CptS 122 — Homework #4 **Due Monday Feb. 25, 2013 midnight (24:00)**

Becoming a Zen Master of Pointers: Looking at Life in Whole New Ways

Problem 1: Dynamic Allocation of a 2D Array and Life as a Multi-subscripted Array

In the previous assignment you had to implement Conway's Game of Life by allocating, for each board, a single block of memory (an array of `char`s). A pointer pointed to the start of this block of memory and then all "cells" in the 2D board were obtained by calculating an offset from the start of the block. This offset was a function of both the row and column index. You had the option of implementing a toroidal universe (which we'll talk about more in the second problem). However, for the remainder of this discussion, let's assume you implemented the grid with the "dead zone" buffer—so that cells on the edge of the grid were ignored and cells in the interior of the grid were all treated in the same way.

A solution to the previous assignment will be posted to the class Web site shortly after the due-date. In that solution, in the file `life.c`, there are two `char` pointers, called `g0` and `g1`, that point to memory sufficient to contain two different realizations of the Life universe. There are two additional `char` pointers, called `g_current` and `g_next` that are associated with the "current" and "next" generations of the game. For one cycle of the game `g_current` equals `g0` (meaning `g_current` points to the same memory as `g0`) while `g_next` equals `g1`. Then, for the next cycle of the game, the values of `g_current` and `g_next` are swapped so that `g_current` equals `g1` while `g_next` equals `g0`. In this way, the two blocks of memory take turns representing the current and next generations.

These pointers are all declared as static global variables, meaning that every function in the file `life.c` can see these variables but these variables are not visible to functions defined in other files. The following is code from the file `life.c` that is relevant to the current discussion:

```c
static char* g0=NULL;
static char* g1=NULL;

static char* g_current=NULL;
static char* g_next=NULL;

#define Current(I, J) g_current[((I) + 1) * ncol_t + ((J) + 1)]
#define Next(I, J)    g_next[((I) + 1) * ncol_t + ((J) + 1)]

// Function to initialize the boards.
void lifeInit(int the_nrow, int the_ncol) {
      .
13     .
14     .
15  // Allocate space for g0 and g1.
16  g0 = (char*)malloc(ntot_t * sizeof(char));
17  g1 = (char*)malloc(ntot_t * sizeof(char));
18
19  // Initial assign of g_current and g_next.
20  g_current = g0;
21  g_next = g1;
22     .
23     .
24     .
25 } // lifeInit()
26
27 // Function to create the next generation.
28 void lifeStep() {
29     .
30     .  // Code to populate the next generation.
31     .
32  // Swap the current and next grids.
33  if (g_current == g0) {
34    g_current = g1;
35    g_next = g0;
36  } else {
37    g_current = g0;
38    g_next = g1;
39  }
40
41  return;
42 } // lifeStep()
```

In lines 1–5 the pointers are declared. Lines 7 and 8 define macros that are used to provide a convenient, easily readable way of accessing the elements of the "current" and "next" boards. In the function `lifeInit()`, as shown in lines 16 and 17 above, memory is allocated for `g0` and `g1`. Then, in lines 20 and 21, `g_current` and `g_next` are assigned these values. Also shown above, between lines 28 and 42, is a fragment of code associated with the function `lifeStep()`. This function creates the current generation from the previous generation. Just prior to returning, in lines 33 to 39, this function swaps the memory associated with `g_current` and `g_next`.

Accessing elements of `g_current` and `g_next` requires the calculation of an offset based on the desired row and column. Look at the macro above and count the number of operations involved in the calculation of this offset: there are three additions and one multiplication, thus four integer operations.

Let's assume, we have a grid that is 20 rows by 30 columns, for a total of 600 cells. To update a cell, we need to know its current status and the status of its eight neighbors. Obtaining those nine statuses requires 36 integer operations (four operations for each status). Thus, to update the entire grid requires 36 $\times$ 600 = 21,600 integer operations. That isn't huge and our Game certainly runs as fast as we could possibly want, but there is another approach to accessing the elements.

Instead of using a pointer with a 1D offset to access the memory, for the first part of this assignment we want to write a new version of the code where "multi-subscript" notation is used to specify values in the 2D grid, i.e., we'll use a doubly-subscripted array. Instead of using expressions such as the following to access an element of `g_current`

```
    g_current[(row + 1) * ncol_t + (col + 1)]
```

in the new version of the code we will write

```
    g_current[row + 1][col + 1]
```

Note that we still have two integer operations that are involved in accessing the cell. But, we have eliminated one multiplication and one addition.

In this new implementation, which you'll store in a file called `life1.c`, `g_current` is a pointer to a `char` pointer, i.e., the following declaration is given in the file `life.c`:

```
static char** g_current=NULL;
```

where this is a persistent (static) global variable. We'll say a bit more about memory allocation in a moment. For now, simply assume that a block of memory sufficient to hold the entire grid has been allocated and `g_current` has been associated with this memory such that if we dereference `g_current` twice, we get to the "first" cell in the grid (i.e., cell "0,0"). Said another way, we have that `**g_current` is a character and this character corresponds to what would be written in double-subscript notation as `g_current[0][0]`. Each dereference corresponds to one set of brackets (or subscripts). Shown below are some expressions where, on the left, pointer notation is used, while on the right subscript notation is used:[1]

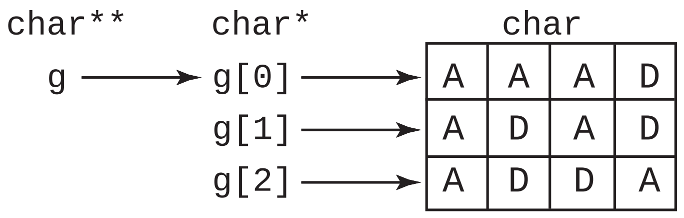
[1]In some of these expressions we are adding an integer to a `char`. That is a valid operation.

```
            **g_current  ⇔  g_current[0][0]
          *(*g_current)  ⇔  g_current[0][0]
        **g_current + 1  ⇔  g_current[0][0] + 1
      **(g_current + 1)  ⇔  g_current[1][0]
  **(g_current + 1) + 3  ⇔  g_current[1][0] + 3
*(*(g_current + 1)) + 3  ⇔  g_current[1][0] + 3
*(*(g_current + 1) + 3)  ⇔  g_current[1][3]
             *g_current  ⇔  g_current[0]
           *(g_current)  ⇔  g_current[0]
       *(g_current + 2)  ⇔  g_current[2]
```

You don't have to look at this very long before you realize you probably want to stick to subscript notation—terms on the right are much easier to read than the terms on the left! But, the important message here is that each subscript is the equivalent of one dereference operation.

If we remove one set of subscripts (or one dereference operation), as was done above in the last three expressions on the right, the result is a pointer to `char` rather than a `char`. So, although `**g_current` is a `char`, `*g_current` is a pointer to a `char` (it contains a memory address that tells us where a `char` can be found—in this case the `char` at the start of the block of memory). Now, it seems natural to ask: For a given value of `i`, what does the pointer `g_current[i]` point to? If we have done our allocation and assignments properly, for any value of `i`, this should point to the start of a row of the associated 2D array.[2]

The following figure serves to illustrate the structure of this array where we assume that the variable `g` is a `char**`. We also assume that we want to work with a 3×4 array of `char`s, i.e., a 2D array with three rows and four columns. To the right of the figure we show the 2D array as we naturally think of where each element of the array is either `A` or `D`. Here we are assuming that all memory has been properly allocated and the pointers point to their proper location.



A more detailed picture of the arrangement of this array in memory is shown in the picture on page six. In this figure the assumed addresses where values are stored is indicated above the respective memory (you may have to expand the view to be able to read these addresses!). Only the last two hexadecimal values for the addresses are shown. The leading `X`'s merely indicate there would be more digits in the addresses than are shown. For the sake of brevity, we are assuming 32-bit addressing so

[2]But, the value of `i` must be less than the number of rows.

that an address occupies four bytes (a byte is indicated by a single rectangle). Characters occupy a single byte. It should be emphasized that this figure (and the one above) indicate the arrangement of things when we dynamically allocate a 2D array. When an array is statically allocated, there are some subtle but important differences from what is being shown here.[3]

The way in which 2D allocation is accomplished was covered in lecture. Here is a quick summary of the steps. Assume we want to allocate a 2D array of characters that has `nrow` rows and `ncol` columns. Further assume that we have a variable `g` that is a `char**`.

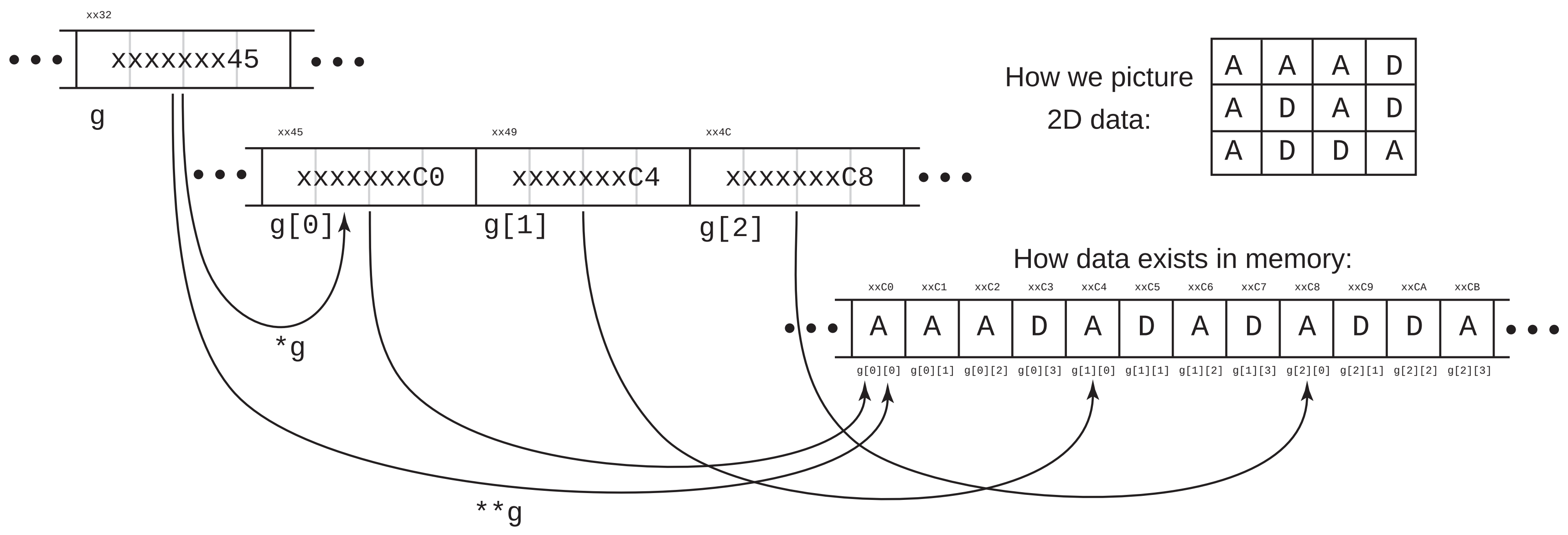
1. Allocate memory for `nrow` character pointers and assign the address of this memory to `g`. Essentially we now have the `g[]` array of pointers—but they don't point to anything yet.
2. Allocate space for `nrow` $\times$ `ncol` characters and assign this address to `g[0]`. (In lecture, a temporary pointer was used for this step. That is not strictly necessary although I think that makes the code easier to read.)
3. For each of the remaining `i` rows, assign the address corresponding to the start of the row to `g[i]`.

For this assignment you must do the following: Create a file called `life1.c`. I suggest you start by copying your `life.c` solution to the previous assignment to `life1.c`. This file should define an identical set of functions as you were required to write for the previous assignment. However, instead of using 1D arrays as in the previous assignment, use dynamically allocated, doubly-subscripts arrays (by dynamically allocated we mean that the memory associated with the arrays is allocated at run-time). To distinguish the functions you must write for this problem from the functions you wrote for the previous assignment, append a "`1`" to each function name. So, for example, you previously wrote the function `lifePrint()`. Change that to `lifePrint1()`. Change `lifeStep()` to `lifeStep1()`, and so on.

From the class Web site you can get the header file (`life1.h`) that should be compatible with your `life1.c`. You can also obtain the associated test and "game launch" program (`life1-test12.c`), but the code in that file is designed to test not only the code you write for this problem, but also the code you must write for the second problem. So, if you just want to test the code for this problem, you can use the `life-test.c` that was used for the previous assignment but update the names of the function calls by appending a `1` to the names.

If things go smoothly (and you are careful), you can solve this problem very quickly!

[3]But those differences are beyond our current concerns and we will not explore them.


xx32
xxxxxxx45
g
xx45
xx49
xx4C
xxxxxxxC0
xxxxxxxC4
xxxxxxxC8
g[0]
g[1]
g[2]
*g
**g
How we picture 2D data:
A A A D
A D A D
A D D A
How data exists in memory:
xxC0 xxC1 xxC2 xxC3 xxC4 xxC5 xxC6 xxC7 xxC8 xxC9 xxCA xxCB
A A A D A D A D A D D A
g[0][0] g[0][1] g[0][2] g[0][3] g[1][0] g[1][1] g[1][2] g[1][3] g[2][0] g[2][1] g[2][2] g[2][3]

Problem 2: Life as a Linked Toroidal Universe

There are many ways in which data structures can be put together (e.g., linked lists, doubly linked lists, binary trees). We have already discussed in lecture singly linked lists as well as stacks.

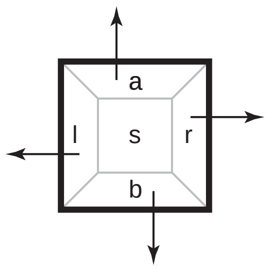
For this problem we will again implement the functions for the Game of Life. But, instead of using characters in a 1D or 2D array, each cell in the Life universe will be a Cell structure. The Cell will contain a character where its status is stored (i.e., a character indicating whether the Cell is alive or dead). Additionally, the Cell will contain four pointers to Cells. These pointers will point to the neighbors that are above, below, and to the left and right.

The appropriate code to define a Cell structure is:

```c
typedef struct cell* CellPtr;

typedef struct cell {
  char s;               // status: alive or dead
  CellPtr a, b, l, r;   // above, below, left, right
} Cell;
```

A depiction of one of these Cells is shown below where the arrows correspond to pointers and the central `s` is the status of the cell:

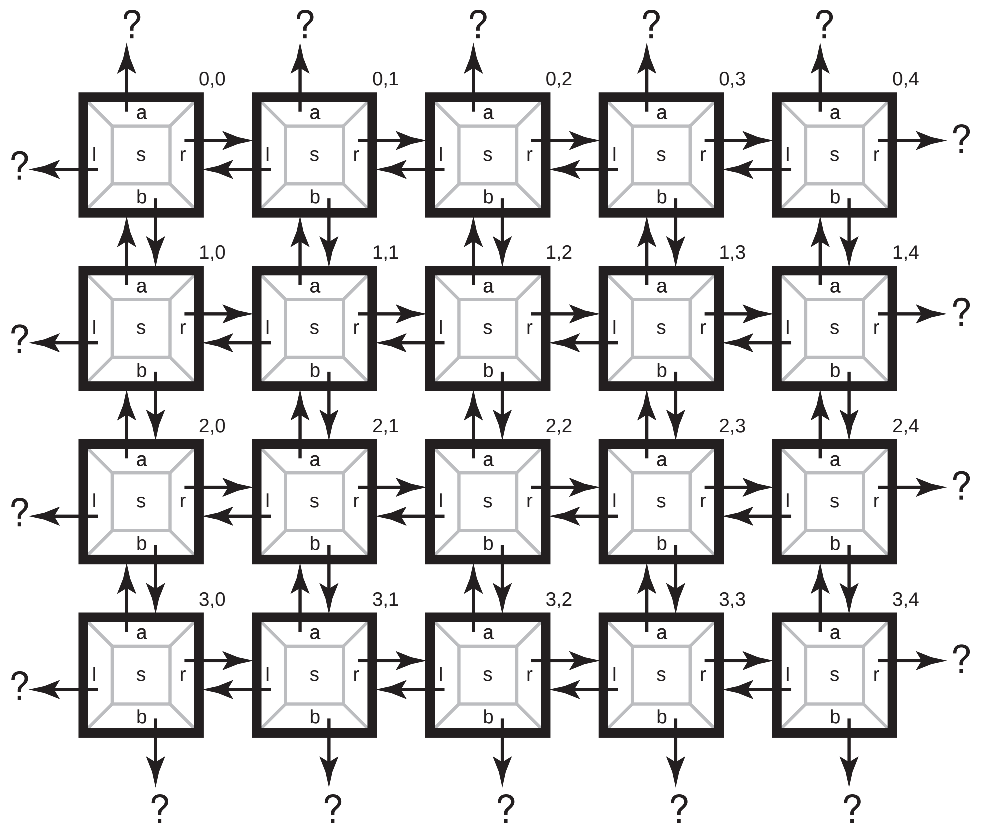


Assume we have a pointer to a Cell, that we'll call `cPtr`, that is pointing to a properly allocated and initialized Cell. Further assume that the Cell to which `cPtr` points is part of the grid of Cells which constitutes the game of Life. We can get the status of this cell, i.e., the character that tells us if the cell is alive or dead, by writing `cPtr->s`. If we want the status of the neighbor above this cell, we would write `cPtr->a->s`. If we want the status of the neighbor below and to the left, we could write either `cPtr->b->l->s` or `cPtr->l->b->s`.

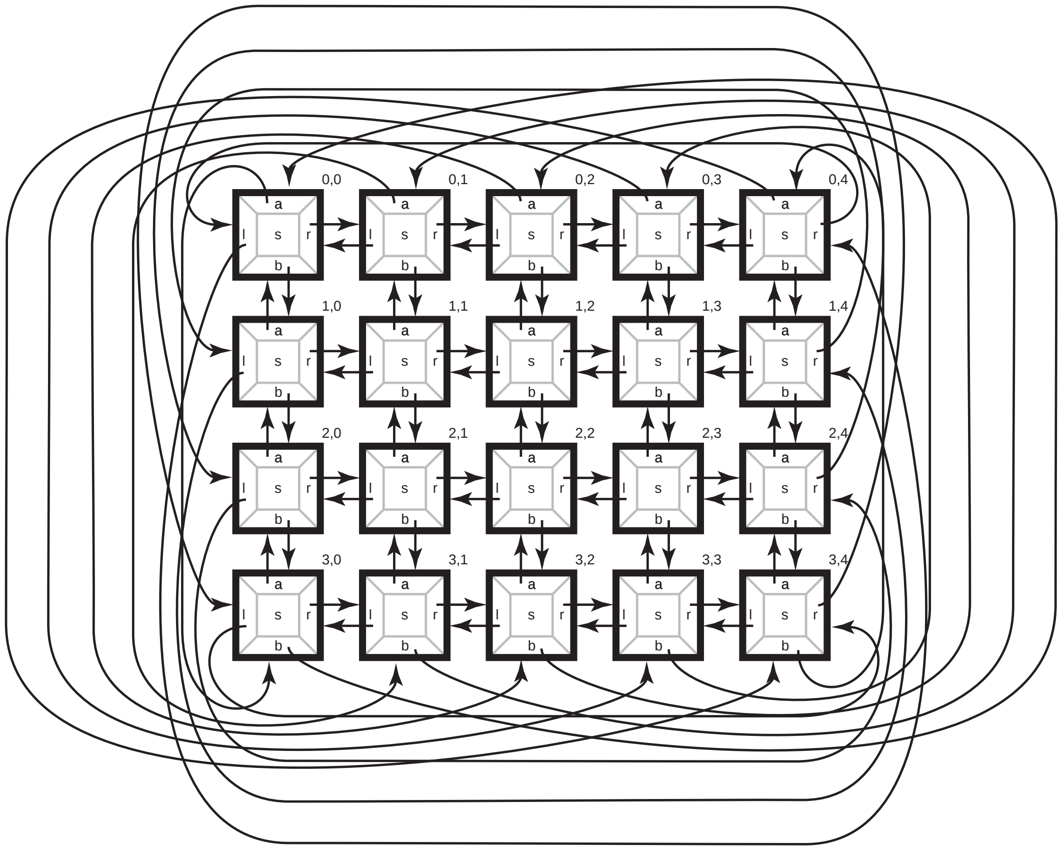
We want to implement the Game of Life using a collection of these Cells. I will provide various details concerning my implementation. Your implementation details are up to you—you don't have to do things the way I did—but the functions you ultimately write must provide the functionality of the Life functions you wrote for the previous assignment. Furthermore, the grid of Cells must form a toroidal universe as described below.

We can start with just two Cell pointers that serve as the “origins” for the current and the next generations. When we initialize the game, the first thing to do is to allocate memory for these origins and make the pointers point to them. Then, we build the grid one cell at a time, row by row, until we have all the cells we need. As we add cells we need to ensure they are, to the extent possible, properly linked to their neighbors. However, we cannot complete all the linking until all the Cells have been allocated.

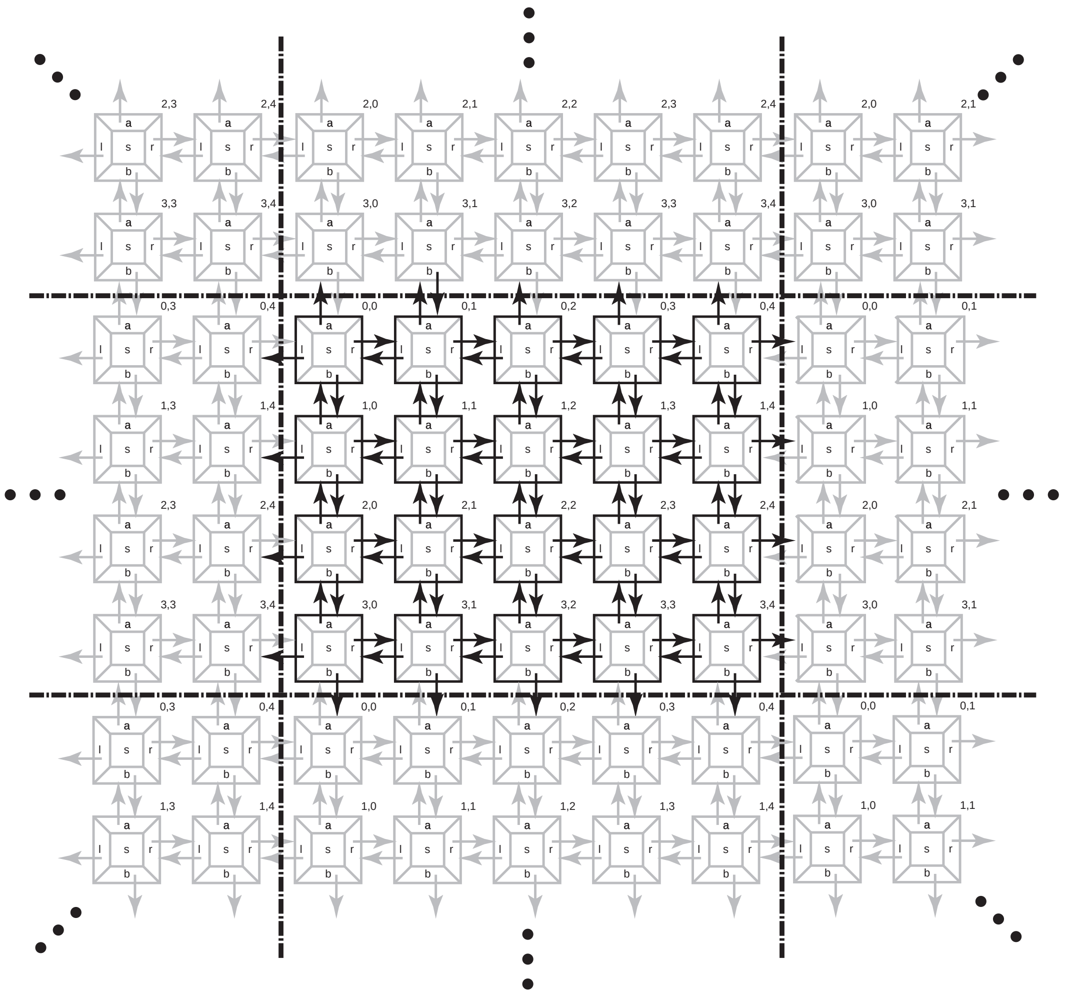
As I build the first row, I can do the left-right linking, but not the above-below linking (since there is nothing above or below). However, once I start adding a new row I can do the left-right linking as well as the up-down linking. Depending on your initial construction, you might end up with something that looks like the following collection of Cells:



What should be done about Cells on the edges? We could include a dead-zone buffer as before, but instead we will implement the toroidal universe mentioned in the previous assignment. In the previous assignment the toroidal behavior would be realized with various conditional statements. However, with the linked Cells that we are now using, we can actually realize the toroidal behavior in a much more elegant way. By doing this, there is no distinction between the interior of the grid and the edge of the grid, because there is no edge! What we will do is link Cells in such a way that the neighbor to the right of the “right-most” Cell in a given row is the “left-most” Cell in that row. The Cell above the “top-most” Cell in a column is the “bottom-most” Cell in that column. And so on. This circular (or toroidal) linking is depicted in the following figure.



This figure is cluttered but you should be able to trace some of the lines to get a feel for the linking. Alternatively, we can depict the resulting Life universe in another way by simply repeating the grid over and over again, linking to itself. That representation is shown in the following figure, where one complete group of `nrow` × `ncol` Cells is shown in back and the surrounding copies are shown in gray. In this figure, and in the previous two, the “origin” corresponds to the Cell labeled “0,0.”



Note that in this implementation we cannot take row and column indices and locate a Cell by plugging these values into an array or use them to calculate a pointer offset. All we have is an origin and a bunch of linked Cells! Nevertheless, we will continue to think in terms of row and column indices (such indices have been included in the figures above). Also, it can be useful in an implementation to access individual Cells based on a row and column index. We will return to this point in a moment, but first let us discuss memory allocation.

In my current solution I again define four global, `static` pointers. One of the pointers, `g0`, always points to one particular Cell (the memory for which must be allocated at run-time) and the other pointer, `g1`, points to a different Cell (which also must be allocated at run-time). These Cells serve as the origins for two different grids. I also have two other Cell pointers, which I call `g_current` and `g_next`, that are used to point to the origins of the current and next generations. The Cells to which these pointers point are swapped at each step of the game. (This is no different from what was done before!) Thus, at the start of my solution I have the following declarations:

```
static CellPtr g0=NULL;
```

```c
static CellPtr g1=NULL;

static CellPtr g_current=NULL;
static CellPtr g_next=NULL;
```

You must create a file called `life2.c` that contains all the code necessary to realize the Life functions. It is the task of the function `lifeInit2()` to ensure that the pointers described above ultimately point to "origins" that are properly linked to all the necessary Cells to form the grids. You will probably find that 90 percent of the challenge of this problem lies in the implementation of this one function. Actually that may not necessarily be true. In my solution I wrote a new function called `grid_creation()`. The prototype for this function does not appear in the header file that accompanies this code, i.e., it does not appear in `life2.h`. Instead, this function is just for local use within the `life2.c` file and is used create a linked grid when passed a pointer to an origin. I call this function once using the `g0` origin and once using the `g1` origin. The "heavy lifting" has been moved to this function so that we can call it once for each grid. The following is the prototype for the `grid_creation()` function and the entire `lifeInit2()` function in my `life2.c` file:

```c
// Prototype for grid_creation().  The static qualifier ensures
// this function is not visible outside of this file.
static void grid_creation(CellPtr grid);


void lifeInit2(int the_nrow, int the_ncol) {
  nrow = the_nrow; // nrow and ncol are global variables.
  ncol = the_ncol;

  // Check if the grids were previously initialized.
  // Free memory if previously initialized.
  if (g0 != NULL) {
    free(g0);
    g0 = NULL;
  }
  if (g1 != NULL) {
    free(g1);
    g1 = NULL;
  }

  // Obtain "origins" for the two grids.  BUT, also allocate space for
  // the entire mesh at the same time.
  g0 = (CellPtr)calloc(nrow * ncol, sizeof(Cell));
  g0->s = ' ';
  g1 = (CellPtr)calloc(nrow * ncol, sizeof(Cell));
  g1->s = ' ';

28    if (g0 == NULL || g1 == NULL) {
29      printf("lifeInit2: memory allocation failed.  Terminating...\n");
30      exit(-1);
31    }
32
33    grid_creation(g0);  // Create the g0 grid.
34    grid_creation(g1);  // Create the g1 grid.
35
36    g_current = g0;
37    g_next = g1;
38
39    return;
40  } // lifeInit2()
```

Notice in lines 17 and 18 that I've used `calloc()` rather than `malloc()` to allocate memory. The two functions are similar except `calloc()` returns a pointer to cleared memory and takes two arguments instead of one (the arguments correspond to the number of elements to allocate and the size of a single element).

Returning to the point raised above about accessing a cell based on the row and column indices, there is one more local function I wrote that proved useful in my solution. When given an "origin" and what are effectively the row and column indices for a desired Cell, this function returns a pointer to that Cell. I called this function `get_cell()` and it is shown in its entirety below. This function works by starting at the origin (setting `cPtr` to the argument `g` as shown in line 3). Then, for however many times as specified by the column index `col`, the for-loop on line 5 resets `cPtr` to be its neighbor to the right. Effectively we walk `cPtr` to the right `col` times. (In my code `ncol` is a global variable corresponding to the total number of columns. In `get_cell()`, `col` is a local variable that is between 0 and `ncol-1`.)

```
1   CellPtr get_cell(CellPtr g, int row, int col) {
2     int i, j;
3     CellPtr cPtr = g;  // set cPtr to "origin"
4
5     for (j=0; j<col; j++)
6       cPtr = cPtr->r;
7
8     for (i=0; i<row; i++)
9       cPtr = cPtr->b;
10
11    return cPtr;
12  } // get_cell()
```

As mentioned previously, once you have a pointer to a Cell, you can easily inspect the status of its neighbors using expressions such as `cPtr->a->r->s`. If you do write your own version of `get_cell()`, you should call it rather sparingly. Nevertheless, this function does allow us to recycle much of the Life code that was written for the previous implementations.

For this problem you must create a file called `life2.c` that defines functions corresponding to all the Life functions that you wrote for the previous assignment. Implement the grids as a collection of linked Cells as described above. To distinguish the functions you must write for this problem from the functions you wrote for the previous assignment, append a “2” to each function name. So, for example, you previously wrote the function `lifePrint()`. Change that to `lifePrint2()`. `lifeStep()` becomes `lifeStep2()`, and so on.

From the class Web site you can get the header file (`life2.h`) that must be compatible with your `life2.c`. You should also obtain the associated test and “game launch” program `life-test12.c`. Ensure that this program works with your code. The menu that is provide allows you to toggle between using the functions in `life1.c` and `life2.c`. (As explained in the help listing that the program prints, you accomplish this toggling by entering “`t`.”)