Marian Zlateva

mzlateva@ucsc.edu

1/31/2021

CSE13s Fall 2020
Assignment 3: The Game of Life

Description:

In this lab, I wrote up a simple algorithm to generate subsequent generations of a grid according to the rules of the "The Game of Life." The program starts with reading the input flags and the input file either from the stdin or a specified file. After that, it generates a grid and calculates the next generations, a specified number of times. The inputs to this program are a series of flags:

Flags:

-t	Specify that the Game of Life is to be played on a toroidal universe.
-s	Silence ncurses. (don't display the animated output and jump straight to the final output)
-n:	Specify the number of generations that the universe goes through. The default number of generations is 100.
-i:	Specify the input file to read in order to populate the universe. By default the input should be stdin.
-o:	Specify the output file to print the final state of the universe to. By default the output should be stdout.

Files:

- Life.c
  - Contains main function
  - Reads the flags
  - Play's the "game"
- Universe.c
  - Contains:
    - the struct definition for Universe.c
    - getter functions for variables inside Universe instance
    - functions to assist the game running algorithm in Life.c
- Universe.h
  - Header file for Universe.c

Functions:

Life.c

int main()

- Reads the flags
- prints output
- calls `void calculateNextGen()`

- Read program inputs (as specified in Flags on pg 1)
- Error handling for invalid input file (infile)
    - Output file (outfile) doesn’t need error handling since a new file gets created if specified file doesn’t exist
- Scan the first line of the file for the specified rows and cols in the universe
- Create a Universe pointer with the specified rows and columns as well as the user’s specification for toroital
- If -s flag was not given
    - initialize the screen
    - loop until you reach the the genCount (could be specified by -n, else its 100)
        - call `void calculateNextGen()`
        - clear the screen
        - in each slot print “o” if the slot is alive, “ ” if it is dead
        - refresh the screen
        - sleep for 5000 microseconds
    - close the screen
- If -s flag was given
    - loop until you reach the the genCount
        - calculate next gen
        - print final generation using: `void uv_print()`
    - delete Universe * using: `void uv_delete()`
    - close infile
    - close outfile

void calculateNextGen(Universe **pointerToA, Universe **pointerToB)

- calculates the next generation of universe A
- new gen inside Universe *a
- old gen inside Universe *b
- pointerToA - the pointer of the point to Universe A
- pointerToB - the pointer of the point to Universe B

- loop through every slot in 'a' and write next gen in 'b'
    - Any live cell with two or three live neighbors survives.
    - Any dead cell with exactly three live neighbors becomes a live cell.
    - All other cells die, either due to loneliness or overcrowding.
- swap pointers a and b

void malformedInput()

- prints error and exits the program
- should be used for invalid inputs

- print an error message: “Malformed input.” in the terminal
- exit the program

Universe.c

struct Universe *uv_create(int rows, int cols, bool toroidal)

- creates a pointer to a Universe
- params
    - rows - the amount of rows in the Universe's grid
    - cols - the amount of cols in the Universe's grid
    - toroidal - value of toroidal in the universe
- return - pointer to the created Universe

```
➔ Universe *u
        ➝ use calloc to allocate space for Universe
➔ u->rows
        ➝ = rows
➔ u->cols
        ➝ = cols
➔ u->grid
        ➝ empty 2D array of size [rows][cols]
        ➝ use calloc to allocate space for rows and row container
➔ u->toroidal
        ➝ = toroidal
```

void uv_delete(Universe *u)

- frees the memory from a universe
- params
    - u - the pointer to the Universe

```
➔ free every row of memory in the grid
➔ free the grid's pointer and the Universe
```

int uv_rows(Universe *u)

- Getter for a Universe’s rows
- params
    - u - the pointer to the Universe
- return - the number of rows in the Universe

```
➔ return Universe->rows
```

int uv_cols(Universe *u)

- Getter for a Universe’s columns
- pointers
    - u - the pointer to the Universe
- return - the number of rows in the Universe

```
➔ return Universe->cols
```

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

- Sets a cell to alive (true)
- params
    - u - the pointer to the Universe
    - r - row of the cell
    - c - column of the cell

```
➔ set grid[r][c] of u = true
```

void uv_dead_cell(Universe *u, int r, int c){

- Sets a cell in a Universe to dead (false)
    - u - the pointer to the Universe
    - r - row of the cell
    - c - column of the cell

```
➔ set grid[r][c] of u = false
```

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

- getter for cell status (if out of bounds -> false)
- params
    - u - the pointer to the Universe
    - r - row of the cell
    - c - column of the cell
- return - status of the cell (alive/true or dead/false)

```
➔ if cell is out of bounds, cell can be read as dead
    → return false
➔ else, return value inside passed in cell
```

bool uv_populate(Universe *u, FILE *infile)

- populates a universe according to the specifications of the file
- params
    - u - the pointer to the Universe
    - infile - the file that contains the coordinates of the alive cells
- return - false if error / true if no error

```
➔ while true loop
    → stop loop if we reach EOF
    → if scan doesn't return 2, then there was an error so return false
    → if scanned values are out of bounds, return false
```

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

- calculates the amount of live neighbors to a cell in a toroidal or non-toroidal universe
- params
    - u - the pointer to the Universe
    - r - row of the cell
    - c - column of the cell
- return - amount of live neighbors

```
int aliveNeighbors = 0
loop through all neighbors with a nested for loop:
    for(int rn =-1; rn <= 1; rn ++ )
        for(int cn =-1; cn <= 1; cn++ )
            get row and col of neighbor by:
            rowOfNeighbor = r + nr
            colOfNeighbor = c + nc
            if toroidal, make out of bounds values loop around
                rowOfNeighbor = (rowOfNeighbor + u->rows) % u->rows
                colOfNeighbor = (colOfNeighbor + u->cols) % u->cols
            if cell is alive
                increment aliveNeighbors
```

void uv_print(Universe *u, FILE *outfile)

- prints the universe into a file
- params
    - u - the pointer to the Universe
    - infile - the file that you want to universe printed in

```
loop through all slots of the Universe's grid
    if the cell is alive, print "o" in the file
    if the cell is dead, print "." in the file
    print a newline after every row
```

Design Process

The hardest part of making the program was writing the main function. I didn’t have any idea how to use getopt() with inputs assigned to flags. However, after looking through the manual for getopt(), I found what I needed to do. Another hard part of this assignment was coding the main function. Since I didn’t write pseudo code for most of the main at the start, I kept running into logical errors. On the other hand, in universe.c, I was able to code with almost no logical errors since I went through the process of designing nearly all the functions before coding. I also did this for calculateNextGen() in a separate document. In my next project, I will definitely be designing the entire program before coding.

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

```
Universe *uv-create(rows,cols,toroidal){
    Universe *u = (Universe *) calloc (1 , sizeof(Universe)
```

always type cast when allocating memory for a variable

1 ↑ count (# of items), ↑ size

don't put sizeof(u) b/c thats only 8 bits b/c u is a pointer

-allocates the requested memory and returns a pointer to it

-also zeros out the memory (unlike malloc)

```
    u→rows =rows
    u→cols=cols
    u→toroidal=toroidal
    u→grid=(bool **)calloc(rows,sizeof(bool *));
    for(int r=0; r<rows ;r+=1){
        u→grid[r]= (bool *) calloc(cols,sizeof(bool));
    }
    return u;
}
```

2D Array:
{{0,0,0},
{0,0,0},
⋮
}

```
void uv_delete (Universe *u){
    for(int r=0; r<rows ;r+=1){
        free(u→grid[r]);
    }
    free(u→grid)
    free u;
    return;
}
```

frees memory

```
int uv_rows(u){
    return u→rows;
}
```

↳ you can also say (*u).rows

```
int uv_cols(u){
    return u→cols;
}
```

```
void uv-live-cell(*u,r,c){
 if r & c are in bounds ↴
    u→grid[r][c] = true;
}
```

row col

FILE * from <stdio.h>

```
bool uv-populate(Universe *u,FILE*infile){
    fscanf(infile,"%d␣%d\n",...)EOF
```

-1

int uv_census(u,1,1)

	0	1	2
0	1 false	2 true	3 false
1	4 true	5 false	6 true
2	7 true	8 false	9 true

uv_census (u,1,1) → 5

uv_census(u,0,0) → 2 or

flat ↑

[1][1]
↓
[0][0] dead
[0][1] true
[0][2] dead
[1][0] true
[1][2] true
[2][0] true
[2][1] false
[2][2] true

~~||||~~ 5 alive neighbors

[1,0]
[2,2]
[2][0]
[0][-1]
[0][2]

so we check
✓ [r] [c-1] left
✓ [r] [c+1] right
✓ [r-1][c] top
✓ [r+1][c] bottom
✓ [r-1][c-1] top left
✓ [r-1][c+1] top right
✓ [r+1][c-1] bottom left
✓ [r+1][c+1] bottom right

looks like these could be written in a for loop

```
for(i=-1; i≤1 ; i++){
    for(j=-1; j≤1 ; j++){
        if(i==0 & j==0){
            continue;
        }
        check spot at [r+i][c+j]
    }
}
```

I quickley compried this to check my work and its all good!

Quick test:

```
*u = vu-create (4,4,false)
uv-live_cell(u,0,0)
uv-live_cell(u,2,2)
uv-print(u)
uv-dead_cell (u,0,0)
uv_print (u)
```