TCSS 380, Autumn 2019, Lab 2

The objective of this lab assignment is to give you more practice with Unix tools, C pointers, arrays, strings, and memory management.

Work with a partner assigned to you to complete this lab. The instructor will check off first two programs during the lab and if you get checked off, you don't need to submit the code. At the end of the lab submit all the parts you complete during the lab session that haven't been checked off – first 2 programs must be completed during the lab session to receive full credit. If you do not complete some part within the two-hour session, finish at home by the designated due date. Make sure that each one of your programs contains a comment at the top with your name and the name of your partner. Upload the following to Canvas:

- Updated codes:
  - lab02a.c (if not checked off during class)
  - lab02b.c (if not checked off during class)
  - lab02c.c
  - lab02d.c

All together there should be at most two submissions associated with each student: one during the lab and one afterward. For all C labs you should be using Unix environment and Unix command line. Today you should work on your Linux VM. Open **Firefox** and download **lab02a.c, lab02b.c, lab02c.c,** and **lab02d.c** from **Canvas** into your local subdirectory.

I. Function pointers

lab02a.c

1. Open **lab02a.c** and replace numbered comments with actual statements. Compile and run. The program should print:
   sum(sin): 0.459308
   sum(cos): 0.840758

end of lab02a.c

II. Pointers and arrays

lab02b.c

1. Open **lab02b.c** in an editor of your choosing and look for the comments that indicate what to add to this code – there are 6 numbered comments where you have to add code.
2. After adding code, save the file. Fix all the mistakes and re-compile.
3. To verify proper memory usage, run the executable with **Valgrind** (http://valgrind.org/), a tool that enables extensive checking of memory allocations and accesses:
   valgrind ./a.out
   Read through the generated report thoroughly. It will tell you whether you have memory leaks in your code or other errors. If you do, fix them, recompile, and run Valgrind again. A quick gude to Valgrind is also available here:
   http://web.stanford.edu/class/cs107/guide_valgrind.html

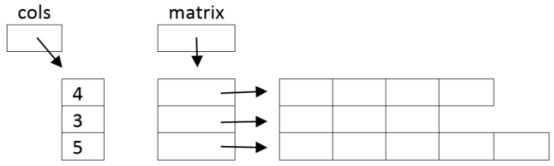
end of lab02b.c

lab02c.c

1. Open **lab02c.c** program in an editor of your choosing and replace numbered comments with appropriate statements. The program is to generate a dynamic jagged 2D array and a helper

1D array. Since the 2D array is jagged, we need a helper array to store the number of columns used in each row of the 2D array in order to be able to process the 2D array later on.

a. First, in *main*, the user is prompted for the number of rows they want in the 2D array, and a helper array is allocated of that size, as well as the array of pointers (2D array). Then, for every row the user is prompted for the number of columns in that row. The program allocates each row of the 2D array and stores the size of the row in a 1D array. You can think of these variables as follows:



b. Then, both arrays are passed to function *myprint* that prints initialized contents. After that both arrays are passed to function *assign* that populates the contents of the 2D array. Finally, the arrays are passed to *myprint* function again to print contents again.

2. After you are done, compile and run the program. Once the program runs properly, check for memory leaks with **Valgrind** and apply more fixes, as needed.
3. After the changes, is your program robust enough to handle a user entry of 4,000,000,000 for the number of rows and then the same number for each column? Fix the program so that if the matrix cannot fit into memory, your program prints an error message and exits.

end of lab02c.c

III. Pointers and strings

lab02d.c

1. Open **lab02d.c**. Right now the program contains a function we discussed in class that repeats a string *n* number of times and tests this functions. Run the program in Valgrind and fix mistakes.
2. Then, add another string function and test it in *main.* The new function should be named *strip_puncts,* and it should take a string as an argument and modify it so that all punctuation marks are stripped from the original string (hint: *ctype* library). For example, if the original string passed to this function were “hi, how are you?”, then the updated string in the calling block should contain: “hi how are you”.
3. Then, add another string function and test it in *main*. The new function should be named *add_substring*, and it should take three parameters: the substring to be inserted, the original string, and the location where the insertion is to begin. The function should allocate a new, resulting, string dynamically and return it. The new string is to be the exact size in memory as the number of characters actually needed for the resulting message. For example if add_substring(“are ”, “all men created equal”, 8) is called, then “all men are created equal” should be the resulting string – don’t forget the null character at the end. Assume that the caller of your function will always pass valid integer values, i.e. there is no need to handle things such as negative indexes or other invalid values. Once the program runs properly, check for memory leaks with **Valgrind** and apply more fixes, as needed.

end of lab02d.c

--- CONGRATS – YOU ARE DONE – UNTIL NEXT LAB THAT IS ☺ ---