05OGDLP - Algorithms & Programming
Laboratory 5

Learning objectives

- Dynamic memory allocation
- Structured data types
- Sorting

1 Cycling performance analysis

During a training session, each athlete of a group of professional cyclists is checked during each lap. For each athlete, all lap times are stored in a file with the following format. The first line of the file stores the number of cyclists in the group. Then, for each cyclist, the file stores:

- On the first line, their name (a string of 30 characters at most), identifier (integer value), and number of laps performed.
- On the second line, all lap times, time_1 time_2 ... time_N, stored as real values.

Write a program that, after reading the file and storing its content in a proper data structure, is able to reply to the following menu inquiry:

- `list`: the program prints out the number of athletes, their names, identifiers, and number of laps performed. The list must be in ascending order by identifier.
- `detail <name>`: given an athlete's name, the program prints out their identifier and all lap times.
- `best`: the program prints out the name, identifier, all lap times, and the average lap time for the athlete whose average lap time is smaller.
- `stop`: end the program.

Notice that all operations can be performed more than once till the stop command is issued.

Example

Let the following be the input file:

```
4
Rossi 100 3
1.30 1.38 1.29
Bianchi 101 5
1.46 1.43 1.42 1.51 1.28
Neri 117 2
1.26 1.34
Verdi 89 4
2.01 1.45 1.43 1.38
```

The following is a run example of the program:

```
Input file name: cyclist.txt
Command? list
Number of athletes : 4
Name: Rossi #Id:100 #Laps:3
Name: Bianchi #Id:101 #Laps:5
Name: Neri #Id:117 #Laps:2
Name: Verdi #Id:89 #Laps:4
Command? best
Name:Neri #Id number:117 Laps:2 Times: 1.26 1.34 (Average:1.30)
Command? details Bianchi
#Id:101 #Laps:5 Times: 1.46 1.43 1.42 1.51 1.28
Command? stop
Program ended.
```

2 Rectangle sorting

A file defines a set of rectangles with the following format:

- Each row of the file contains 1 string and 2 real numbers:
  - The string (4 characters long) is the rectangle identifier,
  - The two numbers specify the x and y coordinates, respectively, of one of its vertices
- For each rectangle, there are two lines in the file, specifying the coordinates of two opposite vertices (top-right and bottom-left or top-left and bottom-right).

Notice that, in general, the two rows defining a rectangle are not consecutive, and that it is not known which vertex they specify. In any case, suppose the maximum number of rectangles is 100.
Write a C program that receives 3 file names on the command line:

- The first file is an input file, and it contains all rectangle specifications as previously indicated

- The second file is an output file, and it must contain the name of the rectangles ordered by ascending area values.
- The third file is an output file, and it must contain the name of the rectangles ordered by ascending perimeter values.

Examples

Let us suppose that the program receives the following three parameters:

```
rectangles.in area.out perimeter.out
```

Moreover, let us suppose that the content of `rectangles.in` is the following:

```
rct2    1.5 3.5
xxyy    -0.5 3.0
xxyy    1.5 2.0
abcd    1.0 4.5
ktkr    -2.5 1.5
abcd    2.0 2.0
rct2    3.5 -2.0
trya    2.5 -1.0
ktkr    1.5 3.5
trya    4.0 4.0
```

As areas and perimeters of the rectangles are:

```
rct2    area=11.00  perimeter=15.00
xxyy    area= 2.00  perimeter= 6.00
abcd    area= 2.50  perimeter= 7.00
ktkr    area= 8.00  perimeter=12.00
trya    area= 7.50  perimeter=13.00
```

The program has to generate the following two files:
`area.out`:

```
rct2
ktkr
trya
abcd
xxyy
```

`perimeter.out`:

```
rct2
trya
ktkr
abcd
xxyy
```

Implementation notes

Use an array of structures where each element of the array stores the name and the two extreme coordinates of a rectangle.

3 Taking it further: Rectangle sorting with dynamic memory

Extend the previous program with dynamic memory allocation, i.e. dynamically allocate an array of structures to store information on all rectangles.
This time, the number of rectangles stored in the input file is specified on the first row of `rectangles.in` itself, for example:

```
5
rct2    1.5 3.5
xxyy    -0.5 3.0
xxyy    1.5 2.0
abcd    1.0 4.5
ktkr    -2.5 1.5
abcd    2.0 2.0
rct2    3.5 -2.0
trya    2.5 -1.0
ktkr    1.5 3.5
trya    4.0 4.0
```

4 Conway's Game of Life (part 2)

In Exercise 2 of Laboratory 3, you started working on The Game of Life by writing the functions for building the board and initializing it to a random state. In this second part, we are going to extend the program so that, given a starting state, it can compute the next one. The set of milestones you will have reached by the end of 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
4. Calculating the next state of the board

4. Calculating the next state of the board

Each cell in Life's 2D grid examines its 8 immediate neighbors (the red squares in Fig. 1a), then updates its liveness according to the following rules:

- 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 (reproduction)

Write a function

```
grid_t * grid_next_state(grid_t * grid)
```

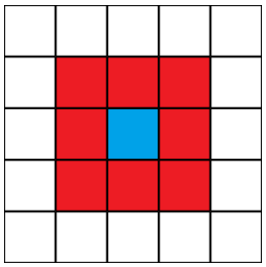
that, given the grid, computes the next state and returns the updated grid.

Implementation notes

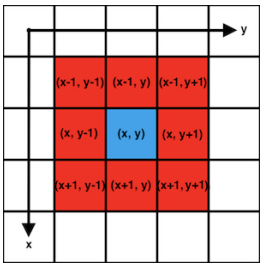
- **How to store the newly computed state?** You cannot write the new liveness value of a cell directly on top of the grid, as this would modify the neighborhood of the next cells. Hint: why does the `grid_next_state` function receives a `grid_t *` parameter **and** returns a `grid_t *` value?
- **How do you iterate over the neighborhood?** The neighbors of a cell at coordinates `(x, y)` are located 1 space either side of it (Fig. 1b). Write a helper function (i.e. implement it in the `life.c` file but do not add it to the `life.h` header)

  ```
  int grid_alive_neighbors(grid_t * grid, int i, int j)
  ```

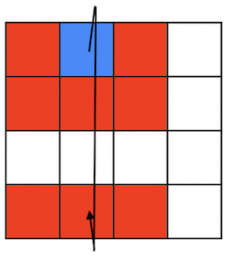
  that, given the coordinates `(i, j)` of a cell in the grid, returns the number of alive neighbors. The function iterates over the neighborhood in a nested loop and adds up the number of alive neighbors. The caller function (`grid_next_state`) uses this helper function on each cell in the board, then decides its fate according to the rules.



(a) A cell's (blue) neighborhood (red)



(b) Iterating over a cell's neighborhood



(c) Wrapping the board round in a circle

Figure 1: Game of Life cell's neighborhood

- **What about the cells on the edge?** While the cells in the center of the board have 8 neighbors, those on the edge have 5, and those in the corners have just 3. In your `grid_alive_neighbors` helper function, wrap the board round in a circle (Fig. 1c) so that cells on the top and bottom edges count as each others' neighbors, as do those on the left and right edges. This means that every cell has 8 neighbors.