CS 2110 Timed Lab 5: C

Lauren Chen, Manley Roberts, Daniel Becker, Shannon Ke, and Maddie Brickell

Spring 2019

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

1.1 General Rules

1. You are allowed to submit this timed lab starting at the moment the assignment is released, until you are checked off by your TA as you leave the recitation classroom. Gradescope submissions will remain open until 7:15 pm - but you are not allowed to submit after you leave the recitation classroom under any circumstances. **Submitting or resubmitting the assignment after you leave the classroom is a violation of the honor code - doing so will automatically incur a zero on the assignment and might be referred to the Office of Student Integrity.**
2. Make sure to give your TA your Buzzcard before beginning the Timed Lab, and to pick it up and get checked off before you leave. **Students who leave the recitation classroom without getting checked off will receive a zero.**
3. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. **The information provided in this Timed Lab document takes precedence.** If in doubt, please make sure to indicate any conflicting information to your TAs.
4. Resources you are allowed to use during the timed lab:
    - Assignment files
    - Previous homework and lab submissions
    - Your mind
    - Blank paper for scratch work (please ask for permission from your TAs if you want to take paper from your bag during the Timed Lab)
5. Resources you are **NOT** allowed to use:
    - The Internet (except for submissions)
    - Any resources that are not given in the assignment
    - Textbook or notes on paper or saved on your computer
    - Email/messaging
    - Contact in any form with any other person besides TAs
    - Any compiler that outputs LC3 code
6. **Before you start, make sure to close every application on your computer.** Banned resources, if found to be open during the Timed Lab period, will be considered a violation of the Timed Lab rules.
7. We reserve the right to monitor the classroom during the Timed Lab period using cameras, packet capture software, and other means.

1.2 Submission Rules

1. Follow the guidelines under the Deliverables section.
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 Gradescope.

3. Do not submit links to files. We will not grade assignments submitted this way as it is easy to change the files after the submission period ends.

1.3 Is collaboration allowed?

Absolutely NOT. No collaboration is allowed for timed labs.

2 Overview

2.1 Description

In this timed lab, you'll be writing two functions which will act on a linked list of `pokemon` structs. The first of these, `copy_list()`, takes in a pointer to a list and returns a pointer to a "deep" copy of this list, with *all* dynamically-allocated data duplicated. The second, `destroy()`, takes in a pointer to a list and destroys this list, freeing all dynamically-allocated memory associated with it.

3 Instructions

You have been given one C file - **tl5.c** - in which you should complete the `copy_list()` and `destroy()` functions according to the comments.

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.

3.1 Writing `copy_list()`

The function `copy_list()` takes in one argument, a pointer to `struct list`, and returns a pointer to a new `struct list`. The list returned should be identical to the list passed in, but should be an entirely different list—that is, the `list` struct, all `pokemon` structs, and any dynamic attributes of the pokemon struct will be copied over into new dynamically allocated memory. This is known as a **deep** copy.

NOTE! When you make this copy, you should note every place you have allocated memory...because when you destroy a list, you'll have to free all the dynamically allocated memory.

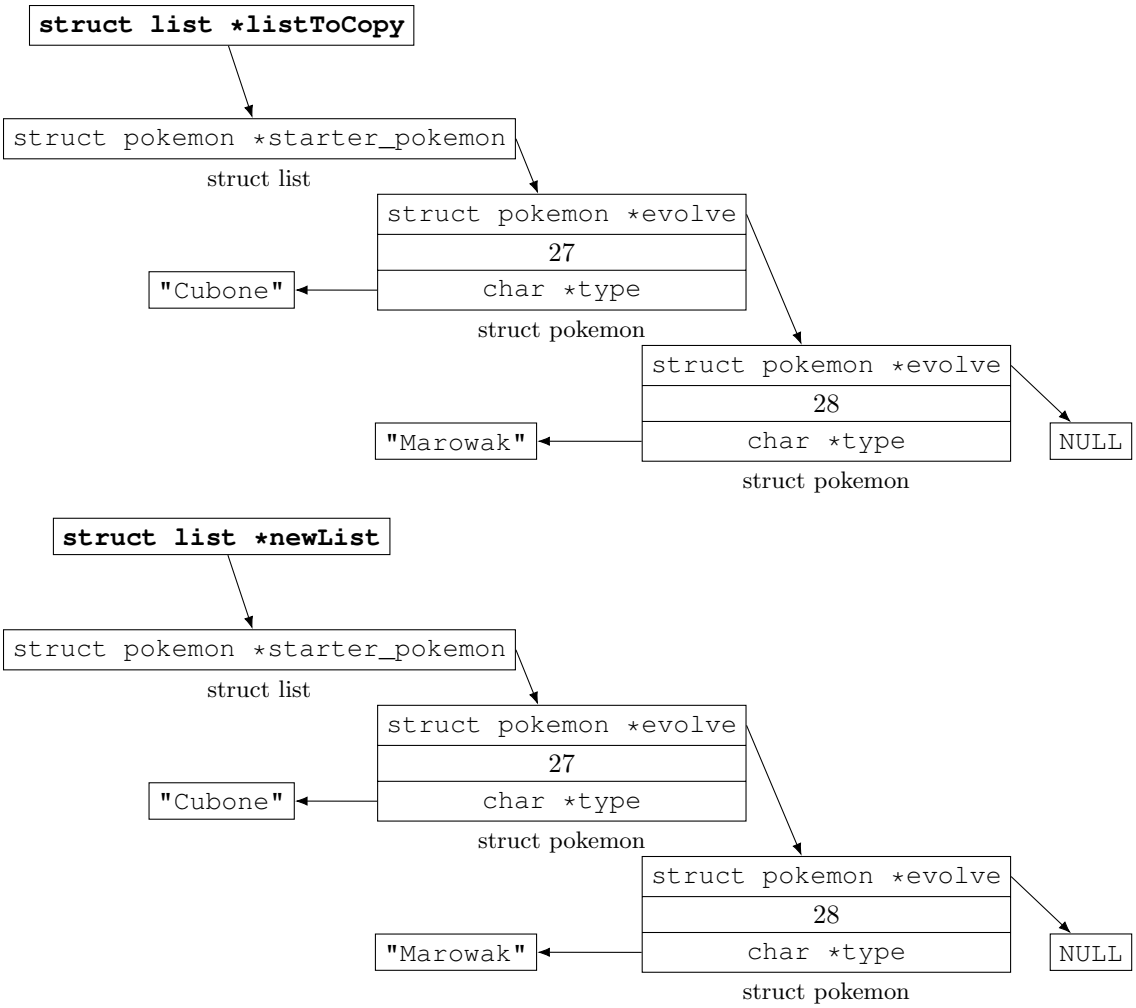
BONUS NOTE! If any memory allocation failures occur, you **must** destroy the entire list and free all dynamically allocated memory before returning NULL. This means that your `copy_list()` function may depend on your `destroy()` function!

The following diagram highlights the nature of a deep copy. Notice that *every* piece of data is duplicated. The diagram shows the state after the following code is executed successfully, without memory allocation errors:

```
//pre-condition: listToCopy points to a valid list, populated with data.

struct list *newList = copy_list(listToCopy);

//post-condition: the list pointed to by listToCopy is not altered in any way,
//              and the list pointed to by newList is in the state shown below.
```



3.2 Writing `destroy()`

The function `destroy()` takes in one argument, a pointer to `struct list`, and does not return anything.

After this function executes, all dynamically allocated memory associated with `listToDestroy` must be freed. This includes any `pokemon` structs in the list, the list itself, and *any* other heap data referenced by any of these structs.

3.3 Useful `man` pages

You might find the following abbreviated `man` pages useful:

- **Name**

  `strlen` - calculate the length of a string

  Synopsis

  ```
  #include <string.h>

  size_t strlen(const char *s);
  ```

  Description

  The `strlen()` function calculates the length of the string `s`, excluding the terminating null byte `('\0')`.

  Return Value

  The `strlen()` function returns the number of bytes in the string `s`.

- **Name**

  `strcpy`, `strncpy` - copy a string

  Synopsis

  ```
  #include <string.h>

  char *strcpy(char *dest, const char *src);

  char *strncpy(char *dest, const char *src, size_t n);
  ```

  Description

  The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte `('\0')`, to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy. Beware of buffer overruns!

  The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied. Warning: If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.

  If the length of `src` is less than `n`, `strncpy()` writes additional null bytes to `dest` to ensure that a total of `n` bytes are written.

  Return Value

  The `strcpy()` and `strncpy()` functions return a pointer to the destination string `dest`.

4 Rubric and Grading

4.1 Autograder

We have provided you with a test suite to check your linked list that you can run locally on your very own personal computer. 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. Also keep in mind that this file does not have debugging symbols so you will not be able to step into it with gdb (which will be discussed shortly).

Your process for doing this lab should be to write one function at a time and make sure all of the tests pass for that function—and if one of your functions depends on another, write the most simple one first! 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. Further down, 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. Your grade on Gradescope may not necessarily be your final grade as we reserve the right to adjust the weighting. However, if you pass all the tests and have no memory leaks according to valgrind, you can rest assured that you will get 100 as long as you did not cheat or hard code in values.

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.

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.

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.

4.2 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 run the tests without valgrind or gdb: `make run-tests`
3. To run your tests with valgrind: `make run-valgrind`
4. To debug a specific test with valgrind: `make TEST=test_name run-valgrind`
5. To debug a specific test using gdb: `make TEST=test_name run-gdb`

   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/list_suite.c:420`, or `b list.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

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

```
suites/test_utils.c:14:E:test_list_copy_basic_easy:test_list_copy_basic_easy: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.

Note: The checker may not reflect your actual grade on this assignment. We reserve the right to update the checker as we see fit when grading.

5 Deliverables

Please upload the following files to Gradescope:

1. tl5.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!