Ever, Jessi Chen

Summer 2022

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to introduce you to dynamic memory allocation in C.

Some questions you will be able to answer by the end of this homework include:

- How do we allocate memory on the heap?
- How do we de-allocate it when it is no longer used?
- How do we manually manage memory?
- When do we allocate memory?
- When do we free pointers?

1.2 Task

In this assignment, you will be implementing a **circular** singly-linked list data structure. Your linked list nodes will have data contained in a union (holding either type `struct student*` or type `struct instructor*`). You can find details about these structs and their fields in `list.h`. You will write functions to add, remove, mutate, and query the data stored within the **circular** singly-linked list data structure. It is highly recommended to view `list.h` and understand the data and functions you will be using in this assignment.

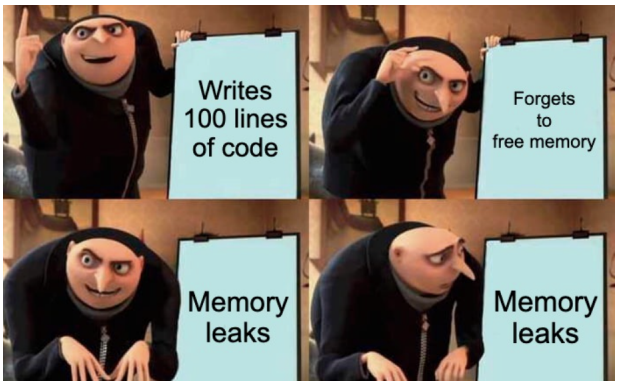
You will be writing code in

1. `list.c`
2. `main.c`

The `main.c` file is included only for your own testing purposes.

1.3 Criteria

You will be graded on your ability to implement the linked list data structure properly. Your implementation must be as described by each function's documentation. Your code must manage memory correctly, meaning it must be free of memory leaks. Remember, a memory leak is a when a block of memory is allocated on the heap, and never de-allocated from the heap before the program loses all references to that block of memory. The efficiency of your operation (time complexity) will not affect your grade.



2 Instructions

2.1 Implementation Overview

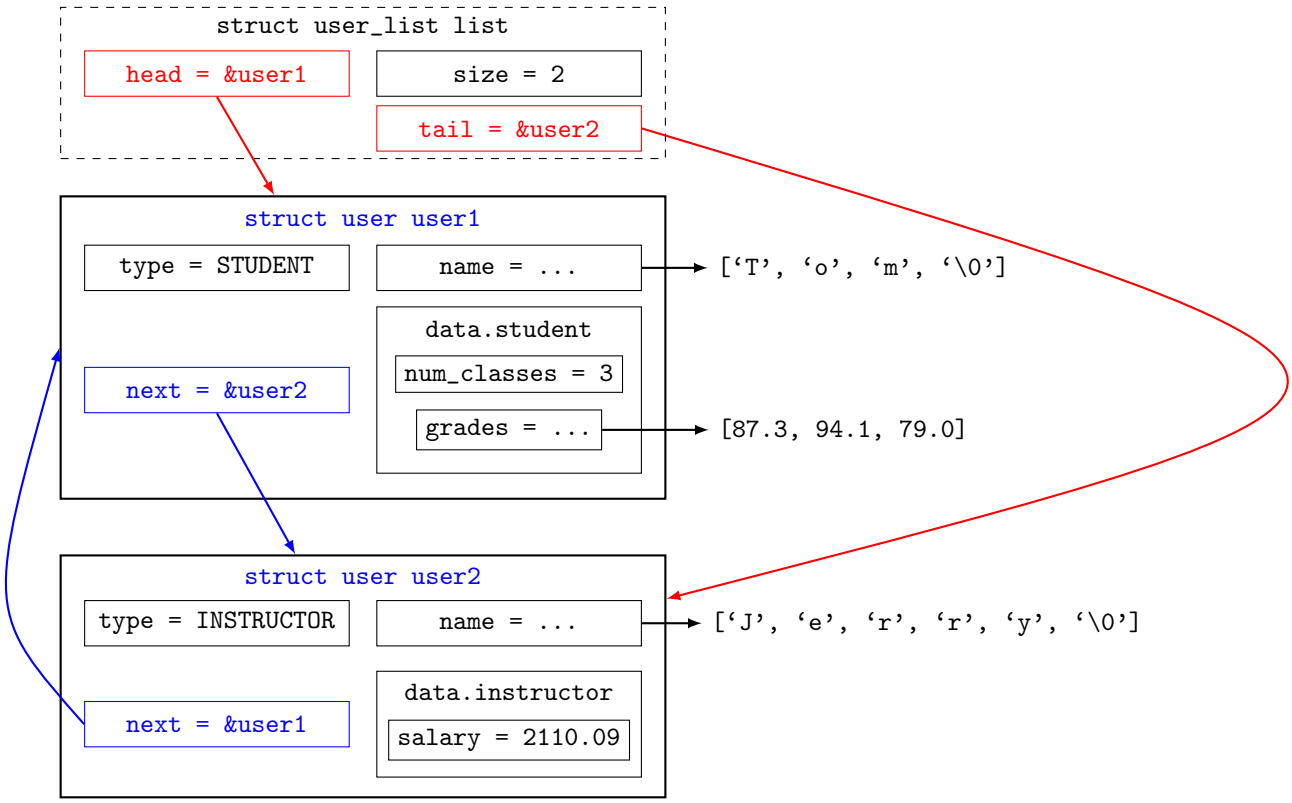
In `list.c` you will complete the implementation for the function prototypes found in `list.h`. Each function has a block comment that describes its expected behavior. A driver file, `main.c` can be *optionally* used to manually test your code. Be sure **not** to modify `tests.c`. For your convenience a copy of the tests run by the autograder can be found in `suites/list_suite.c`

You are given a `struct user_list`, which contains the meta-data for the linked-list. It has a `head` pointer that points to the first node in the list, a `tail` pointer that points to the last node in the list, and a `size` which represents the length of the linked list.

The nodes of the linked list are of type `struct user`. Consult `list.h` for a detailed explanation of each member within this node.

Note that the linked list is circular so the tail node's next pointer will point to the head. If you have not taken a data structures course yet, or could benefit from a refresher, please take some time to understand what circularity means in a linked list, and how to maintain circularity while modifying the data structure. Many of the functions you have to implement in this homework rely on your understanding this data structure. *To get you thinking*: some edge cases to consider are if there are no elements in the linked-list, or if there is only one element in the linked list.

2.2 Diagram



Above you will find a diagram expressing the structure of the circular singly-linked list described thus far. Here is a list of important observations you should make:

- the `user_list` simply contains meta-data for the linked list, and is not part of the linked list itself. That is why it is given a dashed outline
- the actual nodes of the linked list are given in blue. Circularity is represented through the use of blue arrows. They indicate that the tail points to head and the head points to the first node in the list, followed by any subsequent nodes.
- "..." denotes an unknown pointer address (pointing to some memory allocated elsewhere on the heap). We call this unknown since at the time that the containing struct is malloced, it only creates enough space for a pointer. If we want to make that pointer useful, then we need to make an additional allocation on the heap for it. Consider `struct user` for example: when it is malloced, `char *name` only has enough space for a `char *` to some string of characters. The actual characters have to be malloced elsewhere since they may take up more "`sizeof`" than a `char *`. To think about this from another perspective, the `sizeof(struct user)` does **NOT** depend on the `strlen(name)`, since the actual string lives elsewhere on the heap. `user` simply points to `name`. A similar concept applies for the grades array found in `struct student`. Understand that the choice to draw these arrays, is intended to visualize this concept. Forgetting to free this data when it is removed can cause memory leaks (memory that is no longer being used, but is never freed). Freeing memory when it should be returned to the user can create dangling pointers (memory that's freed but still being referenced), causing use-after-free bugs. Keep both of these in mind when writing your code. It might be a good idea to leave your self a note about this while you are implementing functions that remove nodes from the linked list.
- The data stored inside the `union user_data data` depends on whether the user is a student or instruc-

tor. To determine what type of user is stored inside the union, each user also has an `enum user_type type`.

- For `user1`, the union field 'instructor' contains garbage data and for `user2`, the union field 'student' contains garbage data, so they are omitted from the diagram. If you are unsure where this garbage data comes from, it is worth reviewing the definition of this union in `list.h` and understanding how a union is laid out in memory.

2.3 Data Out Concept

Throughout this assignment, you will see parameters called `dataOut`. Remember C is pass by value by default. That means if we want to pass information into a function, modify it, and see the modifications outside the scope of the function we need to emulate *pass-by-reference* via pointers. The concept of dataOut is just passing a certain parameter of a function by reference. If you unsure how pass by reference works, then it is advised that you revise that material since it is a huge aspect of this assignment. It is common convention in C programming to reserve the return value of a function to be a error/status code. In this assignment the two status codes we have are `SUCCESS` and `FAILURE`, aptly named to indicate how a function has returned. That is why certain functions such as `pop_front` have an int return type, while the intent of these functions is to remove a linked list node.

2.4 Implementation Tips

- Once you've implemented the functions in the `list.c` file, or after implementing a few, compile your code using the provided Makefile. You may test your functions manually by writing your own test cases in the provided `main.c`, or run the autograder's test suite. See the Testing section below for more information.
- If you do no know where to begin with implementing a function, read through the comments prior to it and consult the entirety of this pdf too.
- Remember we are not grading for time complexity. For example, we will **NOT** dock points for completing a method in `O(n)` time when there is an `O(1)` way to complete it. However, we are grading for memory safety, so if you are using the local autograder, you have to run the valgrind tests to make sure there are no memory issues. Details about this are given in section 2.5.
- Helper functions: There are three helper functions defined in `list.c`: `create_student`, `create_instructor`, `student_equal`. Please do not make any modifications to them, but rather use them to your advantage in other functions that you complete. NOTE: As seen in `list.c`, the helper functions are static, which means they are not part of the file's public interface and therefore will not be tested by the autograder. **Improper implementations of these helpers may cause other tests to fail**, so be sure to check them if your other functions fail any tests.
- If your code is segfaulting, it is most likely due to dereferencing a pointer which is NULL. Therefore it is advised to NULL check any pointers before using them.
- Remember that the struct data passed to certain functions (`struct user *data`) is malloc-ed data.
- Push/add functions: For these functions, make sure to read the documentation about what to do when `malloc` fails. In most cases, this means that you need to return 1 to indicate something went wrong.
- When modifying the list, remember that there are cases when both the `tail` pointer and `head` pointer need to be updated.
- Modifying functions: For these keep in mind that you might not need as much memory as the data originally needed, or you may need more, and in either case you should reallocate memory accordingly.

- Please COMPILE OFTEN. The compiler will reveal many syntax errors that you would rather find early before making them over and over throughout your code. Waiting until the end to compile for the first time will cause big headaches when you are trying to debug code. We speak from experience when we say compile often. :)

2.5 Testing and Autograding

To compile and test your code, use the provided Makefile as detailed below. All commands should be run from inside of the CS 2110 Docker container.

2.5.1 Manual testing

To manually test specific functions in your code, fill in the `main` function in `main.c` with tests and run

```
$ make hw9
```

This will create an executable called `hw9`. Then, you can run your tests via your `main` function by running

```
$ ./hw9
```

2.5.2 Running the Autograder

To run the autograder's test suite, run:

```
# Run all test cases
$ make run-tests

# Run a specific test case
$ make run-tests TEST=test_list_contains_NULL_name
```

Note that the above only runs logical tests on your implementation. The local autograder does not test for memory errors by default, though the one on Gradescope does. To test your code for memory errors locally, you must run `valgrind`.

To check your code for memory errors using `valgrind`, run:

```
# Run all test cases
$ make run-valgrind

# Run a specific test case
$ make run-valgrind TEST=test_list_pop_front_nonempty
```

To run a specific test with GDB, run the following command:

```
# Run a specific test case
$ make run-gdb TEST=test_list_pop_front_nonempty
```

3 Deliverables

Please upload the following files to Gradescope:

1. `list.c`

4 Appendix

4.1 Rules and Regulations

4.1.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. As such, please start assignments early, and ask for help early. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

2. If you find any problems with the assignment it would be greatly appreciated if you reported them to the TAs. Announcements will be posted if the assignment changes.

4.1.2 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension, you will still turn in the assignment over Canvas/Gradescope unless instructed otherwise.

4.1.3 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative. In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

4.1.4 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code.

What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.