Minesweeper in C
Part 2

When the player of the game is ready to play, they will use the "new" command to create a board of the desired dimensions – rows, columns, mines – and play. We've covered how to build the board but we haven't finished setting it up with all the required cell attributes. In Part 2 we finish setting up the board so that is ready to play the game.

For your reference, here are the commands to be implemented. So far only "new" and "show" have been implemented.

Command	**Example**	**Meaning**
"new" (Part 1)	"new 5 10 10"	Create new board with 5 rows, 10 columns, 10 mines
"show" (Part 1)	"show"	Show or display current board
"quit" (Part 1)	"quit"	Quit the app
"uncover" (Part 3)	"uncover 3 5"	Uncover cell at row 3 column 5
"flag" (Part 3)	"flag 2 7"	Put a flag on cell at row 2 column 7
"unflag" (Part 3)	"unflag 1 4"	Remove flag from cell at row 1 column 4

Recall that the "command_new()" function is called whenever the player types in a "new" command:

"new 15 20 30"

In Part 1 we covered how to tokenize the command and build a dynamically allocated board of the correct size. But the preparation of the board for game play wasn't complete, we didn't place the mines or calculate the adjacency counts. The original command_new() just built the board:

```
void command_new() { // from Part 1
        // build the board based on requested # of rows and cols
        board = (cell **) malloc( sizeof (cell *) * cols);
        for (int i=0; i<rows; i++) …
        // initialize cells
        for (int i=0; …)
                for (int j=0; …)
                        init_cell(&board[i][j]);
}
```

In Part 2 we add code to command_new() to randomly position mines and calculate adjacency counts

```
void command_new() { // for Part 2
        // build the board based on requested # of rows and cols
        …
        // initialize cells
        …
        // New:  add mines to cells and distribute them randomly
        // New:  calculate adjacency counts for cells that are not mined
}
```

Random Number Generator

The C libraries provide a function called rand() that returns a random value. We need it to provide randomly generated values of row and column on the board. For example if the board has 10 rows (numbered 0 to 9) we might want to pick a row at random by generating a random number from 0 to 9 inclusive. Here's the code for a function **int get_random(int range)** that uses rand() to create and return a random integer on the interval 0 to range-1 inclusive.

```
int get_random(int range) {
      return ((int)floor((float)range*rand()/RAND_MAX))%range;
}
```

We scale the value returned by rand() by dividing it by its maximum value RAND_MAX to get a random float value between 0.0 and 1.0. We multiply it by range, convert it to an int, then compute the result modulo range, leaving a random value from 0 to range-1.

You may need to add

```
#include <stdlib.h>                <- standard library header
#include <math.h>                  <- math library header
```

to your source file to support the rand() and floor() functions.

Initializing the Random Number Generator

Most random number generators create numbers that look random but technically aren't truly random. Generation of truly random numbers on a computer is not simple. Instead, most random number generators use an algorithm called pseudo-congruential random numbers, which creates a large list of numbers that look random and even satisfy some mathematical tests for randomness. But the list of numbers actually defines a cycle. Generate enough numbers from the list and the cycle will wrap around to the beginning and the numbers will start to repeat. As long as the cycle is large compared to the number of random numbers we need, this might be good enough.

If you use a pseudo random number generator, you will probably want to initialize the generator to start at a different point in the cycle each time you run your program. If we don't bother with initialization, then each time we run the program we get exactly the same series of "random" numbers each run, and the results won't look randomized.

At some point in the program during the initialization stages, before we actually use the random number generator – for example, near the beginning of the main() or rungame() function -- we add the following function call, positioned so that it gets called once per application execution and before we generate any numbers with rand():

```
srand(time(0));
```

The name of the srand() function stands for "seed the rand() function". Seeding a random number generator means initializing it so that it starts at a specific point in its cycle. The argument to srand() is itself another function call. The time(0) function call reads the system clock and uses a few of the lower precision bits of the current system time as a seed or starting point for the random number generator. The system clock will have a different time each time the time(0) function is called, and the value returned will also look somewhat random. By using this as the seed value, we will get a series of random numbers that look sufficiently random to support our program.

Add Mines To Board

After the "new" command builds the board, all the cells are initialized with their "mined" property set to 0 (false). In this step we take the number of mines requested by the user and change the "mined" property of requested number of cells to 1 (true). We say that those cells are "mined" or "contain a mine". The location of the mines should appear randomly distributed and different for each game if the game is to be interesting to play.

The player will pick the number of mines to be randomly distributed around the board. What that number should be is up to the player. The minimum number of mines is 0 or 1 (not a very interesting game) and the maximum is all cells on the board which is # rows times # cols (also not interesting). Use some rule of thumb, like # of mines = 20% of # of cells. There is no exact formula to use here, feel free to experiment, too few or too many mines will make the game uninteresting, and some choices make the game too hard to solve. But it is important to randomly distribute them. We will need a random number generator to help us pick rows and columns on the board at random for mine placement.

How to Randomly Distribute the Mines

Approach #1
For each mine to position, generate a random row,col pair and place a mine there assuming it doesn't already contain a mine. If it does contain a mine, pick another row,col pair until you find one that doesn't already contain a mine. Repeat this process until the required number of mines have been generated.

Algorithm

```
For m = 1 to # mines                          // loop for each mine
        r = random row (must be from 0 to rows-1)
        c = random col (must be from 0 to cols-1)
        while cell at r,c already contains a mine
                r = random row                // loop until you find a row,col
                c = random col                // that doesn't already have a mine
        end while
        place mine in cell at r,c
end for
```

The outer for loop counts the required number of mines. But for each pass through the for loop, a nested "while" loop is required to ensure you randomly locate a cell that doesn't already contain a mine. The while loop keeps going until you find a random cell at row,col that doesn't already have a mine. If you don't add the while loop, then you may wind up assigning multiple mines to the same cell and wind up with fewer mines than specified. This approach will work fine so long as the number of mines is a small percentage of the total number of cells, which will normally be the case for our game. If the number of mines is large compared to the number of cells, then the "while" loop may run for a long time before finding a cell that doesn't already contain a mine, especially in the later stages of the "for" loop. As the board fills up with mines, it takes more and more work to find a cell without one.

Approach #2

Initially fill the first so many cells in the board with mines. In other words, start at location 0,0 and go down the rows in order setting each cell to contain a mine until you have the desired number of mines. In other words, we initially pack the first few rows with all the mines packed together. This ensures that the board contains the correct number of mines even though they're not randomly distributed yet. Then "shuffle" the locations of the mines to randomize them. You have to pick a "shuffle" number that is large compared to the number of mines.

Algorithm

Initial Placement of Mines: pack the mines into the first few rows (not randomized so far)

```
For p from 0 to # mines – 1
        r = p/#cols             // r = p divided by # of columns
        c = p%#cols             //c = p modulo # of columns
        Set cell at r,c to mined = 1
End for
```

Shuffle: randomly distribute mines around the board

```
For s = 1 to # of shuffles
        r1,c1 = random row,col
        r2,c2 = random row,col
        x = value of "mined" for cell r1,c1 (either 0 or 1)
        y = value of "mined" for cell r2,c2 (either 0 or 1)
        Set cell at r1,c1 to mined = y
        Set cell at r2,c2 to mined = x
End for
```

The shuffle approach has the advantage that even if the number of cells with mines is large, we don't waste time randomly searching for a cell without a mine. But the randomness of the mine positions depends on number of shuffles or exchanges, which must be large compared to the number of mines. If the # of shuffles is too small, the mines won't appear random. If # of shuffles is too large, the mines will be randomized, but we will just waste time on some shuffles that don't contribute any more randomness.

Mine Adjacency Counts

If a cell is a mine, then the game rules are to figure out that a cell is a mine without uncovering it and instead put a flag on it without uncovering it. If a cell it not a mine, it is okay to uncover it. Once a cell that is not a mine is uncovered, the display of the cell will show its adjacency count. Adjacency count is a number from 0 to 8 that indicates how many of the newly uncovered cell's neighbors are mines. The player uses the adjacency counts to make inferences and figure out which cells are mines and flag them without actually uncovering them.

Neighbor Cells

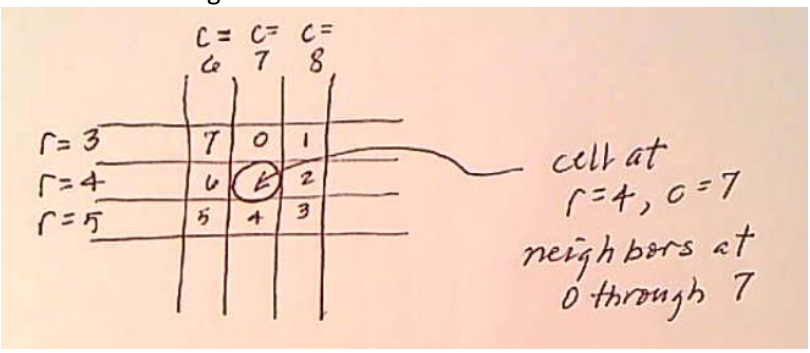
For any given cell, figuring out where are its neighbors is based on the row and col location of the original cell. For example, the cell at row=3, col=5 has 8 neighbors:

Direction	Row	Col
North	3 – 1 = 2	5 + 0 = 5
Northeast	3 - 1 = 2	5 + 1 = 6
East	3 + 0 = 3	5 + 1 = 6
Southeast	3 + 1 = 4	5 + 1 = 6
South	3 + 1 = 4	5 + 0 = 5
Southwest	3 + 1 = 4	5 – 1 = 4
West	3 + 0 = 3	5 – 1 = 4
Northwest	3 – 1 = 2	5 – 1 = 4

More generally, starting from the original location r,c for some cell, the row,col pair for each of its neighbors can be found with the following formulas:

Direction	r2, c2	Meaning
North	r-1, c	One row up, same column
Northeast	r-1, c+1	One row up, one column right
East	r, c+1	Same row, one column right
Southeast	r+1, c+1	One row down, one column right
South	r+1, c	One row down, same column
Southwest	r+1, c-1	One row down, one column left
West	r, c-1	Same row, one column left
Northwest	r-1, c-1	One row up, one column left

Here's a sketch of the neighbors for the cell at row = 4 and column = 7:



The neighbors are indexed from 0 to 7, starting with 0 in the "north" position, then increasing the index in a clockwise order until we get to index 7. There is nothing special about starting at "north" or in moving clockwise to index the neighbors. Any numbering system would work so long as you're consistent about which direction each index represents.

We can implement these neighbor offset values by creating arrays that hold the row offset and column offset for each direction. For example, declare the following variables, either as globals or as local variables inside the function that calculates adjacency counts:

```
int neighborcount = 8;
int rowneighbors[] = {-1,-1,+0,+1,+1,+1,+0,-1};
int colneighbors[] = {+0,+1,+1,+1,+0,-1,-1,-1};
```

The row/col offsets at index 0 are -1 and 0, which corresponds to “north”. The offsets at index 1 are -1 and +1 which corresponds to “northeast”, and so on.

Given the current cell is at row=r and col=c, we can locate its neighbors by looping over all possible directions 0 through 7 and adding the row and column offsets for each direction as shown in the arrays:

```
for (int d=0; d<neighborcount; d++) {
      int rn = r + rowneighbors[d];            // row for direction d
      int cn = c + colneighbors[d];            // col for direction d
      …
}
```

So rn,cn is the location of the neighbor of r,c in direction d. We still have to confirm that rn,cn are valid indexes (>= 0 and < # of rows or # of cols).

```
if (0 <= rn && rn < rows && 0 <= cn && cn < cols) …
```

If the neighbor indexes are valid we check if that neighbor has a mine or not:

```
if (board[rn][cn].mined == 1) …
```

If it contains a mine, we increment our adjacency count for r,c. After looping over all possible directions, finding all valid neighbors and counting the mines they contain, we set that count to be the adjcount value for the r,c cell in the middle:

```
board[r][c].adjcount = minecount;
```

Final Adjcount Calculation

Now that we know how to find all the neighbors of a given cell, we can give each cell on the board that doesn't contain a mine its mine adjacency count (how many of its neighbors contain a mine, a value from 0 to 8).

Algorithm

```
For r = 0 to # rows – 1
    For c = 0 to # cols – 1
        minecount = 0
        For d = 0 to 7 (all directions)
            r2, c2 = neighbor in direction d
            if r2,c2 is valid (on the board) and contains a mine
                minecount ++
            end if
        end for
        set adjcount for cell at r,c to minecount
    end for
end for
```

Displaying Cells

Our temporary version of display_cell() for Part 1 only displayed its "position" property. That was useful for Part 1 development, but now we are ready to code a more realistic one based on its properties. Initially we will set all cells to uncovered so that we can see where all the mines are, the adjacency counts, and confirm they are correct. So we will set the covered and flagged property of each cell to false for now, until we have completed our testing.

For each cell we will display a character like "*" if the cell contains a mine. If it doesn't contain a mine, we will display its adjcount value, with one exception. Since most of the cells on the board don't contain a mine and are not adjacent to any mines, they will display with a "0". To avoid cluttering up the board with a bunch of zeros, we will display a dot "." for cells that are not mined and that have an adjcount of 0. You can experiment with other characters to print for various states to see which one looks best. Our updated version of the display_cell() function is now:

```
display_cell(cell *c) {
      if (c->mined == 1) printf("%2s", "*");
      else if (c->adjcount == 0) printf("%2s", ".");
      else printf("%2d",c->adjcount);
}
```

This is not the final version of display_cell(). The complete version will also have to check the "flagged" and "covered" properties for a cell to determine how to print it. But that will be handled in Part 3. For now, we have all the parts in place to complete Part 2.

Cell Initialization

To simplify testing, we will use several patterns for initializing the properties of cells in the board when they are first created by command_new(). For now we will use the following initialization:

```
init_cell(cell *c, int p) {
      c->pos = p;                  // each cell knows its pos in board
      c->mined = 0;                // mined initially false
      c->adjcount = 0;             // adjcount initially 0
      c->covered = 0;              // covered initially false
      c->flagged = 0;              // flagged initially false
}
```

After all the cells have been created and initialized, the code to position mines at random cells and to calculate adjacency counts will still run and will perform additional updates to the "mined" and "adjcount" fields.

In the actual game, the "covered" field of all cells will be set to 1 (true), so that the player cannot see what's in a cell until it's uncovered (there wouldn't be any point to the game if all the cells are uncovered at the start!). But for Part 2, we actually do need to set all cells to uncovered or "covered" = 0 (false) so that we can see our work and confirm that random mine placement and adjacency count calculations are correct. Once we get to Part 3, we'll change the init_cell() function to set "covered" = 1 (true) so that the content of all cells is initially hidden at the start of the game. We'll add commands to the CLI to "uncover", "flag", and "unflag" individual cells. The player can then pick which cells to uncover and which to flag without uncovering to play the game.

Trace of Running Program

Here are a some traces of how the running program should look at the end of Part 2. Remember to use a fixed-width font like Times or Consolas to display the board. Otherwise the columns will not be printed in correct alignment.

```
>> new 5 10 5
New Board Command
>> show
Show Board Command
  .  .  .  1  *  2  *  1  .  .
  1  1  1  1  1  2  1  1  .  .
  2  *  2  .  .  .  .  .  .  .
  2  *  2  .  .  1  1  1  .  .
  1  1  1  .  .  1  *  1  .  .

>> new 10 20 20
New Board Command
>> show
Show Board Command
  .  .  .  1  *  2  1  1  .  .  .  .  .  .  .  1  *  3  2  1
  .  .  .  1  1  2  *  1  .  .  .  .  .  .  .  1  2  *  *  1
  .  .  .  .  .  1  2  2  1  .  .  .  .  .  .  .  1  2  2  1
  .  .  .  .  .  1  2  *  1  1  1  1  .  .  .  .  .  .  .  .
  .  1  1  2  1  2  *  2  1  1  *  1  .  .  .  .  .  .  .  .
  1  2  *  2  *  2  1  1  .  2  2  3  1  1  .  1  1  1  .  .
  1  *  2  2  1  1  .  .  .  1  *  3  *  1  .  1  *  1  .  .
  2  2  2  .  1  1  1  .  .  1  2  *  3  3  1  2  1  1  .  .
  1  *  1  .  2  *  2  .  .  .  1  2  *  2  *  1  .  .  .  .
  1  1  1  .  2  *  2  .  .  .  .  1  1  2  1  1  .  .  .  .

>> new 20 30 50
New Board Command
>> show
Show Board Command
  .  .  .  1  *  1  .  1  1  1  .  .  .  .  .  .  .  .  .  .  .  1  *  1  .  .  .  .  .  .
  .  .  .  1  2  2  1  1  *  1  .  .  .  .  .  .  .  .  .  .  .  1  1  1  .  .  .  1  1  1
  .  .  .  .  1  *  2  2  1  1  1  1  1  .  .  .  .  1  1  1  .  1  1  1  .  1  1  3  *  2
  .  .  1  1  2  2  *  1  .  .  2  *  2  .  .  .  1  2  *  2  1  1  *  2  1  1  *  3  *  2
  .  .  2  *  2  1  1  1  .  .  2  *  2  .  .  .  1  *  4  *  2  1  2  *  1  1  1  2  1  1
  .  .  2  *  2  .  .  .  .  .  2  2  2  .  .  .  1  1  3  *  2  .  1  1  1  .  .  .  .  .
  .  .  1  1  1  .  .  .  .  .  1  *  1  .  .  .  .  .  1  1  1  .  .  .  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  1  1  1  .  .  .  1  1  2  1  2  1  1  .  1  1  1  .  .  .
  1  1  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1  *  2  *  2  *  1  .  1  *  1  .  .  .
  *  1  .  1  1  1  .  .  .  .  .  .  .  .  .  .  1  2  2  2  1  2  1  1  .  1  1  1  .  .  .
  1  1  .  1  *  1  .  .  .  .  .  .  .  .  .  .  1  *  1  .  .  .  1  1  1  .  .  .  .  .  .
  .  .  .  1  1  1  .  .  .  .  .  .  .  .  .  .  1  1  1  .  .  .  1  *  1  .  .  .  .  .  .
  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  1  1  1  .  .  .  .  1  1  1  .  1  2  2  1  .
  .  .  .  .  1  1  1  .  .  .  .  .  .  .  .  1  *  1  .  .  .  .  .  .  .  .  1  *  *  1  .
  1  1  1  .  1  *  1  .  .  .  .  1  1  1  1  1  2  1  1  .  .  .  .  .  .  1  2  4  3  2  .
  2  *  2  .  1  2  2  1  .  .  .  1  *  1  .  .  1  *  2  1  1  1  1  1  1  *  2  *  1  .
  2  *  2  .  .  1  *  1  .  .  1  2  2  2  1  1  1  1  2  *  1  1  *  2  2  2  2  1  1  .
  1  1  1  .  .  1  1  1  .  .  1  *  2  2  *  2  1  .  1  1  1  2  2  3  *  1  .  .  .  .
  1  1  1  .  .  .  .  1  1  1  1  2  *  3  3  *  1  .  .  .  .  1  *  3  2  2  .  .  .  .
  1  *  1  .  .  .  .  1  *  1  .  1  1  2  *  2  1  .  .  .  .  1  1  2  *  1  .  .  .  .
>>
```

In Part 3, we will add commands for uncover, flag, and unflag, and set the initial state of all cells to "covered = 1". One more complex algorithm to be implemented before we're ready to play has to do with how to uncover a cell. Here are the possible situations when uncovering a cell.

- Cell contains a mine:
    - game is over, you lose
- Cell contains an adjacency count > 0:
    - simply uncover that one cell
- Cell contains an adjacency count == 0:
    - we must uncover not only the original cell, but we must recursively find all the neighbors of that cell, and neighbors of the neighbors, etc., that also have an adjacency count of 0, and uncover them all at the same time as a group in one operation. The game will take too long to play without this little optimization. It's an interesting problem that we will solve in Part 3. Think of ways to solve it in the mean time.