Assignment 2

The Binary Game of Life

Revised Due Date:
Friday, March 6, 2015
before midnight

Purpose:

C finds its most frequent modern use in embedded systems development—at the level of the processor/controller. Working in this field involves dealing with binary and hexadecimal values on a daily basis. Unfortunately, given that the virtualization software employed on your PC—not to mention the Windows operating system beneath it—makes communication with the hardware somewhat difficult, an embedded programming project is not currently possible in this course. Nonetheless, we can explore some of the power and speed of C's low-level operations by implementing a popular computer game, known as the Game of Life, using bitwise operators. As an added bonus, you'll explore the use of functions and forward declarations as well.

You will have completed this assignment when you have:

- ☐ Implemented the rules of the game of life without error, according to the code shown below;
- ☐ Implemented and utilized the various functions, variables and constants described in this document;
- ☐ Used forward declarations for *all* functions in your code;
- ☐ Used the binary operations described to process the information 64-bits at a time;
- ☐ Addressed boundary issues correctly so that the grid 'wraps' around at the edges;
- ☐ Written code that displays one grid at a time, using keypresses to advance to the next screen;
- ☐ Produced code which is anonymous; it must *not* generate any output which identifies you as its author either during execution or in the executable file itself;
- ☐ Submitted your fully-functioning C program via Canvas. See Submission Guidelines below for details.

Assignment 2

The Binary Game of Life

I. Introduction

The 'Game of Life' (or GoL) was developed in 1970 by John Conway. It started as a computer recreation designed to simulate the behaviour of a population of creatures living on a 2D surface, whose subsequent evolution is dictated by just three simple rules (see below). Historically, it forms the starting point of the subject of cellular automata. More recently (and controversially), it has been suggested to be the foundation for a new view of physics (see http://en.wikipedia.org/wiki/A_New_Kind_of_Science). A good introduction to the GoL can be found at http://arxiv.org/ftp/cs/papers/0406/0406009.pdf, which includes an introduction to how logic functions can be implemented using the GoL Rules. The Wikipedia page is especially worth a look, at http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

II. The Rules

In the GoL, one typically starts with a 2D grid consisting of *n X m* squares. Each square in this grid is initially assigned the value `1` (ALIVE) or 0 (DEAD) at random. The game then proceeds according to the following three simple Rules.

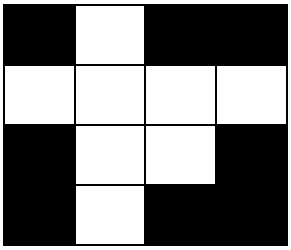
1. If the total number of ALIVE cells in the eight adjacent neighbours is less than two, the square dies of loneliness, and becomes DEAD. But if the number of ALIVE neighbours is more than three, the cell dies of starvation, and becomes DEAD as well.

2. If the total number of ALIVE neighbours is 2 or 3, the cell stays ALIVE.

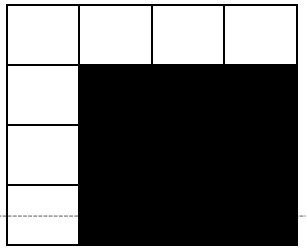
3. If a DEAD cell has exactly three ALIVE neighbours, then the DEAD cell can become ALIVE again.

We'll refer to cells that follows Rule 3 as ZOMBIE cells; ALIVE cells that follow Rules 1 and 2 will be referred to as NORMAL.

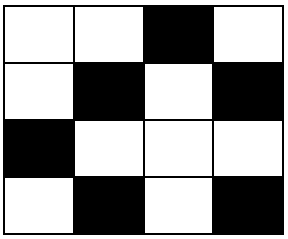
As an example of these rules in action, consider the following 4 X 4 grid, where ■ = ALIVE and □ = DEAD.



The top left square is ALIVE, but it has three DEAD neighbours, in white. Hence, according to rule 1 above, it will be DEAD on the next iteration of the game. The square to its right is initially DEAD, and has only two ALIVE neighbours on either side. Hence, by Rule 3, it stays DEAD. But the cell directly below this cell is a ZOMBIE cell, having three ALIVE neighbours. Hence by Rule 3 it becomes ALIVE on the next iteration of the game, as do the two cells immediately below it. Working through each square according to the above rules, the situation on the next iteration will be:



And on the next iteration still, it will be:



Make sure you fully understand how these rules work and how they affect the next generation of cells before proceeding.

Note that the above rules are implemented according the number of DEAD and ALIVE cells in each generation; your code determines who is DEAD or ALIVE based upon the DEAD or ALIVE neighbours in the *current* iteration of the grid, and generates the next iteration *only after* the current iteration has been computed.

III. Implementation

For our purposes, the size of this grid will be fixed at 64 X 32 throughout this assignment. Since each square contains a `1` or a 0, the most appropriate way to handle the modeling is to use a one-dimensional array of `unsigned long long int`s. An `unsigned long long int` is a 64-bit data type. To simplify our notation, we'll assume the following typedef:

```
typdef unsigned long long int uLLInt;
```

(Hence `uLLInt` becomes a shorthand for an `unsigned long long int` in the following discussion.) Therefore, a 64 X 32 grid of DEAD and ALIVE cells can be stored in an array declared as

```
uLLInt Grid[ARSZ];
```

where `ARSZ` is defined as 32. This effectively giving us a binary grid of 64 X 32 bits.

This approach has the great advantage that we can use bitwise operations to calculate the outcome of each cycle 64 bits at a time, essentially reducing the execution time from $\mathcal{O}(64 \text{ X } 32)$ to $\mathcal{O}(32)$ for most operations. While this speed advantage won't have a noticeable effect on a 3 GHz PC, it would have a significant performance impact when much larger grid sizes, say 4k X 4k, are used.

Consider `Row[2]`, highlighted in boldface print in the box below. We'll assume each row has been assigned a random 64-bit number. If we wish to know if any living cell in this row has a living neighbor in the row directly above it, we can bitwise AND the two rows:

```
Row[1] & Row[2];
```

Mathematically this is just (showing the leftmost 32 bits only, due to lack of space):

```
  Bit: 63                                                              0

Row[0]  00101110101110000001010101111010001011101011100000010101011110 10
Row[1]  10000101110101001111111010110101000110101110111011100001010010 10
Row[2]  00101110100010000101011010100000111101001000110100101001010001 01
Row[3]  00000100010111110111011111000010111011101100000010101011001000 10
Row[4]  11001010111000101001011010110000111110110000010100101110100011 11
Row[5]  00011011100010110010010110111010011011111111010110100101100101 11

   ...
```

```
1000010111010100111111010110101…
00101110100010000101011010100000…
00000100100000000010101101000000…
```

Wherever a `1` appears in the resultant `uLLInt` (indicated in bold on the third line above) this indicates that, for an ALIVE cell in `Row[2]`, it has an ALIVE neighbor in the row directly above it. Similarly, to find out whether there are ALIVE neighbours in `Row[3]`, bitwise AND `Row[2]` with `Row[3]`:

```
00101110100010000101011010100000…
00000100010111110111011111000010…
00100100000010000101011010000000…
```

To find which cells in `Row[2]` have a neighbor immediately to their right, make a copy of `Row[2]`, bitshift it left by one bit, and AND the result to itself, i.e.

```
Row[2] & (Row[2]<<=1)
```

ANDing the shifted `Row[2]` with itself gives:

```
00101110100010000101011010100000…
01011101000100001010110101000000…
00001100000000000000010000000000…
```

Again, whenever a `1` occurs, this indicates an ALIVE neighbor to the right of an ALIVE cell. Similarly, to find the number of cells having an ALIVE neighbor to their left, you'll need to right-shift a copy of the row by `1` and AND it to itself. For the top-right, top-left, bottom-right and bottom-left calculations, you will need to right- and left-shift both the row above and the row below and AND it in similar fashion.

Recall that Rule 3 states that a DEAD cell may be brought back to life if it has exactly three ALIVE neighbours—the ZOMBIE condition. How do we use bitwise operations to test to see that a DEAD cell ('0') has ALIVE neighbours? (ANDing a 0 with an ALIVE cell only generates 0's.) We can temporarily bring DEAD cells back to life by performing a bitwise inversion on a row, i.e. `~Row[2]` converts

```
00101110100010000101011010100000…
```

to

```
11010001011101111010100101011111…
```

If we bitwise AND this row with its nearest neighbour rows, then the location of the resultant `1`'s tells us the locations of the DEAD cells that have ALIVE neighbours. This follows the same logic as finding the neighbours of the ALIVE cells in the discussion immediately above.

[With one small exception, which you may otherwise miss: to find out if a DEAD cell has an ALIVE neighbour to its left or right (including in the rows immediately above and below), you *do not* AND a ZOMBIE row with its shifted shelf, since this only tells the DEAD cells how many DEAD neighbours they have—useful for a full-blown zombie version of the game, but not helpful here. To find out how many ALIVE neighbours a DEAD cell has, create a ZOMBIE row by inverting each cell's condition first (from 0 to `1` or `1` to 0, as indicated above), and *then* AND the result with the shifted *NORMAL*, i.e. ALIVE neighbours.]

To handle these various conditions, it is first necessary to create four primary functions, called `T()`, `B()`, `L()` and `R()`, to handle the Top, Bottom, Left and Right conditions respectively, using the logic outlined above. Each of these functions takes two variables, the row number (an `int`) and a binary value (which checks for both NORMAL or ZOMBIE cells), and returns an `uLLInt`, the result of the requested operation.

The four additional functions, TL(), TR(), BL() and BR(), do not need to be *derived* from the existing functions (T(), B(), L() and (T()), but will computationally function as combinations of them. For example TL() operates by ANDing the right-sifted version of the row above it.

Each of your functions only need to pass the row number of the row in quesitons, and *not* the value of the `uLLInt` stored in that row.

Once these eight functions have been created, it is useful to call them inside a function that sums the total number of ALIVE neighbours in each row and returns the resultant information as an `uLLInt` to the calling program—in `main()`, in this case. This function, called `sumNeighbours()`, needs to perform the following operation, which will eventually be illustrated in the diagram at right. For now, note that `sumNeighbours()` takes two arguments, the current row number and the NORMAL/ZOMBIE binary value; it returns a `uLLInt` containing the next generation for the current row.

First, define the constants TOP, BOT, LFT, RGT, TLF, TRT, BLF, and BRT in your code and them set to values 0 through 7 respectively. (You can think of these values as standing for the compass values indicating the neighbors immediately North, South, East, West, NorthWest, NorthEast, SouthWest, and SouthEast from the current cell.) Next, assume that an array of `uLLInt`s exists called `Neighbours[]`, and it holds the results from the `T()`, `B()`, `L()`, `R()`, `TL()`, `TR()`, `BL()` and `BR()` functions, as follows:

```
Neighbours[TOP] = T(rowNum, NORMAL);
Neighbours[BOT] = B(rowNum, NORMAL);
Neighbours[LFT] = L(rowNum, NORMAL);
                …
Neighbours[BRT]= BR(rowNum, NORMAL);
```

Now the `Neighbours[]` array contains all the information about the ALIVE neighbours to each cell. [To check for the ZOMBIE condition, you'll need to run `sumNeighbours()` again, this time with the ZOMBIE condition set (rather than NORMAL being set, as above), and then OR each resulting `uLLInt` into the current row to 'revive' the cells that would otherwise stay DEAD, according to Rule 3.]

To calculate the uLLInt returned by `sumNeighbours()`, consider the following example. Let's assume that we've calculated the nearest Neighbors array for a particular row, say row 7. Let's say the `Neighbors[]` array for this row has the following values, where each value is 64-bits long, with the leftmost bit, bit 63, indicated in the box below:

```
Neighbours[TOP]   0100001010000…
Neighbours[BOT]   0001010101100…
Neighbours[LFT]   1010000010000…
Neighbours[RGT]   0000101011000…
Neighbours[TLF]   0001000000000…
Neighbours[TRT]   1010000000100…
Neighbours[BLF]   0000100100100…
Neighbours[BRT]   0100010100000…
```

Then the total number of neighbours of the leftmost cell in row 7 is equal to the sum of all the numbers in the box shown above—in this case 2. Hence, according to Rule `2`, the leftmost bit for row 7, column 63, will be a `1` in the next generation: the cell stays alive.

To extract this leftmost bit from each row, we first need to create a bitmask, called `LBITMASK`, `#defined` as

```
#define LBITMASK 0x8000000000000000
```

The value of `LBITMASK` is a 64-bit value with the leftmost bit set to '`1`'. Copy this value into

an `uLLInt` variable (call it `LMASK`). To sum all the leftmost bits in the `Neighbours[]` array, you need to (1) mask out all the other bits in the row with `LMASK`, (2) determine if the value remaining is non-zero, (3) if it is, add 1 to the total number of neighbours, and (4) loop through for each of the eight neighbours. Surprisingly, this only takes two lines of code in C:

```
for (unsigned int compass = 0; compass < 8; compass++)
   totAliveNeighbors += ((LMASK & Neighbors[compass]) > 0);
```

Make sure you understand this code completely before proceeding. (There *will* be a question on the next midterm and/or final based on this code fragment).

To check the next bits in the column (i.e. bit 62), shift the bits in `LMASK` right by one, and repeat the procedure.

Note that, when you wish to build the next generation array in the GoL (see the paragraph immediately below), `LMASK` can be used to insert a '`1`' into the cell at the current cells row and column.

IV. Possible Complications

a. Your code will operate on an array (called `Rows[]` in the code above) that contains the arrangement of cells in the *current* generation. But it should store the results for the next generation in a second array, also consisting of eight `uLLInt`s; this array will eventually be displayed to the terminal window. At the end of each cycle, you can either (a) copy the contents of the new array into the old array, and begin the process again, using the old array once again for your neighbourly calculations, or (b) set a variable that toggles between the two arrays. Thus the new array for one generation becomes the old array for the next, and vice versa. This saves you having to copy the new array into the old on each cycle. Regardless of which method you use, be sure that the array you store your new data into is not contaminated with old values that confuse your algorithm.

b. Your code must handle the situation at the boundaries of the grid correctly. In particular, they must 'loop' so that the row 'above' `Row[0]` is `Row[31]`. Thus `Row[31]`'s neighbours in the line below are technically those found in `Row[0]`.

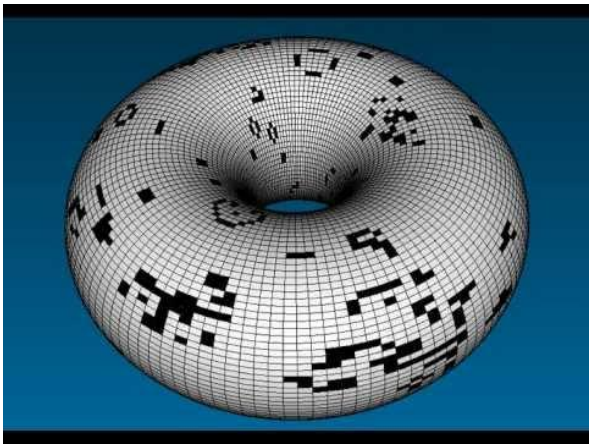
   Hint: use a common, but underutilized mathematical operation to effectively 'wrap' your arrays around without the need for unwieldy 'if' statements.

   Your arrays should also 'wrap' around at the left and right endpoints of the grid, i.e. bit 63 is 'next' to bit 0. This is a 'bit' trickier, and it will require that you mask the end bits, test for a '`1`' or '0', and if the former, bitwise OR a '`1`' onto the opposite end of the `uLLInt` in question…a procedure similar to the bitmasking used in the previous section. For this reason, in addition to `LBITMASK`, you'll probably want to create an `RBITMASK`, like this

```
#define RBITMASK 0x0000000000000001
```

for those situations when you left shift a row and wish to OR the bit shifted off the left end into the rightmost position.

Note that because of the boundary conditions at the top and bottom of the array, and at the left and right edges, your binary life form essentially 'lives' on a torus (or donut), like this:



c. To display the actual output, you can use the following function for each row:

```
void displayBinary(uLLInt x){
uLLInt MASK=0x8000000000000000;
   do {
      printf("%c",
         (x & MASK)?'X':0x20);
   } while((MASK >>= 1)!=0);
}
```

(This uses the same principle as the Binary Display Function in Example `11` of Module 3 in the course notes. The Binary Display Function itself can be used, with modifications, for diagnostic purposes, if you need to output 0's and `1`'s to the screen.)

Note that your program should run in full-screen terminal mode, to accommodate the full width and height of the 64 X 32-bit grid.

d. To clear the screen between printouts of the grid, use

```
system("clear");
```

This helps ensure that you can actually see the progression of your lifeform between successive cycles.

e. To loop through your code one grid at a time, consider using the following code:

```
char ch;
do{

...//your code here

} while((ch=getchar())!='x');
```

WARNING:

***Do NOT** automate your GoL simulation so that it spontaneously advances to the next generation without user input.*

This ensures that your GoL program displays its output one cycle at a time—hit the ENTER key to advance to the next screen. You can then press `x + ENTER` to exit the program.

f. The following code can be used to initialize each `uLLInt` in your initial array. While

not a true 64-bit random number generator, its good enough for the purposes of this assignment:

```
uLLInt init64;

srand(time(NULL));
init64 = rand();
init64 <<= 2;
init64 |= (0x00000003 & rand());
init64 <<= 31;
init64 |= rand();
```

`init64` now contains an approximately-random 64-bit number. This code, of course, should be put into its own function and called in a loop to initialize each of the 32 values in `Row[ARSZ]` at startup (or use whatever name you elect to call this array).

V. Testing

Your code should be completely tested to ensure that it follows the three Rules of the GoL as described above in *Section II. The Rules*.

Particular attention should be paid to the four areas where errors are most likely to occur:

1. The screen width is only 63 bits, not 64;
2. ZOMBIE cells are not brought back to life as required;
3. One row or column, usually at the boundary, becomes inactive, because a loop does not include it correctly;
4. The wrap-around conditions at the margins are not correctly implemented.

The best way to test your program is to (1) comment out `srand()` in your code so that the random-number generator is always seeded to the same value, (2) recompile your code, and (3) compare your results with those of your classmates whose random number seed generator has been similarly inactivated. Assuming they both generate the same sequence of random numbers, both programs should produce the same output. (Be sure to re-engage `srand()` before submitting your finished product.)

VI. Submission Guidelines

Your C code should be submitted on or before the deadline, March 1st, before midnight, to the dropbox set up in Canvas. Late submissions will be penalized according to the schedule outlined in Module 0, Slide 20, 'Late Assignments and Labs.' If there are any problems with Canvas, please ship your code to me at houtmad@algonquincollege.com.

When submitting, please be sure to zip all files, even if you are only submitting one file. Your zip file *must* have the following name:

Assignment2_*YourLastName*_*YourFirstName*.zip

Note that:

(1) pseudocode/PDL is not required for this assignment. However, your ALGORITHM (to be provided in the header of each function) must be sufficient to describe the logic behind the operation of the code in each function.
(2) Your code must be formatted according to the CST8234 Documentation Standard.
(3) See Section *VIII. Notes* for additional information

VII. Marks/Evaluation

Your assignment will be marked as follows:

Requirement	Mark
Code follows the Documentation Standard	6
Followed the instructions in this document (including any late additions/changes), and in particular used the functions correctly, i.e. for the purpose described	9
Code appears to execute correctly, i.e. no obvious defects	4
Clarity of Code (terse is good, but not to the point of incomprehensibility)	6
Total:	**25**

As an added incentive, your executable code will be put out for peer review, i.e. it will be anonymously published for your classmates to review, and bonus marks will be awarded for bugs found according to the following rules:

a. each bug found will garner 1 bonus mark for the finder, up to a maximum of 7 bonus marks total;
b. each bug detected will cost the original programmer of the code 2 marks. You will be informed of this error, and if you fix the code within one week of notification, you only lose `1` only mark; otherwise the full penalty of 2 lost marks applies;
c. The student who detects the most bugs is (anonymously) awarded a certificate as the '*Best C Language Bug Finder for the 2015 Winter Semester*' (potentially beneficial if you are applying for a job as software tester someday and wish to establish your *bona fides* for the position);
d. 32- and 64-bit versions of the code will be posted on Canvas approximately one week after the assignment submission deadline, and an announcement will be made beforehand so that everyone has the chance to participate equally;
e. Only the first student to detect a verifiable bug gets bonus marks for finding it; priority is established according to the time at which your email arrives in drop-box set up for the purpose.
f. Bugs must be reported on a standard bug form (to be made available at a later date). Your bug report must be sufficiently detailed for the bug to be observable on the professor's ancient 32-bit computer;
g. The professor is the final arbiter on all matters related to determining whether a reported bug is a true bug (according to the rules laid out in this document) or merely an 'unorthodox' interpretation of the assignment guidelines, but still within the rules. However, since the rules of the GoL are absolute, any break from these rules should be immediate and obvious to anyone having the patience to step through a few iterations of game. *If you find a bug, report it.*
h. Bug reports must be correctly filled out, including having a proper header that contains both the name of the programmer who submitted the original bug (an alias, for the sake of anonymity), and a brief description of the nature of the bug.
i. Students must check the list of submitted bugs before submitting their own candidate bugs. This is to ensure that they are not re-reporting an already-reported bug. Bugs which are essentially duplicates of previously reported bugs will not be counted.

Additional details will be provided shortly after the submission deadline at the start of March.

VIII. Notes

- Your two `uLLInt` arrays should be declared globally, before `main()` but after the `#include`s, so that they can be accessed inside any function;

- At no point in your program should you need a 2D array to store values; the point of the exercise is to create a well-documented C program using `1`D arrays that employ bitwise operations throughout, and then operate on the `uLLInt` values in special-built functions using the row number and `uLLInt`s to pass information only;

- Your code should *not* use pointers, even if you know how to use them;

- Your executable code must be anonymous—it must not give any indication as to your identity during execution. Nonetheless, you are still required to enter your name in the HISTORY section of the Documentation Standard header. As this is inside a comment block, it will be stripped away during compilation. But this is the only place (aside from using your name in the submission guidelines) where you will indicate your identity;

- Whenever you use a professor's code in your program, even in altered form, you must cite him/her as the original author in the HISTORY section of the header that precedes that function. Remember, Documentation Standard headers should precede each function, except those that merely act as 'setter/getter' functions. Practically speaking, for this assignment, even the shortest functions will require a header, plus, of course, appropriate in-line comments in the body of each code block;

- For aficionados of the GoL, each grid is capable of supporting a menagerie of possibly 'lifeforms.' If you're curious, these sites may be of interest:

  - http://www.binarydigits10.com/articles/conwaysgameoflife
  - http://www.bitstorm.org/gameoflife/lexicon/
  - http://conwaylife.com/w/index.php?title=Main_Page