CS 2110 Timed Lab 4: C

Matthew, Varnika, Kevin, Patrick, Alice, Nicole

Spring 2021

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

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 in a private Piazza post, 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. Contact in any form with any other person besides a TA is absolutely forbidden. **No collaboration is allowed for timed labs.**

2 Overview

2.1 Description

Please read this entire document before starting. If you are running out of time, implement the functions `set_character()` and `draw_hollow_box()` since you will be able to get partial credit. These do not require any dynamic memory allocation knowledge.

In this timed lab, you shall be writing two functions that involve array and pointer arithmetic: `set_character()` and `draw_hollow_box()`, two functions that involve dynamic memory allocation: `create_image()` and `destroy_image()`, and a single function that involves string manipulation AND dynamic memory allocation, `add_extension()`. 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. The one exception is `draw_hollow_box`, where you may find it much easier to implement if you complete `set_character` first (although implementing `set_character` is not strictly required to get full credit on `draw_hollow_box`).

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 `set_character()`, `draw_hollow_box()`, `create_image()`, `destroy_image()`, and `add_extension()` 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 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 The ASCII Image

This timed lab involves creating, modifying and destroying ASCII images in C. ASCII images are images created using text or ASCII symbols. You will be using the provided `ascii_image` struct in `tl04.h` to write all of the required functions in this assignment.

A `struct ascii_image` contains a width and height, representing the dimensions of the ASCII image; a non-null name; and the image itself, represented by `char *data` which is a non-null, 1-dimensional array of size `width * height`. Like the `videoBuffer` from HW8, this one-dimensional array represents a 2D array or image whose rows/scanlines are stitched together into a 1D array.

3.2 Writing `set_character()`

This function writes a character into the given image at the given row and column coordinates. It takes in a pointer to an ASCII image, a position (row and column), and the character to set that position to, as given in the function header below:

```
int set_character(struct ascii_image *image, int row, int col, char c);
```

This function will set the character in the ASCII image to the given character at the row and column coordinates.

- If the image passed in is `NULL` or its data is `NULL`, return `FAILURE`
- In the case that the row or column supplied is invalid, i.e. row is not in the range `[0, image->height - 1` or column is not in the range `[0, image->width - 1]`, return `FAILURE` without modifying the image
- If the function sets the character successfully, return `SUCCESS`

HINT: Think about how we used `videoBuffer` in HW8

3.3 Writing `draw_hollow_box()`

`draw_hollow_box()` is used to draw a hollow box on the ASCII image supplied, given the coordinates of the top left corner of the box and the dimensions of the box. The function takes in an ASCII image, row and column coordinates to signify the top left corner of the box to be drawn, width and height of the box to be drawn, and a character to be used to draw this box, as detailed in the function header below:

```
int draw_hollow_box(struct ascii_image *image,
                    int row, int col,
                    int height, int width,
                    char c);
```

The box to be drawn must be hollowed out, so that it is only one character thick as shown in the example in `tl04.c`. The box drawn on the screen can go out-of-bounds; in this case, draw the parts of the box within bounds of the image, ignoring the part that goes out-of-bounds. For this reason, it is highly encouraged to use `set_character()` for drawing, since it should have built-in bounds checking.

Note: none of our tests will intentionally draw out-of-bounds, but you must avoid drawing out-of-bounds in your code, or else you might get a bad image or a Valgrind error.

- If `image` is `NULL`, or if either the width or height of the box to be drawn are less than 1, return `FAILURE`.
- If there are no issues with the input, you should draw the box and return `SUCCESS`
- You may assume that `image->data` is not `NULL` whenever `image` itself is not `NULL`.

3.4 Writing `create_image()`

`create_image()` allocations a new ASCII image using three parameters: the height and width of the image, as well as its name.

```
struct ascii_image *create_image(int height, int width, char *name);
```

In this function you must dynamically allocate space for a new image, the new image's name and the new image's data.

If the width or height are less than 1, return `NULL` without allocating anything.

The allocated image data must have a size of `width * height`, and must be filled in with '.' (period) characters, representing a blank canvas for us to draw on. The name for the image allocated must be deep-copied.

You must ensure that there are no memory leaks whatsoever. In the case that `malloc` fails, free the remaining dynamically allocated variables in your function and return `NULL`. Otherwise, if all the dynamic allocations succeed, initialize the newly-created `ascii_image` pointer and return it.

3.5 Writing `destroy_image()`

The function `destroy_image()` takes a single argument, the ASCII image to destroy.

```
void destroy_image(struct ascii_image *image);
```

The purpose of this function is to free up all of the dynamically allocated data associated with the image passed in. This means that we need to free the image's name and data, as well as the image itself. It is possible for an image to be non-null and have either its data or name be `NULL`. You should not have to handle these cases differently since `free(NULL)` is a harmless operation. Be careful to not dereference the image if it is NULL (e.g. `image->data`), as this will segfault. As usual, take care to not have any memory leaks in the program.

3.6 Writing `add_extension()`

This function appends a file extension to the end of the name of an ASCII image. `add_extension()` takes in the ASCII image to modify and the extension to be used:

```
int add_extension(struct ascii_image *image, char *extension);
```

For example, adding the extension `".png"` to an image with the name `"puppy"` should change the name of the image to `"puppy.png"`. If we then called `add_extension(image, ".zip")`, the image should now have the name `"puppy.png.zip"`.

When appending the extension to the image name, it is required that you dynamically allocate a new pointer to it either using `malloc` or `realloc`. This point should be assigned to `image->name`, and it should have enough room for both the old image name and the extension combined. If you `malloc` space for your new name, make sure to deallocate the space used for the previous name of the image.

Make sure that there are no memory leaks in your code! You are welcome to use any of the `string.h` functions in your code, these are quite handy! Make sure to allocate enough memory for both strings combined, or else you will get "write to invalid memory" Valgrind errors.

- If the image or the extension passed in are `NULL`, return `FAILURE` without allocating or modifying the image
- If the image is not `NULL`, you can assume that `image->name` is also not `NULL`
- If any dynamic memory allocation fails, make sure to return `FAILURE` and free any other memory that you have allocated

3.7 Testing your program with `main()`

`main()` can be used to test all of the functions that you've written so far. You can use `print_image()` (provided in `tl04.c`) to print out the image you pass in. From here, you can set up your own test cases, print out images, 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 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 Aborted signal.

- 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 code in `main.c`: `make tl04`
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/list_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

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!