CSCI 1120 (Low-Level Computing), Spring 2021

Homework 7

Credit: 20 points.

1 Reading

Be sure you have read, or at least skimmed, the assigned readings for classes through 03/24.

2 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to my TMail address with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 1120 hw 7" or "LL hw 7"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (20 points) For this assignment your mission is to write a C program that sorts the lines in a text file (considering each line as a string) using the library function `qsort`. So for example if the file contains lines `"hello world"`, `"abcdef"`, `"the end"`, and `"aaaa"`, the program should print

   ```
   aaaa
   abcdef
   hello world
   the end
   ```

   The program should take the name of the file to sort as a command-line argument (and print appropriate error messages if none is given or the one given cannot be opened) and write the result of the sort to standard output.

   To do this, I think you will need to read the whole file into memory. There are various ways to do this and perform the sort; the one I want you to use is somewhat involved but intended to give you more practice working with pointers. *To get full credit you must use the approach described here.*

   - First, read the whole file into memory. To do this you will need to know its size, and interestingly enough there doesn't appear to be any truly portable and reliable way to find that out! My suggestion is therefore to just open the file, read it a character at a time, counting the number of characters but not trying to save them, and close it again. Reading the file twice (once only to find its size) is of course inefficient but will give the desired result (unless some other application is changing the file at the same time) using only standard and portable C functions, and coming up with a nicer way to accomplish this task is beyond the scope of this assignment.

- Once you have the (best estimate for) file size, you can allocate a single array for the file using `malloc()`, something like this:

  ```
  char * data = malloc(size_in_bytes);
  ```

  You can now operate on `data` as if it had been declared as an array of `char`. (Check first that `malloc()` succeeded.)

  Now you can read in the contents of the file; a character at a time is probably simplest. Note that as you do this you will get the newline characters at the ends of lines. (You might write this much of the program and check that it works before going on.)

- Once you have the whole file in memory, the objective is to sort it with `qsort`. `qsort` needs four parameters: an array to sort (of elements of fixed size), a count of elements, a size for each element, and a comparison function. Sample program sorter-improved.c has an example of using `qsort`.

  Your first thought may be to wonder how this can work, since text strings aren't of fixed size. But we can play a trick ...

  The idea will be to build an array of pointers pointing to starts of lines, sort *the pointers* so the first one points to the first line to print, etc., and use them to print the lines in order. (If you think you know at this point how to proceed, you could try doing so, and then come back and read the rest of this description.)

  So the first step is to build the array of pointers to lines. How many do you need? You could figure that out as you're reading the file into memory. Say you have that in a variable called `N`. Then you can allocate space for an array of pointers like this:

  ```
  char ** lines = malloc(sizeof(lines[0]) * N);
  ```

  The first one should point to data[0], which you can accomplish like this:

  ```
  lines[0] = &data[0];
  ```

  Then the idea is to go through the rest of the characters, and make `lines[1]` point to the character after the first newline, `lines[2]` point to the character after the second newline, etc.

- Once you get this array built, you can check it by printing the file contents out again using the array of pointers; for example, to print the first line you can write

  ```
  printf("%s\n", lines[0]);
  ```

  (You'll need this code anyway, so I say you might as well write it now and check that it works. You may get a surprise when you first run it, as a result of which you may decide you need to do more processing of your `data` array. More-explicit hint in a footnote[1] so you can at least try to figure it out for yourself first.)

- Now the missing piece of the puzzle is to use `qsort` to actually sort the lines before printing them. Its parameters were described earlier; the only one you don't yet have is the comparison function. It can look a lot like the one in the sample program; all that's different is that you're sorting pointers-to-strings rather than integers. You will probably want to use the C library function `strcmp()` for the actual comparison. Read its `man` page to find out what it does and what parameters it takes.

You can check your program's output by using the `sort` command to sort the input file and comparing its result (captured with I/O redirection!) with your result (also captured

[1] `printf` with a `%s` conversion specification prints a string, which is assumed to end with the null character (`'\0'`). It's happy to print strings containing any number of newlines.

with I/O redirection). Traditionally `sort` just sorted based on what `strcmp()` returns for the lines in the file, but depending on configuration it may instead do a comparison that is case-insensitive and ignores leading whitespace. For this problem I just want you to do the simple comparison. You can get the `sort` command to do that by overriding the default configuration, thus:

```
LC_COLLATE=C sort filetosort
```

(Remember that one way to view differences between text files — including of course program source — is with `vimdiff`. I like to use `vimdiff -o`.)

3 Pledge

For programming assignments, this section should go in the body of the e-mail or in a plain-text file `pledge.txt` (no word-processor files please). For written assignments, please put it in the text or PDF file with your answers.

Include the Honor Code pledge or just the word "pledged", *plus* at least one of the following about collaboration and help (as many as apply). Text *in italics* is explanatory or something for you to fill in.

- I did not get outside help *aside from course materials, including starter code, readings, sample programs, the instructor.*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*
- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

4 Essay

For programming assignments, this section should go in the body of the e-mail or in a plain-text file `pledge.txt` (no word-processor files please). For written assignments, please put it in the text or PDF file with your answers.

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what if anything you think you learned from the assignment, and what if anything you found interesting, difficult, or otherwise noteworthy.