Programming Assignment #2: Word Search

COP 3502, Fall 2015

Due: Friday, September 25, *before* 11:59 PM

Abstract

In this programming assignment, you will implement a word search puzzle solver. In doing so, you will gain advanced experience working with dynamic memory management, string manipulation, structs, and file I/O in C. You will also learn to manage projects that import local header files (as opposed to standard system header files).

Attachments

WordSearch.h, dictionary.txt, large-dictionary.txt, sample-puzzle-1.txt, sample-output-1.txt

(Note: *large-dictionary.txt* is included to help you test larger puzzles of your own creation.)

Deliverables

WordSearch.c

(Note: Capitalization of your filename matters!)

1. Overview

In this program, you will be given a dictionary and a word search grid, and your goal will be to print out all the words from the dictionary that occur in the word search puzzle, along with a count of the number of unique occurrences of those words. Words in the grid may be spelled vertically, horizontally, or diagonally, in any direction (forward or backward, up or down, and so on). For example, given the following dictionary and word search puzzle, the output would be as shown:

dictionary.txt	sample-puzzle-1.txt	sample-output-1.txt
5	3 5	ab (1)
ab	abcde	be (1)
be	efghi	he (2)
he	jklem	hi (1)
hi		
him		

(Notice that “he” occurs twice in the grid: once diagonally up and right from the 'h', and once vertically, going down from the 'h'.)

For a large dictionary, it would be really inefficient to take each word, then search for how many times it occurs in the word search puzzle. Instead, you will have to construct every possible string of characters in the puzzle, one-by-one, and then search the dictionary to see if that string is an actual word. To make this process even faster, you will have to search the dictionary using binary search.

2. Input Specifications

You will read the dictionary from a file called `dictionary.txt`. You should assume the dictionary file will be in the directory where we run your program, so open it like so:

```
FILE *ifp = fopen("dictionary.txt", "r");
```

Do **NOT** hard-code a file path when opening the file. For example, this is undesirable:

```
// WRONG! Do not hard-code a directory!
FILE *ifp = fopen("C:\Users\John\Desktop\Program2\dictionary.txt");
```

You should use `fscanf()` to read the contents of the dictionary file. The file will start with a single integer, *n*, on a line by itself. Following that, there will be *n* lines, sorted in alphabetical order, each containing a single word consisting entirely of lower-case letters (no spaces, digits, or other symbols). The maximum length of any word in the dictionary is defined in the attached `WordSearch.h` file. You should refer to that `#defined` identifier instead of hard-coding a max word length in your source file.

The wordsearch puzzle itself should be read from stdin (i.e., the keyboard); so, use `scanf()`, not `fscanf()`, to read the puzzle. The puzzle input will start with two integers, separated by a space: the *height* ($3 \leq height \leq 300$) of the puzzle grid, followed by the *width* ($3 \leq width \leq 300$) of the puzzle grid.

Following that will be *height* lines of text, each containing *width* number of characters.

Because the size of the dictionary and puzzle are not known ahead of time, you will have to use dynamic memory allocation to get them into memory in your program.

3. Output Specifications

Every time you find a word in the word search puzzle that is also in the dictionary, you will increment a counter within the dictionary. (A struct for storing the dictionary and counting these occurrences is described in the following section.) Then, at the end of the program, you will loop through the dictionary and print out (in alphabetical order, which is the same order in which words are stored in dictionary.txt) each word that occurred at least once in the word search puzzle. After every such word, you should print a single space, followed by the number of times that word occurred (in parentheses).

Although we have included some sample input files to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

4. WordSearch.h

The header file included with this assignment contains struct definitions and functional prototypes for the word search functions you will be implementing. You should #include this file from `WordSearch.c`, like so:

```
#include "WordSearch.h"
```

The “quotes” (as opposed to <brackets>), indicate to the compiler that this header file is found in the same directory as your source, not a system directory. So, you will need to place this in the directory where you compile your code. For instructions on importing this header file into a project with Code::Blocks, see Section 6, “Compilation and Testing (CodeBlocks).”

You should not modify `WordSearch.h` in any way, and you should not send `WordSearch.h` when you submit your assignment. We will use our own copy of `WordSearch.h` when compiling your program.

If you write auxiliary functions in `WordSearch.c` (which is strongly encouraged!), you should ***not*** add those functional prototypes to `WordSearch.h`. Just put those functional prototypes at the top of your `WordSearch.c`.

`WordSearch.h` is here to help you. Including this file (without modifying it) will help you ensure that you’re using the correct struct definitions and functional prototypes necessary to earn a perfect score on this program. Also, learning to include custom header files is an essential skill for software developers. This is a gentle gateway to building projects with multiple header and source files.

The basic structs you will use to implement the word search are defined in `WordSearch.h`, and are as follows:

```
typedef struct WordSearchPuzzle
{
    char **grid;   // dynamically allocated 2D character grid
    int height;    // height of the wordsearch puzzle character grid
    int width;     // width of the wordsearch puzzle character grid
} WordSearchPuzzle;

typedef struct Dictionary
{
    char **words;  // pointer to array of strings
    int *counts;   // number of times each word occurs in the puzzle grid
    int size;      // number of words in dictionary
} Dictionary;
```

These structs hold information about the dictionary and word search puzzle grid you'll read in, and having all those pieces bundled up in structs means that you can pass them as neat little packages to your functions, instead of passing several distinct (yet interdependent) arguments.

Both structs contain a double `char` pointer that can be used to set up a 2D `char` array (which is just an array of `char` arrays, otherwise known as an array of strings). These will have to be allocated dynamically at runtime, which will probably be the bane of your existence for the next week or so. (See the notes in Webcourses from August 30 for more on strings in C and dynamic allocation of 2D arrays.)

The `WordSearchPuzzle` struct also has `height` and `width` variables, which store the height and width of the puzzle grid. The `Dictionary` struct has a `size` variable that stores the number of words in the dictionary's `words` array, and a `counts` variable that you should initialize to a dynamically allocated array of zeros and then use to count how many times a particular word is encountered in the puzzle. More explicitly, after the whole puzzle is processed, count[i] should indicate the number of times that words[i] occurred in the puzzle.

5. Function Requirements

In the source file you submit, `WordSearch.c`, you must implement the following functions, and you must actually use these functions to implement your solution. You may implement any auxiliary functions you need in order to make this program work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. Also, note that in this section, I often refer to `malloc()`, but you're welcome to use `calloc()` or `realloc()` instead, if you want.

```
int readDictionary(Dictionary *dict);
```

> **Description:** Read the dictionary from `dictionary.txt`. Dynamically allocate space for the dictionary's internals (the `words` and `counts` arrays), properly initialize those fields, and

initialize the struct's `size` member to be the number of words in the dictionary. For each string you place into the dictionary, allocate just enough space – no more – to store that string (and its null terminator). If any calls to `malloc()` fail, be sure to free any memory you've allocated in this function in order to avoid memory leaks before returning.

Returns: 1 if the dictionary was read successfully, and 0 if reading the dictionary failed for any reason (e.g., the file could not be found, or a call to malloc() failed).

```
void destroyDictionary(Dictionary *dict);
```

Description: Free any dynamically allocated memory associated with the dictionary, set all pointers to `NULL`, and set its `size` member to zero.

```
int readPuzzle(WordSearchPuzzle *puzzle);
```

Description: Read the puzzle (starting with height and width) from stdin (i.e., the keyboard). Dynamically allocate space for the 2D grid of characters, and populate them with the word search puzzle. If any calls to `malloc()` fail, be sure to free any memory you've allocated in this function in order to avoid memory leaks before returning.

Returns: 1 if reading the puzzle was successful, 0 otherwise (namely, if any calls to `malloc()` fail).

```
void destroyPuzzle(WordSearchPuzzle *puzzle);
```

Description: Free any dynamically allocated memory associated with the puzzle, set all pointers to `NULL`, and set its `height` and `width` members to zero.

```
int checkString(Dictionary *dict, char *str);
```

Description: Perform a binary search through the dictionary to see if `str` is an actual word. If so, update the `counts` array accordingly: if `str` is found at `words[i]`, increment `counts[i]`.

Returns: 1 if `str` is found in the dictionary, 0 otherwise.

```
int main(void);
```

Description: You must write a main() function to process the puzzle and produce the expected output for this program. Please match the functional prototype given above.

Returns: Your `main()` function must return zero in order for your program to pass test cases.

6. Compilation and Testing (CodeBlocks)

The key to getting your program to include a custom header file in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, importing `WordSearch.h` and the `WordSearch.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "WordSearch").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **each** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Reminder: Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 8, "Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

7. Troubleshooting: File Not Found Errors with Mac OS X

If you encounter a file-not-found error on your Mac, you might need to put your input files in an unusual directory. Use my `pwd()` ("print working directory") function to print the directory where your IDE wants those input files. `pwd()` is defined in file-read.c, posted September 4 in Webcourses. Just plunk it into your source code and call it from `main()`. (Just don't forget to delete that call before submitting your code!)

8. Compilation and Testing (Linux/Mac Command Line)

To compile your program at the command line (in the same directory as `WordSearch.h`):

```
gcc WordSearch.c
```

By default, this will produce an executable file called `a.out` that you can run by typing:

```
./a.out
```

If you want to name the executable something else, use:

```
gcc WordSearch.c -o WordSearch.exe
```

...and then run the program using:

```
./WordSearch.exe < sample-puzzle-1.txt
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./WordSearch.exe > whatever.txt
```

This will create a file called `whatever.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided a sample output file. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample-output-1.txt
```

If the contents of `whatever.txt` and `sample-output-1.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt sample-output-1.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample-output-1.txt
6c6
< he (x2)
---
> he  (2)
seansz@eustis:~$ _
```

9. Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, "Where do I even start with this assignment?! I'm in way over my head!" Don't panic! We're here to help in office hours, and here's my general advice on starting the assignment (as well as a few hints and spoilers):

1. Try to implement just one small piece of the program at a time. Always stop to compile and test your code before moving forward.

2. Start with the `readDictionary()` function. Once you read the dictionary into the Dictionary struct, print out the contents of that struct to make sure everything got in there okay. Test it with a small dictionary, then test it with a few progressively larger dictionaries.

   Here's a big hint for the `readDictionary()` function: One thing that might be helpful would be to create a static char array of length (`MAX_WORD_LENGTH + 1`). (`MAX_WORD_LENGTH` is defined in `WordSearch.h`.) Read words from the dictionary file into that array, then use `strlen()` to figure out how long they are and allocate space in the dictionary's `words` array to copy that string over before moving on to the next one.

3. Having trouble with `readDictionary()`? Move on to `readPuzzle()`. Test it with a small puzzle (print the grid out character-by-character), and then test it out with progressively larger puzzles. Try puzzles that are square, puzzles that are wider than they are tall, and puzzles that are taller than they are wide.

4. Implement binary search in the `checkWords()` function. You're free to adapt the version I showed you in class, as long as you include a comment citing me as your source for that code. (You should not, however, look at code from other outside sources while working on this assignment.)

   To adapt binary search for an array of strings, you'll need the `strcmp()` function. Note that `strcmp(str1, str2)` returns zero if `str1` and `str2` are the same, a negative integer if `str1` comes before `str2` in sorted order, and a positive integer if `str1` comes after `str2` in sorted order. Test out your binary search function by calling it directly a few times from `main()`, passing it words that you know are and are not in the dictionary.

5. Maybe leave the `destroyDictionary()` and `destroyPuzzle()` functions to the end. These might be really tricky, and they might cause your program to crash if you haven't been careful with memory.

6. Constructing all possible strings in the word search puzzle grid is probably the trickiest part of this assignment. Starting with a letter in the grid, build all possible strings moving left, right, up, down, and along all four diagonals. Do this for every letter in the grid. Every time you build a new string, pass it to `checkWords()` to determine whether it's in the dictionary or not.

7. Remember that strings need null terminators. That's really important with the strings in `readDictionary()`, in particular.

10. Deliverables

Submit a single source file, named `WordSearch.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

11. Special Restrictions

1. You must return zero from `main()` in order to receive credit for test cases.
2. Do not use global variables (although `#define` directives are totally fine).
3. Be sure to declare all your variables at the tops of your functions. No mid-function variable declarations.
4. Do not submit the `WordSearch.h` header file with your source code. Just submit `WordSearch.c`. We will use our own version of the header file when testing your program.
5. Be sure to include your name and PID as a comment at the top of your source file.

12. Grading

The tentative scoring breakdown for this programming assignment (not set in stone) is:

50%	Correct Output for Test Cases
20%	Unit Testing (see details below)
20%	Implementation Details (manual inspection of your code)
10%	Comments and Whitespace

Your program must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

For this program, we will likely also be unit testing your code. That means we might create tests where we rip out one of the required functions and then run it independently, with our own `main()` function, to see if it does everything it's supposed to do. So, for example, if your program produces correct output but your `readDictionary()` function is simply a skeleton that does nothing no matter what parameters you pass to it, your program will fail the unit tests.

Additional points will be awarded for style (proper commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `clearDictionary()` function to see that it is actually freeing up memory properly.