Exercises — Game of life

version **#0.0.1**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.forge.epita.fr

File Tree

```
game_of_life/
├── Makefile
├── examples/
│   ├── clown.txt
│   └── gosper_glider_gun.txt
└── src/
    ├── args.c
    ├── args.h
    ├── game.c  (to submit)
    ├── game.h  (to submit)
    ├── grid.c  (to submit)
    ├── grid.h  (to submit)
    ├── gui.c
    ├── gui.h
    ├── io.c
    ├── io.h
    ├── main.c
    ├── terminal.c
    └── terminal.h
```

Authorized functions : You are only allowed to use the following functions

- malloc(3)
- calloc(3)
- free(3)

Authorized headers : You are only allowed to use the functions defined in the following headers

- err.h
- errno.h
- assert.h
- stddef.h

Compilation : Your code must compile with the following flags

- -std=c99 -pedantic -Werror -Wall -Wextra -Wvla

1 Game of Life

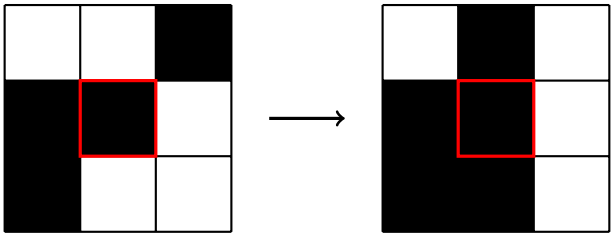
1.1 Introduction

The Game of Life is a zero-player game created by John Coway in 1970. It has become part of popular culture to the point that Google includes an easter egg if you search for “Game of Life”. Its popularity is due to the fact that it shows how complexity can arise from a very simple set of rules.

This “game” is played on a 2D grid made of *cells*. Each cell is either **alive** or **dead** and the next state of a cell is defined by its surrounding:

- A live cell with fewer than two live neighbors dies because of *underopulation*
- A live cell with more than three live neighbors dies because of *overpopulation*
- A live cell with exactly two or three live neighbors stays alive
- A dead cell with exactly three live neighbors comes to life

The neighborhood of a cell is the eight surrounding cells on the grid. Consider the following example:

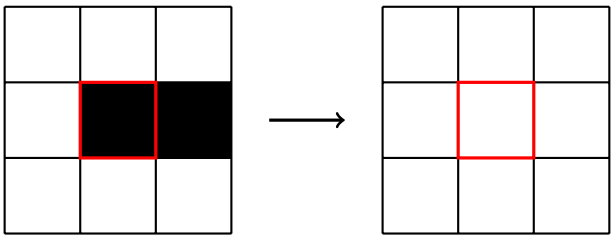


Tips

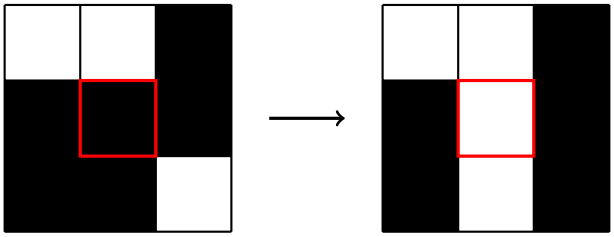
On the examples, black cells are alive while white cells are dead.

Here, the center cell has exactly 3 neighbors so it stays alive in the next generation.

Here is an example of *underpopulation*:

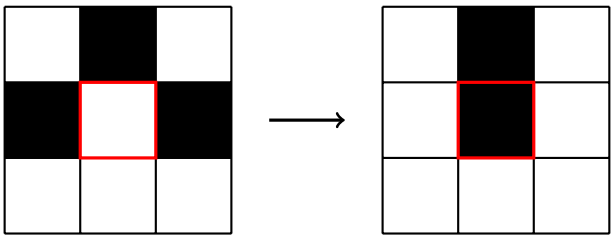


The center cell has only 1 neighbor which makes it die from underpopulation on the next generation. *Overpopulation* works similarily:



In this case, the center cell has 5 neighbors and dies from overpopulation.

Finally, here is how a cell is born:



Initially, the center cell is dead but surrounded by exactly 3 live cells, which results in it coming to life.

1.2 Goal

The goal of this exercise is to implement the *Game of Life*. You do not have to handle the visual part as this is already in the given code. Your task will be to implement the rules of the game.

1.2.1 Grid

Open the file `grid.h`. It contains a structure named `grid`. This is the underlying representation of the Game of Life grid. The enum `cell_state` represents the two possible states for a cell: `DEAD` or `ALIVE`.

Tips

Enums in *C* work roughly like `enum.IntEnum` in Python. Each member of an enum can be used as an integer constant that is equal to its associated value. To access enum members in *C*, you do not need to prefix them by the enum name:

```
printf("DEAD = %d, ALIVE = %d\n", DEAD, ALIVE); // DEAD = 0, ALIVE = 1
```

Since you will have to represent a 2D grid, you can choose between two possible implementations:

- Use an array of arrays.
- Use a contiguous array and compute the indexes using the dimensions of the subarrays.

Both implementations are valid, each having their advantages and drawbacks.

Going further...

The second implementation is generally preferred because allocating a contiguous array prevents memory fragmentation.

Note that the structure body is empty and contains a `// FIXME` comment. It is indeed up to you to fill it with the appropriate field(s) depending on your project architecture.

Implement the functions used to manipulate the grid in `grid.c`.

Tips

You are of course allowed to write your own functions to help you. Note that they still have to comply with the Coding Style.

grid_init

```
struct grid *grid_init(size_t width, size_t height);
```

Return a newly allocated `grid` with the right dimensions. Initially, all cells must be dead.

grid_destroy

```
void grid_destroy(struct grid *grid);
```

Release the memory allocated for the `grid`. After it has been called, the `grid` will not be usable anymore.

grid_get_cell

```
int grid_get_cell(const struct grid *grid, size_t x, size_t y);
```

Return the state (dead or alive) of the cell at position `(x, y)`. If the arguments are invalid, return -1.

grid_set_cell

```
int grid_set_cell(struct grid *grid, size_t x, size_t y, enum cell_state state);
```

Set the state of the cell at position `(x, y)`. Return 0 if the arguments are valid, -1 otherwise.

grid_get_width

```

```

Return the width of the `grid` if the argument is not `NULL`, -1 otherwise.

Going further...

The type `ssize_t` is a signed alternaive to `size_t`. It is often used instead of `size_t` when it is necessary to indicate an error with -1. The C library POSIX specification defines it like this:

The type ssize_t shall be capable of storing values at least in the range [-1, {SSIZE_MAX}]

grid_get_height

```
```

Return the height of the `grid` if the argument is not `NULL`, -1 otherwise.

1.2.2 Game

The structure `game` defined in `game.h` wraps the grid and potentially adds additional information about the game itself like for instance the number of dead cells. Once again, you are free to implement it as you wish, except for the first member `grid` that you must **not** change:

Some functions of `game` simply forward the call to the functions that manipulate the underlying `grid` while others need additional processing. It is up to you to decide which part of the logic belongs to the `grid` and which part belongs to the `game`.

Implement the functions in `game.c`:

game_init

```
struct game *game_init(size_t width, size_t height);
```

Return a newly allocated `game` with the right dimensions. Initially, all cells must be dead.

game_destroy

```
void game_destroy(struct game *game);
```

Release the memory allocated for the `game`. After it has been called, the `game` will not be usable anymore.

game_get_cell

```
int game_get_cell(const struct game *game, size_t x, size_t y);
```

Similar to *grid_get_cell*.

game_set_cell

```
int game_set_cell(struct game *game, size_t x, size_t y, enum cell_state value);
```

Similar to *grid_set_cell*.

game_get_living_cells

```
ssize_t game_get_living_cells(const struct game *game);
```

Return the number of living cells if `game` is not `NULL`, -1 otherwise.

Tips

If you store the number of living/dead cells in the structure `game`, do not forget to update it when calling `game_set_cell`.

game_get_dead_cells

```
ssize_t game_get_dead_cells(const struct game *game);
```

Return the number of dead cells if `game` is not `NULL`, -1 otherwise.

game_next_generation

```
void game_next_generation(struct game *game);
```

Use the rule of the Game of Life to compute the next generation of the game. This function must work in-place and directly edit the `game` passed as an an argument.

Be careful!

You need to perform a copy of the grid for this function, else your result will be flawed by the fact that you change the grid will reading it. As always, do not forget to free the copy once you are done using it.

1.3 Given code

1.3.1 Usage

Once you have implemented the functions above, use the provided files to check that your code works and visualize the Game of Life. The rule `all` in the provided `Makefile` builds a binary file named `game_of_life`. You can print its usage help message with the option `-h`:

```
42sh$ ./game_of_life -h
Usage:
    ./game_of_life -f <filename> [-tg] [-r <fps>]
    ./game_of_life -h
```

The option `-f` allows you to start the game of life from a given file:

```
42h$ ./game_of_life -f examples/clown.txt -g
```

The option `-g` enables the GUI mode which opens a new window while the option `-t` displays the game directly in your terminal.

1.3.2 Implementation details

The GUI mode is implemented in the files `gui.c` and `gui.h`. It uses the SDL (Simple DirectMedia Layer) library to open a new window and draw in it.

The terminal mode uses ANSII escape sequences to clear the window, print full-length characters by entering revserse video mode and save the terminal content. To exit this mode, press the key `q`. The function *`pselect(2)`* is used to read from the standard input while writing in the standard output. You can look at the file `terminal.c` to see how it is done.

The command line options are parsed in `args.c` using the function *`getopt(3)`*. Here are the supported options:

Option	Purpose	Mandatory
`-f <filename>`	Load the initial grid from `filename`	Yes
`-g`	Enable GUI mode (default)	No
`-t`	Enable terminal mode	No
`-r <fps>`	Set the frame rate	No
`-h`	Display the help message and exit	No

Going further...

Some functions are defined with the keyword `static`. This means that those functions are only visible to functions that are in the same file.

I must not fear. Fear is the mind-killer.