Minesweeper

Brynn Caddel

CSC 17A - Spring 2015 - 43950

Project 1 Write Up

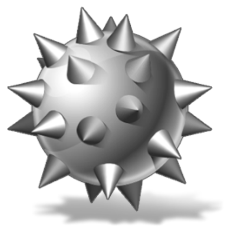


Table of Contents

Introduction:

What is it? A game. Why are you doing it? Because I’m taking CSC 17a. Why is it important? Because Dr. Lehr told me so. But in all seriousness, this is my rendition of that one game that everyone has installed on their computers, Minesweeper. My game is 608 lines long and implements concepts that are the basis of this class.

Summary and Gameplay Instructions:

Just like the original Minesweeper, this game begins by asking the user what difficulty level they would like to play on: Beginner, Intermediate, or Expert. The user is also able to choose what nxn board they would like to play on.

The user is initially presented with a grid of undifferentiated squares. Some randomly selected squares, unknown to the user, are designated to contain mines.Once the game begins, the user is prompted to choose a row and column. The objective of the game is to clear a rectangular board containing hidden mines without detonating any of them, with help from clues about the number of neighboring mines in each field. The game is played by revealing squares of the grid by clicking or otherwise indicating each square. If a square containing a mine is revealed, the user loses the game. If no mine is revealed, a digit is instead displayed in the square, indicating how many adjacent squares contain mines; if no mines are adjacent, the square becomes blank, and all adjacent squares will be recursively revealed.

The player uses this information to deduce the contents of other squares, and may either safely reveal each square or mark the square as containing a mine. In the case that the user loses, the minefield is displayed with the revealed locations of the mines and their proximity to spaces without mines. At the end of the game, the user is prompted to view statistics from previous games.

Development Summary:

When I began this project, I started out trying to program a Pokemon game where the player would navigate through a grid and fight enemy Pokemon. I thought this seemed like a good idea for a game since I would be able to use data structures to store different types of Pokemon (i.e. fire, water, leaf Pokemon). However, I didn’t think that there was enough logic present in the game other than water beats fire, fire beats leaf, and so on. I had successfully created a two-dimensional array that I used as my grid when I was working on my initial game. So I decided to Google “deduction logic games” and what do you know, Minesweeper was one of the first games to appear. Since I already had a grid I could work with, I began to read up on different ways to implement Minesweeper and the logic behind it.

I decided to use enumerated data types in my "Settings" data structure because it was a new concept introduced in the class and because I thought it was an easy way to handle the varying difficulty levels. Initially I had two data structures, one called "Settings" and the other called "Minefield." The former was to house the difficulty levels, mines, and flags, and the latter was to function as a place to store rows, columns, and a pointer to the gameboard. However, I ran into difficulties when I tried to do this, and decided to combine the two into one data structure. With all the "::" used throughout the program, this decision to make use just one structure made it easier to remember and reference throughout the game.

Writing to binary files was another challenge that I had when creating this game. It took many tries to successfully create a function that utilized binary file reading and writing. However, I was pleased that I didn't have much trouble when it came to destroying dynamically allocated memory.

References:

Even though these games was written in Java, the logic behind the functions they used to place and check for adjacent mines helped me start creating this game.

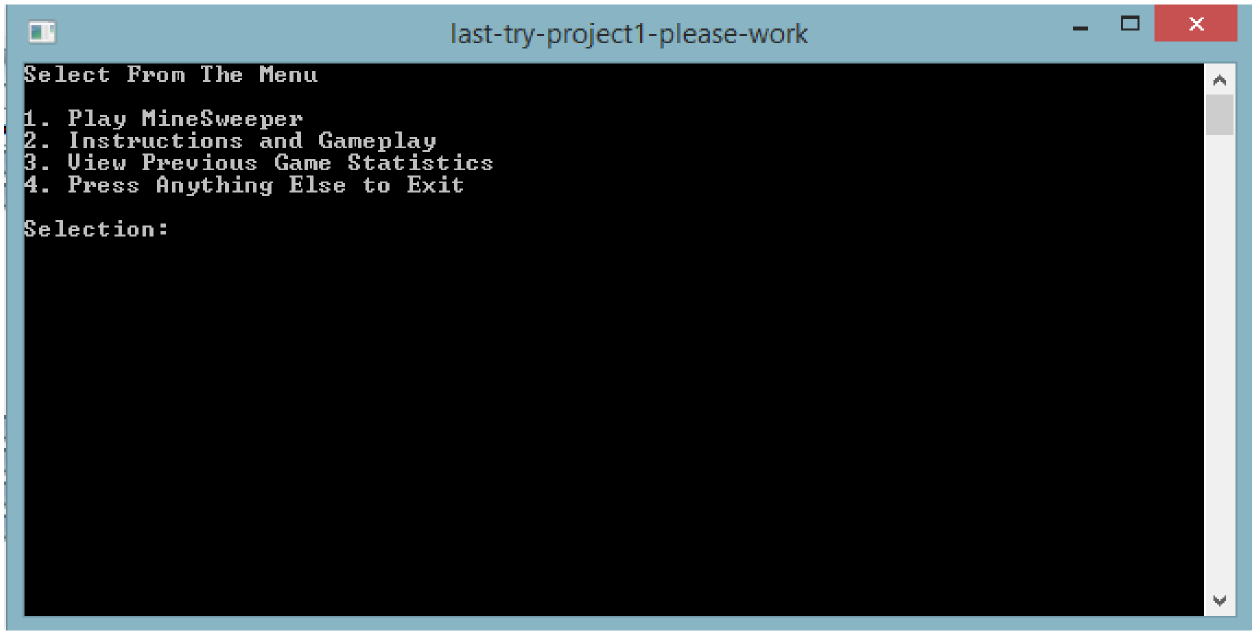
- http://zetcode.com/tutorials/javagamestutorial/minesweeper/
- http://www.progressivejava.net/2012/10/How-to-make-a-Minesweeper-game-in-Java.html

I've never taken a Java course, and even though the syntax differs how that of C++, their implementations of the setting mines, revealing end game boards, and other functions were helpful.

Sample Input/Output:

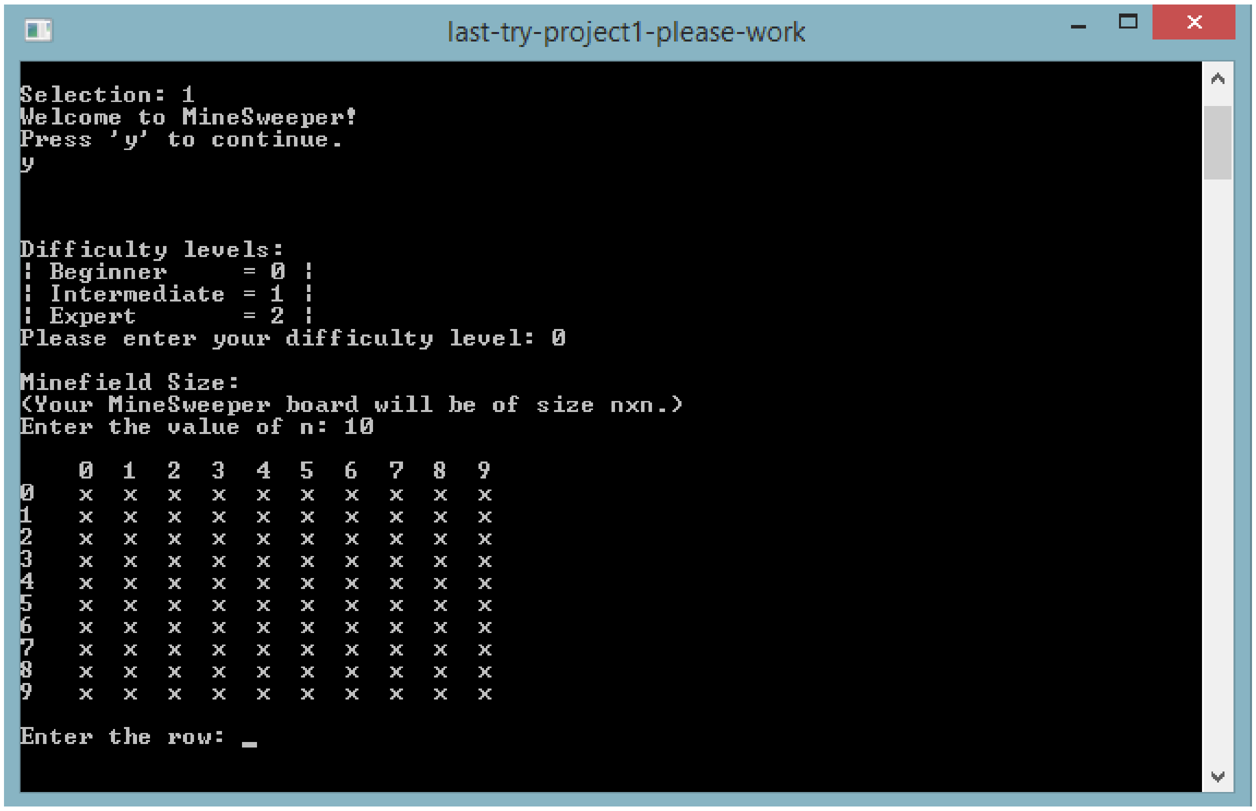
Figure 1.
First screen to display once the program runs. You're given the option of

- 1. Play MineSweeper
- 2. View the instructions and how to play the game
- 3. The option to view previous game statistics
- 4. Exits the program



```
Select From The Menu

1. Play MineSweeper
2. Instructions and Gameplay
3. View Previous Game Statistics
4. Press Anything Else to Exit

Selection:
```

Figure 2.
If option 1 is chosen, the user is prompted to enter a difficulty level: Beginner, Intermediate, or Expert. The difference in difficulty levels will change the amount of mines placed on the board. An description of each difficulty is given in Menu Option 2, which will be addressed later in this write up. The user is then given the option to choose a Minefield Size. Unlike the original MineSweeper game, this game only allows boards that are of size nxn. The many x's on the board represent areas of the board that have not been selected yet. Since this is a new board, the entire board is filled with x's. There isn't much strategy that the user can employ at this point in the game since the first row/column choice is pretty much random. You might might only reveal one number, you might uncover a large portion of the board, or you might hit a mine.



```
Selection: 1
Welcome to MineSweeper!
Press 'y' to continue.
y


Difficulty levels:
| Beginner     = 0 |
| Intermediate = 1 |
| Expert       = 2 |
Please enter your difficulty level: 0

Minefield Size:
(Your MineSweeper board will be of size nxn.)
Enter the value of n: 10

    0  1  2  3  4  5  6  7  8  9
0   x  x  x  x  x  x  x  x  x  x
1   x  x  x  x  x  x  x  x  x  x
2   x  x  x  x  x  x  x  x  x  x
3   x  x  x  x  x  x  x  x  x  x
4   x  x  x  x  x  x  x  x  x  x
5   x  x  x  x  x  x  x  x  x  x
6   x  x  x  x  x  x  x  x  x  x
7   x  x  x  x  x  x  x  x  x  x
8   x  x  x  x  x  x  x  x  x  x
9   x  x  x  x  x  x  x  x  x  x

Enter the row: _
```

I guess, if you really wanted to, you could take the number of mines on the board and the number of available spaces and find the probability of hitting a mine on your first try.

Example:

Number of possible mines: 10

Number of available spaces on a 10x10 board: 100

Probability of hitting a mine on first try:

$$\frac{10}{100} = \frac{1}{10} \text{ chance of selecting a mine on your first try}$$

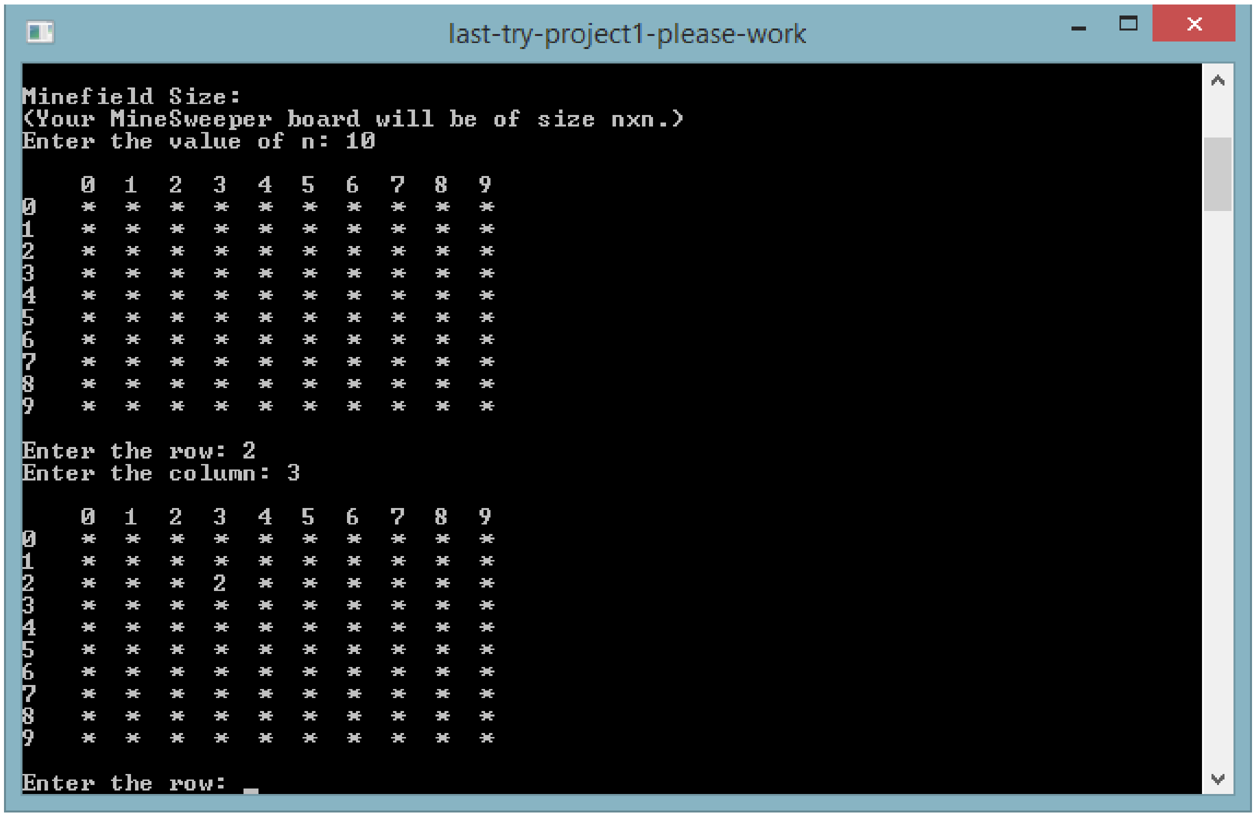
Or in a more generalized example:

x is the number of mines; $x \in$ *Positive Real Number*

n is the number of rows(or column) in the matrix; n $\in$ *Positive Real Number*

Probability of hitting a mine on your first try:

$$\frac{x}{n*n} \text{ chance of selecting a mine on your first try}$$



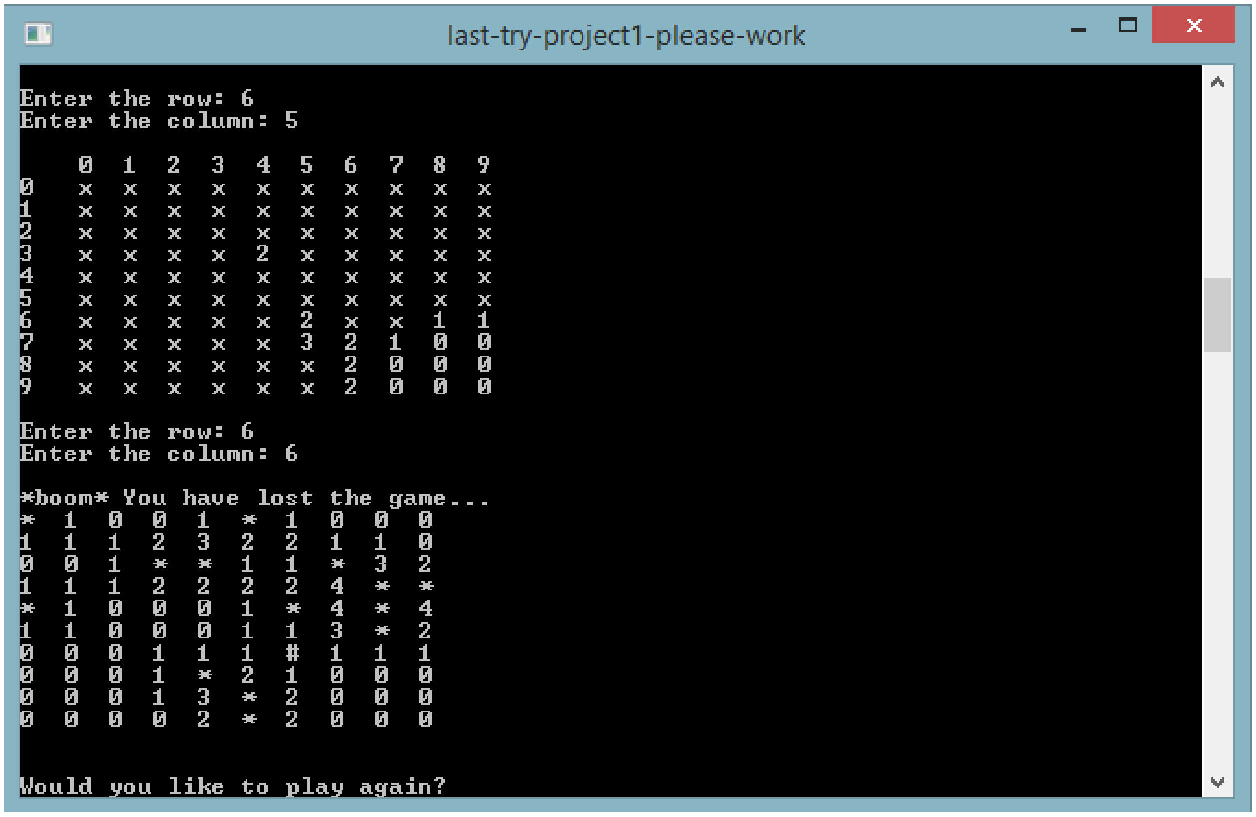
```
Minefield Size:
(Your MineSweeper board will be of size nxn.)
Enter the value of n: 10

    0  1  2  3  4  5  6  7  8  9
0   *  *  *  *  *  *  *  *  *  *
1   *  *  *  *  *  *  *  *  *  *
2   *  *  *  *  *  *  *  *  *  *
3   *  *  *  *  *  *  *  *  *  *
4   *  *  *  *  *  *  *  *  *  *
5   *  *  *  *  *  *  *  *  *  *
6   *  *  *  *  *  *  *  *  *  *
7   *  *  *  *  *  *  *  *  *  *
8   *  *  *  *  *  *  *  *  *  *
9   *  *  *  *  *  *  *  *  *  *

Enter the row: 2
Enter the column: 3

    0  1  2  3  4  5  6  7  8  9
0   *  *  *  *  *  *  *  *  *  *
1   *  *  *  *  *  *  *  *  *  *
2   *  *  *  2  *  *  *  *  *  *
3   *  *  *  *  *  *  *  *  *  *
4   *  *  *  *  *  *  *  *  *  *
5   *  *  *  *  *  *  *  *  *  *
6   *  *  *  *  *  *  *  *  *  *
7   *  *  *  *  *  *  *  *  *  *
8   *  *  *  *  *  *  *  *  *  *
9   *  *  *  *  *  *  *  *  *  *

Enter the row: _
```

Figure 3. (Displayed above) This displays a board that has chosen a row/column. The number "2" is displayed to show how many mines are adjacent to that particular space. The user now has the option of strategically choosing a space around the "2," or choosing another spot on the board that isn't near the "2." If you play MineSweeper without using any type of deduction and instead click randomly, your chances of hitting a mine vary in probability from game to game (since the chances of generating the same board twice are very unlikely). In one game, your first click may reveal a large portion of the board, effectively eliminating the number available spaces. Other games, you may only reveal one space. Each turn possesses its own unique probability of randomly hitting a mine.

However, probability subsumes logic. If you can logically prove a mine must be in a location, its probability must be 100%; if you can prove one cannot be, its probability must be 0%. So probability is all you need, in some sense. Nonetheless, you use logical deduction to detect those 100% situations; sometimes, especially at easier difficulty levels, that's all you need to complete a Minesweeper game; no appeal to probabilities is necessary.

Figure 4.
This displays a losing board. The "*" are the spaces on the board that contained a mine. The numbered spaces are those that show proximity of adjacent mines. The "0" are safe spaces that

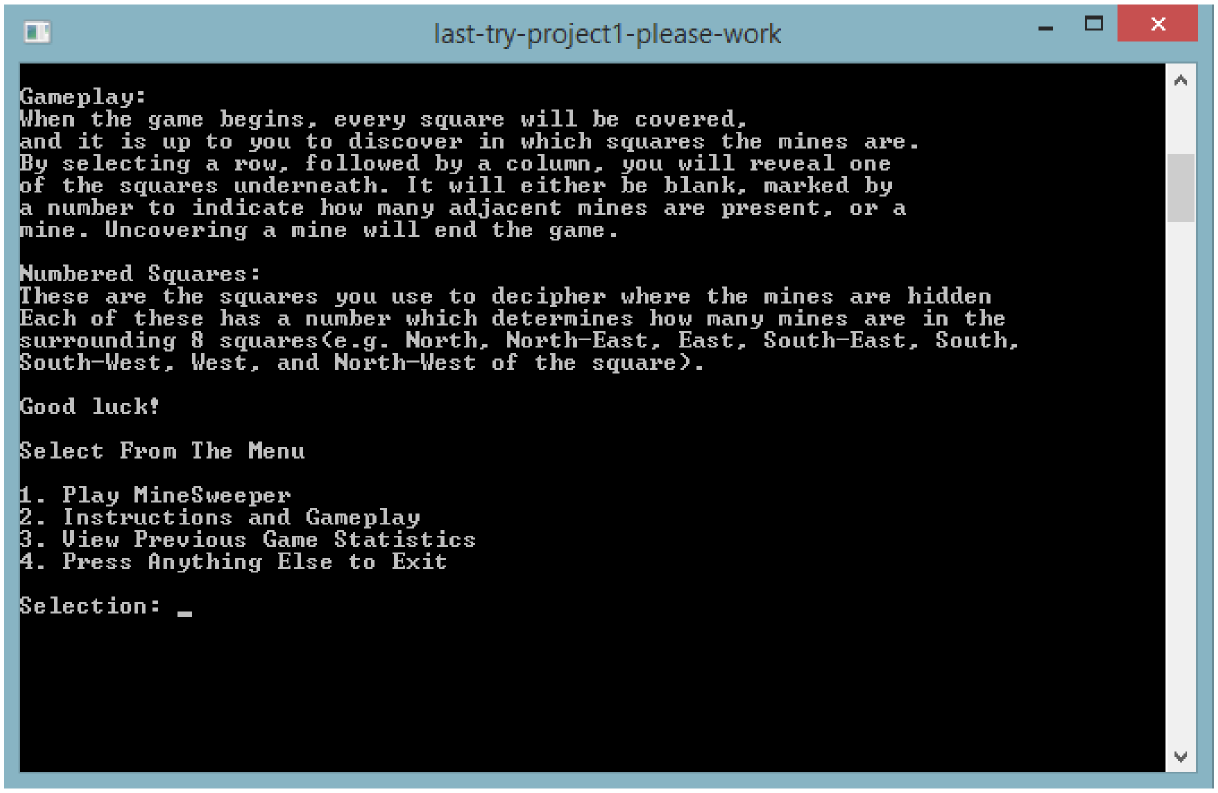
contained no mines and no adjacent mines. The “#” is where the user selected a space with a mine.



last-try-project1-please-work

```
Enter the row: 6
Enter the column: 5

    0  1  2  3  4  5  6  7  8  9
0   x  x  x  x  x  x  x  x  x  x
1   x  x  x  x  x  x  x  x  x  x
2   x  x  x  x  x  x  x  x  x  x
3   x  x  x  x  2  x  x  x  x  x
4   x  x  x  x  x  x  x  x  x  x
5   x  x  x  x  x  x  x  x  x  x
6   x  x  x  x  x  2  x  x  1  1
7   x  x  x  x  x  3  2  1  0  0
8   x  x  x  x  x  x  2  0  0  0
9   x  x  x  x  x  x  2  0  0  0

Enter the row: 6
Enter the column: 6

*boom* You have lost the game...
*  1  0  0  1  *  1  0  0  0
1  1  1  2  3  2  2  1  1  0
0  0  1  *  *  1  1  *  3  2
1  1  1  2  2  2  2  4  *  *
*  1  0  0  0  1  *  4  *  4
1  1  0  0  0  1  1  3  *  2
0  0  0  1  1  1  #  1  1  1
0  0  0  1  *  2  1  0  0  0
0  0  0  1  3  *  2  0  0  0
0  0  0  0  2  *  2  0  0  0


Would you like to play again?
```

Figure 5.
This displays the Option 2 in the menu. The Rules and Instructions of the game are explained and displayed here.



last-try-project1-please-work

```
Gameplay:
When the game begins, every square will be covered,
and it is up to you to discover in which squares the mines are.
By selecting a row, followed by a column, you will reveal one
of the squares underneath. It will either be blank, marked by
a number to indicate how many adjacent mines are present, or a
mine. Uncovering a mine will end the game.

Numbered Squares:
These are the squares you use to decipher where the mines are hidden
Each of these has a number which determines how many mines are in the
surrounding 8 squares(e.g. North, North-East, East, South-East, South,
South-West, West, and North-West of the square).

Good luck!

Select From The Menu

1. Play MineSweeper
2. Instructions and Gameplay
3. View Previous Game Statistics
4. Press Anything Else to Exit

Selection: _
```

Concepts Implemented:

Chapter	Concept	Implementation	Line
7.8	Character Array	playerN()	189
7.9	Two-Dimensional Arrays	create()	590
9.2	Pointer Variables	*create	600
9.8	Memory Allocation	create()	590
9.10	Deleting Pointers	delete player	116
10.7	String Objects	playerN()	190
11.1	Data Structure	Settings	16
11.5	Array of Structures	runGame()	502
11.6	Deleting Structure Memory	destroy()	328
11.12	Enumerated Data Types	Difficulty	25

Major Variables:

Type	Variable Name	Description	Location
int	**field	Holds the game data	Line 18
enum	Difficulty	Determines how many mines are on a board	Line 25
int	columns	number of columns on board	Line 22
int	rows	number of rows on board	Line 20
int	mines	number of mines on board	Line 24

Functions:

Name	Purpose	Line
bool check	Ensures that mines don't exceed number of space on the field	Line 140
bool noMine	Returns true if there are no adjacent mines for a given space	Line 147
bool resume	Reveals what is hidden under each space. If mine, then end game. If clear, resume game.	Line 157
bool winCase	If there are empty spaces left, then the user hasn't won. If no empty space, then player wins.	Line 178
char *playerN()	To fulfill rubric requirement. Enters name as string object and stores in a character array.	Line 189
int nMines	Number of mines on the field. Increments mines by 15 mines for each difficulty level.	Line 206
int proxM	Checks for adjacent mines. A more in depth explanation of this can be found below.	Line 216
void ckborder	Finds perimeter of cleared spaces	Line 294
void destroy	Deallocates memory used for structure.	Line 328
void flagSet	Shows the number of adjacent mines next to each space	Line 338
void hideMines	Prints the field with all mines hidden	Line 348
void instructions	Prints the instructions and gameplay	Line 378
void mineSet	Randomly places mines onto board. Only places mines if the result of rand()%15 == 0	Line 418
void readFile	Reads to a binary file.	Line 442

void revealM	This reveals the location of the hidden mines once the game is over.	Line 462
void revealZ	Reveals the location of clear spaces, in other words, it displays 0's everywhere.	Line 482
void runGame	Function that actually runs through the game.	Line 502
void userIn	User is able to input difficulty level and size of the minefield.	Line 527
void writeBinary	Writes to a binary file.	Line 567
Settings::Difficulty convert(int);	Converts integers to Difficulty level data types.	Line 574
Settings *create(int, int);	Creates the size of the NxN board.	Line 590

Description of Proximity Mine:

proxM: Located in function definitions on Line 216

After the user enters in their chosen row/column position, the Proximity Mine function (**proxM**) checks the entire board for the location of adjacent mines. The method that I used in my approach to this function came from a concept in Linear Algebra, where if *A* is a square matrix, then the minor M_{ij} of the entry a_{ij} is the determinant of the matrix obtained by deleting the *i*th row and the *j*th column of A. Example: If *A* is a 3x3 matrix,

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

then the minors of a_{11} are

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix} = \begin{matrix} \cancel{a}_{11} & \cancel{a}_{12} & \cancel{a}_{13} \\ \cancel{a}_{21} & a_{22} & a_{23} \\ \cancel{a}_{31} & a_{32} & a_{33} \end{matrix} = \begin{matrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{matrix}$$

X	X	X	X
X			X
X			X
X	X	X	X

X			X
X			X
X			X
X			X

X	X	X	X
X	X	X	X

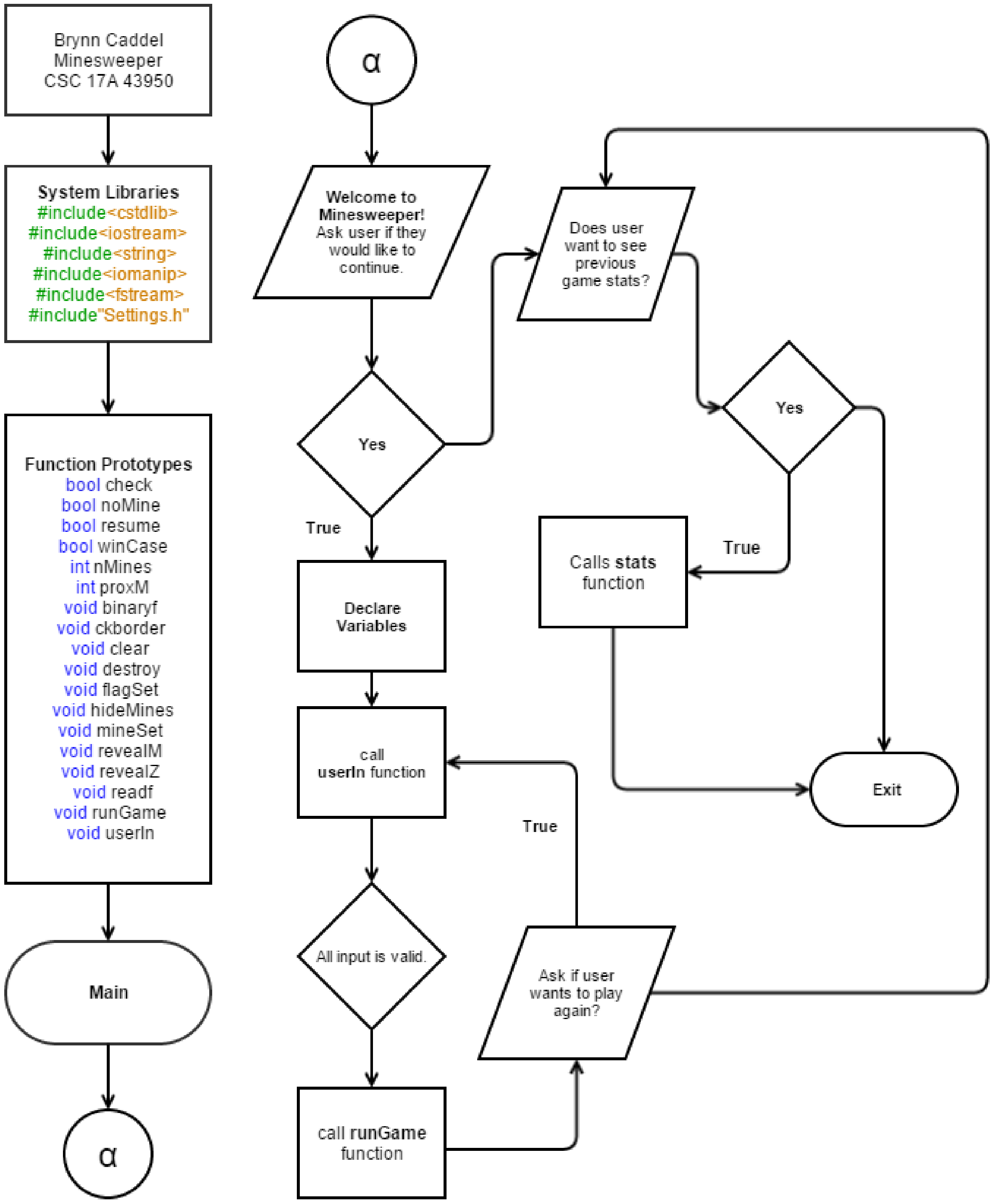
I tried to implement a similar type of logic into this function; checking the minors of each row and column that depend on the constraints given in the for loop conditions. I have two counters that check against one another and increment one if there is an adjacent mine present. The if/if else statements sweep across the board, checking for mines. The last places that are checked are the corners of the board, so the a_{ij} position, themselves. At the very end, the function returns the number of mines. The figures to the right demonstrate this concept. The X's are the rows/columns that are ignored, while the blank space is searched for adjacent mines.

Within this function I added additionally comments that describe each area of the board the function inspects.

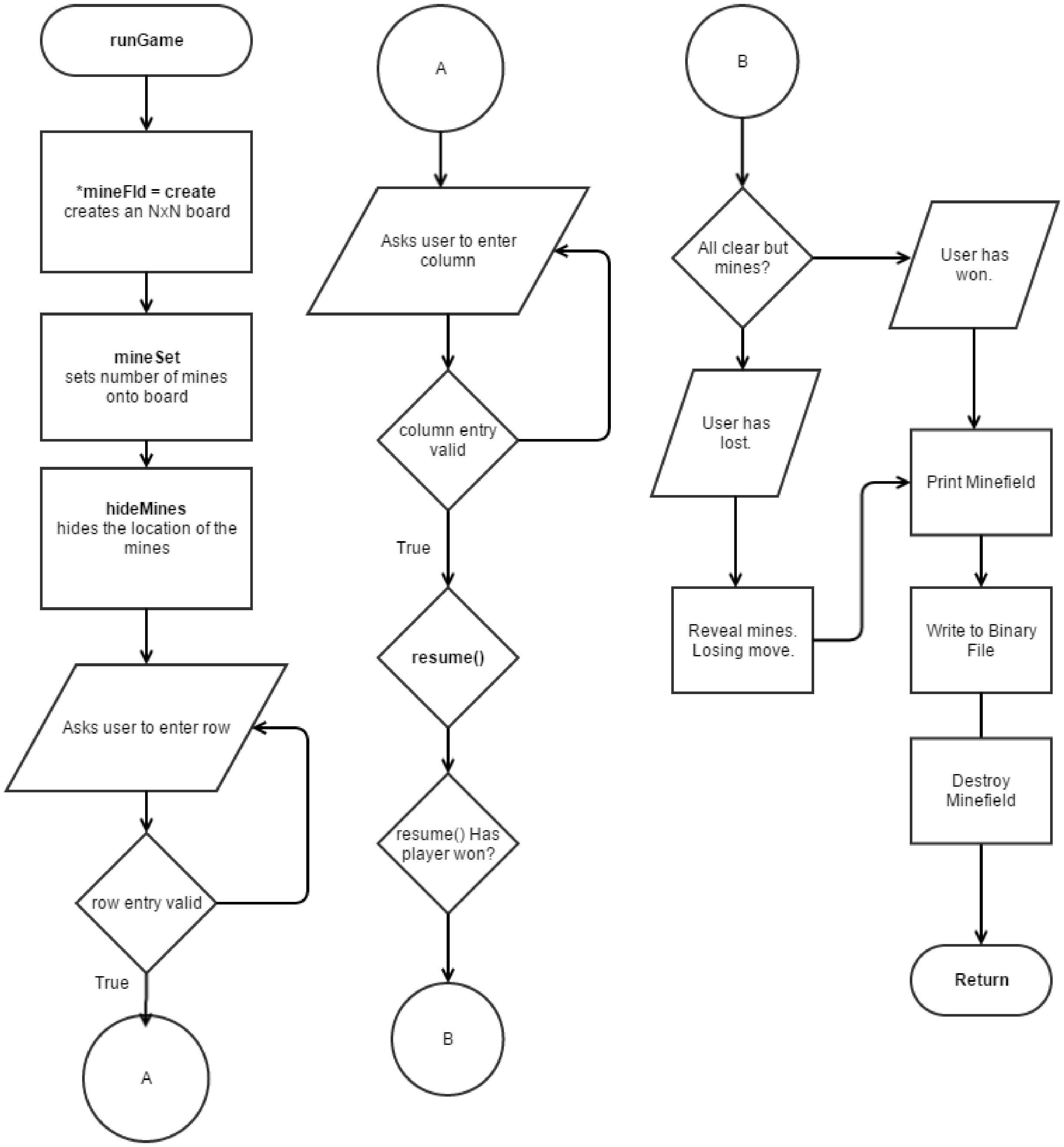
- Case 1:
  - Ignores first column/row and last column/row
  - Looks at interior
- Case 2:
  - Checks first row
  - Doesn't check first or last column
- Case 3:
  - Checks last row
  - Doesn't check first or last column

- Case 4:
  - Checks first column
  - Doesn't check first or last row
  - Sweeps to the right checking each space to the right
- Case 5:
  - Checks last column
  - Doesn't check first or last row
  - Sweeps to the left checking each space to the left
- Case 6-10:
  - These check each corner
  - I couldn't think of a more elegant way I checking these ones, so my method of approach is quite tedious

Main (Before I added my Menu):



runGame Function:



UML:

Settings

int **field;
int rows;
int columns;
int mines;
enum Difficulty {BEGINNER, INTERMEDIATE, EXPERT}
enum Flags {EMPTY = 10, MINE, CLEAR, LOSER}

Program:

```
/*
 * File:   main.cpp
 * Author: Brynn Caddel
 * Project 1 - MineSweeper
 * CSC17A 43950
 */

///System Libraries
#include <cstdlib>
#include <iostream>
#include <string>
#include <iomanip>
#include <fstream>

///Data Structures
struct Settings {
    ///Holds the game's data
    int **field;
    ///Rows on minefield
    int rows;
    ///columns on minefield
    int columns;
    ///number of mines on minefield
    int mines;
    enum Difficulty {BEGINNER, INTERMEDIATE, EXPERT};
    enum Flags {NONE = 10, MINE, NOMINE, LOSER};
};
```

```
using namespace std;

///Function Prototypes
//Descriptions of each function can be found in Function Definitions
Settings *create(int, int);
Settings::Difficulty convert(int);
bool check(int, int, Settings::Difficulty);
bool noMine(Settings *, int, int);
bool resume(Settings *, int, int);
bool winCase(Settings *);
char *playerN();
int nMines(Settings::Difficulty);
int proxM(Settings *, int, int, int = Settings::MINE);
void ckborder(Settings *);
void clear(Settings *, int, int);
void destroy(Settings *);
void flagSet(Settings *);
void hideMines(Settings *);
void instructions();
void mineSet(Settings *);
void readFile(string);
void revealM(Settings *);
void revealZ(Settings *, int, int);
void runGame(int, int, Settings::Difficulty,char*);
void userIn(int&, Settings::Difficulty&);
void writeBinary(Settings *, string);

///Execution Begins Here
int main(int argc, const char * argv[]) {
  ///Declare Variables: Menu
  int choice;
  bool exitMenu=true;
  char *player = playerN();
  ///Loop: Implemented Until Exit
  cout << "    " << endl;
  do{
    ///Output Menu
    cout << "Select From The Menu" << endl;
    cout << "      " << endl;
    cout << "1. Play MineSweeper" <<endl;
    cout << "2. Instructions and Gameplay" <<endl;
    cout << "3. View Previous Game Statistics" <<endl;
    cout << "4. Press Anything Else to Exit" <<endl;
    cout << "      " << endl;
    cout<<"Selection: ";
    cin>>choice;
    switch(choice){
      case 1:
        cout << "Hello, " << player << ", welcome to MineSweeper!" << endl;
        cout << "Press 'y' to continue." << endl;
        char ans;
        cin >> ans;
        cout << "      " << endl;
        cout << "      " << endl;

///This allows the user to choose the size of their minefield and the
///difficulty level of the game
if (ans == 'y') {
    ///Creates the variables in the game: matrix size and difficulty
    int nrows;
    Settings::Difficulty userSet;
    ///This function collects the number of rows and difficulty and stores
    ///it in an array
    userIn(nrows, userSet);
    ///Error Checking: checks validity of user input
    ///If valid, then it creates the array
    if (check(nrows, nrows, userSet)) {
        while (ans == 'y' && check(nrows, nrows, userSet)) {
            runGame(nrows, nrows, userSet,player);
            cout << endl;
            cout << "Would you like to play again?";
            cin >> ans;
            cout << endl;
            ///User has the option to play again
            if (ans == 'y')
                userIn(nrows, userSet);
        }
    }           ///Else, the program exits
    else
        cout << "Error: Minefield too small."
            "The size of the minefield is not compatible with the"
            "difficulty level. Please try a larger minefield if you're"
            "playing on Difficulty:EXPERT";
}
cout << "Game Over.";

///Take Out Trash(clean up)
 delete player;
            break;

        case 2:
            ///Displays instructions
            instructions();
            break;

        case 3:
            ///Reads Binary File
            readFile("result");

          ///Exits Menu
          default: exitMenu = false;
        }

    } while (exitMenu);

    return 0;
}

///Function Definitions

//Function prompts user to enter difficulty level and minefield size
void userIn(int &rows, Settings::Difficulty &userSet) {
    int dLevel;
    cout << "                                    " << endl;
    cout << "Difficulty levels:                  " << endl;
    cout << "| Beginner     = 0 |                " << endl;
    cout << "| Intermediate = 1 |                " << endl;
    cout << "| Expert       = 2 |                " << endl;
    cout << "Please enter your difficulty level: ";
    cin >> dLevel;
    userSet = convert(dLevel);

    cout << "                  " << endl;
    cout << "Minefield Size:  " << endl;
    cout << "(Your MineSweeper board will be of size nxn.)" << endl;
    cout << "Enter the value of n: ";
    cin >> rows;
    cout << "                  " << endl;
}

/// Function returns true if input is valid
bool check(int rows, int cols, Settings::Difficulty dLevel) {
    ///Makes sure that the number of mines does not exceed spots in
    ///the minefield
    return (rows * cols) > nMines(dLevel);
}

///Function allows user to enter rows, columns, and difficulty
void runGame(int nrows, int ncols, Settings::Difficulty dLevel,char *p) {
    srand(static_cast<unsigned int> (time(0)));
    Settings *mineFld = create(nrows, ncols);
    mineFld->mines = nMines(dLevel);
    mineSet(mineFld);
    hideMines(mineFld);
    int row, col;
    do {
        /// Select the row
        do {
            cout << "Enter the row: ";
            cin >> row;
            ///Error checking: checks validity of bounds for rows
        } while (row < 0 || row >= mineFld->rows);
        do {
            cout << "Enter the column: ";
            cin >> col;
            ///Error Checking: checks validity of bounds for columns
        } while (col < 0 || col >= mineFld->columns);
        cout << endl;
    } while (resume(mineFld, row, col) && !winCase(mineFld));

    ///Once the game has either been won or lost
    ///Win Case
    if (winCase(mineFld)) {
        cout << "You win! The Special Forces will now contact you," << endl;
        cout << " as your skill in cleverly evading mines may prove" << endl;
        cout << " to be useful out on the field." << endl;
        flagSet(mineFld);
    }       ///Lose Case
    else {
        cout << "*boom* You have lost the game..." << endl;
        flagSet(mineFld);
        mineFld->field[row][col] = Settings::LOSER;
    }
    /// Print the complete minefield
    revealM(mineFld);

    /// write result to binary file
    writeBinary(mineFld, "result");
    /// deallocate the game area
    destroy(mineFld);
}

///Function generates the grid the game is to be played on
Settings* create(int rows, int cols) {
    ///Dynamically allocating memory for minefield
    Settings *out = new Settings;
    out->rows = rows;
    out->columns = cols;

    ///Creates 2D playing field
    out->field = new int *[rows];
    ///Creates each new row
    for (int row = 0; row != rows; ++row)
        out->field[row] = new int [cols];

    ///Ensures that each space is empty
    for (int i = 0; i != rows; ++i)
        for (int j = 0; j != rows; ++j)
            out->field[i][j] = Settings::NONE;
    return out;
}

///This function takes the int variable and returns it as Settings::Difficulty
Settings::Difficulty convert(int choice) {
    switch (choice) {
        case (0):
            return Settings::BEGINNER;
            break;
        case (1):
            return Settings::INTERMEDIATE;
            break;
        case (2):
            return Settings::EXPERT;
        default:
            return Settings::BEGINNER;
            break;
    }
}

///Deallocating memory function
void destroy(Settings *mineFld) {
    ///This deletes each row that was dynamically allocated
    for (int i = 0; i != mineFld->rows; ++i)
        delete[] mineFld->field[i];
    ///This deletes the dynamically allocated structure
    delete mineFld;
}

///Once the game is won/lost, this function prints the location of the mines
///revealed
void revealM(Settings* mineFld) {
    for (int row = 0; row != mineFld->rows; ++row) {
        for (int col = 0; col != mineFld->columns; ++col) {
            ///
            if (*(*(mineFld->field + row) + col) == Settings::LOSER)
                cout << "#  ";
            else if (*(*(mineFld->field + row) + col) == Settings::MINE)
                cout << "*  ";
            else if (!noMine(mineFld, row, col))
            cout << proxM(mineFld, row, col) << "  ";
         else
            cout << "0  ";
      }
      cout << endl;
   }
   cout << endl;
}

///This function prints the field with mines hidden
void hideMines(Settings* mineFld) {
   //Print the column index
   for (int i = 0; i != mineFld->columns; ++i) {
      //Makes spacing look nice
      if (i == 0)
         cout << "  ";
      cout << setw(3) << i;
   }
   cout << endl;
   for (int row = 0; row != mineFld->rows; ++row) {
      for (int col = 0; col != mineFld->columns; ++col) {
         if (col == 0 && row < 10) cout << row << "  ";
         if (col == 0 && row >= 10) cout << row << " ";
         /// keeps empty spaces and mines hidden
         if (mineFld->field[row][col] == Settings::NONE ||
               mineFld->field[row][col] == Settings::MINE)
            cout << setw(3) << right << "x ";
            /// print out the clear areas
         else if (mineFld->field[row][col] == Settings::NOMINE)
            cout << setw(2) << 0 << " ";
            /// Print out the actual value of the square
         else
            cout << setw(2) << mineFld->field[row][col] << " ";
      }
      cout << endl;
   }
   cout << endl;
}

///There are a varying number of mines laid out depending on the user inputted
///difficulty level
int nMines(Settings::Difficulty userSet) {
   if (userSet == Settings::BEGINNER)
      return 15;
   else if (userSet == Settings::INTERMEDIATE)
      return 30;
   else
      return 45;
```

```
}

///Function randomly places mines in the field
void mineSet(Settings *mineFld) {
    ///Determines how many mines will be used
    int mines = mineFld->mines;

    ///Continues looping through mines until they're all laid out
    while (mines) {
        for (int i = 0; i != mineFld->rows; ++i) {
            for (int j = 0; j != mineFld->columns; ++j) {
                /// place mines if result of rand()%15 == 0
                if ((rand() % 100) % 10 == 0) {
                    ///only place mines if mines are still available
                    /// and current is empty
                    if (mines && mineFld->field[i][j] == Settings::NONE) {
                        /// set the mine
                        mineFld->field[i][j] = Settings::MINE;
                        --mines;
                    }
                     }
            }
        }
    }
}

/// Function returns how  many 'flag' elements surround a given square
int proxM(Settings *mineFld, int row, int col, int FLAG) {
    int adjmine = 0; /// the number of adjacent mines

    ///Ignore first/last row/column
    /// Search interior
    if (row > 0 && col > 0 && row < mineFld->rows - 1 && col < mineFld->columns - 1) {
        /// search the 3x3 grid surrounding a cell
        for (int i = row - 1; i <= row + 1; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (mineFld->field[i][j] == FLAG)
                    ++adjmine;
        }
    }       /// checks first row, doesn't check first or last column
    else if (row == 0 && col > 0 && col < mineFld->columns - 1) {
        for (int i = row; i <= row + 1; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (mineFld->field[i][j] == Settings::MINE)
                    ++adjmine;
        }
    }       /// checks last row, doesn't check first or last column
    else if (row == mineFld->rows - 1 && col > 0 && col < mineFld->columns - 1) {
        for (int i = row - 1; i <= row; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (mineFld->field[i][j] == Settings::MINE)
                    ++adjmine;
        }
    }       /// checks first column, doesn't check first or last row
        ///sweeps to the right
    else if (col == 0 && row > 0 && row < mineFld->rows - 1) {
        for (int i = row - 1; i <= row + 1; ++i) {
            for (int j = col; j <= col + 1; ++j)
                if (mineFld->field[i][j] == Settings::MINE)
                    ++adjmine;
        }
    }       /// checks last column, doesn't check first or last row
        /// sweeps to the left
    else if (col == mineFld->columns - 1 && row > 0 && row < mineFld->rows - 1) {
        for (int i = row - 1; i <= row + 1; ++i) {
            for (int j = col - 1; j <= col; ++j)
                if (mineFld->field[i][j] == Settings::MINE)
                    ++adjmine;
        }
    }       /// checks top left corner
    else if (row == 0 && col == 0) {
        if (mineFld->field[row][col + 1] == Settings::MINE) ++adjmine;
        if (mineFld->field[row + 1][col] == Settings::MINE) ++adjmine;
        if (mineFld->field[row + 1][col + 1] == Settings::MINE) ++adjmine;
    }       /// checks top right corner
    else if (row == 0 && col == mineFld->columns - 1) {
        if (mineFld->field[row][col - 1] == Settings::MINE) ++adjmine;
        if (mineFld->field[row + 1][col] == Settings::MINE) ++adjmine;
        if (mineFld->field[row + 1][col - 1] == Settings::MINE) ++adjmine;
    }       /// checks bottom left corner
    else if (row == mineFld->rows - 1 && col == 0) {
        if (mineFld->field[row - 1][col] == Settings::MINE) ++adjmine;
        if (mineFld->field[row - 1][col + 1] == Settings::MINE) ++adjmine;
        if (mineFld->field[row][col + 1] == Settings::MINE) ++adjmine;
    }       /// checks bottom right corner
    else if (row == mineFld->rows - 1 && col == mineFld->columns - 1) {
        if (mineFld->field[row - 1][col - 1] == Settings::MINE) ++adjmine;
        if (mineFld->field[row - 1][col] == Settings::MINE) ++adjmine;
        if (mineFld->field[row][col - 1] == Settings::MINE) ++adjmine;
    }
    /// return number of mines
    return adjmine;
}

///Returns true if there are no mines adjacent to space
```

```
bool noMine(Settings * mineFld, int row, int col) {
    if (proxM(mineFld, row, col))
        /// there was at least one mine adjacent
        return false;
     /// area was clear
    return true;
}

///clear and area whose values are clear
void revealZ(Settings *mineFld, int row, int col) {
    /// check bounds
    if (row >= mineFld->rows || row < 0 || col >= mineFld->columns || col < 0)
        return;
    if (noMine(mineFld, row, col) && mineFld->field[row][col] != Settings::NOMINE) {
        mineFld->field[row][col] = Settings::NOMINE;
        /// go up one row
        revealZ(mineFld, row + 1, col);
        /// go down one row
        revealZ(mineFld, row - 1, col);
        /// go right one col
        revealZ(mineFld, row, col + 1);
        /// go left one col
        revealZ(mineFld, row, col - 1);
    }       /// space was not clear or already shown
    else
        return;
}

/// Function shows how many mines are adjacent to selected square
/// for the entire minefield
void flagSet(Settings *mineFld) {
    for (int i = 0; i != mineFld->rows; ++i)
        for (int j = 0; j != mineFld->columns; ++j)
            /// don't look for adjacent mines in areas where
            /// mine is already located
            if (mineFld->field[i][j] != Settings::MINE)
                mineFld->field[i][j] = proxM(mineFld, i, j);
}

/// Reveals what is under space
///If mine, you lose the game..If no mine, game resumes
bool resume(Settings * mineFld, int row, int col) {
    /// check if user selected a losing square
    if (mineFld->field[row][col] == Settings::MINE)
        return false;

        /// Square is a zero, clear the surrounding area if necessary
    else if (noMine(mineFld, row, col)) {
        revealZ(mineFld, row, col); /// show cleared area
        ckborder(mineFld);
        hideMines(mineFld);
        return true;
    }       /// Square had adjacent mine
        /// reveal the number to the user
    else {
        mineFld->field[row][col] = proxM(mineFld, row, col);
        hideMines(mineFld);
        return true;
    }
}

/// Function checks whether the player has won
bool winCase(Settings *mineFld) {
    for (int i = 0; i != mineFld->rows; ++i)
        for (int j = 0; j != mineFld->columns; ++j)
            /// if there are empty spaces player has not won
            if (mineFld->field[i][j] == Settings::NONE)
                return false;
    /// there were no empty spaces left. Player has won
    return true;
}

/// Function find the perimeter of the cleared areas
void ckborder(Settings *mineFld) {
    for (int row = 0; row != mineFld->rows; ++row) {
        /// avoid search at left and right edge of array
        for (int col = 0; col != mineFld->columns; ++col) {
            /// when you're not on the bounds of the array
            if (row > 0 && row < mineFld->rows - 1
                    && col > 0 && col < mineFld->columns - 1)
                if (mineFld->field[row][col] == Settings::NOMINE) {
                    /// check that the previous number has mines adjacent
                    if (mineFld->field[row][col - 1] != Settings::NOMINE)
                        mineFld->field[row][col - 1] = proxM(mineFld, row, col - 1);
                    /// check if the next number has mines adjacent
                    if (mineFld->field[row][col + 1] != Settings::NOMINE)
                        mineFld->field[row][col + 1] = proxM(mineFld, row, col + 1);
                    if (mineFld->field[row - 1][col] != Settings::NOMINE)
                        mineFld->field[row - 1][col] = proxM(mineFld, row - 1, col);
                    /// check if the next number has mines adjacent
                    if (mineFld->field[row + 1][col] != Settings::NOMINE)
                        mineFld->field[row + 1][col] = proxM(mineFld, row + 1, col);
                    /// check the adjacent corners
                    if (mineFld->field[row + 1][col - 1] != Settings::NOMINE)
                        mineFld->field[row - 1][col - 1] = proxM(mineFld, row - 1, col - 1);
                    if (mineFld->field[row - 1][col + 1] != Settings::NOMINE)
                mineFld->field[row - 1][col + 1] = proxM(mineFld, row - 1, col + 1);
            if (mineFld->field[row + 1][col - 1] != Settings::NOMINE)
                mineFld->field[row + 1][col - 1] = proxM(mineFld, row + 1, col - 1);
            if (mineFld->field[row + 1][col + 1] != Settings::NOMINE)
                mineFld->field[row + 1][col + 1] = proxM(mineFld, row + 1, col + 1);
          }
        }
    }
}

///Function that displays the rules of the game
void instructions(){
    cout << "The Objective:" << endl;
    cout << "To locate all the mines without clicking/uncovering them." << endl;
    cout << "If you uncover a mine, you lose the game." << endl;
    cout << " " << endl;
    cout << "Difficulty Levels:" << endl;
    cout << "Beginner: 9x9 Area: 10 mines" << endl;
    cout << "Intermediate: 16x16 35 mines" << endl;
    cout << "Expert: 25x25 45 mines" << endl;
    cout << "Disclaimer: If you choose to make a grid that isn't stated " << endl;
    cout << "above, then the number of mines will be increased or decreased." << endl;
    cout << "accordingly" << endl;
    cout << "    " <<endl;
    cout << "Your MineSweeper board will always be of size nxn." << endl;
    cout << "Example: When n=4" << endl;
    cout << "   1 2 3 4 " << endl;
    cout << " 1 x x x x " << endl;
    cout << " 2 x x x x " << endl;
    cout << " 3 x x x x " << endl;
    cout << " 4 x x x x " << endl;
    cout << "    " << endl;
    cout << "Gameplay:" << endl;
    cout << "When the game begins, every square will be covered," << endl;
    cout << "and it is up to you to discover in which squares the mines are." << endl;
    cout << "By selecting a row, followed by a column, you will reveal one " << endl;
    cout << "of the squares underneath. It will either be blank, marked by" << endl;
    cout << "a number to indicate how many adjacent mines are present, or a" << endl;
    cout << "mine. Uncovering a mine will end the game." << endl;
    cout << " " << endl;
    cout << "Numbered Squares: " << endl;
    cout << "These are the squares you use to decipher where the mines are hidden" << endl;
    cout << "Each of these has a number which determines how many mines are in the" << endl;
    cout << "surrounding 8 squares(e.g. North, North-East, East, South-East, South," << endl;
    cout << "South-West, West, and North-West of the square)." << endl;
    cout << " " << endl;
    cout << "Good luck!" << endl;
    cout << " " << endl;
```

```
};

//character array with strings to satisfy rubric
char *playerN() {
   cout << "Enter your name: ";
   string in;
   cin >> in;

   short size = in.size();
   char *name = new char[size+1];
   for (short i = 0; i != size; ++i) {
      *(name+i) = in[i];
   }
   *(name+size+1) = '\0';

   return name;
}

///Writes Settings structure to binary file
void writeBinary(Settings *mineFld, string fileName) {
   fstream out(fileName.c_str(), ios::out | ios::binary);    /// Open the file
   out.write(reinterpret_cast<char *>(&mineFld),sizeof(*mineFld)); ///Write to the file
   out.close();
}

///Reads from Binary File
void readFile(string fileName) {
   char response;
   cout << "Would you like to view a previous game from a binary file?" << endl;
   cout << "Hit 'y' if yes: " << endl;
   cin >> response;
   if (response == 'y') {
      cout << "     " << endl;
      cout << "Results from the last game:" << endl;
      /// Create space to hold the file read
      Settings *result;
      fstream in(fileName.c_str(), ios::in | ios::binary);
      in.read(reinterpret_cast<char *>(&result), sizeof(*result));
      revealM(result);
      in.close();
   }

}
```