CS 2110 Final Exam: C

Your 2110 TAs <3

Fall 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 - `team.c` and `team.h`. For `team.c`, there are two functions that you need to complete according to the comments and descriptions given below.

3 Instructions

For this section of the final exam, you will help **manage the GT football team**. You will be writing functions that manage the team roster. At this point, it would be recommended to open both `team.c` and `team.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 `team.c` and pass the autograder. You will not be submitting `team.h`, so any modifications in `team.h` will not be reflected in the Gradescope autograder.

3.1 Useful Structs

A `struct player` contains `char *name`, the string holding the player's name; `int age`, the player's age; and `int number`, the player's jersey number.

A `struct roster` contains `struct player **players`, a pointer to **element zero** of an array of pointers to players; `int size`, the number of players currently in our roster; and `int capacity`, the **dynamically allocated** size of our `players` array.

3.2 `createPlayer()`

```
int createPlayer(const char *name, int age, int number, struct player **dataOut);
```

This function takes in a pointer to the player's name, an int for their age, an int for their jersey number, and a pointer to a pointer used to return the newly created player if successful.

The function will create a new player struct with the given information by allocating memory for it on the heap. If this function succeeds, it should return SUCCESS and modify dataOut to hold the newly created player. If it fails, it should return FAILURE and leave dataOut unchanged.

If any of the information is invalid, such as a NULL name or negative player number, you will return FAILURE. Specifically, you return FAILURE if: the name is NULL, age is negative, number is negative, dataOut is NULL, or a malloc failure occurs. Otherwise, you will dynamically allocate space for the player, and return SUCCESS if you are able to create a new player successfully.

3.3 `addToRoster()`

```
addToRoster(struct roster *roster, struct player *newPlayer);
```

This function takes in a pointer to the roster struct, and a pointer to the new player struct to add to the roster.

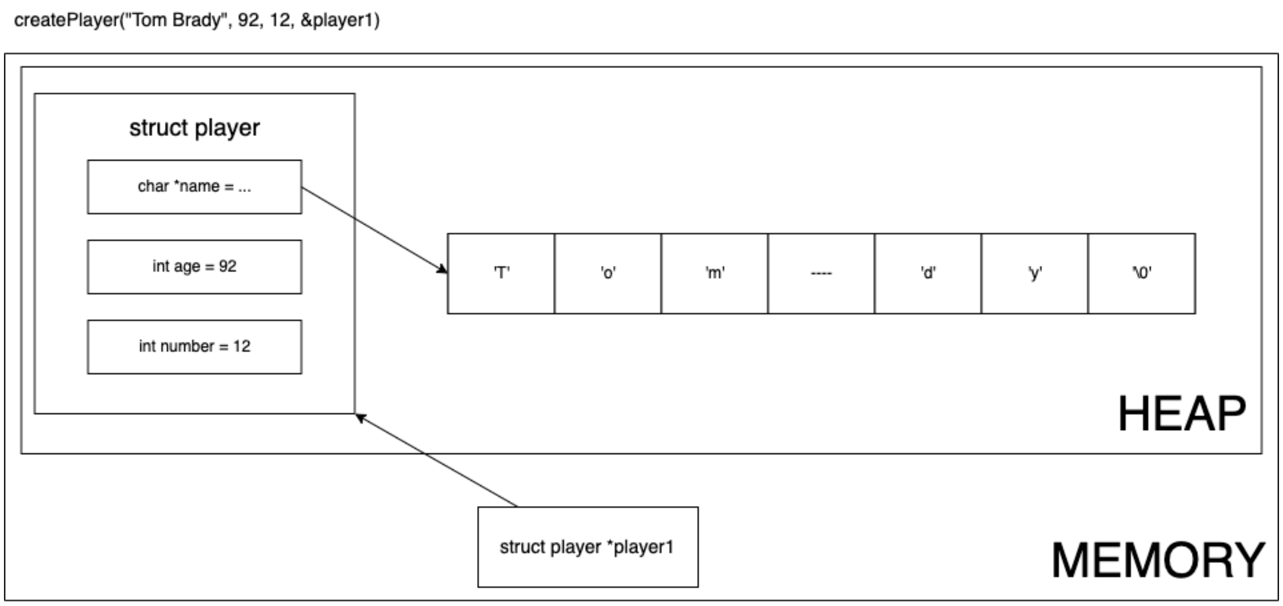
The function will add the passed in player to the end of the players array in the roster struct. You may assume that if the the player struct passed in is not NULL, then it has already been allocated on the heap.

The maximum capacity for the players array is MAX_ROSTER_SIZE, which is defined in the team.h file. If the array is not filled, then add the player to the end of the array. However, if the array is already full, then you must do one of the following: resize the array and double the capacity, resize the array to MAX_ROSTER_SIZE, or leave the array as is if the capacity is already at MAX_ROSTER_SIZE.

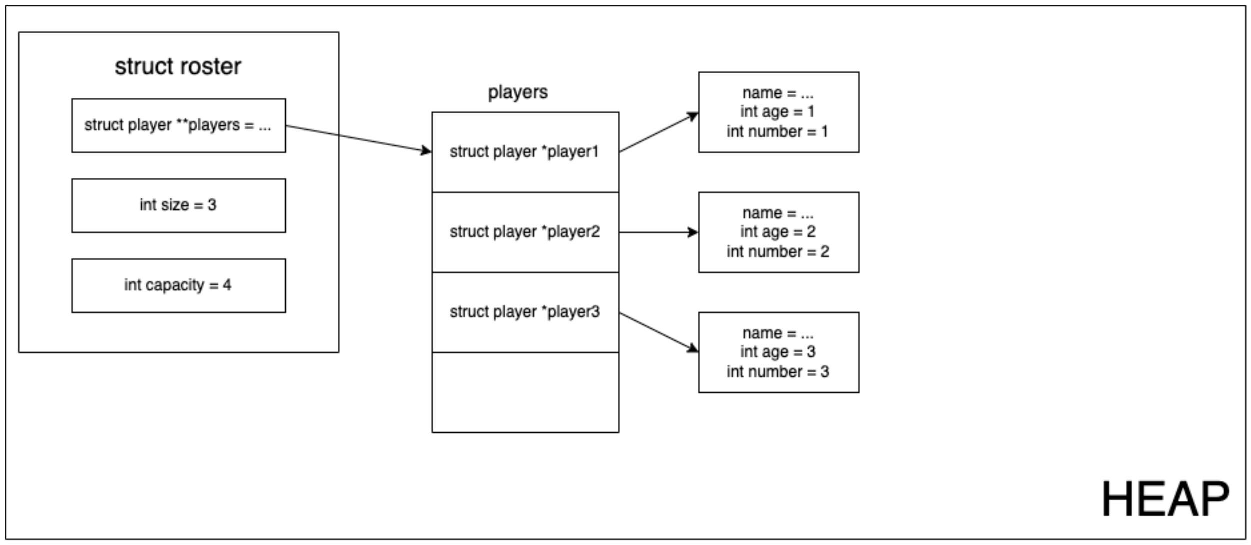
If any of the following are true, return FAILURE: roster or its player array is null, the player is NULL, the size or capacity is negative, the size is greater than the capacity, the capacity is exceeds MAX_ROSTER_SIZE, both the size and capacity are at MAX_ROSTER_SIZE, or dynamic memory allocation fails. Otherwise, you will add the passed in player to the end of the players array and adjust the size and capacity of the array as needed. Return SUCCESS if you are able to add the player to the array successfully.

3.4 Diagram

Refer to the following for a visual representation of the player struct, where "..." denotes an unknown pointer address (pointing to some memory on the heap). In the first diagram, we see the results of `createPlayer("Tom Brady", 92, 12, &player1)` with all the proper data copied to the heap. The name will contain the string `"Tom Brady"`, and the age and number of the player will be 92 and 12, respectively.

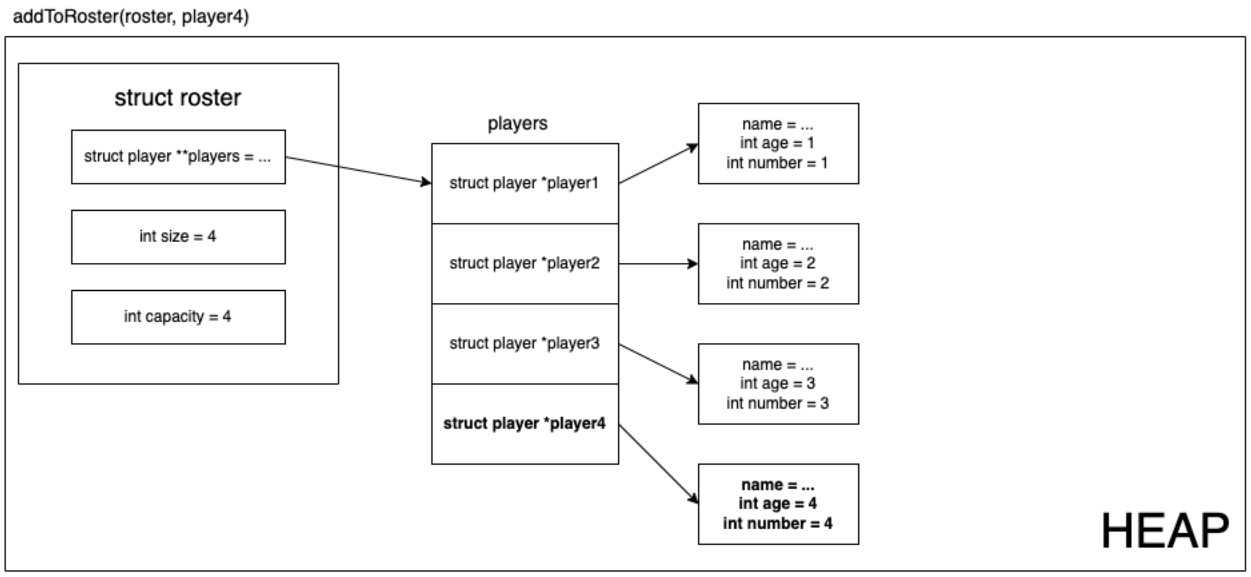


In the second diagram, an existing roster is shown. The players array has already been dynamically allocated to be able to hold 4 players, and the array is currently holding 3 players. The roster's size and capacity fields have been set to 3 and 4 accordingly. *Reminder*: `players` points to the first element of the array-backed roster.



In the third diagram, you can see the results of `addToRoster(roster, player4)`, which adds a fourth player to the previously mentioned roster.

After ensuring that the roster and the player are not `NULL`, we had check to see if adding another player would make us go over the capacity. We know that adding the new element will not make us go over the capacity, so we add the passed in `struct player *` at the end of the array to point to the the passed in player as well as update the size appropriately.



4 Grading

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

- `createPlayer`:
  - If the passed in name is `NULL`, or dataOut is `NULL`, you should return FAILURE.
  - If the passed in name or number is negative, 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 newly allocated player should be stored in dataOut
  - If the function is successful, you should return `SUCCESS`.
- `addToRoster`:
  - If the passed in roster or its player array is `NULL`, or the passed in player is `NULL`, you should return FAILURE.
  - If the roster's size or capacity is is negative or is greater than or equal to MAX_ROSTER_SIZE, you should return FAILURE
  - If dynamic memory allocation fails at any point, you should return `FAILURE`.
  - If there is space for the passed in player, then it should be stored at the end of the array and size should be appropriately incremented.
  - If dynamic memory allocation does not fail when it's needed, the array should be resized to fit the new player and capacity should be appropriately changed.
  - 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. team.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 team`
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/team_suite.c:43`, or `b team.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_player_normal`:

```
suites/stack_suite.c:45:F:test_create_player_normal:test_create_player_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!)