Gregory Mitchell
Oregon State University
CS165 Accelerated Introduction to Computer Science with C++
Week 8 Assignment Report
November 2014

Contents

Chapter 1

Exercises

1.1 logger.cpp

1.1.1 Understanding of logger.cpp

(5) Write a program to log visitors to a small building. You don’t need to keep track of the time. They just need to know who is in the building in case they need to contact someone. They need to be able to enter a name (yes realistically it would also be a room # but we’ll keep it simple), check to see if a name is entered, and remove a name. There is no maximum on the number of people who might be in the building. Use a vector to store the names.

File must be called: logger.cpp

Hint: Use short names, or even just letters for testing. No need to debug if you just mistyped a name Note: for additional (optional) practice, write a class that represents an individual (name, security rating, current room), a class that represents a room (name, security level to enter, current occupants), a class that represents a building (name, list of rooms, list of occupants), and some member functions for trying to enter a room, for checking a room or the building for occupants, and removing someone from a room... this could even be the start of a game or other fun or helpful program.

1.1.2 Design of logger.cpp

1.1.3 Testing of logger.cpp

Expected Results:

1. Prints menu asking user to sign in, show who is logged in or sign out.
2. Log in function adds entered name to vector
3. Log out function removes entered name from vector
4. Show Who is signed in shows full list of logged in names

1.1.4 Implementation of logger.cpp

Please see the attached file **logger.cpp**.

1.1.5 Reflections on logger.cpp

1.1.5.1 What was learned from logger.cpp

This exercise shows how to add and remove strings from a vector, as well as iterate over it.

1.1.5.2 Reflections on Initial Design of logger.cpp

The initial design was adequate and did not need to be altered.

1.1.5.3 Reflections on the Testing of logger.cpp

All tests worked as expected. No surprises here.

1.1.5.4 Reflections on the Implementation of logger.cpp

There were no problems implementing this exercise.

1.1.5.5 Reflections on the Techniques of logger.cpp

As I understood it, this exercise is simply a further use of vectors. No outside sources were necessary.

1.2 quartiles.cpp

1.2.1 Understanding of quartiles.cpp

For some reason, I had a hard time understanding this exercise. I'm not sure why. The problem statement is as follows. Eventually I understood it.

```
(5) Programming Project 12.3 from Absolute C++ (p562 in the 5e book),

File must be called: quartiles.cpp

\a. Compute the median of a data file. The median is the number that
has the same number of data elements greater than the number as there
are less than the number. For purposes of this problem, you are to
assume that the data is sorted (that is, is in increasing order). The
median is the middle element of the file if there are an odd number of
elements, or is the average of the two middle elements if the file has
an even number of elements. You will need to open the file, count the
members, close the file and calculate the location of the middle of
the file, open the file again (recall the \start over" discussion at
the beginning of this chapter), count up to the file entries you need,
and calculate the middle.

b. For a sorted file, a quartile is one of three numbers: The first
has one-fourth the data values less than or equal to it, one-fourth
the data values between the first and second numbers (up to and
including the second number), one-fourth the data points between the
second and the third (up to and including the third number), and
one-fourth above the third quartile. Find the three quartiles for the
data file you used for part a. Note that \one-fourth" means as close
```

```
to one-fourth as possible. Hint: You should recognize that having done
part a you have one-third of your job done. (You have the
second quartile already.) You also should recognize that you have done
almost all the work toward finding the other two quartiles as well."
```

1.2.2 Design of quartiles.cpp

I used the following pseudocode as a basic guideline. It was altered quite a lot throughout the implementation however as my understanding of the problem was weak in the beginning.

Pseudocode

```

fstream fin; // file io stream

// if (argc > 2)
// use file from commandline

// else
// open file
// create random vector
// sort vector
// store vector elements to file


// -
// find median (2nd quartile) of file
//

// from http://en.wikipedia.org/wiki/Quartile
/*
  Use the median to divide the ordered data set into two halves. If
  the median is a datum (as opposed to being the mean of the middle
  two data), include the median in both halves.

  The lower quartile value is the median of the lower half of the
  data. The upper quartile value is the median of the upper half of
  the data.
*/
```

1.2.3 Testing of quartiles.cpp

I used two test cases from http://en.wikipedia.org/wiki/Quartile which were very helpful to understand this exercise:

The program tries to read a filename from the commandline to use, but defaults to the following test file:

```
6 7 15 36 39 40 41 42 43 47 49
```

The results are shown as follows:

```
argc: 1
```

```
argv[0]: ./quartiles
argv[1]: vector (11):6, 7, 15, 36, 39, 40, 41, 42, 43, 47, 49,
h1 (6):6, 7, 15, 36, 39, 40,
h2 (6):40, 41, 42, 43, 47, 49,
q1 (3):6, 7, 15,
q2 (3):36, 39, 40,
q3 (3):40, 41, 42,
q4 (3):43, 47, 49,

quartile medians:
=================
m1: 25.50
m2: 40.00
m3: 42.50
```

This agrees with one of the three given values shown on wikipedia.

1.2.4 Implementation of quartiles.cpp

Please refer to the attached file **quartiles.cpp**.

1.2.5 Reflections on quartiles.cpp

1.2.5.1 What was learned from quartiles.cpp

I didn't realize how poorly defined "quartiles" were before this exercise which caused me a lot of grief trying to understand the problem, but I came through it for the better.

1.2.5.2 Reflections on Initial Design of quartiles.cpp

I had to alter the initial design to ignore files at the beginning and use vectors instead as test cases to begin to understand the problem.

Once I believed that I'd understood what the results should be, I switched back to using files, but I still used vectors as buffers which isn't the greatest of designs, but it works for this small case.

I think that I could probably redesign the program to be more efficient by avoiding the use of vector buffers should the need ever arise.

1.2.5.3 Reflections on the Testing of quartiles.cpp

The tests worked as expected.

1.2.5.4 Reflections on the Implementation of quartiles.cpp

The implementation wasn't so hard as the understanding of the problem. The hardest details involved deciding which results to shoot for.

1.2.5.5 Reflections on the Techniques of quartiles.cpp

This exercise is a rough introduction to file io. The most useful reference I found was http://en.wikipedia.org/wiki/Quartile which gave me some test values to work with.

1.3 merge.cpp

1.3.1 Understanding of merge.cpp

The problem statement was as follows:

```
(5) Programming Project 12.6 from Absolute C++ (p562 in the 5e book),

File must be called: merge.cpp

\Write a program that merges the numbers in two files and writes all
the numbers into a third file. Your program takes input from two
different files and writes its output to a third file. Each input file
contains a list of numbers of type int in sorted order from the
smallest to the largest. After the program is run, the output file
will contain all the numbers in the two input files in one longer list
in sorted order from smallest to largest. Your program should define a
function that is called with the two input file streams and the
output-file stream as three arguments."
```

1.3.2 Design of merge.cpp

There wasn't really much to design for this exercise.

1.3.3 Testing of merge.cpp

Expected Results:

```
1. program opens three files without problems
2. program sorts results from the first two files and merges
   them into the third without problem
```

1.3.4 Implementation of merge.cpp

Please refer to the attached file: **merge.cpp**.

1.3.5 Reflections on merge.cpp

1.3.5.1 What was learned from merge.cpp

I got a better handle on file streams from this exercise.

1.3.5.2 Reflections on Initial Design of merge.cpp

The initial design was adequate. I altered it a bit to include values from the commandline.

1.3.5.3 Reflections on the Testing of merge.cpp

The tests seemed to work as expected.

1.3.5.4 Reflections on the Implementation of merge.cpp

The implementation went without problems. There wasn't anything too difficult to get working.

1.3.5.5 Reflections on the Techniques of merge.cpp

The main technique this exercise focused on is the use of file streams. This is a very usefult technique to know, and could be useful for the final project of this class.

I also used the insertion sort algorithm again from http://en.wikipedia.org/wiki/Insertion_sort. Strangely enough, I don't have this memorized yet.

1.4 getInt.cpp

1.4.1 Understanding of getInt.cpp

```
(5) Programming Project 12.25 from Absolute C++ (p570 in the 5e book),

File must be called: getInt.cpp

\One problem using cin to read directly into a variable such as an int
is that if the user enters non-integer data then the program will
continue with erroneous data and usually crash. A solution to this
problem is to input data as a string, perform input validation, and
then convert the string to an integer. Write a function that prompts
the user to enter an integer. The program should use getline to read
the user's input into a string. Then use the stringstream class to
extract an integer from the string. If an integer cannot be extracted
then the user should be prompted to try again. The function should
return the extracted integer."
```

1.4.2 Design of getInt.cpp

There wasn't really any design necessary in this exercise.

1.4.3 Testing of getInt.cpp

Expected Results:

```
1. The program takes string input and returns integer output using the
   stringstream class.
2. If integer output cannot be created, the program should repeat.
```

1.4.4 Implementation of getInt.cpp

Please refer to the attached file **getInt.cpp**.

1.4.5 Reflections on getInt.cpp

1.4.5.1 What was learned from getInt.cpp

Nothing much was really gained from this exercise. Using the stringstream class is definitely a lot simpler to write than looping over a string and checking each of its characters to see if they are numeric.

But the aforementioned technique is more robust. The stringstream technique generates integers of zero for non numeric input. Which keeps a program running, but it's not very robust and could lead to some bad problems if not known ahead of time.

1.4.5.2 Reflections on Initial Design of getInt.cpp

The initial design was adequate.

1.4.5.3 Reflections on the Testing of getInt.cpp

The tests worked as expected.

1.4.5.4 Reflections on the Implementation of getInt.cpp

The implementation went without problems.

1.4.5.5 Reflections on the Techniques of getInt.cpp

This is a further exercise on the use of C++ data streams. No outside sources were necessary, and this doesn't seem particularly helpful for future projects other than to avoid the use of the stringstream class for converting strings to ints.

Chapter 2

Project: Week 7 Game of Life Design

2.1 gameOfLife.cpp

2.1.1 Understanding of gameOfLife.cpp

1. Write a program that implements Conway's Game of Life based on your design notes (modifications are allowed, but be sure to track how your implementation differs from your design).

(http://en.wikipedia.org/wiki/Conway's_Game_of_Life)

Recall that Conway's Game of Life is a standard example of a cellular automata. This means that you have an array or matrix of cells. Each turn (also called a generation) the value of each cell may change based upon its neighbors. There is a more complete description in the book, Ch 5, p.234, #12. You can also find much information about the game on the Internet. Much much more information... You will now implement the behavior and tests for a program that runs the game of life based on your designs.

Remember that since you may be developing your program through SSH we will limit the \world" to 80 x 22cells. Typing that all in would be a lot of characters (hint, not a good idea to input the whole world). There are numerous ways to address this problem that may involve hard coding arrays or using the keyboard to intelligently input the starting configuration. That's one reason we wanted to design it before-hand!

Remember that you need to give the user some options for the start of the world (a number of startingconfigurations, the ability to enter a listing of cells to mark as active, some random settings, or some other options).

Also remember that a difficult part of the design is what you will need to do to handle the cells on the edges (hint, use the mod operator).

We will partly be grading your implementation based on how well related it is to the initial design. Remember that you can reflect on how you had to change the design after your first ideas. Just be sure to explain what you learned.

Remember to apply your test plan (you might consider having these be some of your pre-set configurations or as an animation while waiting for user input just after the program starts up). Remember, while it's nice to have shooters and other stable configurations you are testing the operation of your software. In this case you will have \boundary" conditions to test, literally in this program. For a cell on any edge or corner you'll need to take into account cells on the other side of the grid.

HINTS-
a. I recommend that you start with some simple pre-set or stable configurations to make sure your understanding and design were correct for the game. This will save time in later debugging since you know how something will work in a pre-determined case.

b. Remember to get your program working without the edges looping. Once you know your code is working then implement the edge calculation.

2. As one website states, the Game of Life is one of the most programmed games in the world. Be very careful about borrowing any code, or ideas you see in someone else's code as you should be learning how this works for yourself.

2.1.2 Design of gameOfLife.cpp

2.1.2.1 Week 8 Design Additions

The biggest challenge I came across this week was trying to get the animation aspect of this project to work more nicely. To this extent I redid everything in ncurses.

I'm still unclear how this will work for grading purposes, so I made three stages for the program.

The first stage simply calls either of two secondary stages. The first secondary stage is simply last week's simple implementation. The next secondary stage uses ncurses to animate through 5 different patterns or so, and allows for speed control and interactive menus.

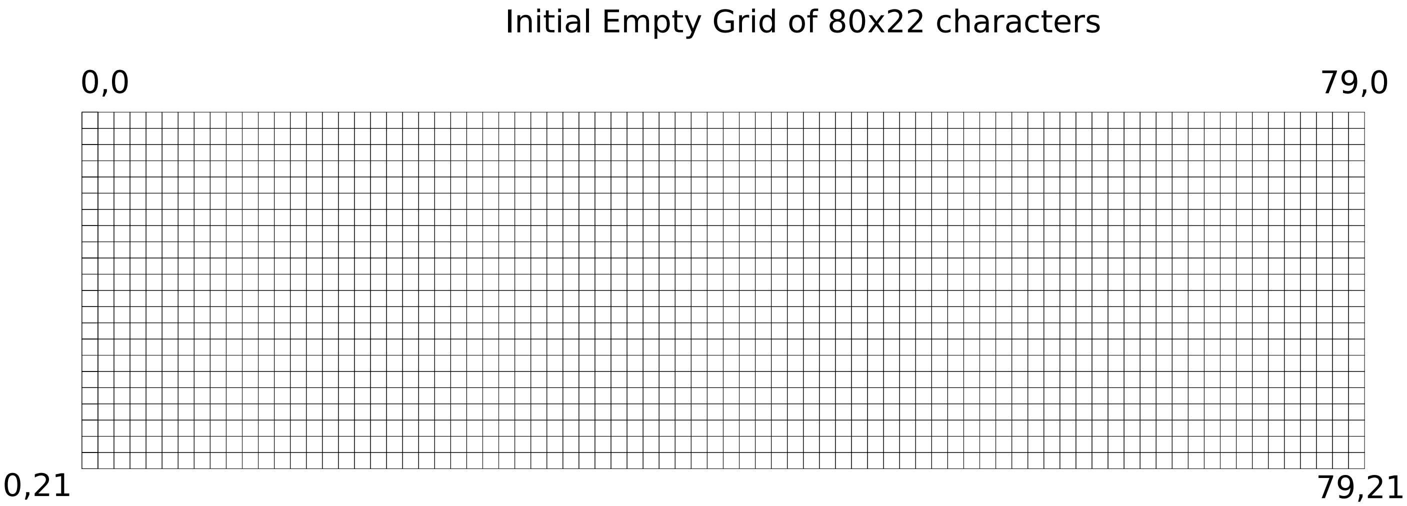
I thought I would have had more time to work on initial conditions this week, but the real world got in the way of that unfortunately.

2.1.2.2 Week 7's Initial Design (pseudocode and rationalizing walkthrough)

At this point I'm still unsure whether or not to use classes and/or structs. I think I want to keep the first draft as simple as possible. This means trying to restrict it to a single main() function if possible just to test the rules.

So off the top of my head, I'll need two strings to hold the game board: one for the current generation and one for the next generation. Hopefully that will work.

The strings are meant to hold 80 columns of characters by 22 rows of characters, so they should each be $80 \times 22 \rightarrow 1760$ chars long.

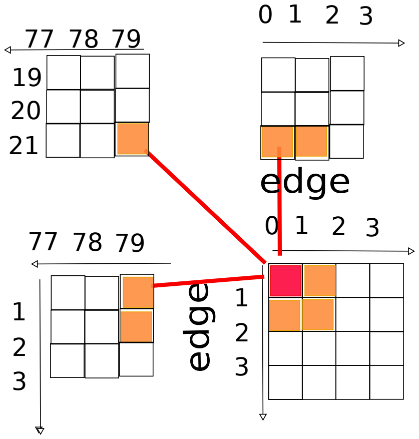


That should be enough to get started on the rules. Rule 1 needs to somehow check all of the eight neighbor cells. I'm going to have to decide at this point whether or not to use wrap-around edges.

Here it's starting to look like I can't constrain things to the main() function as I initially desired, so the program is going to be somewhat complex right off the bat. That's okay though. I thought maybe there was a simple way to implement rule1 using simple loops, but now that I'm considering edge detection, I don't think that's a wise way to start. I want to get as much of the entire functionality setup as soon as possible which means thinking of the big picture and final destination as soon as possible. Hopefully that's a wise route to take.

The reason I want to consider edge detection right away is because it might be one of the trickier parts of the design. The first thing to come to mind is a function that will take one of the 22 row strings and find all of side neighbors for each cell.

Demo of Edge Wrapping



For the cell (0,0): the tangential cells are:

1: (79,21)
2: (0,21)
3: (1,21)
4: (1,0)
5: (1,1)
6: (0,1)
7: (79,1)
8: (79,0)

The problem with this is mod(a,b) isn't defined in C++, but it's pretty easy to define.

Rethinking this, the cells could be contained in a struct, or a struct of structs.

```
struct cell {
  int x;
  int y;
  bool occupied;
}
```

The function could probably find all 8 neighbors in one shot and use a struct to store them.

```

// walk clockwise around the current cell to get 8 neighbors
struct neighbors {
   cell topLeft;
   cell topCenter;
   cell topRight;
   cell centerRight;
   cell bottomRight;
   cell bottomCenter;
   cell bottomLeft;
   cell centerLeft;
   cell all[8];
};
```

Having a struct for neighbors, here's how the function might go:

```
// purpose: return a structure containing all neighboring cells
neighbors getNeighbors (cell current)
{
  neighbors hood;
  hood.topLeft = getTopLeft(current);
  hood.topCenter = getTopCenter(current);
  hood.topRight = getTopRight(current);
  hood.centerRight = getCenterRight(current);
  hood.bottomRight = getBottomRight(current);
  hood.bottomCenter = getBottomCenter(current);
  hood.bottomLeft = getBottomLeft(current);
  hood.centerLeft = getCenterLeft(current);
  // store everything in array for easy iteration.
  hood.all[0] = hood.topLeft;
  hood.all[1] = hood.topCenter;
  hood.all[2] = hood.topRight;
  hood.all[3] = hood.centerRight;
  hood.all[4] = hood.bottomRight;
  hood.all[5] = hood.bottomCenter;
  hood.all[6] = hood.bottomLeft;
  hood.all[7] = hood.centerLeft;
  return hood;
}
```

Which causes me the grief of having to define 8 functions to get each cell. It's not the most eloquent of designs, but it might work. This is where the modular math comes in.

Did I mention already that $mod(a, n)$ is not defined in C++? Well, it's not. But it's easy to define.

```
int mod(int a, int b)
{
    int r = a % b;
    return r < 0 ? r + b : r;
}
```

source: http://stackoverflow.com/questions/11720656/modulo-operation-with-negative-numbers

Now proceeding with the design: I need a function to get the top left cell from the current cell. The current cell is at (0,0). So I think the top left cell will always be at ($x - 1$,$y + 1$) but for these edge conditions we need to remember to use the mod function just defined above. Just to clarify: I expect that for (0,0) topLeft = ($mod(0 - 1, 80)$,$mod(0 - 1, 22)$) which should be (79,21).

$$mod\,(-1, 80) \rightarrow 79$$

$$mod\,(-1, 22) \rightarrow 21$$

So defining the getter functions should be simple from here. I'll provide a listing for the top left getter and leave the rest out since it's pretty straight forward to simply modify the mod call parameters.

```
cell getTopLeft(cell current)
{
  cell block;  // create a new cell
  block.x = mod(current.x - 1, 80);
  block.y = mod(current.y - 1, 22);
  return block;
}
```

Reflection on Current Design

The fact that I'm calling these getters suggests that I ought to be using classes and objects, which is a good possibility. But again I want to try to keep this as simple as possible, and I don't think that classes are necessary yet.

In the process of writing all of those functions out, I realized how difficult it would be to change world sizes, so I replaced all instances of "80" with "COLS" and all instances of "22" with "ROWS". I then defined "ROWS" and "COLS" in the top of the source code.

I'm at a point where I need to decide how to deal with occupancy on each cell. I think I had it in mind to use a "*" character to indicate occupancy, and a blank character to indicate vacancy.

Is it straightforward to determine a cell's vacancy or occupancy? I think it ought to be simple enough to get the character at the ith and jth position. Hopefully that assumption is true. Here is my untested idea to do so:

```
// purpose: search the world array for current cell to determine if it is empty
bool isVacant(cell current, char* world)
{
  char occupied = world[current.x][current.y];
  if (occupied == ' ')
    return true;
  else
    return false;
}
```

So I've got a lot in place now to start working on Rule2 and Rule3 which will determine how the next generation world will look. This will be carried out in a driver function to evaluate neighbors.

The functionality to implement is that we have to get the number of occupied/vacant neighbors.

```
// purpose: get number of occupied/vacant neighbors
int numberOfVacantNeighbors(cell current, char* world)
{
  int vacancies = 0;
  neighbors hood = getNeighbors(current);

  // determine vacancy for each neighbor
  for (int i = 0; i < 8; i++)
    {
      if (isVacant(hood.all[i]))
        vacancies++;
    }
  return vacancies;
}
```

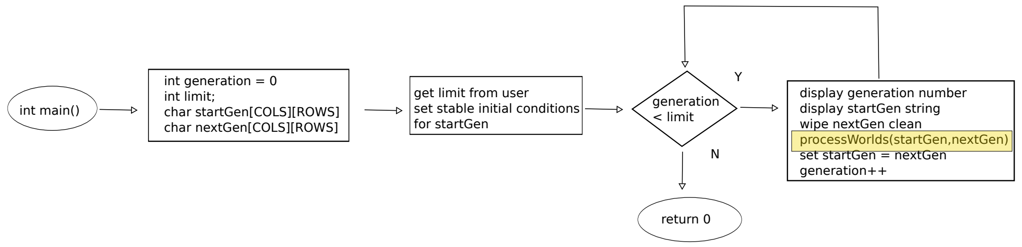
From here, it's a simple matter of applying Rule2 and Rule3. Rule2 only applies to occupied cells, and Rule3 only applies to vacant cells. Starting with Rule2, there are two functions to define:

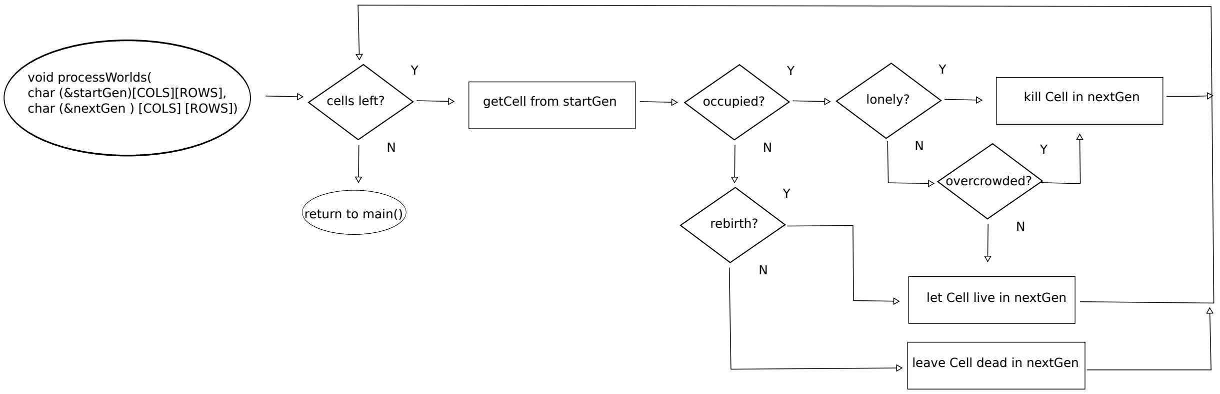
```
// purpose: see if occupied cell can survive loneliness
// ”If an occupied cell has zero or one neighbor, it dies of loneliness.”
bool loneliness (cell current, char* world)
{
  int vacancies = numberOfVacantNeighbors(current, world);
  int occupancies = 8 - vacancies;
  if (occupancies <= 1)
    return true;
  else
    return false;
}

// purpose: see if occupied cell can survive overcrowding
// "If an occupied cell has more than three neighbors, it dies of overcrowding."
bool overcrowding (cell current, char* world)
{
  int vacancies = numberOfVacantNeighbors(current, world);
  int occupancies = 8 - vacancies;
  if (occupancies > 3)
    return true;
  else
    return false;
}
```

I'm kicking myself for defining vacancies instead of occupancies, but this ought to work. Rule3 should be just as easy to define:

```
// purpose: see if vacant cell can regenerate
// "If an empty cell has exactly three occupied neighbor cells,
// there is a birth of a new cell to replace the empty cell."
bool rebirth (cell current, char* world)
{
  int vacancies = numberOfVacantNeighbors(current, world);
  int occupancies = 8 - vacancies;
  if (occupancies == 5)
    return true;
  else
    return false;
}
```

All that's missing now is a driver function to iterate over the world, and some test conditions to try out. The test conditions are probably the hardest thing to implement at this point. Might as well hack out the driver function since it seems obvious. Since this turned out to be lengthy, I'll just show a diagram:





I think I'm at a good point to verify that my design works before I move forward any further. Also, moving forward shouldn't be too difficult. The next step immediate step is to consider how initial conditions should be determined. We're not allowed to use file I/O in this design yet, though it might be allowed next week.

2.1.3 Testing of gameOfLife.cpp

This is where the real fun starts. I started with two simple oscillators to check that the edge wrapping works and a still life block.

The results met my expectations as follows:

```
# ./gameOfLife
Enter a generation limit:
2

generation: 0
--------------------------------------------------------------------------------
               .




..                                                                             .




 ..
 ..








               .
               .
--------------------------------------------------------------------------------

generation: 1

--------------------------------------------------------------------------------




.
.
.



 ..
 ..
```

```
              ...
--------------------------------------------------------------------------------


generation: 2

--------------------------------------------------------------------------------
               .




..                                                                             .




 ..
 ..








               .
               .
--------------------------------------------------------------------------------
```

2.1.4 Implementation of gameOfLife.cpp

2.1.5 Reflections on gameOfLife.cpp

2.1.5.1 What was learned from gameOfLife.cpp

This was a great exercise on editing a char array. I've been wanting to program this game for at least a decade, and I finally have a working implementation which is fantastic.

It really wasn't anywhere near as difficult as I imagined it would be, which is the most important thing I learned for my own sake.

By week 8, I'd gotten considerably better with ncurses for better or worse. There are numerous benefits and struggles to adding a psuedo-GUI to the terminal.

2.1.5.2 Reflections on Initial Design of gameOfLife.cpp

As of week 7:
This initial design works, but it leaves a lot to be desired as far as user input and setting known initial conditions go. I think the next alterations will include means to input some of the many well-known configurations.

As of week 8:
I mostly left the initial design from week 7 alone, but built more interfacing on top of it including animation and menus. I didn't have enough time to implement all the initial conditions that I thought I could.

It turned out to be quite difficult to hard code initial conditions. I'd meant to resolve that problem by allowing the user to draw cells via mouse input, but I couldn't get there in time.

I also intended to implement drop down menus via ncurses, but couldn't work out the details quickly enough.

Still, I managed to make some progress on the program over the week.

2.1.5.3 Reflections on the Testing of gameOfLife.cpp

As of Week 7:

The tests meet the expectations of the game. My main concern at this point was that the game would process the simplest known configurations: still lives and 3 cell oscillators, wrapping edges around the board. These tests succeeded.

As of Week 8:

Two of my 6 test initial condition oscillators that I implemented are broken. I mentioned them in the menu. I'm not sure why, but I think it has something to do with the screen wrapping and the windowing code.

I didn't have time to work out the details on that front. Maybe I'll get around to it come time for the final project, but I wouldn't count on it.

2.1.5.4 Reflections on the Implementation of gameOfLife.cpp

As of Week 7:
The implementation went surprisingly well so far. The most difficult problem I had was that I'd accidentally flipped index variables in my driver functions which caused the board to rotate 90 degrees on every generation.

As of Week 8:
I've made some nice progress on the interface, but left the core functionality alone as mentioned before. It was particularly difficult for me to make the jump into ncurses, but I think it was well worth the effort.

2.1.5.5 Reflections on the Techniques of gameOfLife.cpp

As of Week 7:
I tried to keep the game as simple as possible, which turned out to be more complex than I originally intended. As a result of my intentions however, I avoided the use of classes and left everything in terms of structs. So I

didn't really use any of this week's topics such as vectors, classes or exceptions. I also avoided the use of file IO since the instructions explicitly said to avoid this.

I might re-implement the entire game in terms of classes, for next week, but I'd rather stay with simplicity. There is plenty of work to do trying to create a menu of initial conditions and user interaction for the remainder of next week. It's probably likely that I'll end up using file IO to implement more features such as gliders and other spaceships.

As of Week 8:

As mentioned earlier, my biggest endeavor this week was trying to bring more of a graphical user interface to the terminal which was simultaneously easier and harder than I thought it would be.

It turned out to be easier to do animations than I thought it would be, whereas dropdown menus, forms, and sub windows still seem far out of my reach.

I'd still like to employ these methods towards the final project which is coming up quite soon. There are numerous places where psuedo-terminal GUIs might be useful such as text editing, file browsing and monitoring of various things.