CS 220 - Homework 3

Due: ~~Tuesday, October 2nd~~ Thursday, October 4th by 11pm.
This assignment is worth 60 points.

Learning Objectives
This assignment will allow you to practice using basic control structures in C, strings, file and command-line I/O, and writing and testing functions that deal with arrays. You will also continue to build familiarity with git version control and the make utility.

Collaboration
This is an individual assignment. This means you must not show your working code to another student, and should discuss with each other only the assignment requirements and expectations. See course staff for coding help. Check the office hours doc to see when help will be available in the CS ugrad lab, Malone 122, or visit instructor office hours.

Project Scaffolding ("Starter Code")
In the class public git repository, the folder `cs220fa18-public/homework/hw3/` contains several files you should use as a starting point for this assignment. From within the public repo, type **git status** to confirm your local copy is in good shape, then type **git pull** to copy this folder. Then copy these files to your personal `cs220` repository to begin your work. If you compile the files as posted, you'll see a number of "unused parameter" errors reported, because the files contain incomplete function definitions. As you work on filling in the function definitions, you'll eventually eliminate these warnings. By the time you submit your code for grading, none of these warnings (or any others) should be reported.

The Problem: Word Search
Write a program to solve a word search puzzle. Your program will accept a single command-line argument which is the name of a text file containing the word-grid. After reading the grid of letters from that file, you will read from stdin words for which to search. For each given word, you will print to stdout the location of all copies of that word in the grid.

Required Program Functionality:
- Unless otherwise noted, your program should exit with return value 0.
- The program expects a filename as a command-line argument. If none is supplied, the program should output `"Please enter a command line argument."` and exit with return value 1. If a filename is supplied but cannot be opened successfully, output `"Grid file failed to open."` and exit with return value -1.

- A proper grid file contains an N x N grid of characters where N is at most 10 and all characters in the grid are alphabet letters. That is, there should be N lines containing N letters (any mix of upper or lower case), followed by an empty line, with no spaces between the letters. Anything in the grid file after the empty line can be ignored.
- There are only 3 types of invalid grids that your program must detect and report. In any of the following cases, your program should output `"Invalid grid."` and exit with return value -2:
    1. if two of the grid rows have differing numbers of letters
    2. if the number of rows doesn't match the number of columns
    3. if the number of rows (and columns) is 0 or greater than 10
- Search words read from stdin are separated by whitespace (any amount). The program should continue processing search words until it reaches the end-of-input (ctrl-d if stdin is not redirected).
- Words may appear in the grid forwards or backwards, horizontally or vertically, with the characters in order in adjacent cells. Different words might share characters. The program must search for each word in all 4 directions: right, left, down, up.
- The program should be case insensitive in its search.
- For each appearance of a search word found in the grid, your program should output to stdout a line of text containing the matched word, the row number of the start of the word, the column number of the start of the word, and the direction of the match: R for right, L for left, D for down, U for up (as shown in the examples below). Rows and columns shown in the output are to be numbered starting with 0.
- If multiple copies of a single search word are present in the grid, the program must locate and report each of them. The required order of the output lines for multiple appearances of a single search word w is as follows:
    1. All appearances of w directed Rightwards
    2. All appearances of w directed Leftwards
    3. All appearances of w directed Downwards
    4. All appearances of w directed Upwards

  Multiple appearances within each directional category (R, L, D, U) should be reported in order of occurrence of the **first** letter in the search word, **when the grid is read row-by-row from left to right, starting with row 0**. For example, if word w appears in the grid directed leftwards starting at row 4, column 3 and again directed leftwards starting at row 1, column 6, then the appearance starting at row 1, column 6 should be listed before the other leftward appearance, and both leftward appearances should be after any rightward appearances are reported, but before any downward or upward directions are reported.
- If no matches for a word are found, print the search word followed by `" - Not Found"` , as shown below.

Sample Runs:

grid.txt, example input file with empty line at the end

```
pitk
olpe
pkey
tope
```

Sample run #1, interactive user input shown in bold

```
./word_search grid.txt
tip pop key
tip 0 2 L
pop 0 0 D
pop 2 0 U
key 2 1 R
key 0 3 D
```

Sample run #2, interactive user input shown in bold, interleaved with program output

```
./word_search grid.txt
tip
tip 0 2 L
pop key
pop 0 0 D
pop 2 0 U
key 2 1 R
key 0 3 D
```

Sample run #3, redirected input using Unix echo and pipe

```
echo "tip pop key nope" | ./word_search grid.txt
tip 0 2 L
pop 0 0 D
pop 2 0 U
key 2 1 R
key 0 3 D
nope - Not Found
```

Implementation Requirements:

- The grid must be stored in a 2D array whose size will be set as 10x10 at compile time.
- In the provided scaffolding file `search_functions.h`, a constant named `MAX_SIZE` is defined as 10. Your code must utilize `MAX_SIZE` rather than the literal value 10.

- Not every grid file will use the entire `MAX_SIZE` x `MAX_SIZE` array; your program should be able to handle this. The top left character in the grid (no matter the size of the grid) should be said to reside at row 0, column 0.
- You must not use any global (or extern) variables.
- Your main function must reside in a file named `word_search.c`, but you must factor out reusable code into (more easily) testable functions.
- Minimally, you must supply functions with exactly the following declarations, all of which are provided for you in scaffolding file `search_functions.h`:

```
/* Given a filename and a MAX_SIZExMAX_SIZE grid to fill, this function populates the grid
and returns n, the actual grid dimension. If filename_to_read_from can't be opened, this
function returns -1. If the file contains an invalid grid, this function returns -2.
*/
int populate_grid(char grid[][MAX_SIZE], char filename_to_read_from[]);

/* Each of these 4 functions returns the number of times the given word string was found in
the grid facing the direction indicated in the function name. Parameter n indicates the actual
size of the grid. The function sends corresponding output to the specified file pointer, which
already points to an open stream. Output lines must appear in order of the first character's
appearance in a left-to-right scan of each row beginning with row 0.
*/
int find_right(char grid[][MAX_SIZE], int n, char word[], FILE *write_to);
int find_left (char grid[][MAX_SIZE], int n, char word[], FILE *write_to);
int find_down (char grid[][MAX_SIZE], int n, char word[], FILE *write_to);
int find_up   (char grid[][MAX_SIZE], int n, char word[], FILE *write_to);

/* This function is similar to the 4 functions above, but reports ALL appearances of the given
word, in the required R,L,D,U order.
*/
int find_all  (char grid[][MAX_SIZE], int n, char word[], FILE *write_to);
```

- You must separate out declarations of all non-main functions in your program into a header file named `search_functions.h`. Include there the required functions listed above, plus additional functions you elect to create. The accompanying file `search_functions.c` will implement the functions whose headers appear in `search_functions.h`. The file `word_search.c` will use those functions to implement the required program functionality described above. Both of these .c files must `#include "search_functions.h"`.
- In addition to the source files mentioned above, you are required to submit a testing main function, a working make file, and git log as described below.

Testing

In addition to the source files listed above, you must write and submit a C file named `test_search_functions.c` that will contain a tester `main` function and `#include "search_functions.h"`. The file will also contain at least one helper test function per each of the functions declared in `search_functions.h`. For example, if the declaration for a function named `fun1` appears in `search_functions.h`, then `test_search_functions.c` should include a corresponding function named `test_fun1`. The job of each of the helper functions in `test_search_functions.c` is to test, using `assert` statements, that the corresponding function declared in `search_functions.h` works properly. The `main` function in `test_search_functions.c` will call each helper function in turn, and, if all of the tests pass, should output `All search_functions tests passed!` to indicate success.

Makefile

You must submit a working makefile with at least a `word_search` target that builds your program to create an executable named `word_search` and a `test` target that builds and runs your testing executable. To be considered fully functional, your makefile should correctly build the executables with the appropriate flags when (and only when) one of the relevant source files has changed. The makefile should be named `Makefile` (spelling and case are important).

Git Log

You are expected to use git to backup your progress on this project. Include in your submission a copy of the output of `git log` showing at least five commits to the repository of code from this assignment, with meaningful commit messages. (You should commit early and often---likely every time you have a version that compiles---so you should probably have many more than five of these). Just before submitting your work, save the output into a file called `gitlog.txt` by executing `git log > gitlog.txt`.

Submission Checklist

Your Gradescope submission should contain at least the following files:

1. `word_search.c`
2. `search_functions.h`
3. `search_functions.c`
4. `test_search_functions.c`
5. `test1.txt, test2.txt, test3.txt`, and any additional data files created by you which your tester functions require
6. `Makefile`
7. `gitlog.txt`

Hints, Tips, and a Challenge:

- HINT: To implement a case-insensitive search, consider converting to lowercase all characters read in as part of the grid or as search words.
- HINT: Make the horizontal search as clean and efficient as possible before attempting to write searches in other directions.
- HINT: Instead of writing a new search function to search in reverse for a word, reverse the word itself and apply your original searches (horizontal, vertical) to it.
- TIP: Make use of `gdb` to help debug your program.
- TIP: Write your function tester functions early, rather than waiting until the last minute. Their purpose is to help you catch bugs in the functions early on, so the bugs don't cause problems when used by main. Using this approach as intended will help you track down bugs more easily.
- TIP: If in testing your program's output, you want to determine whether two files are identical (e.g. when one file contains your program's actual output and the other contains the expected output), the completed function named `file_eq` provided for you in the `test_search_functions.c` file will be useful!
- TIP: In addition to writing tests for individual helper functions that are included in your submission, you'll need to create and utilize "end-to-end" tests for your complete word search as well. By "end-to-end" testing, we simply mean running the entire program and checking if it behaves as desired on a number of different inputs. Your end-to-end tests do not need to be handed in.
- **Extra Challenge (not for credit)**: Find words directed diagonally as well, by searching in 8 directions, rather than 4. This is a good way to improve your grasp of the language and concepts.