CS 2110 Timed Lab 4: C and Dynamic Memory Allocation

Vy Mai, Thanasis Taprantzis, Toby Salusky, Rohan Bafna, Isaac, Jeff Shelton, Anastasiya Masalava

Spring 2024

Contents

1 Timed Lab Rules - Please Read **2**

2 Overview **2**

2.1 Description 2

3 Instructions **3**

3.1 Structs and Global Variables 3

3.2 Writing `create_student` 3

3.3 Writing `add_student_to_front` 3

3.4 Writing `remove_student_by_name` 4

3.5 Testing your program with `main()` 4

4 Debugging with GDB - List of Commands **5**

5 Rubric and Grading **6**

5.1 Autograder 6

5.2 Valgrind Errors 6

5.3 Makefile 7

6 Deliverables **7**

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. **No collaboration is allowed for timed labs.** Contact in any form with any other person besides a TA is absolutely forbidden. You are also forbidden from using any materials written with the use of machine learning software such as GitHub Copilot, or ChatGPT.

2 Overview

2.1 Description

The registrar's office has become extremely busy trying to keep track of the student registered in 2110. After thinking about the problem in office hours, the 2110 TAs determined that the simplest method of keeping track of the students is with a dynamically sized array (Arraylist). In this timed lab, you will implement an Arraylist that will be used to keep track of the students in the class.

In order to help you manage the data structure, we have provided you with a `struct Student`, holding information for each registered student, and a struct Arraylist, the dynamic array you will need to manage by adding and removing students. There is `capacity` for the size of the backing array and `current_size` for the current number of students currently held in the backing array. You will be given responsibility to change these variables as needed.

Note that dynamic memory allocation is required for this implementation. You do not know ahead of time how many elements this data structure must hold, so you will have to increase the size of the backing array at compilation (eg. when you need to exceed capacity).

For this stack, you shall be writing three functions that will aid TAs with keeping track of students on the Arraylist: `create_student`, `add_student_to_front`, and `remove_student_by_name`.

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 `create_student`, `add_student_to_front`, and `remove_student_by_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: `struct Student` and `struct Arraylist`. `struct Student` represents Student's data. Each student has a string for his name (`char* name`), as well as a float for his gpa (`float gpa`). For example, we could register a student named `"George Burdell"` with a gpa of 3.5 to our class.

The `struct Arraylist` represents the Arraylist we will use to keep track of the students. The Arraylist has a pointer to a `backing_array` of students, where we will have the actual student data in. There are also two variables. `capacity` holds the `backing_array`'s max capacity at the moment (size of array), while `current_size` keeps track of the number of students in the `backing_array`. Once `current_size` reaches `capacity` the `backing_array` needs to grow.

There are no global variables in this file.

3.2 Writing `create_student`

This function will be called by client code to create a new student record. The caller will supply a pointer to a Student struct that is to be initialized with the student's name and gpa. While the function does not need to allocate the Student struct, it may need to allocate space as it performs a deep copy of all fields in the struct. In particular, the student name's pointer will need dedicated memory allocation.

This function should return `SUCCESS` if the student record was created 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 gpa is invalid: `gpa < 0` or `gpa > 4`
- the student's name `NULL` or
- the pointer to the Student record we are attempting to change (`output`) is `NULL`.

3.3 Writing `add_student_to_front`

This function will be called by client code to create a Student record and add it to the beginning of the Arraylist. `add_student_to_front()` function has two arguments: name and gpa of a student. The function should create a new Student record given this information, and place it at the beginning of the `backing_array` of the Arraylist, and update `current_size` accordingly. If the `current_size` is already at `capacity` the function should increase the size of the `backing_array` by one by reallocating on the heap to make room for its new `capacity`.
If this function succeeds, it should return `SUCCESS`. If it fails, it should return `FAILURE` and leave `backing_array` unchanged. It should fail if and only if:

- name is `NULL`,
- the `students` Arraylist is `NULL`,
- we cannot grow `backing_array` when we need to, or
- we cannot create a Student record (same conditions as the `create_student()` function).

The `backing_array` should hold the actual Student structs, not pointers to them.

Hint: Consider creating a new Student record before shifting elements of the backing array.

3.4 Writing `remove_student_by_name`

This function will be called by client code to find the first instance of a student with the same name as the name passed in as an argument and remove him from the Arraylist. This involves freeing all elements associated within the student's record on the `backing_array` and shifting the array appropriately so that all Student records start from index 0 and there are no empty spots between any two students (ie. the Student records maintain linearity in memory).

If this function succeeds, it should return `SUCCESS`. If it fails, it should return `FAILURE` and leave `backing_array` unchanged. It should fail if and only if:

- name is `NULL`,
- the `students` Arraylist is `NULL`, or
- we cannot create a find Student record with the given name.

3.5 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 are 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!