Name: Bruna Duarte Alves Souza M Chaves.

Neptun: AHNX67

1. Conway's Game of Life

This program implemented the "Game of Life". Conway's game of life is an example of a cellular automaton, developed by the mathematician John Horton Conway in 1970.

Despite being called a game, this is not a conventional game because it does not depend on players and does not have a winner or loser. The game needs only an initial state and a set of rules that are applied to each interaction, and its development can be observed. The initial state is called the "system seed" and is determined by the user.

The game, which is also known as Life, is played in a grid of cells. Each cell can be in one of these two states: alive or dead. A living cell is shown in the matrix with a ‘1’ in its place and a dead cell is shown as a zero. Each cell has 8 neighbors: two horizontally, two vertically, and four on the diagonals.

At each interaction, rules are applied on each cell, taking into account the state of neighboring cells. The rules are:

- Any live cell with fewer than two live neighbors dies, as if by 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, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

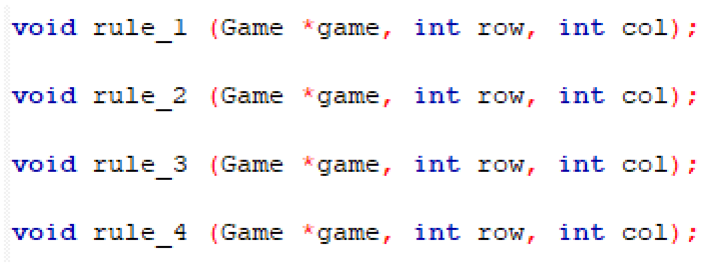
It is worth remembering that the number of living and dead neighbors in a cell is calculated before applying the rule to each cell. The rules continue to be applied repeatedly to create further generations. Life is one of the simplest cell automata invented and several others have been invented, often with the aim of simulating real-world systems.

2. Implementing in C

To implement this Game of Life in C, the program was divided in two blocks: “main.c” and “game.c” together with its header “game.h”.

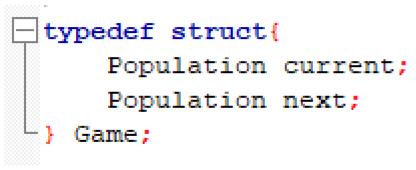
2.1. The Game Module

The module *game* is responsible for implementing the rules of the game, the configuration of the game and its board and printing all the game iterations. As stated before, the game has four rules and each one was implemented through a function as shown in the picture below. Each function receives as input and output parameter one pointer of type Game, and two integers as input parameter to specify the row and column to which the rule is applied. These functions can modify the status of each cell (dead or alive).



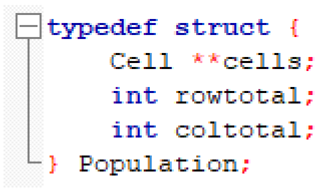
```
void rule_1 (Game *game, int row, int col);

void rule_2 (Game *game, int row, int col);

void rule_3 (Game *game, int row, int col);

void rule_4 (Game *game, int row, int col);
```

Game is a struct containing the “current” and “next” boards. These two boards are needed because the rules need to be applied based on the “current” board, and the modifications are saved in the “next” board. This way the rules applied to one cell do not affect the others cells yet to be checked.



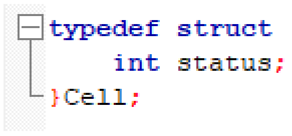
```
typedef struct{
    Population current;
    Population next;
} Game;
```

Each board is of the type “Population”. The Population is a struct that contains one 2D array of type Cell, and two integers that specifies the total number of rows and columns. The 2D array was chosen instead of 1D because it’s easier to access each cell by using only the correspondent row and column (cells[row][col]).



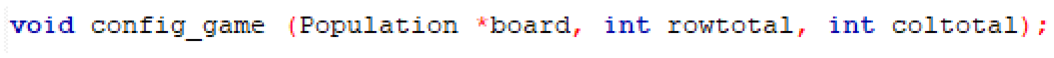
```
typedef struct {
    Cell **cells;
    int rowtotal;
    int coltotal;
} Population;
```

Cell is a struct that contains only one integer that specifies the status of each cell (dead = 0 or alive = 1).



```
typedef struct {
    int status;
}Cell;
```

Another important function implemented in the *game* module is the config_game function that receives as output parameter one pointer to a Population and as input parameter two integers that specifies the total number of rows and columns in the board. The config_game function allocates memory for the game.



```
void config_game (Population *board, int rowtotal, int coltotal);
```

The function get_board_size also implemented in the *game* module, receives as input parameter a FILE type pointer, and as output parameter two integer pointers. This function counts the number of columns and rows from the input file “iniState.txt” and returns these values.

```
void get_board_size (FILE *fp, int *x, int *y);
```

The functions print_game and print_game_file were implemented to print ach iteration of the game. The first one to the standard output and the second one to the output file “fasedavida.txt”.

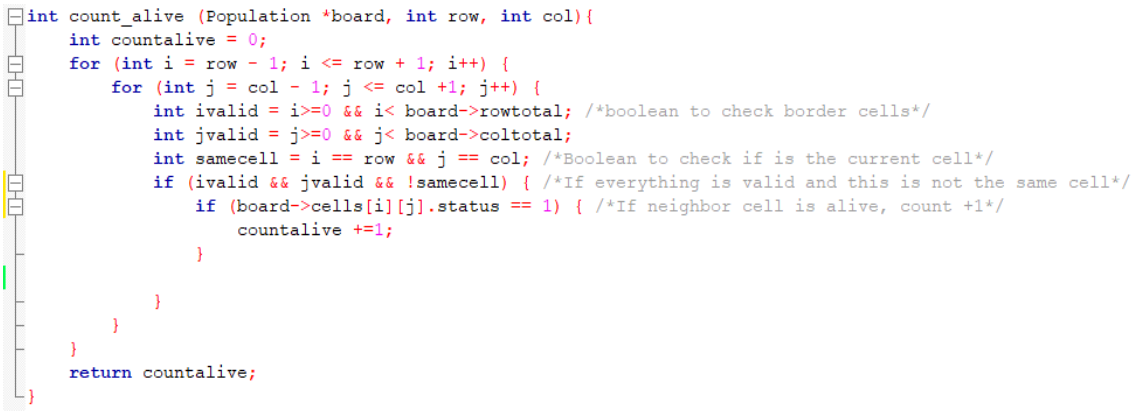
```
void print_game (Population *board);

void print_game_file (Population *board, FILE *fp, int iteration);
```

The only input parameter of the print_game function is a pointer to a board and as for the print_game_file function, it has as output parameter a FILE type pointer, and as input parameter one Population type pointer and one integer, that specifies which iteration the function is printing.

The seed_of_game function, receive as input parameter a FILE type pointer and as output parameter a Population type pointer. This function gets the “seed of the system” from the input file “iniState.txt” that must be filled out by the user, and passes to a board of the game.

```
void seed_of_game (FILE *fp, Population *board);
```

In the *game.c* module exists a local function called count_alive, which receives as input parameters one Population type pointer and two integers that specifies the row and column of the analyzed cell. This function counts how many live neighbors a cell has and return this value to the calling function as an integer.



```
int count_alive (Population *board, int row, int col){
    int countalive = 0;
    for (int i = row - 1; i <= row + 1; i++) {
        for (int j = col - 1; j <= col +1; j++) {
            int ivalid = i>=0 && i< board->rowtotal; /*boolean to check border cells*/
            int jvalid = j>=0 && j< board->coltotal;
            int samecell = i == row && j == col; /*Boolean to check if is the current cell*/
            if (ivalid && jvalid && !samecell) { /*If everything is valid and this is not the same cell*/
                if (board->cells[i][j].status == 1) { /*If neighbor cell is alive, count +1*/
                    countalive +=1;
                }

            }
        }
    }
    return countalive;
}
```

This is an important function because all of the rules are based on the number of alive neighbors. To count the alive neighbors it has a counter ‘countalive’ that was initialized with zero. For each cell [i] [j] we have to check the neighbors from i-1 to i+1, and from j-1 to j+1, which are the 8 neighbors a central cell has. Some cells (border cells) don’t have all the 8 neighbors, in this case we need to check if the neighbors indexes are greater or equal to 0 and less than the board size (board->rowtotal / board->coltotal). Also we don’t need to count the cell itself, only its neighbors, so we check if it is the same cell (samecell = i == row && j == col). If all these conditions were attended, we count countalive +=1.

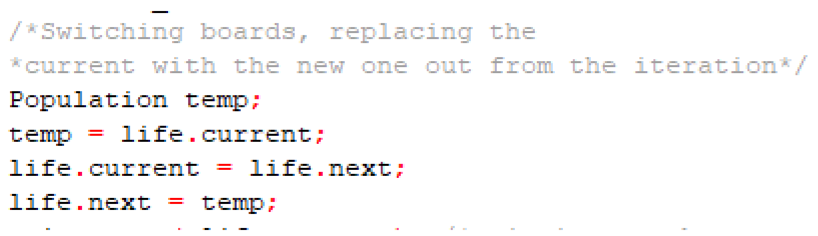
2.2. The Main Module

The *main* module is responsible for presenting the game interface to the user and for calling the functions of the game. It is also responsible for opening and closing the input and output files.

The main function calls the get_board_size to get the size of the board from the input file, then calls the config_game and configures both boards used by the program (‘current’ and ‘next’). Additionally it calls the seed_of_game function to get the seed of system described by the user in the input file, and then it prints to the standard output the menu options.

The menu of the game works accordingly to the user’s choice through a switch-case. The user can choose to run the game continuously (case ‘1’), step-by-step (case ‘2’) or exit the game (case ‘3’).

In both cases 1 and 2, the main function will apply the four rules to the current board, saving the result in the “next” board. After that the “next” board is switched in order to become the “current” board and this way be used by the functions in the next iteration.



```
/*Switching boards, replacing the
*current with the new one out from the iteration*/
Population temp;
temp = life.current;
life.current = life.next;
life.next = temp;
```

3. Testing

- Test case 1: initial state size 1x1

This tests a situation where the user’s input has only one cell.

Input
1
Output
Iteration 0: 1 Iteration 1: 0

- Test case 2: initial state size 6x4

This tests the game for case menu ‘1’, and the ‘ESC’ was pressed at the 6th iteration.

Input

```
1 0 0 1 1 1
0 1 0 0 0 1
1 0 0 0 1 1
0 1 0 1 1 1
```

Output

```
Iteration 0:

1 0 0 1 1 1
0 1 0 0 0 1
1 0 0 0 1 1
0 1 0 1 1 1

Iteration 1:

0 0 0 0 1 1
1 1 0 1 0 0
1 1 1 1 0 0
0 0 0 1 0 1

Iteration 2:

0 0 0 0 1 0
1 0 0 1 0 0
1 0 0 1 0 0
0 1 0 1 1 0

Iteration 3:

0 0 0 0 0 0
0 0 0 1 1 0
1 1 0 1 0 0
0 0 1 1 1 0

Iteration 4:
```

```
0 0 0 0 0 0
0 0 1 1 1 0
0 1 0 0 0 0
0 1 1 1 1 0

Iteration 5:

0 0 0 1 0 0
0 0 1 1 0 0
0 1 0 0 0 0
0 1 1 1 0 0

Iteration 6:

0 0 1 1 0 0
0 0 1 1 0 0
0 1 0 0 0 0
0 1 1 0 0 0
```

- Test case 3: initial state size 3x7

This tests for menu case ‘2’. The ‘ESC’ is pressed after 4 iterations.

Input

```
1 0 0 1 1 1 0
0 1 0 0 0 1 1
1 0 0 0 1 1 1
```

Output

```
Iteration 0:

1 0 0 1 1 1 0
0 1 0 0 0 1 1
1 0 0 0 1 1 1

Iteration 1:
```

```
0 0 0 0 1 1 1
1 1 0 1 0 0 0
0 0 0 0 1 0 1
```

Iteration 2:

```
0 0 0 0 1 1 0
0 0 0 1 0 0 1
0 0 0 0 0 0 0
```

Iteration 3:

```
0 0 0 0 1 1 0
0 0 0 0 1 1 0
0 0 0 0 0 0 0
```

Iteration 4:

```
0 0 0 0 1 1 0
0 0 0 0 1 1 0
0 0 0 0 0 0 0
```

- Test case 4: initial state size 11x11

This tests the game for case menu ‘1’, and the ‘ESC’ was pressed at the 5th iteration.

Input

```
1 0 0 1 1 1 0 1 0 1 0
0 1 0 0 0 1 1 1 1 0 0
1 0 0 0 1 1 1 1 0 0 1
1 1 0 1 0 1 1 0 0 1 0
1 0 0 0 0 1 0 0 1 0 1
0 1 0 1 1 0 0 1 0 0 1
1 1 0 1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 1 0 1 0
1 1 1 0 0 0 1 1 1 0 0
```

```
0 0 0 0 0 1 0 1 0 0 0
0 1 0 1 0 0 1 1 0 0 1
```

Output

```
Iteration 0:

1 0 0 1 1 1 0 1 0 1 0
0 1 0 0 0 1 1 1 1 0 0
1 0 0 0 1 1 1 1 0 0 1
1 1 0 1 0 1 1 0 0 1 0
1 0 0 0 0 1 0 0 1 0 1
0 1 0 1 1 0 0 1 0 0 1
1 1 0 1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 1 0 1 0
1 1 1 0 0 0 1 1 1 0 0
0 0 0 0 0 1 0 1 0 0 0
0 1 0 1 0 0 1 1 0 0 1

Iteration 1:

0 0 0 0 1 1 0 1 0 0 0
1 1 0 1 0 0 0 0 0 1 0
1 0 1 0 0 0 0 0 0 1 0
1 1 0 0 0 0 0 0 1 1 1
1 0 0 1 0 1 0 1 1 0 1
0 1 0 1 1 0 1 1 1 1 0
1 1 0 1 0 0 0 1 0 0 0
0 0 0 1 1 0 0 0 0 0 0
0 1 1 0 1 1 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0

Iteration 2:

0 0 0 0 1 0 0 0 0 0 0
```

```
1 1 1 1 1 0 0 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 1 0 0 1
1 0 0 1 0 1 0 0 0 0 1
0 1 0 1 0 1 0 0 0 1 0
1 1 0 0 0 1 1 1 0 0 0
1 0 0 0 0 1 0 0 0 0 0
0 1 1 0 0 1 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0

Iteration 3:

0 1 1 0 1 0 0 0 0 0 0
0 1 1 0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0
1 0 0 1 0 0 1 0 0 1 1
0 1 0 0 0 1 0 0 0 0 0
1 1 1 0 0 1 0 0 0 0 0
1 0 1 0 1 1 0 0 0 0 0
1 1 1 0 0 1 1 0 0 0 0
0 1 1 0 1 1 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0

Iteration 4:

0 1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0 0 0
0 1 0 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
1 0 1 1 0 1 1 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
```

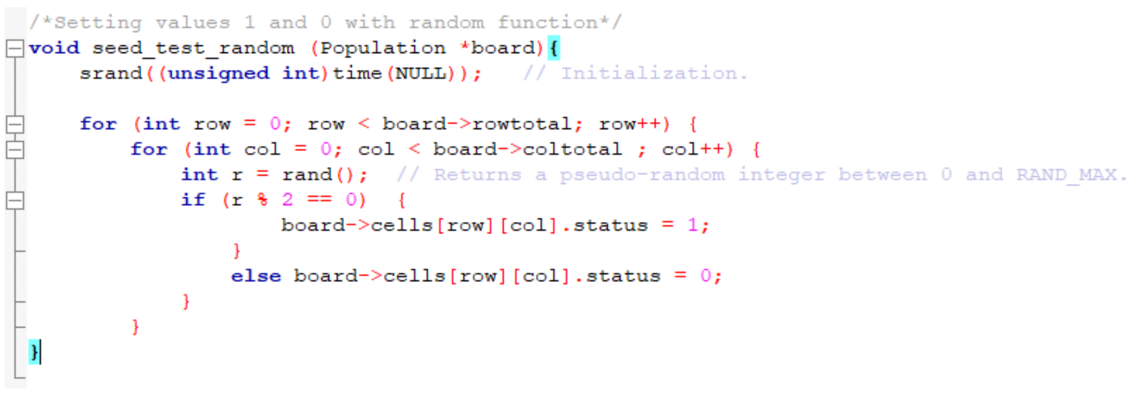
```
1 0 0 0 0 0 0 0 0 0 0
1 0 1 1 1 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0
```

Iteration 5:

```
0 1 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 1 0 1 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 1 0 0 0 0
0 1 0 1 1 1 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0
0 1 0 1 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0
```

- ➢ Test case 5: random iniState.txt

This test was done by a function called "seed_test_random" created to validate the program before inserting the file handling. This function receives a Population type pointer and sets the board cells randomly to 0 or 1.



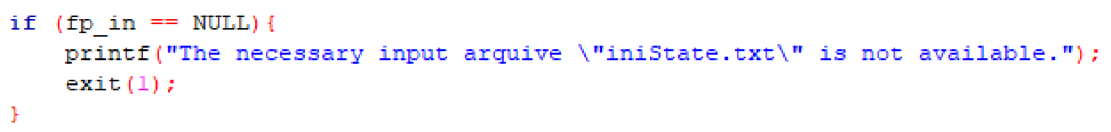
```
/*Setting values 1 and 0 with random function*/
void seed_test_random (Population *board){
    srand((unsigned int)time(NULL));    // Initialization.

    for (int row = 0; row < board->rowtotal; row++) {
        for (int col = 0; col < board->coltotal ; col++) {
            int r = rand();  // Returns a pseudo-random integer between 0 and RAND_MAX.
            if (r % 2 == 0)   {
                    board->cells[row][col].status = 1;
                }
                else board->cells[row][col].status = 0;
            }
        }
}
```

- ➢ Test case 5: user push many buttons at menu option 2

In order to avoid unwanted iterations when the user presses more than one button, the function _getch() was used instead of scanf() in case menu 2 code.

- ➢ Test case 6: missing iniState.txt file

To prevent the program to crash in case of the input file "iniState.txt" is somehow missing, the following was implemented in the main function:



```
if (fp_in == NULL){
    printf("The necessary input arquive \"iniState.txt\" is not available.");
    exit(1);
}
```

- Test case 7: invalid iniState.txt file (not covered)

Due to too many possible ways that an input file can be invalid, this was not covered. This way, the input file has to be provided exactly as specified in the manual.