CMPUT 201 - Winter 2023
Assignment 2
Henry Tang -- University of Alberta

Concepts Covered

- Arrays – ch. 8
- Ways to input / output data – ch. 22
- Expressions – ch. 4
- Functions – ch. 9
- Program organization and variable scope - ch. 10, 15
- Selection statements – ch. 5
- Loops – ch. 6
- Strings (including `<string.h>`) – ch. 13
- Pointers – ch. 11, 12

GitHub Instructions

- By now, you should have already associated your GitHub account with your CCID when you used the invitation link for the Labs repo
- The GitHub Classroom assignment invitation link is: https://classroom.github.com/a/tkukBpAw
- Please update the `LICENSE.md` file with your name where it says `student name`
- Make sure to continuously commit and push to your repository as you are developing your assignment and not simply when you are done. This serves two purposes: (1) you always have a backup of your code and can navigate through history to see your changes or revert back to an old version of the code that worked better and (2) showing your progress through the assignment and having the ability to explain the changes you were doing is of great help to you if you are suspected of plagiarism. To avoid overloading GitHub actions, you can commit locally as much as you want but push only once every hour.

Repository layout

The following files have already been made for you, though many are blank:

```
.
├── License.md
├── Makefile
├── README.md
├── assignment2_description.md
├── .gitignore
├── .github
│   └── .workflows
│       └── continuous-integration.yaml  # Don't modify this file
├── src
│   ├── puzzle2D.c
│   ├── puzzle2D.h
│   └── wordSearch2D.c
```

```
└── tests
    ├── github_actions
    │   └── github_actions.sh  # This is run by Github Actions. Try reading it
    ├── local
    │   ├── expected_solution.txt
    │   ├── puzzle.txt
    │   ├── sample_test.sh
    │   └── wordlist.txt
    └── sample_tests
        ├── student_1_sols.txt      # Don't modify this file
        ├── student_1_table.txt     # Don't modify this file
        ├── student_1_wordlist.txt  # Don't modify this file
        ├── student_2_sols.txt      # Don't modify this file
        ├── student_2_table.txt     # Don't modify this file
        └── student_2_wordlist.txt  # Don't modify this file
```

This is the bare minimum repository structure you'll be submitting. You can add more files if you'd like, particularly in `src/`. Do not submit your compiled executable `./wordSearch2D`. Do not modify `tests/github_actions/` or `tests/sample_tests`

Github Actions will automatically run `tests/github_actions/github_actions.sh` every time you push. Go into the "Actions" tab on Github to see the status for each commit you make. You can also run it yourself like this:

```
bash tests/github_actions/github_actions.sh
```

Problem Description

Bob and Alice are competitive word-search puzzle solvers. However, even to the best of Bob's abilities, he was never able to find words in the puzzles faster than Alice. As Bob's friend and amazing programmer, you have decided to help him out by creating a solver for word-search puzzles.

A word-search puzzle is solved when, given a grid of random letters and a set of words, the program finds the location of the input words in the grid (if they exist). The orientation of the words can be horizontal, vertical, left diagonal, or right diagonal. The words can be read left to right, right to left, top to bottom (whether vertically or diagonally), and bottom to up (whether vertically or diagonally). Note that a diagonal can start at any index, not necessarily on a central diagonal.

You have decided to approach the problem using the Rabin-Karp algorithm. The Rabin-Karp algorithm uses what is called a *hash function* to find a substring of size x in a string of size $y >= x$.

Algorithm Description

You may wonder, why use hash functions at all? Why don't we just use brute-force search? We certainly could do that, but it would be highly inefficient. If you don't know anything algorithm complexity, and symbols like O(n), O(logn) or O(1) sound foreign to you, review this article: Complexity of Algorithms. So, as you can imagine, using a special algorithm for string matching makes the process much, much faster.

Let's assume that we're searching for a word "dog" in a string "gadogado". We need to compare "dog" with any substring of "gadogado" that has the same length ("gad", "ado", "dog", "oga", "gad", "ado"). So, in a sense, you are "rolling" the word "dog" over the string "gadogado". To compare the word with the substrings, instead of comparing them character by character (which would be very time consuming!), wouldn't it be nice if we had some fixed value that represents each substring? Then, we can just quickly compare these values! Here comes the idea of a hash function.

What is a hash function? It's a function that can be used to map data of arbitrary size to fixed-size values. For example, a hash function can map strings to integers. Probably, one of the simplest hash functions would be the sum of ASCII codes of the characters of the string:

```
s = "gadogado"

H(s[0..3]) = H("gad") = 103 +  97 + 100 = 300
H(s[1..4]) = H("ado") =  97 + 100 + 111 = 311
H(s[2..5]) = H("dog") = 100 + 111 + 103 = 314
```

However, it's not the best hash function because it allows many **collisions** when different strings have equal hash values. For example:

```
H("dog") = 100 + 111 + 103 = 314
H("axa") =  97 + 120 +  97 = 314
H("esb") = 101 + 115 +  98 = 314
```

So we need to have a better hash function, that can take into account the order and position of letters. Here is how it works in the Rabin-Karp algorithm.

Study this resource to understand the algorithm.

Let's see an example of how to find the word **dog** in the string **gadogado** using the Rabin-Karp algorithm.

1. We need to know the size of our alphabet -- **D**. We use ASCII codes, so **D = 256**.
2. Also, we need to choose a large enough prime number. The higher it is, the less collisions we will see. As a rule of thumb, it should be larger than the alphabet size. For this example, we'll use **P = 577** (in your assignment, you'll use a different value, see below).
3. Let's calculate the value of **h**. It's a rehashing coefficient, and we'll need it later, on Step 7. `h = D^(len-1) % P` (len is the length of the word we're searching ("dog"), so **len=3**) In this example, **h = 335**.
4. Now, let's calculate the hash of the word "dog", letter by letter, using this step-by-step formula:

   ```
   hash = (D * <previous_hash> + <current letter>) % P;
   ```

   So:

```
# For step "d":
(256 * 0 + 'd') % 577 = 100
# For "do":
(256 * 100 + 'o') % 577 = 323
# For "dog":
(256 * 323 + 'g') % 577 = 280
```

So, the hash of the word "dog" is **280**. See how we use modulo (remainder) here. In fact, we could use the same formula without the last part (% 577). But using modulo helps us avoid extremely large numbers that won't fit into C integer types.

5. Let's now loop through the string "gadogado", calculating "rolling" hashes of all three-letter substrings of it: "gad", "ado", ...

6. First, let's calculate the hash of the substring "gad". Applying the same formula as above, we get the result: hash of "gad" is **6**.

7. Now, we start "rolling" through the string "gadogado", or "rehashing" -- basically, moving to the right. Moving from "gad" to "ado" means that we need to "subtract" the letter "g" and "add" the letter "o". In the modular arithmetic of the Rabin-Karp algorithm, it's done the following way:

   ```
   hash = (D * (<previous_hash> - <letter to subtract> * h) + <letter to add>)
   % 577
   ```

   Thus, hash of "ado" is **506** (if the result if negative, just subtract it from P). Rolling one more step to the right, we can get from "ado" to "dog". Applying the same formula, we get the hash of "dog": **280**!

8. We see that we have found the hash we were looking for. So, with some probability, we have found the **word** we're looking for. But beware of collisions! Even with large enough P, collisions can happen. So, the final step.

9. To be certain that we found the right substring, we need to compare the word with the found substring character by character.

IMPORTANT: For the purposes of this assignment, you must fix **P** (the prime number) to **7079**, and **D** (alphabet size) to **256**.

Task

Write a program that uses the Rabin-Karp algorithm to find the location of individual words in a given word-search grid and outputs their starting position and direction, if they exist.

Enumerate the directions as follows:

1. horizontal right (→)
2. horizontal left (←)
3. vertical down (↓)
4. vertical up (↑)

5. top left to bottom right diagonal (↘)
6. bottom right backwards to top left diagonal (↖)
7. bottom left to top right (↗)
8. top right backwards to bottom left (↙)

How to run your program

Your program runs as follows:

```
./wordSearch2D -p <puzzle_file> -l <word_length> -w <wordlist_file> [-o
<solution_file>]
```

where:

- `puzzle_file` is a required argument. It is preceded by the `-p` flag. It is a text input file that contains the puzzle grid. The provided puzzle file will have *n* lines of single strings of length *n*, providing an *n* by *n* square grid of letters. Each line in the file represents a row, and each line will be of same size *n*. The positions go from *0* to *n - 1* on each axis, with indices read from left to right and top to bottom. Note that 0 <= *n* <= 100. The puzzle file can contain any ASCII character with values [32, 126], not necessarily English letters only.
- `word_length` is a required argument. It is preceded by the `-l` flag. It is a number in the range [1, n] that represents the fixed length of the words the program will search for.
- `wordlist_file` is a required argument. It is preceded by the `-w` flag. It is a text input file that contains the list of words to look for. Each word will be provided on a separate line and will have exactly `word_length` characters; otherwise, the wordlist file is invalid. The wordlist file can contain any ASCII character with values [32, 126], not necessarily English letters only.
- `solution_file` is an **optional argument**. It is the name of the text output file to which your program will write its solution. Note that the `solution_file` may not necessarily exist before your program runs. **If this argument is not provided, the output should be written to a file called `output.txt`.** For each word in the input `wordlist_file`, a corresponding line will be outputted in the `solution_file` with the following format `word;(y,x);d`, where *word* is the input word, *(y,x)* are the coordinates of the first letter of the word in the puzzle (*y* is the row index and *x* is the column index) and *d* is the direction of the word according to the enumerated list above.

If the input word is not found in the puzzle, the output would be `word;(0,0);0`. Note that the top left corner of the puzzle corresponds to coordinate `(0,0)`, but since the direction is 0, this indicates that the word is not found. **Note that you must output the solution of each input word in the same order they were provided in the `wordlist_file`.**

You must account for any edge cases for command-line arguments. We may run with too many arguments, not enough arguments, invalid arguments, etc. **Also, note that, similar to assignment 1, the order of the arguments is not fixed.** For example, `./wordSearch2D -w wordList.txt -p puzzleFile.txt -l 20` is valid.

Error Checking

Your program must report and **print an error to `stderr`** and quit in the following cases. Note the return code that needs to be returned in each case. The return code is the integer that your program returns upon exit (e.g., `return 0;` in `main` or `exit(2)` from anywhere in the program). The error message should be the exact error message provided:

- If the input `puzzle_file` is not found (angle brackets are placeholders here, replace with appropriate file name)
    - Error message (in stderr):

```
Error: Puzzle file <name-of-input-file> does not exist
```

    - Return Code 4
- If the input `wordlist_file` is not found (angle brackets are placeholders here, replace with appropriate file name):
    - Error message (in stderr):

```
Error: Wordlist file <name-of-input-file> does not exist
```

    - Return Code 5
- For any other incorrect program usage (e.g., forgetting a mandatory argument), your program should report (angle brackets need to appear as is here since they indicate the placeholders needed for correct usage):
    - Error message (in stderr):

```
Usage: ./wordSearch2D -p <puzzle_file> -l <word_length> -w <wordlist_file>
[-o <solution_file>]
```

    - Return Code: 6
- Unless explicitly stated in the assignment description or in the assumptions below, there are no simplifying assumptions in this assignment. You should handle all corner cases you can think of. If there is a case of incorrect input that prevents the program from proceeding correctly, your program should report:
    - Error message (in stderr): `Encountered error`
    - Return code: 7

Assumptions

- Only characters with ASCII values between 32 and 126 will appear in the puzzle and word files
- `a` is not the same as `A`. This is already the case, given their ASCII codes
- The input word file can have any number of words, but each line will contain a single word. Each word will have a maximum size of 100 characters
- A word exists at most once in the puzzle

Example with Expected Output

Here is a full example.

Given the following 5*5 puzzle file, `puzzle.txt`:

```
batlw
hello
gblfk
cikkl
jdgod
```

and the following word list, `wordlist.txt`:

```
wok
dog
bat
elk
xxx
```

then running the program with

```
./wordSearch2D -l 3 -w wordlist.txt -p puzzle.txt -o mysolution.com.png
```

would write the following to a file called `mysolution.com.png`:

```
wok;(0,4);3
dog;(4,4);2
bat;(0,0);1
elk;(1,1);5
xxx;(0,0);0
```

Note that the directions correspond to the enumerated list provided above.

Program Organization

Your program must have the following files

- Only your main function and any helper methods for reading arguments should be in a file called `src/wordSearch2D.c`
- All functionality related to solving the word puzzle should be in file `src/puzzle2D.c` and its corresponding header file `src/puzzle2D.h`. You must divide the logic of solving the word puzzle into multiple functions.
- Feel free to add additional files and functions to further divide the functionality in your program.

- **You must not use any global variables!** You can define macros for predefined constants such as the maximum puzzle size or the prime number, but you cannot use any global variables.

All `.c` and `.h` files must be in `./src`. Your `Makefile` must be at the project root and the binary it produces (`wordSearch2D`) is also at the root.

Refer to the repository-layout to know how your repository should look (though you can always add MORE files). Use the following command on our lab machines to see what your repository structure:

```
tree -a -I .git -I wordSearch2D
```

Submission Guidelines

- You must use `gcc -Wall -Wextra -Werror -std=c99` to compile your program. This command needs to be in your `Makefile`. **Notice**: This is command has more flags than we use for labs! These additional flags are more rigorous at detecting problems with your program
- You must create a Makefile to compile and link the files in your program. Your Makefile must also contain the `clean` target. Your program must compile into an executable called `wordSearch2D` . If your TA cannot run `./wordSearch2D` after running `make`, your assignment is considered incomplete as we cannot test your program. You will get a 0 for the assignment.
- Create a `README.md` file that includes at least the following sections. We encourage you to use markdown syntax to make a proper README. Consider looking at the one you're reading right now as an example!
- Information about yourself
- The purpose of your program
- The exact commands we need to run your code
- The files submitted in your assignment, how they are structured, and what they do
- references (e.g., any online resources you used in accordance to the course policy)
- **Copying any code off the Internet or other teams is not allowed.** Please take this seriously as plagiarism cases will be reported to the Faculty of Science.
- Write one test case in `tests/local/` with the following files:
    - `puzzle.txt` contains a given puzzle
    - `wordlist.txt` contains a word list to find in the given puzzle
    - `expected_solution.txt` contains the solution to the given word list and puzzle.
    - `sample_test.sh` is an executable shell script that runs your test case using the appropriate parameters. **DO NOT SPECIFY AN OUTPUT FILE**. We will run your test case as `./tests/local/sample_test.sh` and compare the produced `output.txt` file to your provided `expected_solution.txt`.
- Your `Makefile` and all your code files must contain commented text at the top of your file that specifies your information, as indicated in the general assignment instructions on eClass. You must also have the License information as a comment on the top of all your code files (i.e., `.h` and `.c` files).
- Summary of files that need to be on Github for submission:
    - `src/wordSearch2D.c`
    - `src/puzzle2D.c`
    - `src/puzzle2D.h`
    - `Makefile`

    - `README.md`
    - `License.md`
    - Your test case files:
        - `tests/local/puzzle.txt`
        - `tests/local/wordlist.txt`
        - `tests/local/expected_solution.txt`
        - `tests/local/sample_test.sh`
    - Any additional `.h` or `.c` files that are necessary to compile your code must be in `./src`
    - Any additional test cases should be in `./tests/local`

To check you have all the required files, and everything is spelled correctly, run `./tests/github_actions/github_actions.sh`

Important Notes

- You must be able to handle any puzzle size up to 100 * 100.
- You must implement and use the Rabin-Karp algorithm as described above. Your program must run sufficiently quickly, even when tested with large puzzles. If your program takes more than 0.3 seconds to solve a given puzzle on the lab machines, you will fail that test case. To test how long your program takes to run, use the `time` command (e.g., `time ./wordSearch2D ...`) and look at the Real value displayed.
- Your program must compile without any warnings or errors (using `gcc -Wall -Wextra -Werror -std=c99`). If compiling your program results in errors or warnings, you will receive a 0 for the assignment.
- Any missing files will result in 0 in the assignment. If you forget the README, `Makefile`, one of the source files etc., you will get a 0. Please make use of the scripts we provide you to check the format of your submission.
- Please follow the instructions carefully. Any extra spaces you put after any word will result in a failed diff between the expected output and your solution. Your GitHub repo already has sample test cases and a script to show you how we will compare the program output to the expected output. **PLEASE USE THIS SCRIPT!** It will tell you if your output differs from what we expect. The continuous integration status will also let you know if there is anything wrong. Please do not push every commit you make right away because this overloads the Travis CI build server. We advise you to commit locally as you work, then test locally using the `./tests/local`. Once you verify that things work, then push your commits. In general, try to push at least once a day but not every 10 minutes.

Grading

Total: 50 marks

- README file contains necessary information according to the general assignment instructions posted on eClass **(1 mark)**
- running `make` compiles files into an executable called `wordSearch2D` without warnings or errors **(1 mark)** -- 0 for whole assignment if this step does not work
- `make clean` should delete all object and executable files that are generated in the compilation process **(1 mark)**
- Proper structuring of your code into `puzzle2D.h`, `puzzle2D.c`, and `wordSearch2D.c` **(3 marks)**
- Solution is implemented using the Rabin-Karp algorithm described **(7 marks)**

- Pass your own test case. Obviously, if your test case is missing or has the incorrect format, you will not get the mark **(2 marks)**
- Argument checking. Ensure you cover all cases for command line parameters **(10 marks)**
- Passing of word-search test cases **(23 marks)**
- Proper code formatting and commenting where necessary **(2 marks)**
- **10 marks will be deducted from your grade if global variables are used**