CS 1713, Introduction to Computer Science
Assignment 9, Spring 1998
The Game of Life, Due April 16, 1998

Life. For this assignment you are to write a C program that will simulate the game of "Life." Life takes place on an infinite 2-dimensional rectangular grid of cells. Initially (the 0th generation, or at time $t = 0$) some finite collection of these cells are "live"— the rest are "dead." At each successive generation, cells may remain live or dead, or may switch states according to fixed rules. Thus the course of the game depends on the initial configuration and on the fixed set of rules. What happens to a cell depends on its state (live or dead) and on the states of its 8 *neighbors* (cells touching the given cell on an edge or a corner).

Here are the rules:

BIRTH: A cell that's dead at time t becomes live at time $t+1$ if exactly three of its eight neighbors were live at time t.

DEATH by overcrowding: A cell that's live at time t and has four or more of its eight neighbors live at time t will be dead at time $t+1$.

DEATH by exposure: A live cell that has only one live neighbor or none at all at time t will also be dead at time $t+1$.

In summary, a live cell stays live if it has exactly 2 or 3 live neighbors—otherwise it dies. A dead cell becomes a live one if it has exactly 3 live neighbors—otherwise it stays dead. These simple rules lead to an astonishing complexity.

A Finite Grid. For this course it is too hard to think about an "infinite" grid of cells, so your simulation should work on a fixed rectangular $M \times N$ grid of cells—with M rows and N columns. Since we want you to display your results on an ordinary terminal screen, you should choose $M = 24$ and $N = 80$, *i.e.*, a 24×80 rectangular grid. I suggest you use an array of `char`, using a blank for dead and a star (`*`) for live.

Notice that cells on the boundaries of the grid do not have a full eight neighbors, and your count of dead and live cells must stay inside the grid. (You must not have a row or column index outside the ranges from 0 to $M-1$ and from 0 to $N-1$.)

The Initial Program. Your program should use *two* $M \times N$ arrays, for the tth and $(t+1)$st generations. These arrays should be passed as parameters. You must use a separate C function `neighbors` with the row and column number as parameters that returns the number of live neighbors of a given cell. (The function `neighbors` must also handle the boundary correctly.) You must also have a function `next_gen` with the two arrays as parameters that updates from one generation to the next (using the `neighbors` function). Another function `get_config` should read a finite list of pairs of integers for the initial configuration. There will be a number of sample initial files to choose from. Finally, you need a function `disp_array` to output each new generation to the screen. The simplest way is just to write 24 lines of 80 characters each. Here is a sample outline for this program:

```
#include <stdio.h>
#include <stdlib.h>
#define M 24
#define N 80
#define MAX_GEN 40
void get_config(char b[M][N]);
void blank_array(char b[M][N]);
void disp_array(char b[M][N]);
void next_gen(char b1[M][N], char b2[M][N]);
void assign(char b1[M][N], char b2[M][N]);
int neighbors(char b[M][N], int i, int j);

void main(void)
{
        char b1[M][N], b2[M][N];
        get_config(b1);
```

```
                disp_array(b1);
                fflush(stdout);
                int dummy;
                for (dummy = 0; dummy < MAX_GEN; dummy++) {
                        next_gen(b1, b2);
                        disp_array(b2);
                        fflush(stdout);
                        sleep(1);
                        assign(b1, b2);
                }
        }
```

Note that `next_gen` takes the grid for the old generation `b1` and produces a grid `b2` for the next generation. Because the program moves from `b1` to `b2`, it is necessary to copy `b2` back to `b1`, overwriting the old contents of `b1`. This is carried out by the function `assign`. There is also a system function `sleep(1)` that puts the program to sleep for 1 second, to slow down the output. Finally, on runner it is necessary to force output with a system call `fflush(stdout)`. (One could cleverly use `next_gen(b1, b2)` followed by `next_gen(b2, b1)` in the for loop to avoid the `assign` function.) I intend for you to terminate this program by typing ctrl-C. (This way you can watch as many generations as you like. However, be sure to use a for loop as shown with a maximum number of generations, so that you won't produce a huge amount of output by mistake.)

A Better Program. During initial debugging you should try to get your program to work just using this simple form of output. For full credit (and a more interesting program), you should handle the writing to the screen in a more sophisticated way. One would like to just change those screen grid locations where there was a change from one generation to the next. Let's assume that you use a blank for dead cells and a star for live ones. Initially you should blank out the screen using a special C function `clear_screen`, supplied below. (This and the next function work on VT-100 terminals, that accept special sequences of characters, starting with "escape" or \033, that cause special actions by the terminal, like to blank out the screen or to move the cursor to a new location.)

```
        void clear_screen(void)
        {
                printf ("\033[H\033[J");
        }
```

In going from the tth to the $(t+1)$st generation, you should only redo those positions that changed, either from a blank to a star or vice-versa. Below is another C function `move_cursor`

```
        void move_cursor(int x, int y)
        {
                printf ("\033[%d;%dH", y, x);
        }
```

that will move the cursor to position (x, y) on the screen. (This moves to the column x and the row y, where $1 \leq x \leq 80$ and $1 \leq y \leq 24$. To fit C conventions, you will need the call `move_cursor(i+1, j+1);`.) Assuming that this grid position changed with the new generation, you can then write the appropriate character (blank or star) to the screen. I suggest writing a function `disp_diff` with the two grids as parameters that will make changes based on the differences from one generation to the next. This if a grid positions changes, you should make the change by moving the cursor and writing a blank or a star. Continue in this way for each changed position, first moving and then writing.

I will give a number of possible interesting initial configurations. You should use at least one of them in a final run of your program. Here are three interesting ones:

```
lifedata.abomb    lifedata.glider     lifedata.monster
      ***               *                   *
       *                 *                ***
       *               ***                 *
       *
```