EE231002 Introduction to Programming

Lab11. The Game of Life

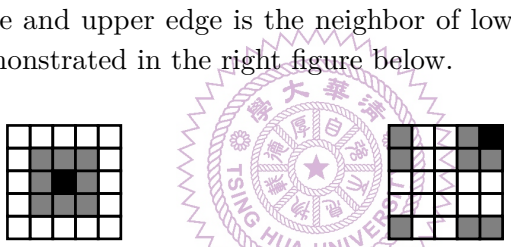
Due: Dec. 19, 2020

Before you start writing the program, please try out the demo program on EE workstations by key in

```
$ ~ee2310/lab11/demo 1 < ~ee2310/lab11/pat1.dat
```

You will see how the patterns evolves with time. This is the Game of Life. To stop the program execution, press `cntl-C`.

As you can see, the Game of Life is played on a board of a rectangle grid. In our case, we have 20×20 cells in the grid. Each cell have two status: `LIVE` or `DEAD`. Given the rules, which are shown below, these cells can be born (become `LIVE`), die (become `DEAD`), or simply maintain the same status, depending on the number of `LIVE` neighbors they have. The neighbors of a typical internal cell is shown on the left figure below. For edge or corner cells, in order to simulate an infinite universe, the edges of the grid are assumed to be wrapped around. That is, the left edge is the neighbor or right edge and upper edge is the neighbor of lower edge. Thus the neighbor cells of a corner cell are demonstrated in the right figure below.



The Game of Life was invented by a British mathematician John Horton Conway in 1970. References can be found on Wikipedia:

`http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

The rules for determining the fate of a cell are:

1. A dead cell with three live neighbors becomes a live cell.
2. A live cell with two or three live neighbors stays alive.
3. A live cell with less than 2 or greater than 3 live neighbors dies.

With these simple rules, the patterns of the live cells evolves with time. In some cases, all the live cells die after a period of time. In other cases, the cells can live forever. For the latter cases, the pattern might even become stationary. The stationary patterns are also called still patterns. In other cases, the pattern may become oscillating. The `pat1.dat` above is an example that reaches an oscillating state.

In this assignment we'll use the structure defined in `life.h` header file. Two `enum` types are defined: `STATUS` and `COLOR`. Thus, your program must use these `enum` types whenever it applies. The members `row` and `col` represents the position of a cell in the grid; `current` and `next` are the status of the cell. They can be either `LIVE` or `DEAD`. The member `age` represents how long a cell has been alive, and `color` requests the `showGrid` function to show the cell in a particular color. The member `Nnbr` should be the number of live neighbors of the cell. A global variable `grid[N][N]` is defined in the `main.c` file. It defines a grid of 20×20 that represents the universe that these cells reside.

The `main` function is rather simple, and the code is attached at the end of this assignment. It starts by processing the command line argument that represents the delay time between different generations. Then it reads in the initial grid from `stin`. This initial patterns should be read into the `next` member of each cell, while the `current` status should be initialized to `DEAD`. Other cell members should also be properly initialized. Then the `main` function check if a still life pattern is reached. This can be done by comparing the `current` and `next` members of each cell. If they are all the same, then a still life pattern has been found. In this `stillLife` function, after the comparison operations, the `current` status of each cell is also updated to the `next` state. If a still life pattern has not been reached, the program will show the updated grid on the terminal, followed by calling the `nextGen` function, which will determines the `next` state of each cell by the rules given above. Of course, all other members of each cell structure should also be updated accordingly. In the demo program, cell color is determined by it's age: `GREEN` for 1-year old, `YELLOW` for 2-year old, and `RED` for longer than 2-year old. But you are free to make your choice. You can make the game more colorful and interesting. After `nextGen` function call, the `main` function loops back to still pattern checking. If a still pattern is found, the program stops after display the grid contents for the last time.

Thus there are 4 functions, in addition to the `main` function, needed for this program. And they are declared in the `life.h` header file. One of them, `showGrid`, is provided already. Your assignment is to implement the other 3 functions to complete the Game of Life program.

1. `void readGrid(CELL grid[N][N]);`
   This function reads the initial pattern and store it to the `next` member of each each cell. It also initializes the cell contents to ensure proper execution of the program.

2. `int stillLife(CELL grid[N][N]);`
   This function checks for `still life` pattern by comparing the cell members `current` and `next`. If a still pattern is found, it returns 1 otherwise it returns 0. Before returning, the cell status should also be updated, that is, the `next` state is copied to the `current` state.

3. `void nextGen(CELL grid[N][N]);`
   This function determines the status of each cell according to the rules given above. Other structure members, such as `age` and `color`, should also be updated.

All these functions should be defined in a single file, `life.c`. Once that is done, you can use the following command to compile this file individually to make sure there is no compiler errors before complete program compilation.

```
$ gcc -c life.c
```

If all compiler errors and warnings have been removed. You can compile the entire program and execute it by the following commands.

```
$ gcc main.o life.o
$ ./a.out 1 < pat1.dat
```

Remember to press `cntl-C` to stop the execution. Other pattern files can also be tested for your own entertainment.

Notes.

1. There is another program, **`rgen`**, provided to demonstrate the Game of Life. It generates random patterns that can be fed to the **`demo`** program. It takes one **`int`** command line argument that specifies how many `LIVE` cells initially. Thus you can try the following:

   ```
   $ ~ee2310/lab11/rgen 90 | ~ee2310/lab11/demo 1
   ```

   And watch the patterns evolves. Of course, you can change the number 90 to other numbers to see the density effects on the Game of Life.

2. Create a directory **`lab11`** and use it as the working directory.

3. Name your program source file **`life.c`**.

4. The first few lines of your program should be comments as the following.

   ```
   // EE231002 Lab11. Game of Life
   // ID, Name
   // Date:
   ```

5. After you finish verifying your program, you can submit your source code by

   ```
   $ ~ee2310/bin/submit lab11 life.c
   ```

   If you see a "submitted successfully" message, then you are done. In case you want to check which file and at what time you submitted your labs, you can type in the following command:

   ```
   $ ~ee2310/bin/subrec lab11
   ```

   It will show the submission records of lab11.

6. The main function needs not be implemented. The definition of this function is shown below for your reference.

```
    // the main function of the Game of Life program

#include "life.h"

CELL grid[N][N];                        // global variable
int delay = 3;                          // delay time between generation

int main(int argc, char *argv[])
{
    int gen = 1;                          // generation of cells

    if (argc > 1) {                       // command line argument for delay
        delay = atoi(argv[1]);
        if (delay < 1) delay = 1;
        else if (delay > 20) delay = 20;
    }
    readGrid(grid);                       // read init patterns from stdin
    while (!stillLife(grid)) {            // check for still life
        showGrid(gen++, grid);            // display grid
        nextGen(grid);                    // calculate the next pattern
    }
    showGrid(gen++, grid);                // display grid
    printf("Still pattern reached!\n"); // stopped due to a still pattern
    return 0;
}
```