Minesweeper in C
Part 1

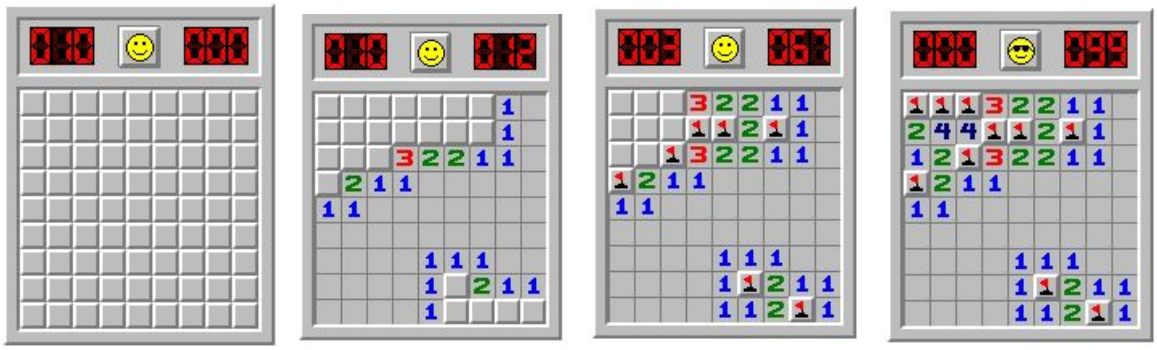
This project is an exercise in C programming to create a version of the **classic Microsoft board game Minesweeper**. It uses no graphics libraries, but instead provides a text-based command-line interface or CLI, and displays the two-dimensional board textually with a monospaced font. We will build the app in several parts that all integrate together at the end to implement the finished program. In Part 1, we implement the CLI, plus two user commands: the "new" command to create a new board, and the "show" command to display the board.

How To Play

You don't have to be an expert player to code the game, but knowing the basic rules is recommended. The game starts by displaying a board of cells that are all initially covered. Some of the cells have hidden mines, others are empty. The goal is to uncover all the empty cells and plant a flag on all the cells with mines without uncovering them. A left click on a cell uncovers it, and once a cell is uncovered it can't be covered. A right click of a covered cell plants a flag on it without uncovering it. Another right click removes the flag. To uncover a flagged cell you first have to remove the flag. All empty cells maintain an integer attribute called the adjacency count that indicates how many mines are adjacent to it. Counting left, right, up, down, and all diagonals, the attribute runs from 0 to 8. After an empty cell is uncovered, if its adjacency count is 0, it displays as empty; if the attribute is 1-8, it displays that number. Your job is to use the adjacency counts of the uncovered cells to figure out where the mines are and plant a flag on them without uncovering them. If you uncover a mined cell, you lose and the game is over. If you uncover all the empty cells and plant flags on all the mines, you win. To speed the game along, if you click on an empty cell with adjacency count zero, that cell is usually in the middle of a big area of other adjacency count zero cells and all the adjoining empty cells are uncovered as a group, which is much faster than the user having to uncover them individually. At any time the user will want to know how many mines remain to flag. That count is displayed in a counter at the upper left. Note that the counter drops whenever you place a mine, but this only counts how many flags you have planted, it is no guarantee that you have planted the flag in the right place (on a mine). You can customize the game by making the board bigger or changing the number of mines.

The minesweeper game came bundled with Microsoft Windows for many years. It's no longer provided automatically, but you can find many versions on the web. An online version is available at https://minesweeperonline.com/#beginner. There's a rumor that Bill Gates was such a Minesweeper addict that coworkers staged occasional interventions to get him to important meetings on time.

Here are a couple of screen captures from the website at the above link.



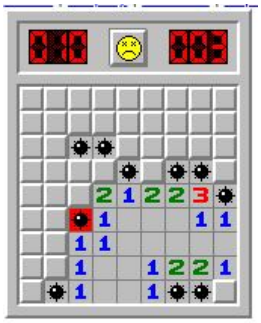
The first figure shows the initial board will all cells covered and the number of mines to flag is 10.

The second figure shows the board after the user starts the game and enters a left click in the middle of the big empty area. All empty cells adjacent to the one cell actually clicked are uncovered at the same time. Since the user has no idea where the mines are, the location of the first left click is a guess.

The third figure shows the board after the user has planted a few flags with right clicks on the cells based on the adjacency counts and has uncovered a few additional empty cells. Seven flags have been planted, and the remaining mine count in the upper left is 3.

The fourth figure is the board after the user has won (smiley face with sunglasses).

In case you want to see what happens when you lose, see the figure below. It shows the board if the user left clicks (uncovers) a cell that contains a mine. The uncovered mine is displayed with a red background, the locations of the other mines are revealed, and the game ends with a loss (frowney face).



What will the C version that we implement look like? Here are a couple of examples from a completed implementation.

```
PS C:\Users\rick\> ./minesweeper
>> new 10 20 50
>> show

 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /

>> uncover 3 1
>> show

 .  .  1  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 .  1  2  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 .  2  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 1  3  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /  /

>>
```

Without using a graphics library and entering clicks via the mouse, all game play must be implemented by textual commands from the keyboard. It makes play more tedious but still preserves the full logic of the game. Note that this trace is from the completed app, but in Part 1, we only implement a subset of the features. See the example shown in the **Example Run** section at the end of the document to get an idea of the state of the app at the end of Part 1.

Major Functional Blocks

Implementation of the complete program includes the following functional blocks, only a subset of which are in Part 1. Part 1 functions are displayed in bold. Other functions will be added in later assignments.

- **Command Line Processor or CLI**
  - **Read in keyboard input and interpret the command it represents**
- **Commands Part 1**
  - **New: Build New Board**
    - **Dynamic Memory Allocation for Board**
    - Randomization of Location of Mined Cells
    - Calculation of Adjacency Counts for Non-Mined Cells
  - **Show: Display Board**
- Commands Part 2
  - Uncover: Uncover Selected Covered/Unflagged Cell
  - Flag: Place Flag on Selected Covered/Unflagged Cell
  - Unflag: Remove Flag From Selected Covered/Flagged Cell
- Command Validation
  - Confirm name of command, number/type/range of command arguments
- Recursive Uncover: whenever the user uncovers a non-mine with adjacency count = 0, to speed up play, the game will automatically and recursively uncover all adjacent non-mine cells, up to one layer of surrounding non-mines with adjacency counts > 0.
- Improved Board Display
  - Add row/column headers
- Win/Loss Detect
  - Loss: User uncovers a mine
  - Win: User flags all mines and uncovers all non-mines

In the remaining sections, we will design the CLI and two of the commands to complete Part 1.

CLI: Command Line Interpreter

The CLI is driven by a prompt-response loop that prompts the user for the next command, then reads in the response from the user, parses it, and executes it. The loop then returns and prompts for the next command. Initially we will skip validation and error detection, but will return to it in a later section.

Since there is no native "string" datatype in C, C provides "array of char" as a substitute. No matter how you handle input, you will need an "array of char" variable to hold string data. A one-dim array will hold one string, a two-dim array will hold a collection of strings (each row holds one string). To input a string from the keyboard into an array of char, there are two basic styles:

Style 1: Use the library function "scanf"

```
char line[80];    // declare array of char named "line"
scanf("%s",line); // read data of type "%s" (for string) into "line"
```

Style 2: Use the library function "fgets"

```
char line[80];          // declare array of char named "line"
fgets(line,80,stdin);   // read max of 80 chars into "line"
```

The first reads text from the keyboard buffer until a **terminating whitespace char** is reached in the input. The second reads all text from the keyboard buffer from the beginning until a **terminating newline character** is reached. The first is simple but inflexible, especially regarding error detection and recovery. The second is complex but more flexible. For example, one command input that defines the dimensions of a new board is

"new 10 20 25"

which the player enters to request a new board with 10 rows, 20 columns, and 25 mines. We could capture this input using style 1 as follows:

```
char command[80];
int rows;
int cols;
int mines;
scanf( "%s%d%d%d", command, &rows, &cols, &mines );
```

This is the simplest solution and will work fine so long as the player makes no input errors. But in case of errors, which is always the case, there is no easy way for the program to recover from the error and continue.

The other approach is to capture the entire input line as a string, then break it down into pieces called tokens, and convert tokens that represent numbers from strings to integers in a process called parsing. In this version, we can add all needed error checking called command validation. If an error is detected, an error message can be displayed, the erroneous input rejected, and the CLI can continue with a new prompt for input.

```
char line[100], linecopy[100];      // line to be read and copy of line
char tokens[5][20];                 // breakdown of line into tokens
char *p;                            // pointer for tokenizing
int tokencount;                     // number of tokens found

fgets(line,100,stdin);        // read one line of input into line
strcpy(linecopy,line);        // create copy of line into linecopy

// break linecopy into tokens based on space " " separator

p = strtok(linecopy, " ");          // initialize p to first token
tokencount = 0;                     // initialize token count to 0

while (p != NULL) {                 // keep going until p is NULL
      strcpy(tokens[tokencount],p); // copy latest token from p
                                    // to tokens[] array
      p = strtok(NULL," ");         // advance p to next token
      tokencount++;                 // increment tokencount
}
```

In this version we read in the entire string, spaces and all, into a single string variable. We then break it into individual tokens by separating the string into substrings based on the position of separator characters. In this case we use space or " " as the separator, but we could use whatever character is required. Each token is placed as a string into one row of the two dimensional array tokens[][]. If the original input read into line and copied into linecopy was

"new 10 20 25"

then the contents of the tokens[] array after tokenization is complete will be

Row 0 = tokens[0]: "new"
Row 1 = tokens[1]: "10"
Row 2 = tokens[2]: "20"
Row 3 = tokens[3]: “25"

The final step is to parse the strings representing numbers into the integers that they represent. The parse function in C is named **atoi()** which stands for “ASCII to Integer”

```
int rows  = atoi(tokens[1]);        // convert "10" to 10
int cols  = atoi(tokens[2]);        // convert "20" to 20
int mines = atoi(tokens[3]);        // convert "25" to 25
```

Functions for Input Processing

Each line of text that the user enters at the prompt represents a command to the CLI. The line is parsed as described above, and then is "dispatched" by an if-else statement to the function that implements that command. We will implement the "new", "show", and "quit" commands for Part 1, then complete the "flag", "unflag", "uncover", and "resign" commands in Part 2.

We will define the C function named **rungame()** that provides the CLI. We will also provide the **rungame()** function with several "helper" functions that break down the task into smaller parts.

int rungame()
    declare variables to hold string input for processing
    initialize state of the game to "new game"
    while(1)
        print prompt ">>" for user input on the screen
        read line of input using fgets()
        tokenize input line into command and arguments
        dispatch command to be executed by its specific function
        if command == "quit", break out of loop
    end loop
    return 1 (may use choice of return values in later versions)
end function

void getline(char line[], int maxlinelen) to call fgets() and get one line of input
    fgets(line, maxlinelen, stdin);
    int linelen = strlen(line);
    if (line[linelen-1] == '\n') line[linelen-1] = '\0';
void gettokens(char line[], char tokens[][MAXTOKENLENGTH],int *count)
    to call strtok() to break the current input line into command and arguments.

    copy line into linecopy
    use strtok() to break linecopy into tokens (code example was given above)
    store tokens into tokens[][]
    store number of tokens found into *count
int processcommand(char tokens[][MAXTOKENLENGTH], int tokencount)
    to dispatch control to specific command function based on input

    check command string in tokens[0]
    if command is "new"
        call command_new()
    else if command is "show"
        call command_show()
    else if command is "quit"
        return 0
    return 1

For "getline()" I've provided the complete suggested code. It improves upon fgets() by replacing the final newline char '\n' with the null character '\0'.

Also note the parameter "tokens" which is a static 2D array. When passing such an array as a parameter to a function, C only passes a reference to the original array to the function, it does not copy the array to the stack. For such an array, we can leave the number of rows empty, but we must explicitly show the number of columns. That's why the MAXTOKENLENGTH expression appears in the 2nd index position to show the number of columns for the parameter that represents a static 2D array.

char tokens[][MAXTOKENLENGTH] <= parameter to function that represents array

The rules for passing a reference to a dynamically allocated 2D array are different, we'll see how to do that in a later section.

Application Entry Point: main()

All C programs need an entry point function named **main().** Depending on how the customer plans to use the code, the developer may or may not need to provide it. If the task is to provide the customer with a library, then a main() function might not be required, but if the task is to build a complete and ready-to-run app, then a main() function is required. In this project, we provide a minimal version that does nothing but call **rungame()** to start the game. In general, it is important to maintain a clean and simple interface between the developer code (the game) and the main function (the client). Whatever it is that needs to be done by the client to run the app should be as simple and straightforward as possible.

```
int main(void) {
      rungame();        // <- interface from client code to game
      return 0;
}
```

Summary of Input Processing Functions

Here is the function call relationship between the command-related functions:

- main() calls rungame()
- rungame() creates a loop and repeatedly calls
    - getline()
    - gettokens()
    - processcommand()
- processcommand() calls the individual command functions
    - command_new()
    - command_show()
- processcommand() returns 0 when "quit" command is processed, rungame() will end when processcommand() returns 0

More Details on Function rungame:

void rungame() { }

We need storage for the strings that the user enters from the keyboard, which we break down into commands and arguments. Rather than use dynamic memory allocation, we will define them as **static arrays** that are **local variables** inside the rungame() function. The pros/cons are that local variables are allocated on the stack when their containing function (rungame) is called, and are automatically from the stack when the containing function terminates (pro), but static arrays have sizes that are restricted to compile-time constants (con). There are other possible approaches, but we will use the "static arrays as local variables" approach here just for practice. We also use **#define** to define constants for the maximum dimension of the arrays that hold the strings.

```
#define MAXTOKENCOUNT 20
#define MAXTOKENLENGTH 20
#define MAXLINELENGTH 400

void rungame() {

	char line[MAXLINELENGTH];
	char tokens[MAXTOKENCOUNT][MAXTOKENLENGTH];
	…
}
```

The function now starts a forever loop that prints a prompt and calls getline() to read keyboard input and store into "line". Note that getline() needs parameters "line" and "MAXLINELENGTH".

```
void rungame() {
	…
	while (1) {
		printf(">> ");
		getline(line,MAXLINELENGTH);
	}
}
```

Next, we take the content of "line" and pass it to gettokens() to break it into tokens. The input will come from the string stored in "line" and the result will be deposited in "tokens". Also the function will store how many tokens were actually found in variable "tokencount", which we pass to the function via pointer.

```
void rungame() {
	…
	while (1) {
		int tokencount;
		int result;
		printf(">> ");
		getline(line,MAXLINELENGTH);
		gettokens(line,tokens,&tokencount);
	}
}
```

Once we have the tokens, we pass them to processcommand() to actually determine which command has been chosen and to execute it.

```
void rungame() {
      …
      while (1) {
            int tokencount;
            int result;
            printf(">> ");
            getline(line,MAXLINELENGTH);
            gettokens(line,tokens,&tokencount);
            result = processcommand(tokens,tokencount);

            if (result == 0) break;
      }
}
```

The processcommand() function will return 1 for all commands except “quit” for which it will return 0. rungame() will use the return value of 0 as a cue to end the function.

Also for practice, we see two different ways for a called function to return info to the calling function. The most obvious way is shown in the statement

result = processcommand(…);

This is the most natural way for the called function (processcommand) to return a value to the calling function (rungame).

Since C provides the programmer flexible access to pointer expressions (not available in all programming languages), the programmer can use pointers as an alternative way to pass values between a calling function and a called function. Specifically, the calling function (rungame) can create a local variable such as tokencount, then pass a **pointer** to the variable to the called function. The called function can use the pointer expression to both receive values from the calling function and to send values back to the calling function. For example, see the line:

gettokens(line,tokens,&tokencount);

In this call, the calling function (rungame) passes 3 pointers to the called function (gettokens). The first two parameters are pointers to char arrays. When passing a pointer to an array, it is not necessary to use the address operator “&”. The third parameter is a pointer to the integer variable tokencount. In the case of an integer, it **is** required to use the address operator and apply it to the name of the variable. Inside the called function, a value can be returned to the calling function by assigning a value to a pointer variable which indirectly assigns the value back into the storage for the variable inside the calling function. So in summary, if a calling function passes a pointer to a variable to the called function, the variable can act either as an input to the called function, an output from the called function, or both.

Processing User Commands

Once the CLI has read text from user input and tokenized it, we are ready to decode and dispatch the command. This means we compare the string entered by the user to a series of known commands. If we get a match, we call the command function for that command. In Part 1 we will implement command-specific functions for two commands:

- command_new(): to implement the "new" command
- command_show(): to implement the "show" command

Both of these commands deal with the state of the game board and how to display it.

Building and Displaying the Game Board

The game board is a two-dimensional array with each location selected by its row-column pair. At each row-column location we store a few key attributes about that location. So in order to store multiple attributes at a single location we define a structured type that we will call a "cell". The board is then a two-dimensional array of cells.

Defining a Cell using "struct"

Each location in the game board is a cell with several state items that are distinct for that cell. Although C is not an OOP language and does not support user-defined classes, it has a simplified older version of the class feature called "struct" which is short for "structure". Here is the complete definition for the cell structure to support the game.

```
struct cell {
      int position;     // value used to show location of cell
                        // within its 2-D board
      int adjcount;     // value from 0-8 indicating how many
                        // adjacent cells are mines
      int mined;        // 0/1 to indicate if this cell
                        // contains a mine (1) or not (0)
      int covered;      // 0/1 to indicate if this cell
                        // is currently covered (1) or not (0)
      int flagged;      // 0/1 to indicate if this cell
                        // is currently flagged (1) or not (0)
};
typedef struct cell cell;
```

The "struct" definition defines a structure named "cell" and specifies the data fields for each instance. The one line "typedef" command that follows creates a shortcut so that the phrase "struct cell" can be replaced in later code with shorter phrase "cell". We can use this shortcut and use the phrase "cell" as if it were a datatype to create variables of type "cell".

Creating a 2-D Table or "board" of Cells
We could create a statically allocated board of cells with a set of definitions like this:

```
#define ROWCOUNT 10
#define COLCOUNT 20
cell board[ROWCOUNT][COLCOUNT];
```

This is a simple solution and easy to code. But the dimensions of this array are restricted to **constants at compile time**. This means that the number of rows and columns is fixed when the application is compiled by the developer and cannot be modified later at runtime by the user.

Therefore, a better solution is to use **dynamic memory allocation** which will allow the array size to be deferred until runtime. Even better, the board can be deallocated and reallocated during game play. We can run the game as a session with multiple games and a different size board for each game. This will result in a more positive user experience. Dynamic memory allocation is one technique that gives the user or player more control over app behavior, rather than reserving all important decisions and parameters to the developer.

```
// global variables
cell **board;          // "board" is the 2-D array declared as
                       // type "pointer to pointer to cell"
int rows;              // number of rows
int cols;              // number of cols
int mines;             // number of mines
```

In this version, "board" has no fixed size. The pointer "board" holds the address of where the board will be allocated, but initially the pointer points to nothing. The variables "rows" and "cols" will hold the values of number of rows and cols once the board is allocated, but the values are no longer restricted to compile time constants. At runtime, the user will enter the desired values from the keyboard and they will be stored in these variables. Even better, the size of the board can change multiple times during the game based on the latest input from the user.

Based on user input from the “new” command to the CLI, the user will request a new board of cells with the number of rows and cols given as arguments to “new”. These values from the user will be stored in the rows and cols variables, and we can then allocate space for the board in several steps:

```
void command_new(int r, int c, int m) {

      // assign r/c/m to rows/cols/mines
      …

      // allocate initial “row vector”
      board = (cell **) malloc( sizeof(cell *) * rows );

      // for each element of row vector, allocate a row
      for (int i=0; i<rows; i++) {
            board[i] = (cell *) malloc( sizeof (cell) * cols );
      }

      // for each cell, call init_cell() to set its initial values
      for (int i=0; i<rows; i++) {
            for (int j=0; j<cols; j++) {
                  init_cell(&board[i][j]);
            }
      }
```

There are still two important board configuration steps to complete before the board is ready for playing the game: randomly positioning the mines across the cells and calculating the adjacency counts for all cells that are not mined (how many mined cells are adjacent to a cell that is not itself mined). We’ll return to those in the next assignment after we successfully build and display the board.

Display the Board
The player will execute the “new” command to build a new board for game play. The “show” command will show the current state of the board whenever the user wishes to see it.

Display a “cell”
We first define a function to display a single cell. Keep in mind the general logic of the game to guide you in how to display a given cell based on its current attribute values.

- Game starts with all cells covered.
- A covered cell may or may not be marked with a flag.
- First check if cell is flagged. If so display the flag indicator for that cell.
- If not flagged, but still covered, display a covered indicator.
- If not covered, display the adjacency count value. If adjacency count = 0, it may improve the readability of the grid to display a space character ' ' to show the cell as empty.
- Also note that we assume an uncovered cell is not mined. Uncovering a cell that is mined results in the game ending with a loss for the player.

We can now write a definition of the display_cell() function. We pass the cell to be displayed as a parameter, either the actual cell copied to the stack, or preferably a pointer to the cell.

```
void display_cell(cell *c) {
      …
}
```

We ensure that each display case takes up exactly the same number of characters so that when the grid of cells is displayed, they line up correctly into a 2D grid. We can pick whatever character we want to display for each case, you might want to experiment here to decide what looks the best in the final display.

For example, suppose we use the character upper-case P 'P' for a flag, since it looks kind of like a flag. Then the first case for display_cell would look like this:

```
void display_cell(cell *c) {
      if (c->flagged == 1) printf("P ");
      else if …
}
```

In this case we will print the flag character followed by one space for a total width of two characters. The display of each cell will need at least one character of white space, otherwise the board will look too compact and will be hard to read. You can experiment here, but whatever you decide, remember that each display option must use exactly the same number of characters with a monospaced font in order for the grid to display with proper alignment.

Show or Display Complete Board

Now that we can display a single cell, we can complete the function that displays the complete board. This is organized as a doubly-nested loop with the outer loop counting the rows and the inner loop counting the columns:

```
void command_show() {
      for (int i=0; i<rowcount; i++) {
            for (int j=0; j<colcount; j++) {
            }
      }
}
```

Inside the inner loop, we use i to choose the row and j to choose the column. We can now access the cell at that location (row i and column j):

board[i][j]: cell at row i and column j

We pass a pointer to that cell to the display_cell function we wrote earlier:

display_cell(&board[i][j]);

For a given row, we want to display all the cells in that row without entering any newline characters. Once the end of a row is reached we want to enter a single newline to move the display of the next row onto the next line. The newline display must be placed outside the inner column loop but inside the outer row loop:

```
for (int i=0; i<rowcount; i++) {
      for (int j=0; j<colcount; j++) {

            …           // put display_cell call here
                        // to display each individual cell
      }
      printf("\n");           // put newline here at the end of
                              // each row
}
```

This will provide a simple display of the grid. Later you can add additional features such as row and column headers that show the number of the rows and columns. Row headers must be placed at the start of every line. Column headers must be printed with a separate loop at the beginning before starting the display of the actual board. Conceptually there is no big problem, but getting all the pieces to align properly will take some trial and error.

The "pos" or Position Attribute for Cells

In later sections, when looking at an individual cell, it will be helpful to know where in the board the cell is store, in other words its row and column indexes. We can store both row and column indexes in a single attribute named "pos" for each cell by using the following relationships:

Let rows = # of rows in the board

Let cols = # of columns in the board

Let r and c be the row and column indexes of some cell, where 0 <= r < rows, 0 <= c < cols

Then pos = r X cols + c

For example if a cell is in row 2 and column 3, in a board with 10 columns per row, then its position is

2 X 10 + 3 = 20 + 3 = 23

If the cell is in row 2, then there are rows 0 and 1 above it with a total of 2 X 10 = 20 cells. Within row 2, the cell is in column 3, so we add 3 to 20 to get 23 for its position, which is a unique value for this cell within the whole board.

Going in the other direction, if I want to use the position value of a cell to figure out its row/col position, the relationships are

r = pos/cols

c = pos%cols

When a new board is initialized with command_new(), calculate the position of each cell and initialize the pos value of the cell with the formula described here.

When we display the board with command_show, we use a double loop to iterate over rows and cols and display each individual cell with display_cell(). For our initial version here in Part 1, for each cell, just display its pos value:

```
void display_cell(cell *c) {
        printf("  %d", c->pos);
}
```

We will write a more detailed version of display_cell() in later parts of the project.

Command Validation (not implemented in Part 1)
Later on we will add command validation to include error checks like

- Compare tokens[0] to an array of strings containing all valid command names. Is tokens[0] one of the valid commands, or is it an error (non-command)?
- Given the command in tokens[0], are the correct number of additional tokens supplied? For example, “new” requires 3 additional tokens, “uncover” requires 2, and “show” requires 0.
- If a token represents a number, are all individual characters in the token a digit character? For example, the string "123" can be parsed to an integer, while "p4qx5" cannot.
- For some commands, the arguments represent a row or column value in an existing board. Are the values in the range 1 to # of rows for row and 1 to # of cols for col? For example, if the board is defined for 10 rows, then row 3 exists, but row 15 does not.

With command validation in place, the user can have a high degree of confidence that any erroneous input will be caught by the command validation before it leads to a runtime error that will crash the program and lose the state of the game play. After playing a game for 10-15 minutes and entering a large number of inputs to work towards a solution, the user would be very disappointed to accidentally enter erroneous input that crashes the program and loses the state of the game. Command validation will result in a game with a much higher level of player satisfaction.

File Organization
For now, all code will fit in a single C source file named cli.c. Here is a suggested organization for the file:

```
// --------------------------------
// application:  minesweeper
// file:  cli.c
// Author:  Rick Covington
// Date:  09/01/2023
// --------------------------------

#include <stdio.h>

// ---------------------------------
// global definitions
// sizing constants
// definition of “cell” struct
// function prototypes
// for all functions defined
// in this file
// ---------------------------------

// ---------------------------------
// input processing functions
// ---------------------------------

// ---------------------------------
// command functions for “new” and “show”
// ---------------------------------

// ---------------------
// main function
// ---------------------
```

It is always important to add the appropriate level of comments to **any** source file. At a minimum, use banner comments as shown here to divide the source into easy-to-follow sections. Provide a clear banner comment at the top that includes info such as name of app, name of this file, and optionally other information such as author/coder and date. Line-by-line comments or one comment for each line of code is usually not advisable since it may make the code harder to read, not easier, but might still be advisable for complex code that implements a non-intuitive algorithm.

Later on, as we add more functions, we will divide the code into multiple source (.c) and header (.h) files for a better organization that supports a more standard build process with separate compilation for each .c file ending with a final link step to link the final executable. If you have correctly organized your file with banner comments, it will be much easier later to divide the code up into multiple files.

Example Run
The following shows a complete application run session, with all the functions for Part 1.

```
01 PS C:\Users\rick> ./ms
02 >> new 5 5 10
03 >> show
04 Current Board
05  0   1   2   3   4
06  5   6   7   8   9
07 10  11  12  13  14
08 15  16  17  18  19
09 20  21  22  23  24
10
11 >> new 5 10 10
12 >> show
13 Current Board
14  0   1   2   3   4   5   6   7   8   9
15 10  11  12  13  14  15  16  17  18  19
16 20  21  22  23  24  25  26  27  28  29
17 30  31  32  33  34  35  36  37  38  39
18 40  41  42  43  44  45  46  47  48  49
19
20 >> quit
21 end of game
22 PS C:\Users\rick>
```

Explanation of each line:

Line 01: User launches “ms.exe” app from PowerShell command line
Line 02: Game provides “>>” prompt, user enters game command “new 5 5 10”
Line 03: Game provides “>>” prompt, user enters game command “show”
Lines 04-10: result of “show” displays board
Line 11: Game provides “>>” prompt, user enters game command “new 5 10 10”
Line 12: Game provides “>>” prompt, user enters “show”
Lines 13-19: result of “show” displays board
Line 20: Game provides “>>” prompt, user enters “quit”
Line 21: Game displays exit message and exits
Line 22: Return to PowerShell command line

Use this example as a final test to confirm completion of Part 1.

To Submit
This version of the project requires a single C source file. Submit your .c source file by uploading it to the Canvas assignment page for the Minesweeper Part 1 assignment.