05OGDLP - Algorithms & Programming
Laboratory 4

Learning objectives

- Files
- Pointers
- Dynamic memory allocation
- Structured data types

1 Characters frequency

A file contains text with an unknown number of lines, each line having a maximum length equal to 100 characters.
Write a C program able to find and count up the number of:

- horizontal sequences of 5 adjacent `'h'` characters on the same row
- vertical sequences of 5 adjacent `'v'` characters on the same column.

Notice that, as the length of the file is unknown, it cannot be stored entirely at the same time in the main memory (i.e., it can only be stored in the main memory a piece at a time). Moreover, it is not possible to read the file more than once.
The program:

- receives on the command line the name of the file
- prints out (on `stdout`) the number of times the `hhhhh` sequence appears on a row, and the number of times the sequence `vvvvv` appears on a column.

Example

Let the input file be the following one:

```
Regression file:
Line with two hhhhhorizontal sequences hhhhh!
From hereinafter  vv
v all rows        vv
v include         vv
v two vertical    vv
```

```
v sequences       v
v and one horizontal ---> hhhhh!
```

The output shall be:

```
Horizontal sequences: 3
Vertical sequences: 2
```

Implementation notes

- Read the file on a line-by-line basis using `fgets`:

  ```
  while(fgets( ... ) != NULL) { ... }
  ```

- Looking for horizontal sequences of "h"s is straightforward, as it can be done by storing a single line of the file in the main memory. To check for vertical sequences of "v"s instead, there are two main possibilities:
  - Work on single lines, keeping an array of (100) counters
  - Work on a matrix that stores 5 rows of the file at the same time. Once the check has been performed, scroll ("shift") one or more rows up and read the new ones from the file.

2 Word counter, reloaded

Write a program that receives two file names as command line arguments. The first file stores a text with an undefined number of rows. Each row is shorter than 100 characters. The second file stores a list of words. Each word is shorter than 20 characters, and the number of words is at most equal to 100.
Your program should count the number of times each word from the second file appears in the text from the first file, and print on `stdout` each word with its absolute frequency.
Both files have to be read just once. Capital and small letters have to be considered equivalent, e.g., `word`, `WORD`, and `WoRd` are considered equivalent.

Example

Let the first file be the following one:

```
Watch your thoughts ; they become words .
Watch your words ; they become actions .
Watch your actions ; they become habits .
Watch your habits ; they become character .
Watch your character ; it becomes your destiny .
Lao-Tze
```

and the second file be the following one:

```
watch
words
become
```

The program has to print out the following:

```
watch - 5 occurrence(s)
words - 2 occurrence(s)
become - 4 occurrence(s)
```

Taking it further

Extend the previous program with dynamic allocation, i.e., dynamically allocate an array of structures to store the list of words read from file and their absolute frequency. This time, the number of words stored in the second file is specified on the first row of the file itself, for example:

```
3
watch
words
become
```

Write two versions of this program:

Version A

```
typedef struct {
    char word [MAX_WORD_LENGTH + 1];
    int occurrences;
} index_t;
```

Version B

```
typedef struct {
    char *word;
    int occurrences;
} index_t;
```

3 Conway's Game of Life (part 1)

This guided exercise is the first part of a larger programming project through which you will learn how you can build complex programs out of simpler pieces, and how to plan, structure, and organize code.

The set of milestones to be reached in this part are

1. Building a data structure to store the board state
2. Initializing the board to a random state
3. Printing the board on standard output

The game and its rules

The Game of Life is a cellular automaton developed by the British mathematician John Conway in the 1960s. It shows how complex structures and behavior can emerge from the interactions between particles following very simple rules. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

Life's universe is a two-dimensional grid where each square represents a cell that starts the game as either dead or alive. During each turn, each cell looks at its eight neighbors (all its adjacent cells) and counts up how many of them are alive. According to this number, the cell updates its own liveness:

- Any live cell with fewer than two live neighbors dies because of underpopulation
- Any live cell with two or three live neighbors lives on to the next generation
- Any live cell with more than three live neighbors dies because of overpopulation
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed, alive or dead. The rules continue to be applied repeatedly to create further generations.

Many different types of patterns occur in the Game of Life, which are classified according to their behavior. Common pattern types include still lifes, which do not change from one generation to the next; oscillators, which return to their initial state after a finite number of generations; and spaceships, which translate themselves across the grid. You can see some examples on this page.

It is possible for objects to interact with each other in interesting ways. For example, if two gliders are shot at a block in a specific position, the block will move closer to the source of the gliders. This sliding block memory can be used to simulate a counter. It is possible to construct logic gates such as AND, OR, and NOT using gliders. It is possible to build a pattern that acts like a finite-state machine connected to two counters. This has the same computational power as a universal Turing machine, so the Game of Life is theoretically as powerful as any computer with unlimited memory and no time constraints. That is, it is Turing complete. (source: en.wikipedia.org)

Structure of the project

We are going to organize the code into a library module – `life` – containing all the data structures and functions for the game, and a client module – `main.c` – using the library to run the game. The `life` library is split into

- A `life.h` header. This file contains the data structures and interface **declarations** (type signatures) of the functions available to the client
- A `life.c` source file, containing the **definitions** of all the data structures and functions declared in `life.h`. In addition, `life.c` might also contain helper functions that are useful for organizing your code but should not be directly exposed to the client.

Let *GameOfLife* be the name of a new CLion project. It should already have a `main.c` file. Right-click on the project folder in the left pane, add a new `C` source file with an associated header and name it *life*.

1. Building a data structure to store the board state

We can break down this milestone into two steps

- Definition of a data structure
- Definition of the functions for dynamic memory allocation and release

Definition of a data structure

The Life world is a 2-D grid of a given height and width, where alive and dead cells can be represented by 1 and 0 values, respectively.

We want to provide a flexible structure so that the client can choose height and width for a given Life instance. Therefore, we are going to model the inner 2D array with pointers instead of fixed-size arrays. The data structure we are going to design should also include the height and width of the 2D array, for easier and safe manipulation of the array itself.

Let `grid_t` be the name of the new datatype. Let `my_grid` be a variable of type `grid_t *`, that is, we want the client to be able to declare a new instance of a life grid as follows

```
grid_t * my_grid;
```

To do this, in the `life.c` file we need to define a `struct` encapsulating the `int **` array and the two `int` variables storing height and width. Then, `typedef` it to `grid_s`.
In the life.h use `typedef` again to declare `grid_t` to be an alias for the type `struct grid_s`.
This way, we are (partially) hiding the details of the data structure implementation and are free to change it without affecting its usage.

Definition of the functions for dynamic memory allocation and release

We want our `life` library to use only dynamic memory allocation. Therefore, `my_grid` variable declared above must be allocated first, and then freed before the program ends.
To do this, our library should contain two useful functions having the following signatures:

```
grid_t * grid_new(int, int);
void grid_free(grid_t *);
```

`grid_new` expects two integer parameters (height and width of the grid) and returns a pointer to a `grid_t` variable (i.e. a chunk of memory large enough to store a `grid_t`). Internally, it uses `malloc` to properly allocate memory for the `grid_t` variable itself (e.g., `my_grid` declared above) and any other variable inside of it requiring dynamic memory allocation.

`grid_free` expects a `grid_t *` parameter (e.g., `my_grid` variable) and returns nothing. Internally, it uses `free` to deallocate the memory previously allocated by a call to `grid_new`. Be careful to

1. avoid double free (i.e., check if the pointer you are freeing is already `NULL`)
2. release the memory in the opposite order it was allocated by `grid_new`

Put the two signatures above in the `.h` file of our library, and implement the corresponding functions in the `.c` file.

2. Initializing the board to a random state

The Game of Life can be initialized with a *soup*, i.e. a random initial pattern, or from a known configuration of dead and alive cells that can be read from file.

In this step, we are going to initialize the board with a soup. Let `void grid_random_state(grid_t *);` be the signature of the function that expects a `grid_t *` variable as parameter and returns nothing.
Implement it so that each cell in the 2D grid is initialized to 0 or 1. To this end, you can use the `rand` function and the remainder operator.

3. Printing the board on standard output

Let `void grid_show(grid_t *);` be the function prototype. Implement it to print the board, using a space `char` for dead cells, and a character of your choice (e.g. `#`) to represent alive cells.

Write an additional helper function to print the header and footer borders like in Figure 1.

Testing the library

Use the following client code to test your library, and check it for memory leaks and dangling pointers.

```
#include "life.h"

int main() {
    grid_t * grid = grid_new(18, 38);
    grid_show(grid);
    grid_free(grid);
    return 0;
}
```

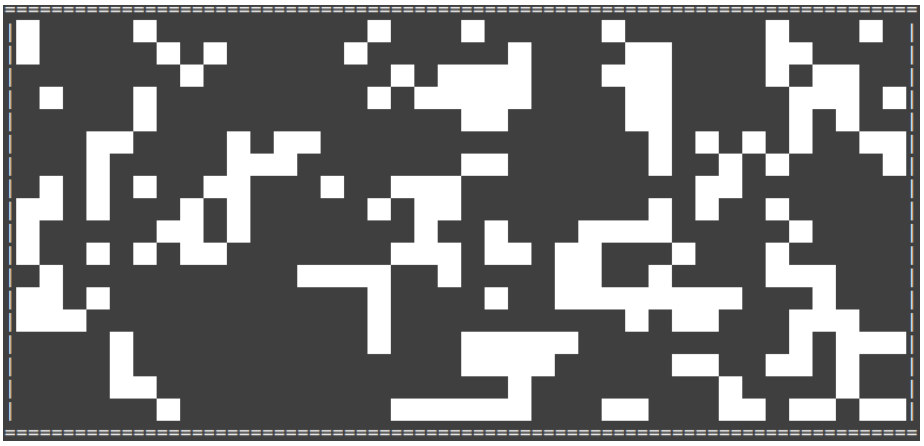

Figure 1: An instance of Life on a 18 by 38 board initialized with a *soup*, and using the Unicode character `U+2588` to represent alive cells