Shawn S Hillyer

09/28/2015

CS 162 – Assignment 1 – Conway's Game of Life

Assignment 1 - Reflections

Design

Looking at the complexity of the specifications for the Game of Life, I am opting to go with a design using Object-Oriented Programming rather than straight function calls. My thought process is that I will want to validate input and perform various actions on abstract data types, like a grid representing the cells in the world.

After researching the problem and reading the assignment, I know I will need to avoid creating one "Do It All" class. I will need to abstract out the simpler pieces so that the highest-level class – probably called a Game – will be understandable just by the interfaces that I provide. Here are my notes from the brainstorming session I had:

Design brainstorm for Object Oriented design of Game of Life

Highest level of abstraction: **GameOfLife** object. This will represent the game's most public interfaces. A user of this class will be able to create the game world, load a pattern, and advance the game a generation. Also, they should be able to set the starting location for the pattern. I want to abstract out the grid of cells in another class. Also I want to use a Point struct or class to define the row/column of the offset so that it can be stored in the game.

-The Grid should be is own object in case I want to change the interface for that later. For now, make grid a dynamic 2d array of Cells.

-A Cell will just be an ENUM with two values, DEAD or ALIVE.

-Each element (Cell) of the Grid, which can be a 2d array, should have a DEAD or ALIVE status. I can create an ENUM that has either DEAD or ALIVE. The 2d array can be of type Cell.

-The evaluate_rules will have to decide what every cell will do and update it without overwriting the existing data – I could handle this by making a local temporary copy or maybe having a stateA and stateB type variable. I'm not sure which is more efficient? I'd like to update or copy only the values that need to change, which would be less operations than overwriting an entire array.

-The evaluation of the rules will also have to see if a Cell in the Grid is alive or dead, and then evaluate each of the rules for each of those two cases. If no rules apply, then the cell should stay alive or dead. If I have the grid iterate through every cell and check all 8 of its neighbors, I'm going to have to overcome the problem of trying to read out-of-array-bounds values from the Grid.

-Not sure best way to deal with this. Perhaps a complex set of checks to handle if the cell is in top row, then check if top left else if top right. If not not row, then check if bottom row. Then if bottom left, else if bottom right. In each of those cases, I would know which cells will die by default and just don't call the read/write updates on those cells and treat them as dead (don't count them).

- Handling the edges might be tricky. I think the impact on the pattern as it evolves will be visible if I just handle the edges by treating the "outside" of the viewable area as dead. I could hide this by making my game board bigger than the viewable grid. Say... twice as wide and twice as high should be okay? Looking at images of the glider canon, it seems like that should be enough. I'll know if I get "blowback" from the edges that isn't expected that it isn't big enough.

-Make sure that patterns fall off edge per specifications – "Glider should just go off the screen" basically.

- The game should know how many rows and cols it has.

- The only things I want to be able to set or get publically (outside of constructor) would be the starting point (offset) and the ability to load whichever pattern. Also the size can be set, but only once, upon initialization. And the ability to print to the console.

Every time the game advances a generation, or "tick", it should: print the Grid, evaluate the rules, and update the grid so that the next "tick" will print the new status and so forth.

Therefore, the GameOfLife might look like (after some brainstorming):

GameOfLife
Cell_Status: Grid* Pattern_Offset: Point Rows: int Columns: int Current_generation: int
GameOfLife(): constructor : rows, columns Advance_game(): void, no params To_Console(): void, no parms Set_pattern(Point[]): void, accepts array of Points – or a Pattern? Set_pattern_offset(Point): Evaluate_rules(): void, no parms

On to the **Grid**. I've already discussed that the Grid should be a 2d array of Cells (an enum). It should have rows and columns, which I can have the constructor for the class pass to its composite member Grids.

I should have an initialize method that will update every cell in the Grid to DEAD and call it in the constructor. Make it public in case I want to call it from the GameOfLife object to clear the board between rounds.

I need a way to get the cell status for any given cell so my rules check has access to the 9 cells it needs to look at for every rule evaluation.

Grid
Cells[][] : 2d array of type Cell (enum) Rows: int Columns: int
Grid(): Constructor. Input rows and columns ~Grid(): Deconstructor to free the 2d array memory Initialize(): void, no parameters. Sets the entire board to DEAD. Called by constructor or upon game reset? To_console(): void, no parms.

The idea of a **Pattern** class seems like the best way to allow me to encapsulate the coordinates of every point in an indeterminate list of points that make up a Pattern. Some patterns are 1, 2, 3, etc. Points. So set it up as a Dynamic array of Points. Store the number of points in the object. Deconstructor to delete. The constructor should take a list of Points and the number of points, and probably a getter for each (though no setter – only want to allow updating data members upon creation. If a new pattern is needed, instantiate another one, don't revise the existing one).

Point class also super simple. Will need to be a class, not a struct, just because we don't want to allow any x or y coordinates that are negative (these are points in an array, not in a Cartesian Coordinate system)

Testing

Although I designed top-down, I will write the program bottom-up. I will test each class and build those that are composed inside of others first, and test each method in kind. I typically build an automatic test that just calls a series of pre-determined statements to test each class as I go to ensure it behaves properly. During this testing, I often make my functions display cout for debugging.

First I made will make header file for a Cell enum and test it. I will ensure that I can do some simple things like comparison operator and assignment. I should be able to assign a Cell a value of DEAD or ALIVE, and anything else should give a compiler error.
Calls I should be able to make:

```
Cell dead_cell = DEAD;
Cell alive_cell = ALIVE;
Cell error_cell = alphabet_soup; // compiler error
If (dead_cell == ALIVE)
  // this shouldn't print!
  Std::cout << "Oh noes";
… etc.
```

Testing the Point class next - it is the lowest-level and simplest class. I should be able to only have positive values for any Point, since this is a Point in a 2d array that represents a real index, NOT a Cartesian coordinate system. I should have getters and setters and make this a class, not a struct, because I need to ensure it validates input. The default constructor should be tested and set a Point with 0,0, and it should call the setters (which contain the validation). My test should validate that I can set x and y values, invalid (<0) input is set to 0, and make sure I can create an array of Points without any

problems (I'll need this for my Pattern class). I can test the getters by creating an array of Points and printing their x and y values by calling the get_xxx() methods.

For the Pattern, I will do a similar test. I need a constructor that takes an array of Points and the # of points, so I can test that first. I can use the Points test earlier to populate the array of Points in the Pattern. Once that's done, I should test that my deconstructor deallocates the memory. Finally test that I can retrieve a Point at a certain index in the list by testing a getter function, and get the number of points in the list since it's a data member I may need to access in the rest of my program.

The grid itself will be a 2d array of Cells. It also needs to know the rows and columns quantity. So I'll test a constructor and appropriate setters to ensure that it will properly set up a grid without array access errors, allocating the 2d array and setting the rows and columns to default values if a negative value is passed in (or no value). I need a way to see the elements in the array in order to verify what's in it, so I'll verify the print_to_console method and use it to validate the data matches what I expect. Specifically, I expect my grid to have all dead cells to start.

I should be able to report the status of a cell or update the status of a cell in the array, so I will test a get_ and set_cell_status method by testing a few values.

Finally, putting it all together. Once I have the classes in a somewhat working order, I need to test the GameOfLife client itself. SO to do this, I'll need to see if the constructors will set up a Grid that I can print, advance the game, and load a pattern that I set up. Once I'm satisfied that it's actually reading a patterns initial state, THEN I will start running the rules in small increments for about 5 seconds per screen to take notes of what's happening.

I'll know my rules evaluate correctly when I can test a few stable patterns. I have animations of the glider pattern, the Gosper glider cannon, and a simple oscillator (3 alive cells stacked in a straight column) that are easy to watch for problems. I'll know the rules evaluate well when I can run them for about 1000 iterations and have no problems.

Testing Results

Okay, most of the testing went well, but two things happened. I decided to build the Game as a current_state and next_state, and have a toggle_current_state method that updates. For some reason, the game toggles between the initial state, and a secondary state, then back again. It does this forever. I ended up sleeping on it and realized the next morning that the next_state was not completely the same as the current_state before the rules were evaluated.

In other words, I just kept swapping the two states back and forth and never fully captured the changes.

Another problem I'm seeing is that my method of checking all 8 neighbors if they are alive or dead is not evaluating all cases properly, and I'm getting a lot of off-by-one errors in my array reads/writes. The game board prints fine blank okay, but when I try to evaluate_rules() it just crashes. This is because the first cell tries to check -1, -1, which are invalid array accessors. [See the next heading to see how I overcame this].

I'm also having a hard time loading in Patterns. If I make a list of Points and store it in a Patten right away, the Pattern loads. Right now the Pattern I'm trying is just 3 living cells in a row. But I tried making

an “assembler” function (outside of any objects) that constructs a Pattern for me and returns it and assigns it to a local variable. I’m getting error messages on that, too. Which is bad, because I want my construction functions independent of the main game logic. I can’t have all of these hard-coded patterns in the middle of my logic loops!

The good news is that the Cell enum works perfectly. I included Cell.hpp in most of the other classes and tested it in main and it’s fine.

Also, I tested Point by calling a series of an inner and outer loop, going from -10 to +10 for each, passing in those variables to the constructor. Then I stored that in an array as I looped and reiterated and printed the get_x_ and get_y_coord() results to the screen. Also, I used the debugger to step through the code and watch (in Visual Studio) the variables go in and come out as 0,0 afterwards for the invalid ones.

The Grid works fine as well, although the destructor was having problems with deleting properly. I had the columns and rows swapped in my mind – I forgot I’d have to free the “inner” loop first in the deconstructor.

Once I sort out these problems I’ll take down some notes on how I fixed these and subsequent problems.

For now, I know that the rules are NOT evaluating right. I tested a simple oscillating pattern and it goes from:

….X…

….X…

….X…

To:

……. Nothing

And then back again.

Challenges and Solutions During Implementation

The first MAJOR problem I ran into was handling the edge cases and actually just ensuring the rules worked. The method I first came up with in my design was to just handle all of the cases by manually checking if the row/cell I was about to read ALIVE or DEAD from was outside of bounds. This actually ended up being about 4 pages of conditional checks (I printed it out while at work, where I wrote it). Since I didn't have a compiler, I decidd to come up with a more elegant solution.

When I have a really bad method in a class, I know its time to evaluate if it's in the write class at all. Also, is the method trying to do to much?The answer is that the rule checking was in the right place, but that I figured out that each Cell in the Grid would probably be able to more easily just count its own neighbors. I decided the rule method should just ask each cell in the grid, "How many neighbors do you

have?" This allowed me to abstract out the problem of counting neighbors and isolate it to a method in a more appropriate place.

The method, which I called count_neighbors, ended up in the Grid class. When passed a valid index from the GameOfLife, I decided it should query its neighbor cells.

I further abstracted the question of whether any given cell is_alive or !is_alive by putting that in another method of Grid. When it receives a row and column to check, it handles the arrays out of bounds problem elegantly. It simply says returns false if the row or column is negative. All it needs is a coordinate pair and it can determine if its alive or not, and its caller just needs to know who it is, so it can ask everybody around it “are you alive?!”. And every cell that is queried just asks itself, if *where* I am is *outside* of the defined Grid's valid index ranges, then I'm dead. And I report that. Otherwise, I will tell you if I'm an alive or dead cell by just looking at myself.

This ended up working really well, in fact, and reduced the code to two really small methods. IT also kept my run_rules() method from turning into a giant frankenstein. Less code smell!

The way my game client loads the patterns required the creation of a copy-constructor. I decided to create hard-coded methods that create the Pattern objects which I load in. The downside of this was that the method could create it localy in a function and pass it back, but in so doing, the deconstructor is called in the process. This deletes the array of Points inside the Pattern right after the pointer to this array has been assigned in the copy one level below in the function call stack.

The solution was just to implement an overloaded copy constructor, but this was the first time I'd actually worked myself into this kind of problem in my design, so it took a lot of time to figure out the problem. I have written them quite a few times (I have already done every Programming Challenge in the textbook). Once I realized that making the patterns locally inside the function worked fine, and putting identical code into its own standalone function and returning it was the source of the crash, I isolated the problem. After google searching the error, somebody said about this specific error to another that they were "probably trying to delete the same memory twice." IN this case, I think the memory was trying to be deleted twice (once the local copy fell out of scope, the destructor was called. Then the game function that called it dropped out of scope and it lost the data. Either that or the local copy just knew that its’ member variable just got the axe right as it was being instantiated? Not entirely sure…).

I had issues trying to make the “clear” and “sleep” commands cross-platform – I decided to cut my losses and just port my code to unix-only, using usleep instead of sleep because 1 second iterations are super boring to watch.

Oh, and I ended up making the next_state copy the state from current_state during every run_rules() call. There’s probably a way to use less overhead, but there’s no apparent memory leak (I ran the cannon for 100k iterations the other night) and its uber fast on FLIP to check these cells. No sense fretting the algorithmic complexity… leave good enough alone!

And so I shall. I left a few commented-out bits for loading a Pattern that I just don’t feel the need to implement, but it shows my thought process for some file in/out to make the Patterns easy to input. It would avoid having to hard-code any more Patterns into the program. I also left in the current_generation private member of GameOfLife class, even though I don’t think I access or use it. If I have time between writing this and the due date, I’ll revisit these two bits.