Minesweeper in C
Part 3

In this part we'll finish the commands for "uncover", "flag" and "unflag". Uncovering a cell that has an adjcount of zero requires a **group uncover** step to uncover the original cell plus the surrounding group of similar cells all at the same time (you can choose between a recursive or non-recursive solution). We will also add some state checks to avoid some illegal operations. For example, if the user tries to uncover a cell that is currently marked with a flag, the app will issue a message that a flagged cell cannot be uncovered and to unflag the cell first, then uncover it.

Here are the commands. The first 3 were implemented in Part 1, the last 3 are implemented in this part.

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

I also suggest adding to the CLI "secret" commands "coverall" and "uncoverall" to cover/uncover all cells on the board. These are "cheat" commands that are needed during development, but should be removed from the final version of the game (or hidden by naming them something sneaky).

Flag/Unflag Commands
These are simple commands to change the "flagged" state of the indicated cell.

```
"flag 3 5"
        In the CLI:                 command_flag(3,5)
        In command_flag(r,c):       board[r][c].flagged = 1;
"unflag 3 5":
        In the CLI:                 command_unflag(3,5);
        In command_unflag(r,c):     board[r][c].flagged = 0;
```

Slightly better: add a check that the command is valid (can only flag an unflagged cell)

```
command_flag(r,c):      if (board[r][c].flagged == 0) {
                                board[r][c].flagged = 1;
                        }
                        else {
                                printf("cell already flagged\n");
                        }
```

Uncover

There are three cases

- uncover a mine: game is over, you lose
- uncover a non-mine with adjcount > 0: just uncover the one cell and continue
- uncover a non-mine with adjcount == 0: must do a group uncover of surrounding cells.

Recursive Uncover Group

Call this function from the uncover function if the original cell is a non-mine with adjcount == 0.

Algorithm

```
uncover_recursive(int r, int c)                     // r=row, c=col for cell d
    uncover d                                       // uncover cell d at board[r][c]
    if (adjacency count of d == 0)                  // only do recursion if d.adjcount 0

        for each neighbor n of d
            if (n exists and n is covered)          // check that n exists
                                                    // and n.covered 1
                uncover_recursive(nr, nc)           // nr=row, nc=col for n
```

Note the following facts about the recursive uncover group:

- The original cell d is first uncovered, regardless of its property values.
- If the adjacency count of d > 0, then the algorithm is done, none of its neighbors are checked.
- If the adjacency count of d == 0, then the algorithm move on to the recursive step.
- In recursive step, call function recursively for each neighbor n of d if n exists and if n is covered.
- Use the same calculation to find neighbors as was used in part 2 to calculate adjacency counts.
- Don't recursively call the function if neighbor n has already been uncovered. This check is needed to avoid infinite recursive calls. Two different cells a and b can have a common neighbor q that is a neighbor to both a and b. Whichever recursion reaches q first (from a or b) will uncover q. If recursion from the other neighbor reaches q later a second or third time, recursion should stop since q was uncovered earlier.

Example

In the following example board (10 rows, 10 columns, 20 mines), we first view the board with all cells uncovered. We see there is a group of non-mine cells with adjcount == 0 at row=0, col=0. We cover the cells, then uncover 0 0, and see the resulting board. All the cells in the group adjacent to 0 0 have been uncovered, including a boundary of cells with adjcount > 0.

```
>> show                       (board with all cells uncovered for clarity)
  .  .  .  .  .  .  1  *  2  1
  .  1  2  3  2  1  1  2  3  *
  .  2  *  *  *  1  .  2  *  3
  .  2  *  4  3  2  1  2  *  2
  1  2  1  1  1  *  1  1  1  1
  *  2  .  .  2  2  2  .  .  .
  *  4  2  2  2  *  1  .  .  .
  *  5  *  *  2  1  1  .  1  1
  3  *  *  4  2  .  .  .  1  *
  2  *  4  *  1  .  .  .  1  1
>> coverall
>> show                       (board with all cells covered as in the game)
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
>> uncover 0 0
>> show                       (cell(0,0) uncovered plus surrounding cells)
  .  .  .  .  .  .  1  /  /  /
  .  1  2  3  2  1  1  /  /  /
  .  2  /  /  /  /  /  /  /  /
  .  2  /  /  /  /  /  /  /  /
  1  2  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
  /  /  /  /  /  /  /  /  /  /
```

Final Version of display_cell

In part 2, we modified the function to display a single cell. Now with all the functions implemented, we can write the final version. For any cell, the way it is displayed depends on its state. Check the state of the cell **in the following order**. For each case, pick a character to display. Make sure each case prints in the same field width in order to properly line up the display of the board

```
void display_cell(cell *c) {
      if c is flagged, print "P"
      else if c is covered, print "/"
      else if c is mined, print "*"
      else if c->adjcount > 0, print c->adjcount
      else print "."
}
```

You can experiment with which character to print for each state to get a board that is easy to read. Note that we won’t usually print a cell that is mined and uncovered since that would imply the game has been lost. Also note that the last case is for a cell that is not mined, uncovered, and adjcount == 0. There can be a lot of such cells. Pick a character with mostly white space to keep the board from being filled up with too much info. You can use a space " " here, but that might make the board look too empty.

Determining Win/Lose

Lose

The only way to lose the game is to uncover a mine. The command_uncover() therefore needs to return a code (1 = continue, 0 = end with loss) that will be read by rungame(). In turn, rungame needs to return a code to indicate if the game continues or ends with a win or lose.

Win

The only way to win the game is to put a flag on all mines and uncover all non-mines. So a win can occur either during command_flag() when the last mine is flagged, or command_uncover() when the last non-mine is uncovered.

Algorithm

```
    For all rows r
        For all cols c
            If cell at r,c is a mine AND not flagged return 0 (false)
            Else if cell at r,c is not a mine AND is still covered return 0 (false)
        End for
    End for
    Return 1 (true)
```

This function searches the board for a counter example that disproves the “win” hypothesis. If any one cell is found that violates the win condition, we stop and return false. We don’t need to continue to check the rest of the board because it only requires one violation to set “win” to false. If we complete the double loop with no violations detected, we can then confirm the win and return true.

Note that when checking for win/lose, a result of false when checking for “win” doesn’t mean “lose”, and a result of false when checking for “lose” doesn’t mean “win”. Most of the game will be spent in a state of “neither win nor lose” until the specific conditions for win or lose are confirmed. So a “false” from a check for “win” just means “not a win (yet)”. In that case we just keep playing.

Malloc and Free

The only use of malloc() in the game is inside the command_new() function to create a new board. So when does the malloc’d space get freed? At the next call to command_new(). The function should first free any previously malloc’d board, then malloc the new board. There is one exception: when the very first board is malloc’d, there is no previous board to free.

During game initialization, set board to NULL. In command_new(), if board != NULL, free previous board then allocate new board, else proceed directly to allocate new board.

Note that each row of the row vector is freed, followed by free of the row vector. Some programmers recommend freeing mem in reverse order from malloc. It is debated if this results in better heap management performance or not.

Error Checking

It will be frustrating for the player if the app crashes due to an error in the syntax of the command input. We will try to catch most of these errors before they crash the app and instead give the player a chance to reenter the correct command. Some of the error checks will require quite a bit more code, but the effort will pay off in the form of a much more robust app.

1 Illegal Command (command isn't one of the commands recognized by the CLI)
Confirm command is a valid command by checking it against the official list of recognized commands.
Add an array of strings to hold names of all legal commands

```
#define LEGALCMDCOUNT 6
char *legalcommands[] = { "new", "show", "quit", "uncover", "flag", "unflag" };
```

Suppose current command in tokens[0]. Check that tokens[0] is in the legalcommands[] array.

```
int cmdlegal = 0;
for (int i=0; i<LEGALCMDCOUNT; i++) {
        if (strcmp(tokens[0],legalcommands[i]) == 0) {
                cmdlegal = i;           // record index i if command is legal
                break;
        }
}
if (cmdlegal > 0) …        // command is legal, continue processing
else …                     // command not legal, print error message and return
```

Note that the for loop will not only find whether the command exists, but if it exists it will find the index of the command in the array. We'll use the index in the next check.

Command Legal, But Wrong Argument Count For That Command
Each command has a specific number of args. For each command, confirm that the player provided the correct number of args for that command. Create an array with the required number of args for each command, using the same command indexing of the previous array.

```
int[] legalcommandargs = {3, 0, 0, 2, 2, 2};
```

Use the index from the previous check for legal command and compare it to the number of tokens which was calculated earlier. Subtract 1 from tokencount to get the number of arguments, since tokencount includes the command itself as one of the arguments.

```
if (tokencount-1 == legalcommandargs[cmdlegal]) …     // arg count correct
else …                                                // arg count incorrect
```

Previous Checks Passed, But Incorrect Format for Numerical Args
Each arg to a command must be an integer. For example, if the user enters
"new 15 24 3x"
Then the command is legal and the number of args is correct, but "3x" is an invalid format for integer number of mines. Confirm that the characters in the argument tokens consist only of digit characters. The C library function **int isdigit(char c)** can be used to check a character, returns 1 (true) if the char is a digit char, returns 0 (false) otherwise. May need #include <ctype.h>

```
int alldigits(char *s) {
    int len = strlen(s);
    for (int i=0; i<len; i++) {
        if (! isdigit(s[i])) return 0;          // return 0 (false) if any nondigit char found
    }
    return 1;                                   // return 1 (true) if all chars are digits
}
```

Apply the alldigits() function for each of the argument tokens, since for the commands implemented so far, the only kinds of arguments being used are integer arguments. Don't apply to tokens[0] since that is the command itself and not one of the arguments. If any argument fails the alldigits() check, then you have found an error. In order to continue with no error, all arguments must pass the alldigits() test.

Previous Checks Pass, But Numerical Args Out of Range
Args that represent row and column values must be a number in the right range. Confirm that a row argument is between 0 to row count – 1 (inclusive), and col argument is between 0 and col count – 1 (inclusive).

These checks are added to the CLI text processing step and are executed before the command function is called. If any of the checks fail, the CLI will issue an error message to the player and return to the top of the CLI loop. In this way, any error in an input command will be caught before it has a chance to crash the app. The player can read the error message and reenter the command without interrupting the game. Also, perform the checks in order, since each check assumes the previous checks passed.

Improved Formatting

Indexes Starting from 1 Instead of 0

Suppose we have a board with 10 rows and 20 columns. Internally in C, array indexes run from 0 to N-1. In this example, the rows are indexed from 0 to 9 and columns from 0 to 19. But on the user interface, most players of the game would more naturally number the rows and columns starting at 1 rather than 0. The more natural numbering would be rows from 1 to 10 and columns from 1 to 20. We can add a formula to expect user input in the range 1 to N and convert it to 0 to N-1, where N represents number of rows or number of columns.

Display Headers for Rows/Columns

So the user will enter a reference to column, say, 3. But when processing the command and its argument, the code will have to subtract one from the value entered by the user to convert it to internal index range, which would be 2 in this case.

For a large board, it will be hard to identify which row and column a cell occupies unless there are headers that display the row and column indexes.