CSC230 Homework 4

This homework is to be done individually.

You may use any functions in the standard library on this assignment.

Learning Outcomes

- **Write small to medium C programs** having several separately-compiled modules.
- Explain what happens to a program during **preprocessing**, lexical analysis, parsing, code generation, code optimization, linking, and execution, and **identify errors that occur during each phase**. In particular, they will be able to describe the differences in this process between C and Java.
- **Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.**
- **Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.**
- **Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow**.
- **Explain, inspect, and implement programs** using structures such as enumerated types, unions, and constants and **arithmetic, logical, relational, assignment, and bitwise operators**.
- **Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.**
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- **Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.**
- **Write, debug, and modify programs using library utilities**, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, **standard I/O, and file I/O**.
- **Use simple command-line tools to design, document, debug, and maintain their programs.**
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- **Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.**

Program - wordsearch.c & wordsearch_test.c

You will write a word search solver. A word search is a puzzle where a matrix of characters are provided and the player attempts to find the words hidden in them. In the following puzzle, there are four words: `"apple"`, `"pie"`, `"food"`, and `"cake"`.

```
a e k a c
p f o o d
p i z a s
l d e n d
e v q i o
```

Words can be oriented in any direction. For example, in the above puzzle, the words could be vertical (e.g., `"apple"`, which is vertical on the left side of the puzzle), left-to-right (e.g., `"food"`, which is on the second row), right-to-left (e.g., `"cake"`, first row), and diagonally (e.g., `"pie"`, which starts in column one, row two and goes diagonally to the lower right).

Your task is to write a program that will find all of the words in the word search.

Requirements

Program Initialization

The wordsearch program shall receive a command line argument that provides the name of a file that contains information for creating a word search puzzle. If no command line argument is provided, the user should be prompted for a wordsearch file by prompting with the string **`"Word search file? "`**. If the file does not exist, the error message **`"Invalid file\n"`** is printed and the program exits with an error code of 1.

Word Search File Processing

The word search file shall be formatted in the following way:

- Line 1: Contains an digit with the number of rows in the word search. The digit is immediately followed by a new line character (e.g., `'\n'`). The digit must be greater than or equal to 3 and less than or equal to 9.
- Line 2: Contains an integer with the number of columns in the word search. The integer is immediately followed by a new line character (e.g., `'\n'`). The integer must be greater than or equal to 3 and less than or equal to 9.
- Line 3 through Line (rows + 2): Contains column number of characters. The last character is followed by a new line character (e.g., `'\n'`). A valid character is between `'a'` and `'z'`, inclusive or between `'A'` and `'Z'`, inclusive. If the character is uppercase, it should be transformed into a lower case letter.

If a file does not meet the format described above, the error message **`"Improperly formatted file.\n"`** is printed and the program exits with an error code depending on the improper format. The improper format cases and their error codes are listed in the table below. If more than one case is possible, exit with the lowest error code.

Improper File Format Case	Error Code
The file is empty	2
Line 1 does not contain a single digit followed by a new line character	3
Line 2 does not contain a single digit followed by a new line character	4
Line 1's row value is less than 3	5
Line 2's column value is less than 3	6
There are too few elements in a row	7
There are too many elements in a row	8
A row is missing	9
There are too many rows	10
A character in a word search row is not between 'a' and 'z', inclusive or between 'A' and 'Z', inclusive	11

Word Searching

The program shall print the wordsearch using the format specifier **`"%c "`** for each character (this means that a line will end in a space followed by a new line character) and prompt the user for a word they have found in the word search using the prompt **`"Word (or ! to quit)? "`**. The word is read in and the program will search through the word search for the word. If the word is found, the lowercase letters for the word are replaced with upper case letters and the wordsearch is printed. The printing and prompting is repeated until the user enters a string that begins with the character `'!'` to quit. The program exits with an exit code of 0 signaling a valid exit condition.

Design

Your program must contain the following functions. You can create additional functions if that helps, but the following three will be tested.

`int matches( char *target, int row, int col, int direction )`

This function starts with the character at the intersection of row and col and begins looking in the specified direction for the target word. The directions are as follows:

```
1: Right
2: Up-Right Diagonally,
3: Up
4: Up-Left
5: Left
6: Down-Left
7: Down
8: Down-Right
```

This function returns `1` if the word is found, a `0` if the word is not found, and `-1` if an error occurs. The errors that must be checked are running past the edge of the matrix and providing an illegal direction (not a value between 1 and 8).

This function should have no side effects.

`int find( char *target, int *row_start, int *col_start, int *row_end, int *col_end )`

This function finds the target word in the word search. If the word is found, the position of the first character is saved in **`row_start`** and **`col_start`** and the position of the last character is saved in **`row_end`** and **`col_end`**. If the word is not found, none of the positions are changed.

If the word is found, this function returns `1`. If the word is not found, it returns `0`. This function should have no side effects beyond the modification of **`row_start`**, **`col_start`**, **`row_end`**, and **`col_end`** if the word is found.

The rows and columns are zero indexed. The upper left corner of the wordsearch is `[0][0]`. The bottom right corner of the wordsearch is **`[puzzle_h - 1][puzzle_w - 1]`**, where **`puzzle_h`** is the height or number of rows and **`puzzle_w`** is the width or the number of columns.

`int load( char *filename )`

This function loads a word search puzzle file located at **`filename`** formatted as specified in the requirements. It will load the word search into a static array named **`puzzle`**, and set the global variables **`puzzle_w`** and **`puzzle_h`** to the width and height of the loaded puzzle.

This function returns 0 on success and non-zero in the event of a file I/O error. The value returned is the error code specified in the requirements. During normal operation, the **`main`** function should print the error message "**`Invalid file\n`**" or "**`Improperly formatted file\n`**" and quit with the returned error code if the return value of **`load`** is non-zero. The **`load`** function should not quit the program.

Assumptions

- There will be no palindromes in the word search (no words that are the same backwards and forwards, such as "racecar")
- There will be no filenames longer than 10 characters.

Implementation

We have provided three files to help you start your program: **`wordsearch.h`**, **`wordsearch.c`**, and **`wordsearch_test.c`**.

`wordsearch.h`

> **`wordsearch.h`** is a header file. A header file is a place to store information that is shared between different C modules. In this case, the required function prototypes are shared between **`wordsearch.c`** and **`wordsearch_test.c`**. Do not modify the header file!

`wordsearch.c`

> You will write the wordsearch functionality in this file. **`wordsearch.c`** must define the behavior of the three required functions. We have started the file for you by including **`wordsearch.h`**. Therefore, you do not need to have function prototypes in **`wordsearch.c`** for the three required functions.

`wordsearch_test.c`

> You will write unit tests in this file. We have started the file for you by including **`wordsearch.h`** and an implementation of **`assert_equals()`**. Therefore, you do not need to have function prototypes in **`wordsearch.c`** for the three required functions.

Testing

For Homework 4, you will write unit tests and run system tests on your wordsearch application.

Unit Testing

> C only allows a single main per compiled executable. When unit testing, we typically create a separate file for our tests; however, with C, we're unable to compile together two files that each have a main. To mitigate the problem, you will use conditional compilation to either include or not include the main function in the wordsearch.c program. When the preprocessor directive **`TESTING`** is defined and equal to 1, the **`main`** function in **`wordsearch`** should not be compiled and instead the **`wordsearch_test main`** will be compiled allow for the execution of unit tests that will exercise the required functions for correctness.
>
> You will write the following unit tests:

Function	Number of Unit Tests	Notes
`load()`	12	Each unit test must consider one of the program exit codes listed in the requirements. You may use the provided **`in_*`** files for your unit tests.
`find()`	4	Each unit test must be from a different equivalence class. At a minimum, you should consider 4 different directions.
`matches()`	4	Each unit test must be from a different equivalence class. At a minimum, you should consider 4 different directions.

Each unit test must use the following structure. You may write helper functions and loops to run many similar tests:

```
printf("Testing [function]([arg1],[arg2],...);\n");
int actual = [function]([arg1],[arg2],...);
assert_equals(expected, actual);
```

Use the provided **`assert_equals`** function in **`wordsearch_test.c`**:

```
void assert_equals(int expected, int actual)
{
    if (expected != actual) {
        printf("ASSERTION FAILED: Expected %d, got %d.\n", expected, actual);
    } else {
        printf("ASSERTION PASSED\n");
    }
}
```

When compiling your program for unit testing, you must use the following command:

```
% gcc -Wall -std=c99 -DTESTING wordsearch.h wordsearch.c wordsearch_test.c
-o wordsearch_test
```

The command will utilize a header file that contains the function prototypes for the three required functions, your source file and your unit test file to generate an executable named **`wordsearch_test`** that will run the 20 unit tests. The regular main functionality MUST NOT be included!

You may also use the provided Makefile

```
% make clean
% make all
```

You can redirect the output of your unit tests to a file for inspection using the following command:

```
% ./wordsearch_test > unit_test_output
```

System Testing

When compiling your program, you must use the following command:

```
% gcc -Wall -std=c99 wordsearch.h wordsearch.c -o wordsearch
```

OR you may use the provided Makefile:

```
% make clean
% make all
```

Some test cases for the program are provided. We have also provided a script, **`test.sh`**, that will execute the provided tests and report the results. There should be no difference between your actual output and the expected outputs. ***Grading will test additional inputs and outputs, so you should test your program with inputs and outputs beyond the provided example.*** You may edit the **`test.sh`** file to help you with additional tests.

The test cases, testing script, template file, and Makefile are provided for you in **`hw4_starter.tar`**. You can untar the starter file using the command:

```
% tar xvf hw4_starter.tar
```

...edit **wordsearch.c**, compile it, test it, etc...
When you are ready to run the full suite of tests, execute:

```
% chmod +x test.sh

% ./test.sh
```

Instructions for Submission

Submit the following:

- **wordsearch.h**
- **wordsearch.c**
- **wordsearch_test.c**
- **Makefile**
- **unit_test_output** - the redirected output from running your unit tests
- **actual_*** - all of the actual outputs generated through system testing

in a **tar** file to **"Homework 04 Submit"** locker through the Moodle Course Website. By submitting the unit test results and the actual results from the tests, you will prove that you have tested your program with the minimum set of the provided acceptance tests.

We recommend that you keep the files for Homework 4 in a separate directory from your other course materials. Assuming that this is so and you're currently in the Homework 4 directory, use the following commands to build and **tar** your files:

```
% make clean
% make all
% ./wordsearch_test > unit_test_output
% ./test.sh
```

(Make sure all tests passed - otherwise fix your code)

```
% tar cvf unityid_04_homework.tar wordsearch.* wordsearch_test.c Makefile
actual_* unit_test_output
```

This command will **tar** all the specified files (in this case, **wordsearch.h, wordsearch.c, wordsearch_test.c, Makefile, unit_test_output,** and all of your expected output files of the pattern **actual_***) in the current directory to a file named **unityid_04_homework.tar**. Use your unity id instead of the literal string "unityid". As the **tar** is created, all the files added to the **tar** will be listed so you can confirm that the appropriate files are submitted.

Follow the CSC 230 Style Guidelines. Make sure your program compiles on the common platform cleanly (no errors or warnings), with the required compile options.

There will be a **5 point deduction** for submissions that do not include **wordsearch.h, Makefile, unit_test_output** and ALL actual output files in the tar in addition to **wordsearch.c** and **wordsearch_test.c**.

There will be a **5 point deduction** for submissions that are not tarred or where the tar file is named incorrectly.

There will be a **5 point deduction** for files (**wordsearch.h, wordsearch.c, wordsearch_test.c, Makefile, unit_test_output,** and all of the actual outputs generated through testing **(actual_*)**) that are named incorrectly.

There is a 24 hour window for late submissions. After the main locker closes, submit all late work to **"Homework 4 Late Submit."** There is an automatic **20 point deduction** for late work, even if you submit part of the assignment on time.

Rubric

+10 for compiling on the common platform with **`gcc –Wall –std=c99`** options, with no warnings

+40 for passing teaching staff test cases, which demonstrates a correct implementation.

+5 for compiling on the common platform with **`gcc -DTESTING -Wall -std=c99`** options, with no warnings

+20 for good, passing test cases

+10 for appropriate preprocessor directives that will conditionally compile the main function for unit testing or normal function

+20 for comments, style, and constants

-5 for meaningless or missing comments

-5 for inconsistent formatting

-5 for magic numbers

-5 for no name in comments

Total: 105 points

Global deductions FROM the score you earn above:

-10 points for any compilation warnings. Your program must compile cleanly! (if it doesn't compile, you will not receive any credit for test cases or compilation)

-5 points for not including the **`wordsearch.h`**, **`Makefile`**, **`unit_test_output`**, and ALL actual output files in the tar in addition to **`wordsearch.c`** and **`wordsearch_test.c`**.

-**5 points** for submissions that are not tarred or where the tar file is named incorrectly

-5 points for files (**`wordsearch.h`**, **`wordsearch.c`**, **`wordsearch_test.c`**, **`Makefile`**, **`unit_test_output`**, and all of the actual outputs generated through testing **`(actual_*)`**)that are named incorrectly.

-20 points for late work, even if you submit part of the assignment on time.

You can not receive less than a 0 for this assignment unless an academic integrity violation occurs.