CS 2110 Timed Lab 4: C

Varnika, Corey, Nicole, Mudit

Fall 2020

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. Gradescope submissions will remain open for 75 minutes (ends at 4:45pm EDT). **Submitting or resubmitting the assignment after time is up is a violation of the honor code - intentionally doing so may incur a zero on the assignment and might be referred to the Office of Student Integrity.**
2. You can ask TAs clarification questions on Piazza, but you are ultimately responsible for what you submit. **The information provided in this Timed Lab document takes precedence.**
3. Resources you are allowed to use during the timed lab:
   - Assignment files
   - Previous homework and lab submissions (this includes homework PDFs)
   - Class Notes (Open Net, Open Book)
   - Your mind!
4. Resources you are **NOT** allowed to use:
   - Email/messaging
   - Contact in any form with any other person besides TAs

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. Under no circumstances whatsoever will we accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over 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

Please read this entire document before starting. If you are running out of time, implement the `OFFSET` macro and the functions `decimal_string_to_int()` and `ints_to_rgb()` before finishing the work on `my_main()` function since you will be able to get partial credit.

In this timed lab, you shall be writing a macro and two functions involving RGB color values and strings in C. `my_main()`, `decimal_string_to_int()`, `ints_to_rgb()`, and the `OFFSET` macro. Please read all the directions to avoid confusion.

3 Instructions

You have been given three C files - **tl04.c, tl04.h, and main.c** (which is only there for the auto-grader; it should not be modified). For `tl04.c` and `tl04.h`, you should implement the `my_main()`, `decimal_string_to_int()` and `ints_to_rgb()` functions according to the comments as well as the `OFFSET` macro.

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 `OFFSET(row, column, width)`

Please complete this macro at the top of the **tl04.h** file. This is required for the next function you need to implement: `my_main()`. This macro will be used to index the `videoBuffer` to get to the correct pixel. We have provided a `VBUF_WIDTH` constant for you.

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

3.2 Writing `decimal_string_to_int()`

`decimal_string_to_int()` takes in one parameter and returns an int as detailed in the function header below:

`int decimal_string_to_int(char *str)`

The argument *str is a decimal string containing ASCII digits that should be converted to an int. You may assume that any inputs to this function will be valid decimal strings, and will contain no characters other than the digits `'0'-'9'`. The int returned must be the corresponding decimal value of the string.

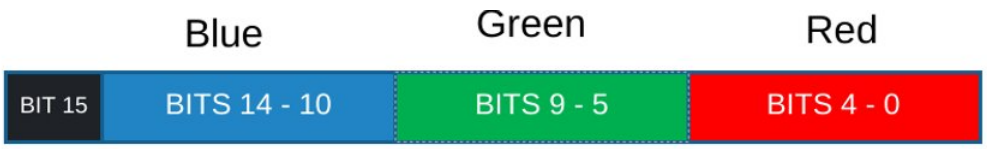
For example, `decimal_string_to_int("2110");` should return an int of value 2110.

3.3 Writing `ints_to_rgb()`

`ints_to_rgb()` takes in three int parameters and returns a short as detailed in the function header below:

`unsigned short ints_to_rgb(int r, int g, int b)`

The arguments **r**, **g**, and **b** represent the red, green and blue values of a color respectively. You may assume that each of these values is a positive number with no more than 5 significant bits (less than 32). You must return an unsigned short containing the combined 16 bit RGB color value following the GBA color format as shown below:



For example, `ints_to_rgb(31, 7, 10);` should return an int of value 10495.

HINT: Bitshifting in C works the same as in Java!

3.4 Writing `my_main()`

The function `my_main()` takes in arguments `int argc` and `char const *argv[]`. `argc` is the number of command line arguments. Recall that the name of the executable is always argument 0. `argv` is an array of pointers to `const char`. The character string at index 1 in `argv` represents a set of pixel coordinates, and each subsequent character string represents an RGB color, which this function will parse. The purpose of this function is outlined in detail below.

You are given a global array of `short` called `videoBuffer` which represents the screen you shall be drawing on. Each entry corresponds to a pixel, and you write a color value to the corresponding pixels. `my_main` should do the following things:

- Parse the character string corresponding to the x, y coordinates before parsing any of the color strings. Use `decimal_string_to_int()` to convert the string representations of these numbers to integers.
- Check if these coordinates are in bounds. `tl04.h` contains macros related to the size of the videoBuffer which should be used to perform bounds checking. If the coordinates are invalid, `my_main` should stop parsing and return 1 to indicate failure.
- For each color, parse the r, g and b values. Once again, use `decimal_string_to_int()` to help convert the string representations of the numbers to integers.
- Check whether r, g and b are valid color components, i.e., less than 32. As before, if the values are invalid, `my_main` should stop parsing and return 1 to indicate failure.
- Once you have converted the individual r, g and b values to integers and confirmed that they are valid, the next step is to convert them to a short representation of an RGB color. Call `ints_to_rgb()` to help with this.
- Store the RGB color short at the coordinates of the videoBuffer given with the help of the `OFFSET` macro.
  - **WARNING: don't confuse `x` and `y` with `row` and `column`!** x corresponds to the column and y corresponds to the row from the top-left of the screen.
  - The first color should be stored at the specified coordinate, and each subsequent color should be stored at the very next pixel.

  - For example, if the given coordinates are (0, 0), the first color should be stored at index 0, the next color should be stored at index 1, the next at index 2, etc.
  - If the end of a row is reached, wrap around to the start of the next row.

- `my_main` should return 0 on successful completion of this task, or 1 if there is an error (an invalid coordinate or color value is encountered).

Example command line usage:

- `./tl04 "   x:30   y:52 " "r:5g:10    b:1" "  r:7 g:10  b:0"`
  - Should store the color `r = 5, g = 10, b = 1` at row 52, column 30, then store `r = 7, g = 10, b = 0` at row 52, column 31.
- `./tl04 "   x:1   y:1 " "r:5g:10    b:1" "r:1 g:10 b:5"`
- `./tl04 "x:100y:109" "r:15g:0b:10"`
- `./tl04 "x:100       y:109" "r:15g:0b:10     "`

NOTE: Unlike Homework 7, there will be multiple character strings to parse, however there is no maximum number of arguments.

RULES:

- The `x` and `y` coordinates represent the position of the pixel in the videoBuffer. **x is the column and y is the row**. `r`, `g`, and `b` represent the RGB components of a color to be used to color the aforementioned pixel in the videoBuffer.
- The coordinates will always be the first argument; there will always be a coordinate and at least 1 color.
- Each term in an argument consists of a single strictly lowercase letter, followed by a colon, followed by a decimal number, with no spacing in between. The ordering of terms will always be consistent (ie. x always before y; r always before g always before b). There will **never** be missing terms.
- **There will be a variable amount of spaces preceding and following every term. In some cases, there may be no spaces at all. You cannot simply index the character string to access information.** You will need to loop through the character string until you find x and y in the first string, and then r, g and b in subsequent strings.
- Numbers within argument strings will be non-negative and up to 5 digits long (but can consist of fewer digits); the macro `MAX_DIGITS` is given in `tl04.h` for this purpose. You do not have to check for this; all numbers your code will be tested with will contain 1-5 digits.
- **If a coordinate or color value is invalid (its value is outside the bounds of videoBuffer, if a coordinate, or its value is greater than 31, if a color), return 1 to indicate failure.**
- You will not be tested with any code that attempts to write past the end of the videoBuffer.
- You may declare as many local variables as you need inside of the main function to help you loop through each character string in `argv` and loop through the character string itself.
- You may use any function from **`string.h`**

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
• r	**run** your code (be sure to set a break point first)
• n	**step over** code
• s	**step into** code
• p/x variable	**print** variable in current scope in 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 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.

5.2 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 compile the tests: `make tests`
3. To run a specific test without gdb: `make run-case TEST=test_name`
4. 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 tl4.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_install_internet_copies_name`:

```
suites/tl4_suite.c:50:F:test_install_internet_copies_name:install_internet_copies_name: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.

6 Deliverables

Please upload the following files to Gradescope:

1. tl4.c
2. tl4.h

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!