CS 2110 Homework 9
Dynamic Memory

Sameer Suri, Justin Hinckley, Nicholas Ade, Kevin Fu, Saahir Dhanani

Fall 2021

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to introduce you to dynamic memory allocation in C. You will also learn how to implement a singly-linked list data structure in C.

How do we allocate memory on the heap? How do we de-allocate it when it is no longer used?

1.2 Task

In this assignment, you will be implementing a singly-linked list data structure. Your linked list nodes will have struct data (of type `struct student*`). You can find details about this struct and the fields it contains in the included `list.h` file. Your linked list will be able to add, remove, and mutate, and query the data stored within it.

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.

Note: A memory leak is when a block of memory is allocated for some purpose, and never de-allocated before the program loses all references to that block of memory.

2 Instructions

2.1 Implementation Overview

You have been given one C file, `list.c`, in which to implement the linked list data structure. Implement all functions in this file. Each function has a block comment that describes its expected behavior.

Be sure not to modify any other files. Doing so will result in point deductions that the autograder will not reflect.

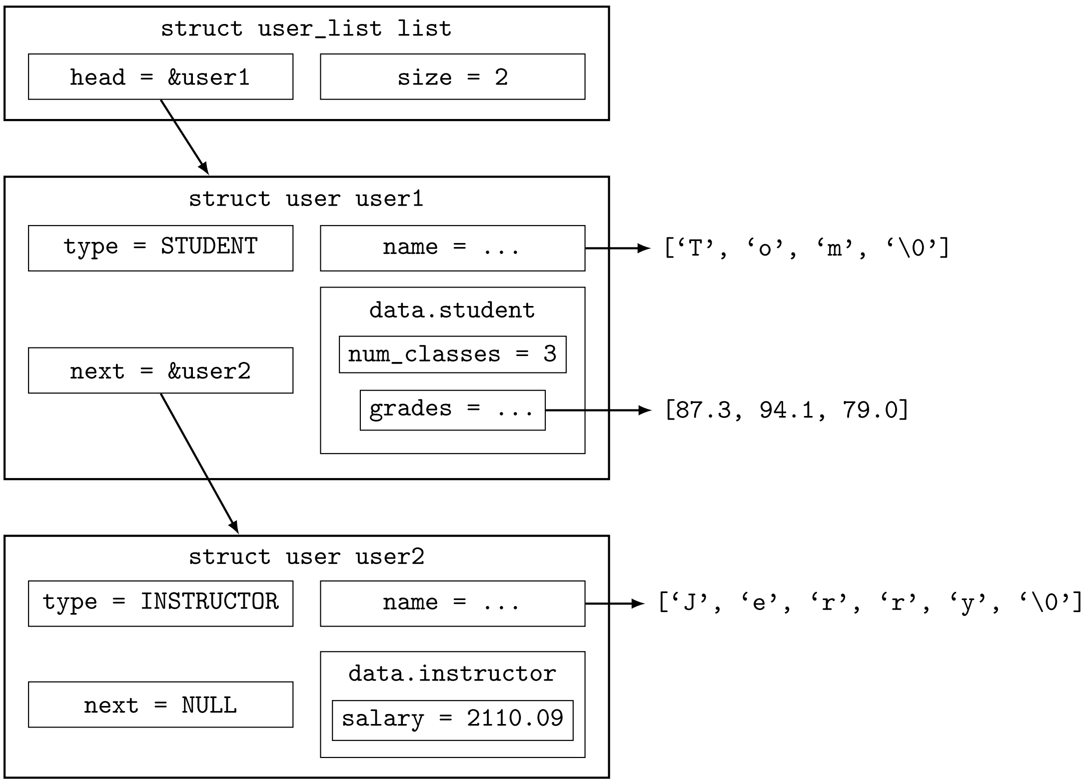
Remember that the struct data passed to certain functions (`struct user *data`) is malloc-ed data.

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 created 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.

You are given a `struct user_list`, which is a linked list with a `head` pointer that points to the first node in the list and a `size` which represents the length of the linked list. You also have a `struct user` that has a next pointer to the next user as well as a union user_data pointer that points to a memory address with all the data for that user.

The data stored inside the union user_data depends on whether the user is a student or instructor. To determine what type of user is stored inside the union, each user also has an enum user_type.

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):



Note that 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.

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.

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. :)

2.2 Implementation Tips

- Helper functions: There are three helper functions defined in list.c. You should use them to your advantage. 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.

- 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.

- Pop/remove functions: As we mentioned, you should free things that are no longer used to avoid memory leak, but you also don't want to free prematurely. Here, the potential candidates for freeing are: the nodes, the struct user data inside the node, and the pointers inside the struct user. Think carefully about what you're doing in these functions (hint: look at return type) and figure out what should be freed.
- 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.

2.3 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.

2.3.1 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
```

2.3.2 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
```

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.