Homework 2

File Processing (Term I/2018–19)

built on 2018/10/15 at 13:47:49 ***due:*** *Mon, Oct 22 @ 11:59pm*

This assignment will give you practice on C programming. You will implement a few C programs, test them thoroughly, and hand them in. To work on Tasks 3 and 4, you will need a few starter files. They can be found in the directory `/handout/a2/` on our Hamachi server machine.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. Zip up all the scripts and name it `a2.zip` i.e.

   ```
   > zip a2.zip vert_hist.c tabutil.c life_engine.c solve_jumble.c
   ```

2. Find out the MD5 hash of your zip file. You will need to submit this code on Canvas. We use it for keeping track of your submission time. You may resubmit your work but the MD5 hash has to match.

   ```
   > md5sum a2.zip
   ```

3. Copy the zip file to the directory `/subm/u5712345` where `u5712345` is your student ID.

   ```
   > cp a2.zip /subm/u5712345
   ```

4. Log on to Canvas, go to assignment 2submission page and enter the MD5 hash.

If you collaborate with another student, include in your zip file a README file indicating your collaborators and the extent to which you work together.

Task 1: Vertical Histogram (15 points)

You will write a program that prints a histogram of frequencies of letters in the English alphabet. The input will come from stdin. Your program should downcase all input letters and ignore symbols besides `a` to `z`. The output is a histogram rendered vertically.

Code and Compilation: Your code will be one standalone file called `vert_hist.c`. Use the following command to generate an executable `vert_hist`. Your code must compile clean with this command.

```
gcc -Wall -W -pedantic -o vert_hist vert_hist.c
```

Example: Below is an example assuming `vert_hist` has been compiled and lives in the current directory. Each * represents a count of 1.

```
echo This Is a TeST--Hello, MississippI. Yippie | ./vert_hist
```

```
        *
        *          *
        *          *
        *          *
        *       *  *
    *   *       *  **
    *  **  *    *  **
*   *  **  ** **  **    *
abcdefghijklmnopqrstuvwxyz
```

(*Hint:* Learn about the function `getchar`. Furthermore, you can use `cat` and pipe to test this program. For example, the command "`cat poem.txt | ./vert_hist`" will read `poem.txt` and send it to the stdin of your program.)

(*Hint #2:* A few functions in `ctypes.h` will prove to be useful. For example, what does `tolower` do? How about `isalpha`? Use the man command to find out.)

Task 2: Entab/Detab Utility (15 points)

The tab character (`'\t'`) is both useful and annoying. You will write a program, called `tabutil.c`, with the following features:

- `tabutil -d <num_spaces>` will read from stdin and print to stdout, replacing every tab occurrence with `num_spaces` consecutive spaces. For example, `tabutil -d 4` will turn each tab into 4 spaces.
- `tabutil -e <num_spaces>` will read from stdin and print to stdout, turn each occurrence of `num_spaces` consecutive spaces into a tab (i.e., does the reverse of `-d`).

Code and Compilation: Your code will be one standalone file called `tabutil.c`. Use the following command to generate an executable `vert_hist`. Your code must compile clean with this command.

```
gcc -Wall -W -pedantic -o tabutil tabutil.c
```

Task 3: Game of Life (35 points)

In this problem, you will be implementing Conway's Game of Life. Back in 1970, British Mathematician J. H. Conway created the Game of Life, a simulation of a population of lifeforms over generations. The simulation takes place on a 2-dimensional grid. We provide a description of the Game of Life below.

For many of you, this is your very first time programming in C. For this reason, we have done the design work for you, providing function prototypes, each of the function skeletons, and ample comments to help you get started.

The Game of Life

The Game of Life takes place on a 2-dimensional grid. Each grid cell can house zero or one organism. The game starts with an arbitrary initial grid—that is, some cells are inhabited and some are empty. Cells that are occupied are *alive*; the other cells are *dead*.

From this initial grid, the next generation of population is obtained by the following rules:

- a cell has 8 neighbors, those cells that immediately surround it vertically, horizontally, and diagonally (on both diagonals).
- If a cell is alive and has 2 or 3 live neighbors, it will remain alive in the next generation.
- If a cell is alive and has fewer than 2 live neighbors, it will die of loneliness.
- If a cell is alive and has 4 or more live neighbors, it will die of suffocation.
- If a cell is dead and has exactly 3 live neighbors, a new organism will be born in that cell. Otherwise, it remains dead in the next generation.
- **Importantly:** All births and deaths take place at exactly the same time. That is to say, all the cells' neighbors are counted simultaneously, based on the current generation, before the next generation is produced. This means, for example, that it is possible for a new cell to be born out of current live neighbors that will be dead in the next generation.

In the original game, the rules assume an infinite grid in all directions. However, we can only store a finite-sized grid, so what to do with cells that are beyond our finite grid? For this problem, treat any cell that isn't on the finite grid as dead.

To gain a better understanding of the game, you can find a simulator at `http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/`. Use this to ensure that you understand the rules and to determine whether or not your implementation is working correctly.

Your Task

We have prepared for you the following files:

- `life_engine.c` - the file containing function stubs that you will complete, though some functions have already been written for you.
- `life_engine.h` - the header file (that you will not modify) corresponding to functions and datatypes used in the simulation. Even though you aren't changing this file, you should look through it.
- `life_driver.c` - the "driver" code that will make use of your functions in running a game of life simulation. Once compiled (see below), you run it as `life <starting file> <num iterations>`.
- `sample_life.txt` is a sample starting file.

You will be filling in the provided file, `life_engine.c`. You must complete each of the provided functions and should not add any additional ones. Each of your functions should have the behavior specified in the explanatory comments, making calls to other functions as you see fit. Specifically, you will implement the following functions:

- `get_index`
- `is_in_range`
- `is_alive`
- `count_live_nbrs`
- `make_next_board`

Code and Compilation: This task is special in that it involves multiple files. Our goal is to generate an executable called `life`. The following command will compile the source files and link them (i.e., combine) as `life`. Your code must compile clean with it.

```
gcc -o life -Wall -W -pedantic life_engine.c life_driver.c
```

Task 4: Unscramble (35 points)

You will write a program that takes in 2 files: a dictionary file and a file listing jumbled words. Your binary will be called `unscramble` and will be run using

```
unscramble <dictionary> <jumbles>
```

The <`jumbles`> file contains a bunch of scrambled words, one word per line. Your job is to print out these jumbles words, 1 word to a line. After each jumbled word, print a list of real dictionary words that could be formed by unscrambling the jumbled word. The dictionary words that you have to choose from are in the <`dictionary`> file. As an example, the starter package contains two sample input files and the result should look as follows. The order that you display the dictionary words on each line can be different; however, the order that you print out the jumbled words should be identical to the order in your input. The sample out below is skipping a few lines to save handout space (your output must not omit these lines).

```
nwae: wean anew wane
eslyep: sleepy
rpeoims: semipro imposer promise
ettniner: renitent
ahicryrhe: hierarchy
dica: acid cadi caid
dobol: blood
nyegtr: gentry
cukklen: knuckle
warllc: NO MATCHES
addej: jaded
baltoc: cobalt
mycall: calmly
dufil: fluid
preko: poker
... lines omitted ..
milit: limit
```

```
pudmy: dumpy
hucnah: haunch
genjal: jangle
```

(When a word doesn't unscramble to any dictionary word, you'll say "NO MATCHES" as shown in the example above.)

Logistics: Your code should follow good programming style. This includes writing each logical unit of code in a self-contained function that is not too long. We're a little impatient: On the sample input, we are willing to wait only up to around 3 seconds on Hamachi.

The sample files can be found in the handout directory (see the beginning of this handout).

Code and Compilation: Your code will be one standalone file called `unscramble.c`. Use the following command to generate an executable `unscramble`. Your code must compile clean with this command.

```
gcc -Wall -W -pedantic -o unscramble unscramble.c
```

ASSUMPTIONS: For ease, you can assume that no words are longer than 50 letters and the dictionary file contains no more than 500,000 words.

TIPS: You should observe that a word s unscrambles to a dictionary word w if when you sort the letters of s lexicographically and similarly sort the letters of w, they result in the same sequence of letters. For example, `sorted("eslyep")` is `eelpsy`, which is the same as `sorted("sleepy")`. Furthermore, it is easy to sort an array (or a string—i.e., an array of characters) using the built-in function `qsort`.