TCS1-IN: Conway's Game of Life

Author : Christophe Garion
Audience : SUPAERO 1A
Date : February 1, 2021

1 Objectives

The objectives of this project are the following:

- write a naive implementation of Conway's Game of Life using C "matrices". This implementation will allow to load the initial configuration from a text file and produces a list of images corresponding to each step of the simulation.
- write a more efficient implementation using a linked list
- read an initial state of the grid in Conway's Game of Life from a text file

Each objective is presented in the following.

2 Presentation

Conway's Game of Life [5] is a cellular automaton that can be easily simulated. The principle is the following: you manage a 2D grid on which cells live and die. The evolution of each cell depends on its *neighbors*, i.e. the eight cells that are horizontally, vertically or diagonally adjacent to the cell. There is no user interaction besides creating the initial state of the grid, and you just watch the grid evolves. Some patterns are rather amusing, like the glider or guns.
The rules of the game are simple:

- if a cell is alive and has 2 or 3 alive neighbors, it will stay alive at the next iteration, otherwise it dies and the cell will be empty at the next iteration.
- if a cell is empty and has exactly 3 alive neighbors, it will resurrect at the next iteration and become alive, otherwise it remains empty.

Notice that the grid represents a *generation* and that all cells evolve simultaneously.
Although Conway's Game of Life is (rather) funny, it has been proved that this simple cellular automaton has the power of a Turing machine, therefore it could be used to compute whatever a Turing machine can compute. You can refer to [3] for more details.

3 A naive implementation

A first implementation of Conway's Game of Life may use a `char` matrix representing the universe of the game. A convention may be established, for instance `'o'` may represent an alive cell. Making the universe evolve is just updating the matrix according to the rules of Conway's Game of Life.
Unfortunately, such naive implementation suffers major drawbacks:

- simulating universes that expands is not easy as you have to recreate a new matrix if the current one is under-dimensioned for the next simulation step.
- a "big" grid is allocated whereas there are only few alive cells in it. Lot of memory is wasted and you cannot simulate huge grids (because of stack size limit for instance).

4 A less space-consuming storage of cells

To be able to manage universes that expand indefinitely easily and to avoid memory space waste, we may use a *linked list* of cells containing only alive cells. During the computation of the next state of the universe, the linked list will be in an intermediate state where cells are alive (they will remain alive after the computation of the next state) or dying (they are alive in the current state but will be empty in the next state). A list of newborn cells must also be built and contains the cells that will be alive in the next state. There is no need to explore the entire universe as only neighbors of alive cells may become newborns. The list of alive cells for the next state must be built using these two lists. Algorithm 4.1 presents a high-level algorithm of the computation of the next state. Of course, this algorithm may be optimized, for instance you may only create newborn cells if they are not already alive or dying in the list, but beware of complexity of searching cells in the list, as it is linear!

Algorithm 4.1: An algorithm to make the list of cells evolve according to the rules of Game of Life

Data: a list of cells L, an empty list of newborn cells LN

```
foreach cell c in L do
    verify if c is dying and change its status accordingly ;
end
foreach cell c in L do
    foreach neighbor n of c do
        verify if n can become a newborn cell, if so create it and add it to list LN;
    end
end
remove duplicates in LN;
append LN to L;
remove newborn cells that are also dying in L;
remove dying cells in L;
```

5 Implementation details

5.1 Structures for naive implementation

The universe will be represented by a structure containing:

- the height and the width of the universe
- the number of steps to execute
- a pointer to `char` representing the universe elements

The pointer to `char` will point on a *dynamically allocated* memory region that can be assimilated to an array of char. This region will contain the width × height char values representing the grid[1]. To access the grid element at row `i` and column j, you should access the element in the array at index `i * width + j` where `width` is the width of the grid. You should write access and modification functions for the universe in order to ease the readability of your code. You should also provide a `print_naive_universe` function to print the universe on the console.
We will use the following convention:

- character `'.'` represents an empty cell
- character `'o'` represents an alive cell

The corresponding structures and functions should be located in the `naive_universe.h` and `naive_universe.c` files.
You should provide a test in `test-naive-universe.c` that initializes a small universe « by hand » (for instance a 5×5 universe), check that your access functions works on some cells (do not forget to verify at least corner cases) and prints it on the console.

[1] The 2D grid will therefore be "flattened" in a 1D array

5.2 The naive universe loader

The universe loader reads a text file containing the universe description and returns a structure representing the universe (cf. section 5.1). You will find in the `data` subdirectory of your repository several file examples. For instance, `glider.txt` (a simple universe that can be used for testing purposes) is the following:

```
20 21
10
.o..................
..o.................
ooo.................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
....................
```

- the first line contains the width and the height of the map separated by a space
- the second line contains the number of simulation steps
- the other lines contains the universe with the following convention:
  - `'.'` represents an empty cell
  - `'o'` represents an alive cell

Look at appendix F to understand how to read data from a file. The `app-ex-loader.c` file contains a example program that reads a file given as input on the command line, gets the dimensions of the universe and the number of simulation steps and print them and print each line of the grid. You can use it to write your `naive_load` function. You must provide a test for your loader in the `test-naive-loader.c` file. This program should:

1. build the structure corresponding to the `data/glider.txt`
2. use `assert` to verify that the width, height and number of simulation steps are the ones that are expected
3. use `assert` and the `strcmp` function from `string.h` to compare the string you have obtained in your universe with the expected one (the latter is given in the `test-naive-loader.c` file). **Do not forget to add a `'0'` character at the end of your string representation of the universe in order to be able to compare the two strings.**

   Instead of using `strcmp`, you may also use the `memcmp` function from `stdlib.h`.

You must also create an application `app-naive-loader` that use `main` arguments (cf. section D) to get a file name, build the corresponding structure and print it.
We should be able to execute `app-naive-loader` as follows:

```
./app-naive-loader ./data/glider.txt
```

to load the `glider.txt` file from the `data` directory in the current directory.

5.3 Naive implementation of Conway's Game of Life

Starting from a naive universe, implementing Conway's Game of Life is rather simple: you just have to update the matrix represented by the string in the structure accordingly to the rules. Of course, you may need to create first a copy of the string in order to update it. Some remarks:

- you may choose to consider that the universe is a torus, but it is not mandatory
- the size of the universe is fixed: the universe cannot expand

You should therefore provide a `naive_conway.c` file in which there are

- a `naive_step` function that takes an universe and apply the game's rules *once*. You may choose to modify the universe taken as argument or to return a new one, but beware of memory deallocation and do not forget to add a final `'0'` character to strings you create if you want to use the `strcmp` function to compare them in tests.
- a `naive_simulation` that takes an universe and apply the game's rules for the number of steps defined in the universe. You may add a parameter to the function to print or not the universe on the console after each step.

To test your `naive_step` function, you will find on LMS an archive containing 512 files corresponding to all possible configurations of a 3×3 universe. An archive is provided on LMS with all files that you can put in your `data` directory.

Do not add the test files in your repository! Normally, SVN should ignore them.

These files are named `test-XXX.txt` files where `XXX` is a number. The corresponding state of the universe after one simulation step is located in the `test-XXX-sol.txt` file. You should therefore create a `test-naive-conway.c` test file using theses files to verify your implementation of `naive_step`. Of course, you should implement a function that takes a number as parameter and verify that your function is correct with the corresponding file. To build dynamically the name of files `test-XXX.txt` and `test-XXX-sol.txt`, use `sprintf` as presented in section E.
You should also provide a `app-naive-conway.c` file with a simple `main` that takes a data file as parameter, execute the defined number of simulation steps and prints the universe at each step of the simulation. Test it on `data/glider.txt` for instance.

5.4 Generating images

We want to be able to generate pictures of the universe at the different steps of the simulation. We will generate images in the PPM format [6], a simple graphical format that uses text files.
A file containing an image in the PPM format is a text file with a `.ppm` extension and is defined as follows:

- the first line of the file must be `P3`.
- the second line contains the width and the height of the image separated by a space character.
- the third line is a maximum theoretical value for each red/green/blue component of the pixels. For the project, we will use `255`.
- the following lines contains the list of pixels of the image from left to right, then from up to down. Each pixel is coded in RGB format, with the value exprimed as text, each component separated from the next by at least a space character, and each pixel separated from the next by a space character. You can have a line of text for each line of pixels, but it is not mandatory.

The RGB format indicates the quantity of Red/Green/Blue in a pixel. A white pixel has all its components at maximum value, i.e. `255 255 255`. A black pixel will be full off, i.e. `0 0 0`. Blue will be `0 0 255`, dark blue will have less blue and will be `0 0 50`, yellow is a mix of red and green (`255 255 0`) and so on...
For instance,

```
P3
3 2
255
  0   0 255  255 255 255  255   0   0
  0   0 255  255 255 255  255   0   0
```

will display a mini french flag.
You will find in your repository (in the `data` directory) an example of a PPM file representing the letter T. Do not hesitate to open the PPM file in VSCode.
You must provide a `naive_generate_image` function in the `naive_ppm_writer.c` file. This function takes an universe and a number `num` and generates a PPM file in the `out` directory with name `img-num.ppm` (format `num` on 3 digits).
You must provide a test for your generator in the `test-naive-generate-image.c` file. This program should open the `data/glider.txt` universe and generate the corresponding PPM file with number `000`.
You must also modify your `app-naive-conway` program to generate images at each step of the simulation.

> Do not add the PPM files in your repository! Normally, SVN should ignore them.

The `video` target in your Makefile will create a MP4 video `video.mp4` with all the PPM files in your `out` directory. By default, it will create a 1024×768 video, it should be sufficient. If you want to change these dimensions, for instance create a 800×600 video, use the following command:

```
make video WIDTH=800 HEIGHT=600
```

5.5 Conway's Game of Life with a linked list of alive cells

In order to store only alive cells, a linked list may be used. This list will contains cells with their coordinates and a boolean flag indicating if the cell will die at the next state of the computation and therefore must be removed. Conway universe will now be composed of such a linked list, and to allow universe that expands the width and height of the universe will not be stored but rather the current bounds of the universe. The implementation of such a solution may follow the implementation of the naive solution:

- first a linked list of cells must be specified and implemented in the `linked_list_cells.h` and `linked_list_cells.c` files. You may use the solution of lab session M9 on linked lists, but keep only necessary functions.

  It is strongly advised to implement a sorting function for the linked list as it will facilitate the writing of the following files. Choose a merge sort algorithm, as it is the easiest one to write on a linked list.

  You must provide a `test-linked-list-cell.c` test file that shows that your list behave correctly on simple examples: adding a cell, finding a cell in the list given its coordinates, removing dying cells and sorting the list.

- the universe of the game must be defined in the `list_universe.h` and `list_universe.c` files. The universe is a simple structure containing the bounds of the universe (both on abscissa and ordinate), the number of simulation steps and a linked list of alive cells.

  You must also write a `print_list_universe` function that prints the universe on the console. You should not create a **`char`** matrix in order to do so. Considering the list to be sorted should allow you to write an efficient algorithm.

  You must provide a `test-list-universe-print.c` application that creates a small universe « by hand » and prints it.

- a file loader for universes defined as lists must be defined in the `list_loader.h` and `list_loader.c` files. You must provide an application `app-list-loader` that use `main` arguments (cf. section D) to get a file name, build the corresponding structure and print it.

  We should be able to execute `app-list-loader` as follows:

  ```
      ./app-list-loader ./data/glider.txt
  ```

  to load the `glider.txt` file from the `data` directory in the current directory.

- the implementation of Conway's Game of Life with lists must be defined in the `list_conway.h` and `list_conway.c` files. You should provide a `list_step` function that takes an universe and apply the game's rules *once* and a `list_simulation` function that applies the game's rules for the number of steps defined in the universe.

  You may refer to algorithm 4.1 to implement the `list_step` function. Having a sorting function for your list should be helpful, particularly to find duplicates.

You must provide a `test-list-conway.c` test file that uses the provided `test-XXX.txt` files and the corresponding solutions to show that your simulation step is correct. Again, sorting the cells in the universe should facilitate the comparison between the universe you have computed and the expected one.

You should also provide a `app-list-conway.c` file with a simple `main` that takes a data file as parameter, execute the defined number of simulation steps and prints the universe at each step of the simulation. Test it on `data/bigger-example.txt` for instance.

- finally, an image generator for universes defined as lists must be defined in the `list_ppm_writer.h` and `list_ppm_writer.c` files. As you should have written a function to print the universe on the console, it should be easy to adapt it for the PPM format.

  You must provide a test for your generator in the `test-list-generate-image.c` file. This program should open the `data/glider.txt` universe and generate the corresponding PPM file with number `000`.

  You must also modify your `app-list-conway` program to generate images at each step of the simulation.

5.6 Development plan

You may apply the following development plan:

1. `naive_universe.h`, `naive_universe.c` and `test-naive-universe.c`: define necessary structures and the `print_naive_universe` function and test them.

   Notice that you may test your print function by defining a map structure directly in a program (i.e. without loading a file).
2. `naive_loader.h`, `naive_loader.c` and `test-naive-loader.c`: define the loader and test it.
3. `naive_conway.h`, `naive_conway.c`, `app-naive-conway.c` and `test-naive-conway.c`: simulate Conway's Game of Life, test it on one step on the provided tests and create an application to use it.
4. `naive_ppm_writer.h`, `naive_ppm_writer.c` and `test-naive-generate-image.c`: define a function to generate a PPM image from an universe and its associated test file. Modify `app-naive-conway.c` to generate PPM files.
5. `linked_list_cell.h`, `linked_list_cell.c`: define the linked list of cells.
6. `list_universe.h`, `list_universe.c` and `test-list-universe.c`: define necessary structures and the `print_list_universe` function and test them.
7. `list_loader.h`, `list_loader.c` and `app-list-loader.c`: define the loader and corresponding application.
8. `list_conway.h`, `list_conway.c`, `app-list-conway.c` and `test-list-conway.c`: simulate Conway's Game of Life, test it on one step on the provided tests and create an application to use it.
9. `list_ppm_writer.h`, `list_ppm_writer.c` and `test-list-generate-image.c`: define a function to generate a PPM image from an universe and its associated test file. Modify `app-list-conway.c` to generate PPM files.

Do not try to implement several functionalities at once, it will not work! Implement **and test** your code incrementally.
See section 7 to see how you will be graded.

5.7 Summary of files to produce

The summary of files to produce is given on table 1.

6 Work to do and requirements

What you have to do and the project requirements are summarized in the following points. The project is due **03/19/2021 23h59**. **The code retrieved from your repository at this date will be considered as the final version of your project.**
You will have to commit working code for the naive Conway's Game of Life simulation on **03/05/2021 23h59**. If not, penalty will be applied on your final grade.

	specification	implementation
naive universe	`naive_universe.h`	`naive_universe.c`
naive loader	`naive_loader.h`	`naive_loader.c`
naive algorithm	`naive_conway.h`	`naive_conway.c`
naive image generator	`naive_ppm_writer.h`	`naive_ppm_writer.c`
naive loader application		`app-naive-loader.c`
naive application		`app-naive-conway.c`
test of naive universe		`test-naive-universe.c`
test of naive loader		`test-naive-loader.c`
test of naive simulation		`test-naive-conway.c`
test of naive image gen.		`test-naive-generate-image.c`
linked list	`linked_list_cell.h`	`linked_list_cell.c`
list universe	`list_universe.h`	`list_universe.c`
list loader	`list_loader.h`	`list_loader.c`
list algorithm	`list_conway.h`	`list_conway.c`
list image generator	`list_ppm_writer.h`	`list_ppm_writer.c`
list loader application		`app-list-loader.c`
list application		`app-list-conway.c`
test of list universe		`test-list-universe.c`
test of list loader		`test-list-loader.c`
test of list simulation		`test-list-conway.c`
test of list image gen.		`test-naive-generate-image.c`

Table 1: Summary of files to produce

6.1 Requirements about functionalities

[R1] you must implement the following functionalities: a program to simulate naively Conway's Game of Life, a file loader, a PPM writer and tests with also a solution implemented as a list of cells as defined in section 5.

[R2] you may implement one of the following extensions: use a BST to store alive cells to be more efficient, implement a quadtree, the most obvious data structure to implement.. You may also find yourself an interesting extension. These extensions will be taken into account in your final grade only if the basic functionalities are correctly working.

6.2 Requirements about implementation

[R3] your implementation must be written in C.

[R4] your code must compile with the `-std=c99 -Wall -Werror` options.

[R5] your must use the provided Makefile (see appendix B). The rule `compile-all` should compile all your executables and tests.

[R6] your code must work on the ISAE computers under Linux.

[R7] you must write

- a program in the `app-naive-loader.c` file that loads a file representing an universe and print it.
- a program in the `app-naive-conway.c` file that loads a file representing an universe, simulates it and generates the corresponding PPM images.
- a program in the `app-list-loader.c` file that loads a file representing a list universe and print it.
- a program in the `app-list-conway.c` file that loads a file representing an universe, simulates it and generates the corresponding PPM images using the linked list.

If you have not implemented one functionality, do not code the corresponding part of the program.

[R8] you must write the test files as defined in section 5.

[R9] you may add programs to test your implementation. They must be located in files beginning with `app-`.

[R10] your data files (universes) should be located in the `data` directory and have a `.txt` suffix.

[R11] you must complete the given Makefile with at least a `compile-all` target that compiles all your programs and tests. Use the provided Makefile. See section B for more details.

[R12] your code must not produce errors when using the Valgrind tool.

[R13] you may reuse code **YOU** have implemented during the lab sessions.

6.3 Requirements about code conventions

[R14] your header files must be documented, possibly with the Doxygen tool (look at lab sessions and the corresponding refcard on LMS). Do not document preconditions or postconditions, only functions behavior, their returns and their parameters. You should add normal C comments in your code to explain your implementation if they are relevant.

[R15] your C code must follow the code conventions explained during the lecture.

6.4 Requirements about your repository

[R16] you must commit your work at each successful functionality implementation. An intelligible message should be added to the commit.

[R17] you must follow the following directories convention: your `.c` files must be in the `src` directory or in the `tests` directory for tests and your `.h` files in the `include` directory. You may put the `.o` and executable files wherever you want. See section C for more details.

7 Grading

Your final grade will be calculated as presented on table 2. **Beware, your tests will be used to evaluate your work!**

Part	Details	Grade	Comments
Naive implementation			
	universe structure	1	
	loader	1	
	image generator	2	
	naive simulation	4	
List implementation			
	linked list	1	
	universe structure	1	
	loader	0.5	
	image generator	0.5	
	list simulation	3	
Code quality			
	overall quality	4	variables naming, documentation, no unuseful code duplication etc.
	tests quality	2	
	your code does not compile	-5	
	your code produces errors with valgrind	-2	maximum malus, it depends on the number of errors
	incorrect use of SVN	-1	not useful commit messages etc.

Table 2: Grade details

A successful extension implementation will give you up to 4 points **if the basic functionalities are correctly implemented**.

Notice that you may have a really good grade even if you do not implement all the functionalities: **the quality of your code is really important!**

Your code will be analyzed by JPlag [2], a code plagiarism detector. DO NOT TAKE CODE FROM ANOTHER STUDENT, EVEN IF YOU TRY TO DISGUISE IT.
If you are convinced of plagiarism, you will obtain the "R" grade for the module.

A Testing functions

In this section, I will present how to write tests for your project. If you want to test a function or a property for the project, you should

- create a file whose name begins with `test-` in the `tests` directory
- add a rule to build the corresponding executable in your Makefile and add the test to the global `launch-tests` variable (see section B)
- use the C `assert` macro to verify properties
- define one or several functions, each one corresponding to a specific test or property
- call these functions into your `main` function
- use simple `printf` commands to indicate that the tests are satisfied

You will find in your `tests` directory a `test-dummy.c` file with two dummy tests about integer arithmetics that can serve as an example. There is a corresponding rule in your Makefile to build the executable.
If your test needs external information such as a filename, use `main` arguments (cf. section D). Do not add the test to the `launch-tests` variable of your Makefile.

B Makefile

In order to compile your applications and tests, you should use the Makefile provided in your repository. Here are some explanations:

- the `doc` rule generates the Doxygen documentation from your `.h` files. The documentation is located in the `doc` directory
- the `clean` rule removes all generated `.o` files and executables
- the C files in the `src` and `tests` directories can be automatically compiled. For instance, if you want to produce the `.o` file from `src/my_file.c`, just execute the following command:

  ```
  make my_file.o
  ```

- you should put all the `.o` file names that can be generated from your `.c` files as prerequisites of the `check-syntax` rule. Therefore, executing `make check-syntax` should compile all your C source files
- you should create a compilation rule for each of your applications or tests (see examples). You can use the `.o` files as prerequisites as the provided rules generate them from your `.c` files. Do not forget to add the needed header files as prerequisites
- you should add all your executables (applications and tests) as prerequisites of the `compile-all` rule. Therefore, executing `make compile-all` should generate all your applications and tests
- you should add all your test executables in the `ALL_TESTS` variables. Therefore, executing `make launch-tests` should execute all your tests.

  Beware: do not add tests that need an argument on the command line
- the `OPT` variable can be used to add the `-O3 -flto` options to GCC (and thus optimize aggressively your code). To use it, call `make` with `OPT=1` as for instance

```
make compile-all OPT=1
```

- you can put debug messages in your code using `printf`. In order to avoid "polluting" the execution of your applications, you can encapsulate these messages in a preprocessor directive:

  ```
  #ifdef DEBUG
  printf("this is a debug message...\n");
  #endif
  ```

  When compiling your applications, all code between the `#ifdef DEBUG` and `#endif` directives will be wiped except if you use `make` with `DEBUG=1` as for instance

  ```
  make compile-all DEBUG=1
  ```

C Repository organization

Your repository is organized as follows:

- the `include` directory contains your header files (`.h` files)
- the `src` directory contains your C source files, except tests
- the `tests` directory contains your C test files
- the `doc` directory contains the documentation generated by Doxygen

You should compile your files using the provided Makefile (see section B).
You must include a `README.txt` file at the root of your repository to give some explanations on your project, particularly how to run executables that need command-line arguments.

D Passing parameters to a program from command line

You have seen in session M0 that you can pass parameters to a Python program using the `sys.argv` list. There is a similar mechanism in C when you use the following signature for `main`:

```
int main(int argc, char *argv[])
```

`argc` represents the length of `argv` and `argv` is an array of strings containing the parameters given on the command line and the name of the executable in position 0.
Let us take an example: let us suppose that you have such a `main` function in a C file compiled to a `example-main` executable. Then, when executing

```
./example-main hello world
```

- `argc` has the value 3, as there is two parameters
- `argv[0]` is `./example-main`, the name of the executable
- `argv[1]` is `hello`
- `argv[2]` is `world`

You will find in your `src` directory an `example-main.c` file that prints these informations when executing the corresponding executable.

E Building strings from numbers etc.

You may sometimes need to build a string using numbers etc, for instance when defining a filename to write experimental results. To do so, you may use the `sprintf` function declared in `stdio.h`. This function behaves like `printf`, but the resulting string is not printed on the console but assigned to the first argument of the function.
Therefore, after executing the following lines:

```
char my_string[1024];

sprintf(my_string, "filename-%d.%s", 32, txt);
```

`my_string` will be the string `filename-32.txt`.
Of course, declaring a string of length 1024 is not necessary here, so you may have to compute the exact length.
You will find in you `src` directory an `example-string.c` file that gives you an example on how to do it.

F File Input/Output in C

If you want to read or write data to a file in C, you should use the structures and functions provided by the `stdio.h` header file. We will present here only simple cases when wanting to read or write formatted data from or to a text file. You may find more information in [4] or manual pages for the functions that are described in the following.

F.1 Opening and closing files

In order to read from files (or write to files), you must first open them. `stdio.h` provides a type, `FILE`, to represent a data stream from or/and to a particular file. Let us suppose that we want to read the content of a file `data.txt`. To open the file, we will use the `fopen` function and get a pointer to a `FILE` object:

```
FILE *p_file = fopen("data.txt", "r+");
```

Some remarks:

- `"data.txt"` represents the path to the file, it is here a relative path (the file `data.txt` must be in the current directory).
- `"r"` is the mode of the stream and specifies what you want to do with the file. You will mainly use the following modes:
  - `"r"` to read a file
  - `"w"` to write to a file
  - `"r+"` to read and write to a file
- if there are some errors when opening the file, the returned pointer is `NULL`. You should therefore test `p_file`.

When you have finished working with a file, **you should close it.** This is particularly true when writing to a file, as the file may have been put in central memory and the modifications you have done to the file might not be commited by the operating system (if you want to know more, you may refer to [1]). To close a file, simply call the `fclose` function on your pointer:

```
fclose(p_file);
```

F.2 Reading from files

F.2.1 Reading formatted input

If your file contains only **readable characters** and you know that data will follow a given format, then you may use the `fscanf` function. `fscanf` behaves like the `scanf` function we have seen in lab M1, but it works on files instead of standard input.
Let us look at the `fscanf` signature. It needs:

- a `FILE *` pointer to the file you want to read from
- a **format string** in the same format than the `printf` function

- several pointer variables to store what you read

Let us suppose that we want to read **int** values from a file with the following format:

```
2 - 4
4 - 12
5 - 42
13 - 2323
```

Each line of the file has two **int** values separated by the string `" - "`. Let us suppose that `first_value` and `second_value` are **int** variables in which we want to store the read values, then the call to `fscanf` may be:

```
fscanf(p_file, "%d - %d", &first_value, &second_value);
```

The format string is `"%d - %d"`: we expect an integer, then a space, then `-`, then a space, then another integer. `fscanf` will return the number of input items successfully matched and assigned to the variables. Therefore, if `fscanf` returns 2, then the line has been correctly read and the variables assigned.

You may wonder how you may know when the file has been completely read. The `EOF` special value is returned by `fscanf` when it encounters the end of the file.

When using `fscanf`, the pointer `p_file` changes! After calling for instance the previous instruction, `p_file` will then "point" after the two integers. Therefore, you should not reuse directly `p_file` supposing that it points at the beginning of the file.

You will find in the `src` directory of your repository a simple program in the `read-file-formatted.c` file that tries to print all lines of a file respecting the previous syntax (cf. listing 1). You can compile it with the following Makefile command:

```
make read-file-formatted
```

You can execute it on the file `data-toread-formatted.txt` from your `data` directory with the following command[2]:

```
./read-file-formatted data/data-toread-formatted.txt
```

Try it on different inputs to verify that it works correctly, particularly when there are errors in the data file (see `data-toread-error.txt` in the `data` directory).

Listing 1: A simple program to read formatted text files

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    int first_int      = 0;
    int second_int     = 0;
```

[2] `data-toread-formatted.txt` must be in the `data` directory in the command, but you can use relative or absolute paths to call `read-file-formatted` on files that are not in the current directory.

```
    int line_nb       = 1;
    int fscanf_result = 0;

    fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);

    while (fscanf_result != EOF) {
        if (fscanf_result != 2) {
            fprintf(stderr, "Line number %d is not syntactically correct!\n",
                    line_nb);
            exit(EXIT_FAILURE);
        }

        printf("first value at line %d: %d\n", line_nb, first_int);
        printf("second value at line %d: %d\n", line_nb, second_int);

        line_nb = line_nb + 1;
        fscanf_result = fscanf(p_file, "%d - %d", &first_int, &second_int);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

F.2.2 Reading text files

If you want to read simple text files[3], do not use `fscanf` but rather the `fgets` function. `fgets` takes as input:

- a `char *` string `s` that is used as a buffer, i.e. `s` will contain after the call to `fgets`
- an `int` value `size` that indicates the number of characters to be *at most* read
- a `FILE *` argument that represents the file to be read

Notice that `fgets` stops when encountering a newline or the `EOF` character representing the end of the file. Notice also that you should know the maximum length of the lines of the file in order to have correct `s` and `size` arguments. `fgets` will return NULL when encountering the end of the file.
Let us take an example and consider the following file:

```
Sing, O goddess, the anger of Achilles son of Peleus, that brought
countless ills upon the Achaeans. Many a brave soul did it send
hurrying down to Hades, and many a hero did it yield a prey to dogs
and vultures, for so were the counsels of Jove fulfilled from the day
on which the son of Atreus, king of men, and great Achilles, first
fell out with one another.
```

We know that the length of each line will not be more than 80 characters, therefore we could read the file and prints each line as presented on listing 2. Compile the executable with the corresponding Makefile rule and use it on the `data/iliad.txt` file in your repository.

Listing 2: A simple program to read text files

```
#include <stdio.h>
#include <stdlib.h>
```

[3] Or part of a file that contains only text

```c
int main(int argc, char **argv) {
    FILE *p_file = NULL;

    p_file = fopen(argv[1], "r");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot read file %s!\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    // declaration of a buffer of 81 chars: we suppose that each line
    // of the file has no more than 80 chars and we add 1 char for the
    // final '\0' character
    char buffer[81];
    int  line = 0;

    // while not having encountered end of file, read lines
    while (fgets(buffer, 80, p_file) != NULL) {
        line++;

        printf("%2d: %s", line, buffer);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

F.3 Writing to files

If you want to write formatted content to a file, you must first open the file with the `"r+"` or `"w"` modes with `fopen`. You then use the `fprintf` function. It behaves like the `printf` function, but it takes as first argument a `FILE *` pointer to the file you want to write to.
For instance, the program presented on listing 3 writes the factorial of the 10 first natural numbers to the file `fact.txt`. The program is available in the `src` directory of your repository and a compilation target is provided in your Makefile.

Listing 3: A simple program to write some computations in a text file

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *p_file = NULL;

    p_file = fopen("fact.txt", "w");

    if (p_file == NULL) {
        fprintf(stderr, "Cannot write to file fact.txt!\n");
        exit(EXIT_FAILURE);
    }

    int fact = 1;

    for (int i = 0; i < 10; i++) {
        fprintf(p_file, "%d! = %d\n", i, fact);
```

```
        fact = fact * (i + 1);
    }

    fclose(p_file);

    p_file = NULL;

    return 0;
}
```

References

[1] Ulrich Drepper. "What Every Programmer Should Know About Memory". Nov. 21, 2007. URL: http://www.akkadia.org/drepper/cpumemory.pdf.

[2] Karlsruhe Institute of Technology. *JPlag – Detecting Software Plagiarism*. 2016. URL: https://jplag.ipd.kit.edu/.

[3] Paul Rendell. *A Turing machine implemented in Conway's Game of Life*. 2010. URL: http://rendell-attic.org/gol/tm.htm.

[4] Wikibooks. *C Programming, File IO — Wikibooks, The Free Textbook Project*. 2015. URL: https://en.wikibooks.org/wiki/C_Programming/File_IO.

[5] Wikipedia contributors. *Conway's Game of Life*. 2016. URL: https://en.wikipedia.org/wiki/Conway%5C%27s_Game_of_Life.

[6] Wikipedia contributors. *Netpbm format*. 2016. URL: https://en.wikipedia.org/wiki/Netpbm_format.

License CC BY-NC-SA 3.0