Assignment 3
The Game of Life

Prof. Darrell Long
CSE 13S – Winter 2021

Due: January 31$^{\text{th}}$ at 11:59 pm

1 Introduction

The last level of metaphor in the Alice books is this: that life, viewed rationally and without illusion, appears to be a nonsense tale told by an idiot mathematician.

—Martin Gardner

John Horton Conway was an English mathematician active in the theory of finite groups, knot theory, number theory, combinatorial game theory, and coding theory. He also made contributions to many branches of recreational mathematics, most notably in popular culture for the invention of the cellular automaton called the *Game of Life*.

Born and raised in Liverpool (home of the *Beatles*), Conway spent the first half of his career at the University of Cambridge before moving to the United States, where he held the John von Neumann Chair at Princeton University for the rest of his career. On 11 April 2020, at 82, he died of complications from COVID-19.

Research on the Go end-game by Conway led to the original definition and construction of the surreal numbers. Conway's construction was introduced in Donald Knuth's 1974 book *Surreal Numbers: How Two Ex-Students Turned on to Pure Mathematics and Found Total Happiness*. In his book, which takes the form of a dialogue, Knuth coined the term *Surreal Numbers* for what Conway had called simply *numbers*. Conway later adopted Knuth's term, and used surreals for analyzing games in his 1976 book *On Numbers and Games*.

John Horton Conway

Conway first conceived The Game of Life in 1970 to describe how life can evolve from an initial state. The concept builds on ideas that trace back to John von Neumann[1] who was a pioneer of early computing and from whom we get the von Neumann architecture that we still use today. Conways game involves a two-dimensional grid in which each square cell interacts with its neighbors according to a simple set of rules. Over time, these simple interactions give rise to surprising complexity.

Life, is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. It is *Turing complete* and can simulate a universal constructor or any other Turing machine. Turing completeness means that

[1] J. von Neumann and A. W. Burks, *Theory of self-reproducing automata.* Urbana, University of Illinois Press, 1960.

for those who have yet to study computability theory, the Game of Life can run *any program* that can be written for a computer.

2 The Rules

Our noblest hopes grow teeth and pursue us like tigers.

—John Gardner, *In the Suicide Mountains*

The Game of Life should be played on a potentially *infinite*, two-dimensional (2-D) grid of cells that represents a *universe*. Each cell has two possible states: it is either *dead* or *alive*. The game progresses through *generations*, what some might call "steps in time." There are only three rules that determine the state of the universe after each generation:

1. Any *live* cell with two or three live neighbors *survives*.
2. Any *dead* cell with exactly three live neighbors *becomes a live cell*.
3. All other cells die, either due to loneliness or overcrowding.

Your task for this assignment is to implement the Game of Life in **C**. The first hurdle is creating an abstraction for the universe in which the game is played.

3 The Universe

Talking, talking. Spinning a web of words, pale walls of dreams, between myself and all I see.

—John Gardner, *Grendel*

You will now write your first *abstract data type*, commonly referred to as an **ADT**. This ADT will provide the abstraction for a universe, a *finite* 2-D grid of cells. Why not an infinite grid? Because computers work in *finite memory*. The following subsections will walk you through the list of constructor, destructor, accessor, and manipulator functions required for your ADT. You will be supplied `universe.h` (in the Resources repository), the header file for the `Universe` ADT and you **may not** modify it.

3.1 `Universe`

The universe will be abstracted as a `struct` called `Universe`. We will use a `typedef` to construct a new type, which you should treat as opaque — which means that you must pretend that you cannot manipulate it directly. **C**, unlike some more modern languages, *does not enforce opacity*.

Here, `universe.h` *declares* the new type and `universe.c` *defines* its concrete implementation. Once again, **you cannot modify universe.h**

```c
struct Universe {
    int rows;
    int cols;
    bool **grid;
    bool toroidal;
};
```

An instance of a `Universe` must contain the following fields: `rows`, `cols`, and a 2-D boolean grid, `grid`. Since there are two states: *alive* and *dead*, then a natural choice for representing the states is the `bool` type. A cell with the value `false` in `grid` indicates that the cell is dead; the value `true` indicates that the cell is alive. It is also possible for a `Universe` to be *toroidal*, which is indicated by the `toroidal` field. This `struct` definition must be placed in the file `universe.c`.

3.2 `Universe *uv_create(int rows, int cols, bool toroidal)`

This is the constructor function that creates a `Universe`. The first two parameters it accepts are the number of `rows` and `cols`, indicating the dimensions of the underlying *boolean* grid. The last parameter `toroidal` is a boolean. If the value of `toroidal` is `true`, then the universe is *toroidal*. The *return type* of this function is of type `Universe *`, meaning the function should return a *pointer* to a `Universe`. You will be using the `calloc()` function from `<stdlib.h>` to dynamically allocate memory. For more information on `calloc()`, read `man calloc`. Here is an example of allocating a matrix of `int`s:

```c
int **matrix = (int **) calloc(rows, sizeof(int *));
for (int r = 0; r < rows; r += 1) {
    matrix[r] = (int *) calloc(cols, sizeof(int));
}
```

Note that we first allocate a column of pointers to rows, and then we allocate the actual rows.

For an example of a complete ADT constructor function, see the section on ADTs in the course coding standards.

3.3 `void uv_delete(Universe *u)`

This is the destructor function for a `Universe`. Simply put, it frees any memory allocated for a `Universe` by the constructor function. Unlike Java and Python, **C** is *not* garbage collected. Not freeing allocated memory by the end a program results in a *memory leak*. Your programs should be free of memory leaks. In the case of multilevel data structures such as a `Universe`, we must free the inside first. Imagine getting rid of a water bucket: you should dump out the water before scrapping the bucket. Use `valgrind` to check for memory leaks – the graders will too!

3.4 `int uv_rows(Universe *u)`

Since we will be using `typedef` to create *opaque* data types, we need functions to access fields of a data type. These functions are called *accessor* functions. An opaque data type means that users do not need to know its implementation outside of the implementation itself. This means that it is incorrect to have `u->rows` outside of `universe.c` as it violates opacity. This function returns the number of rows in the specified `Universe` (it is possible, but only inside `universe.c`).

3.5 `int uv_cols(Universe *u)`

Like `uv_rows()`, this function is an accessor function and returns the number of columns in the specified `Universe`.

3.6 `void uv_live_cell(Universe *u, int r, int c)`

The need for *manipulator* functions follows the rationale behind the need for accessor functions: we need some way to alter fields of a data type. This function simply marks the cell at row `r` and column `c` as live. If the specified row and column lie outside the bounds of the universe, nothing changes. Since we are using *bool*, we assume that *true* means live and *false* means dead.

3.7 `void uv_dead_cell(Universe *u, int r, int c)`

This function marks the cell at row `r` and column `c` as *dead*. Like in `uv_live_cell()`, if the row and column are out-of-bounds, nothing is changed.

3.8 `bool uv_get_cell(Universe *u, int r, int c)`

This function returns the value of the cell at row `r` and column `c`. If the row and column are out-of-bounds, `false` is returned. Again, *true* means the cell is live.

3.9 `bool uv_populate(Universe *u, FILE *infile)`

This function will populate the `Universe` with row-column pairs read in from `infile`. This function will require the use of <stdio.h> since `infile` is a `FILE *` (`FILE` is defined in the <stdio.h> library). The necessary *include* will be supplied in `universe.h` for you. A valid input file that could be fed into your program would be:

```
36 65
2 32
3 30
```

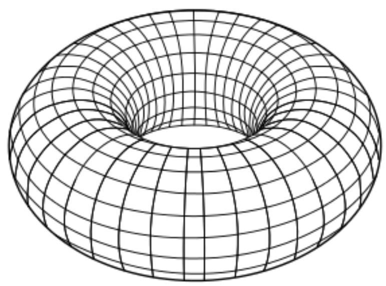
The first line of a valid input file for your program consists of the number of rows followed by the number of columns of the universe. Each subsequent line is the row and column of a live cell. You will want a loop that makes calls to `fscanf()` to read in all the row-column pairs in order to populate the universe. If a pair lies beyond the dimensions of the universe, and return `false` to indicate that the universe failed to be populated. Return `true` if the universe is successfully populated. Failure to populate the universe should cause the game to fail with an error message.

Do not print in functions if you can avoid it – unless that is their job!

3.10 `int uv_census(Universe *u, int r, int c)`

When we consider our universe, a flat universe is simply a grid. Ideally in the Game of Life, that extends to infinity in the $\pm x$ and $\pm y$ directions. Since we cannot have an infinite grid, we are left with a choice: we can either, as the Flat Earthers believe, have things fall off the edge or come up with some other solution. Our solution will be to treat the finite grid as a flat Earth or as a *torus*. An example of a torus can be constructed by taking a rectangular strip of flexible material, for example, a rubber sheet, and joining the top edge to the bottom edge, and the left edge to the right edge, without any half-twists.

It requires some reflection, but if you think about you will see that a torus is any topological space that is topologically equivalent to a torus. A coffee cup and a doughnut are both topological tori.


In the realm of real numbers, the parametric equation of a ring torus is:

$$
\begin{aligned}
x(\theta,\phi) &= (R + r\cos\theta)\cos\phi \\
y(\theta,\phi) &= (R + r\cos\theta)\sin\phi \\
z(\theta,\phi) &= r\sin\theta
\end{aligned}
$$

where θ, and ϕ are angles which make a full circle, so that their values start and end at the same point, R is the distance from the center of the tube to the center of the torus, and r is the radius of the tube. Our torus is finite and discrete: in other words, assuming a square grid $G_{n,n}$ indexed from $0,\ldots,n-1$ (because we are Computer Scientists), the right edge of the grid $G_{i,n-1}$ is connected (succeeded by) $G_{i,0}$, and the bottom edge of the grid $G_{n-1,i}$ is connected to $G_{0,i}$. In other words, *successor* and *predecessor* are calculated using *modular arithmetic* if the universe is a torus, which you have seen these functions before.

This function returns the number of *live neighbors* adjacent to the cell at row `r` and column `c`. If the universe is flat, or non-toroidal, then you should only consider the *valid* neighbors for the count. If the universe is toroidal, then all neighbors are valid: you simply wrap to the other side. Tip: you should calculate the row and column for each neighbor and apply modular arithmetic if the universe is toroidal.

3.11 `void uv_print(Universe *u, FILE *outfile)`

This functions prints out the universe to `outfile`. A live cell is indicated with the character '`o`' (a lower-case O) and a dead cell is indicated with the character '`.`' (a period). You will want to use either `fputc()` or `fprintf()` to print to the specified `outfile`. Since you cannot print a torus, you will always print out the flattened universe.

4 Controlling the Cursor

An orphan's curse would drag to hell
A spirit from on high;
But oh! more horrible than that
Is the curse in a dead man's eye!
Seven days, seven nights, I saw that curse,
And yet I could not die.

—Samuel Taylor Coleridge, *The Rime of the Ancient Mariner*

`ncurses` is a programming library used to develop text-based user interfaces. It is used in vi (vim, neovim), emacs and many other programs. You will use this library to display the state of the universe after each evolution. Here is some code that showcases moving an '`o`' horizontally across the screen:

Short `ncurses` example.

```c
#include <ncurses.h>
#include <unistd.h> // For usleep().

#define ROW 0
#define DELAY 50000

int main(void) {
    initscr(); // Initialize the screen.
    curs_set(FALSE); // Hide the cursor.
    for (int col = 0; col < 40; col += 1) {
        clear(); // Clear the window.
        mvprintw(ROW, col, "o"); // Displays "o".
        refresh(); // Refresh the window.
        usleep(DELAY); // Sleep for 50000 microseconds.
    }
    endwin(); // Close the screen.
    return 0;
}
```

To test this code snippet out, place it in `example.c` and compile it with the command:

```
$ clang -o example example.c -lncurses
```

The `-lncurses` at the end serves to *link* the `ncurses` library. Linked libraries should always be linked at the end. Why? UNIX links binaries from left to right. When an *undefined* reference is encountered, it is expected to be defined *later*. An *unreferenced* item is ignored, so if you list the library first, it will be unreferenced and is thus ignored.

5 Command-line Options

A few dud universes can really clutter up your basement.

Neal Stephenson, *In the Beginning. . . Was the Command Line*

Your program should accept the following command-line options:

- `-t` : Specify that the Game of Life is to be played on a *toroidal* universe.
- `-s` : Silence `ncurses`. Enabling this option means that nothing should be displayed by `ncurses`.
- `-n` *generations* : Specify the number of generations that the universe goes through. The default number of generations is 100.
- `-i` *input* : Specify the input file to read in order to populate the universe. By default the input should be `stdin`.
- `-o` *output* : Specify the output file to print the final state of the universe to. By default the output should be `stdout`.

6 Specifics

Jehosaphat the mongrel cat
Jumped off the roof today
Some would say he fell but I could tell
He did himself away

—John Prine, *Living in the Future*

Here are the specifics for your assignment implementation.

1. Parse the command-line options by looping calls to `getopt()`. This should be similar to what you did in assignment 2.
2. Use an initial call to `fscanf()` to read the number of rows and columns of the universe you will be populating from the specified input.
3. Create *two* universes using the dimensions that were obtained using `fscanf()`. Mark the universes toroidal if the `-t` option was specified. We will refer to these universes as universe A and universe B.

4. Populate universe A using `uv_populate()` with the remainder of the input.

5. Setup the `ncurses` screen, as show by the example in §4.

6. For each generation up to the set number of generations:

   (a) If `ncurses` isn't silenced by the `-s` option, clear the screen, display universe A, refresh the screen, then sleep for 50000 microseconds.

   (b) Perform one generation. This means taking a census of each cell in universe A and either setting or clearing the corresponding cell in universe B, based off the 3 rules discussed in §2.

   (c) Swap the universes. Think of universe A as the current state of the universe and universe B as the next state of the universe. To update the universe then, we simply have to swap A and B. Hint: swapping *pointers* is much like swapping integers.

7. Close the screen with `endwin()`.

8. Output universe A to the specified file using `uv_print()`. This is what you will be graded on. We will know if you properly evolved your universe for the set number of generations by comparing your output to that of the supplied program.

7 Deliverables

It is such a sweet temptation
It gives such grief relief
It is such a false sensation
How come that's so hard to believe?

—Ray Wylie Hubbard, *Loco Gringo's Lament*

You will need to turn in:

1. `life.c`: This file will contain your implementation of the Game of Life.

2. `universe.c` should contain the implementation of the `Universe` ADT.

3. `universe.h` should contain the definitions. You will be supplied `universe.h` and *may not* modify it.

4. `Makefile`: This is a file that will allow the grader to type `make` to build and run your program. Your `Makefile` *must* support the following targets:

   - `all`: Builds your program.
   - `clean`: Removes any compiler-generated files.
   - `format`: Formats any source or header files with the course-supplied `clang-format` file.

5. `README.md`: This must be in *Markdown*. This must describe how to build and run your program. It should also describe the command-line options that your program accepts.

6. `DESIGN.pdf`: This *must* be in PDF. The design document should describe the purpose of your program and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. C code is **not** considered pseudocode. You *must* push `DESIGN.pdf` **before** you push *any* code.

We will provide an example of both `README.md` and `DESIGN.pdf` for your reference.

8 Submission

> *People say that if you're still angry at 52, you're not an angry young man, just a grumpy old git.*
>
> Paul Weller

To submit your assignment through `git`, refer to the steps shown in `asgn0` Remember: *add, commit,* and *push*! Your assignment is turned in *only* after you have pushed. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

We will provide you with an *Ubuntu 20.04* binary of our implementation. Your code must produce *exactly* the same output for *all* inputs in order to receive full credit. You can use the Linux `diff` command to compare results.

9 Supplemental Readings

> *The more that you read, the more things you will know. The more that you learn, the more places you'll go.*
>
> —Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 3 §3.4–3.7
  - Chapter 4 §4.1 & 4.2 & 4.5
  - Chapter 5 §5.1–5.2 & 5.10
  - Chapter 6 §6.1–6.5 & 6.7
  - Chapter 7 §7.2 & 7.5
  - Appendix B §B5
- *The Kollected Kode Vicious* by George V. Neville-Neil
  - Chapter 1 §1.3, & 1.5-1.7
- Martin Martin, *Mathematical Games — The Fantastic Combinations of John Conway's New Solitaire Game 'Life'*, Scientific American (223): 120–123 (October 1970).