

Minesweeper

Brynn Caddel

CSC 17A - Spring 2015 - 43950

Project 2 Write Up

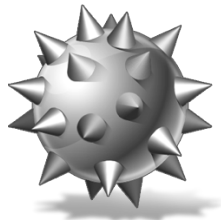


Table of Contents

Introduction	3
Summary of Game and Gameplay Instructions	3
Development Summary	3
References	5
Sample Input/Output	6
Concepts Implemented	11
Major Variables	12
Virtual Void Functions	12
Classes: Public and Private	13
Functions	14
Proximity Mine Function Description	16
Flowcharts	18
UML	20
Program	21

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 1012 lines long and implements concepts that are the basis of this class (pun intended...).

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 $n \times n$ 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:

Project 1 Development : 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.

Project 2 Development: For the updated version of this game, I implemented concepts that we learn about in the latter portion of this class. The first things I did was converted all of my structures to classes. I decided what functions I would make private and which I would make public. I decided to make my private functions those that conceal and reveal mines, randomly distribute mines across the board, and determine where mines are in proximity to the user selected space. The functions I decided to make public were those that require user interaction, such as functions that allows users to select rows/columns, save/load the game, clear the board, and restart the game.

I did my best to utilize inheritance and polymorphism in my game. I created an abstract class, creatively named “AbstractClass.h,” that contained virtual void functions that set rows, columns, configured the game, and printed finished games. My goal was to make it so you could modify the program and make any type of game containing a row by column board. Just as in class, we created an abstract card class that would allow us to create any type of card game from the virtual void card deck functions.

Another concept I implemented in my game was the idea of class hierarchies. The class that all other classes inherit from in the AbstractClasses class. From there, the Board header and its classes inherit from the abstract class. The GameData header and its classes then inherit from the Board header.

- The sequence of inheritance is as follows:
 - Abstract > Board > GameData

Except for the syntax of each function, their roles in the game remained unchanged in the updated version. The only major addition to functions I made was the inclusion of save, load, and time functions. After reading the Gaddis book and doing some searching online, I finally figured out how to successfully implement binary files. The time function I used in the game, while not entirely necessary, allowed the user to keep track of how long each turn took them to complete. When the user ends the game, either by winning or losing, it outputs the duration of the game. I did this so that the game would resemble the original Minesweeper, which has a timer embedded into it.

One feature that I would eventually like to integrate into the game is being able to set flags, just as it's done in the original Minesweeper game. An approach that I thought would be practical would be to assign a different character, maybe an exclamation point, for flags that the user would be able to input, and this character wouldn't reveal the contents of the square, but mark it as a flagged square.

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

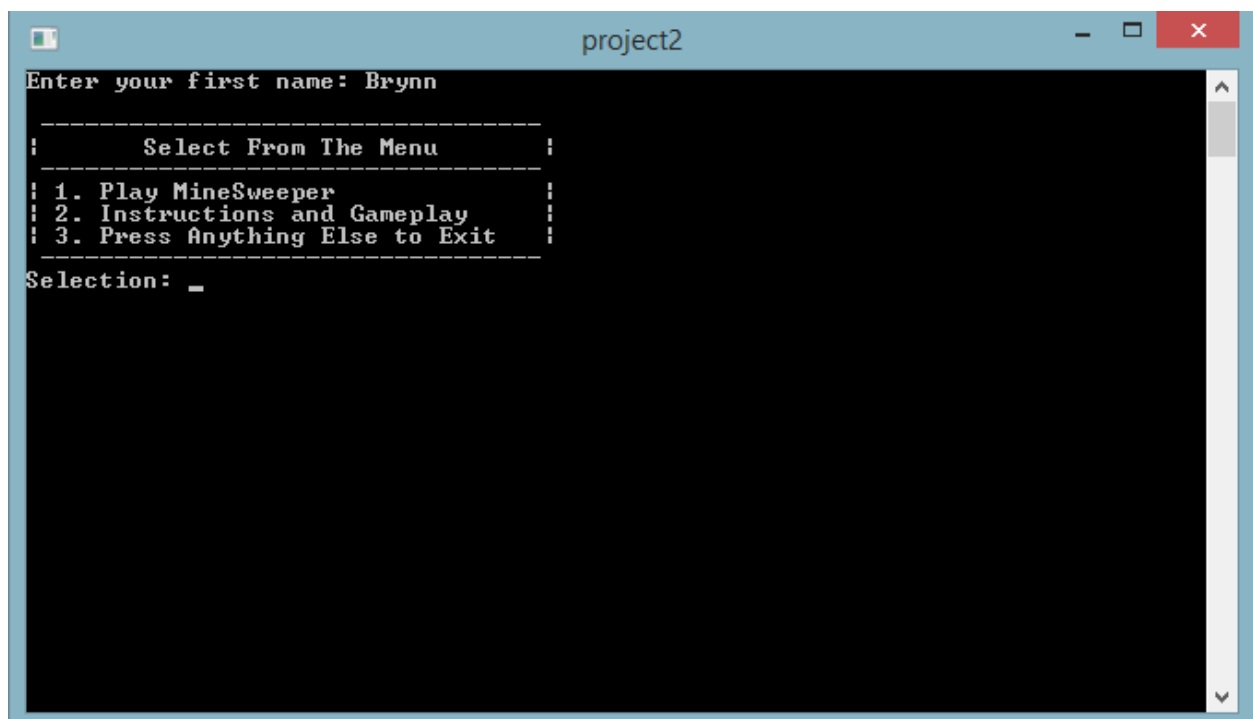
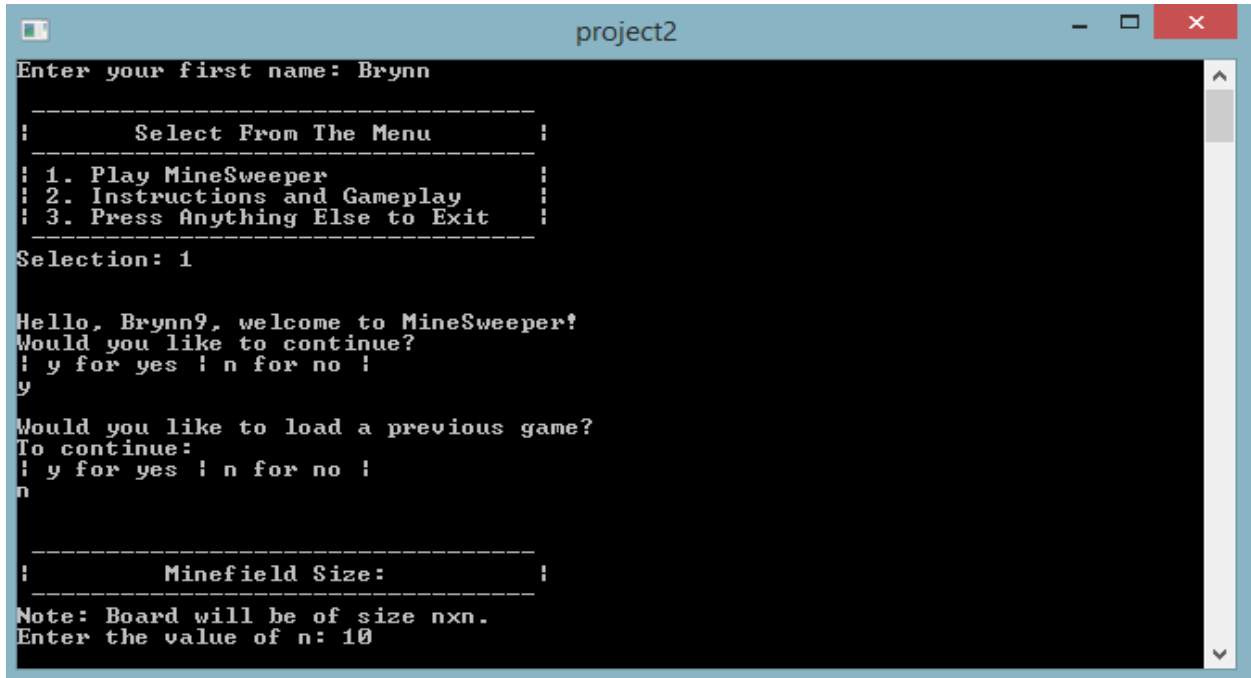


Figure 3.

In the first version of this game that I worked on for Project 1, I didn't include an option to save and load games. In this updated version, I've successfully implemented reading and writing to binary files so that I can save and load boards. In the picture below, I show what it looks like to have the option to either load a previous board, or continue onto a new board.



```

project2
Enter your first name: Brynn

-----
:       Select From The Menu       :
-----
: 1. Play Minesweeper              :
: 2. Instructions and Gameplay     :
: 3. Press Anything Else to Exit   :
-----
Selection: 1

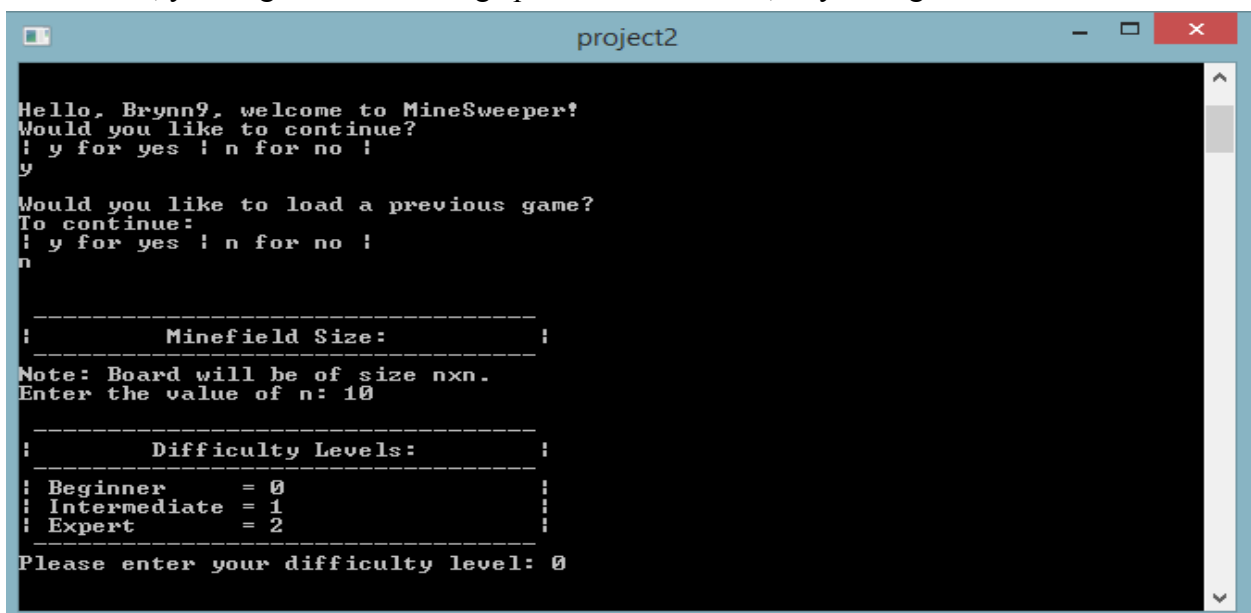
Hello, Brynn9, welcome to Minesweeper!
Would you like to continue?
: y for yes : n for no :
y

Would you like to load a previous game?
To continue:
: y for yes : n for no :
n

-----
:       Minefield Size:             :
-----
Note: Board will be of size nxn.
Enter the value of n: 10

```

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 $n \times n$. 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.



```

project2

Hello, Brynn9, welcome to Minesweeper!
Would you like to continue?
: y for yes : n for no :
y

Would you like to load a previous game?
To continue:
: y for yes : n for no :
n

-----
:       Minefield Size:             :
-----
Note: Board will be of size nxn.
Enter the value of n: 10

-----
:       Difficulty Levels:          :
-----
: Beginner   = 0                   :
: Intermediate = 1                 :
: Expert     = 2                   :
-----
Please enter your difficulty level: 0

```

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; $x \in \text{Positive Real Number}$

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

Probability of hitting a mine on your first try:

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

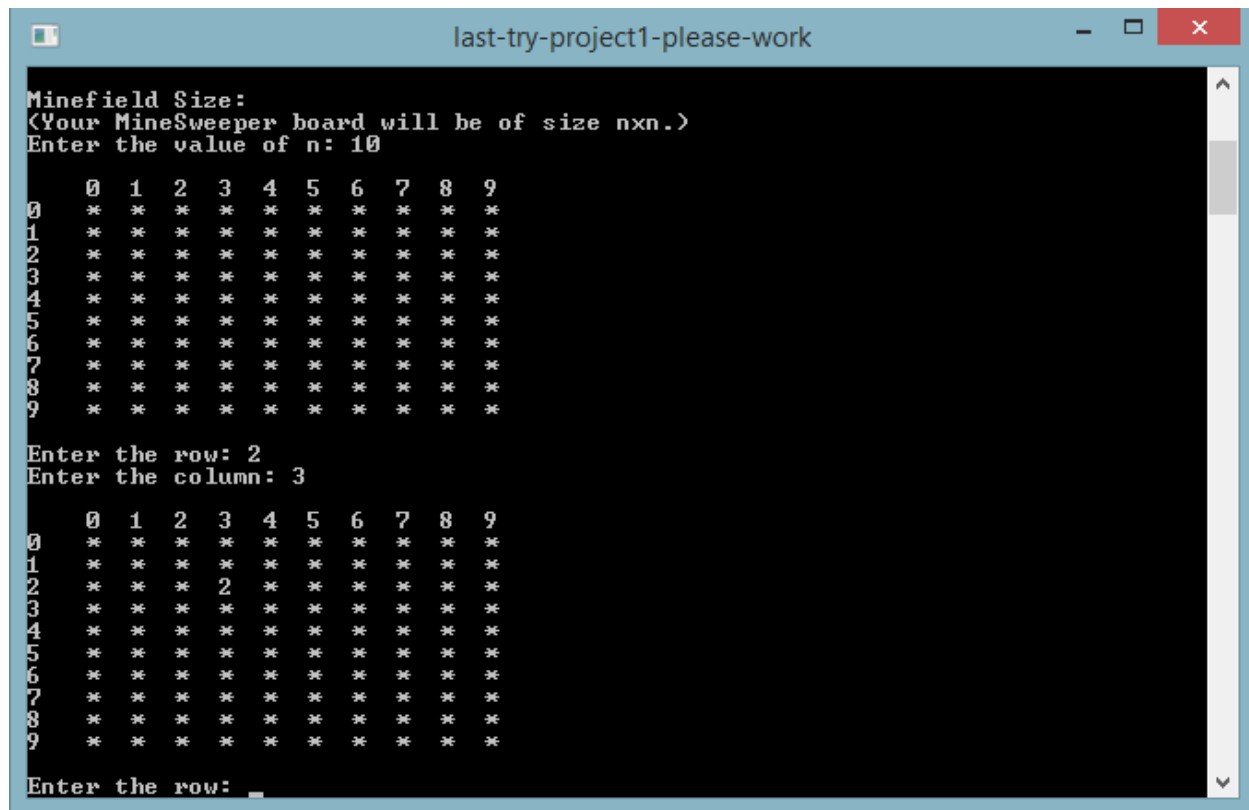


Figure 3. (Displayed above) This displays a board that has chosen a row/column. The number “2” is displayed to show how many mines are adjacent to that particular space. The user now has the option of strategically choosing a space around the “2,” or choosing another spot on the board that isn’t near 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.

```

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.

```

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:

In the **Line** column, every unspecified line can be found in main. “GD.h” corresponds to GameData.h. “Ab.h” corresponds to AbstractClasses.h.” “B.h” corresponds to Board.h.

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
12.7	Binary Files	loadGame, saveGame	584, 592
13.2	Introduction to Classes	class: Settings	GD.h 7
13.3	Defining a Class	private/public components	GD.h 8,35
13.7	Constructors	Matrix(row, col)	GD.h 38
13.9	Destructors	Matrix(row, col)	GD.h 38
15.1	Concept of Inheritance	void Matrix::config()	638
15.6	Polymorphism	void Matrix::config()	638
15.7	Class Hierarchies	Abstract > Board > GameData	Ab,B,GD.h
15.6	Virtual Member Functions	Abstract Header	Ab.h
16.3	Class Templates	Template Header	T.h

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

Virtual Void Functions:

Name	Purpose	Line
virtual void vRows	Sets the rows.	Line 11
virtual void vColmn	Sets the columns.	Line 12
virtual int retRows	Retrieves the rows.	Line 13
virtual int retColmn	Retrieves the columns.	Line 14
virtual void config	Configures game.	Line 10
virtual void print	Prints out outcome of game.	Line 9

Classes: Private

Descriptions of all the functions can be found in the **Functions** table below.

Name	Line
enum Difficulty	10
enum Squares	15
bool check	21
bool cont	22
bool noMine	23
bool winCase	24
char *playerN	25
int nMines	26
void create	27
void ckborder	28
void flagSet	29
void mineSet	30
int proxM	31
void revealZ	32
void userIn	33

Classes: Public

Descriptions of all the functions can be found in the **Functions** table below.

Settings	Line
int retColmn	43
int retRows	47
int getMines	51
void config	54
void instructions	55
void loadGame	56
void run	57
void print	58
void revealM	59
void saveGame	60
void vColmn	61
void vRows	62
void wipeB	63

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	Line 157

	space. If mine, then end game. If clear, resume game.	
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 $\text{rand()} \% 15 == 0$	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 saveGame	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 449

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

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

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/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.

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

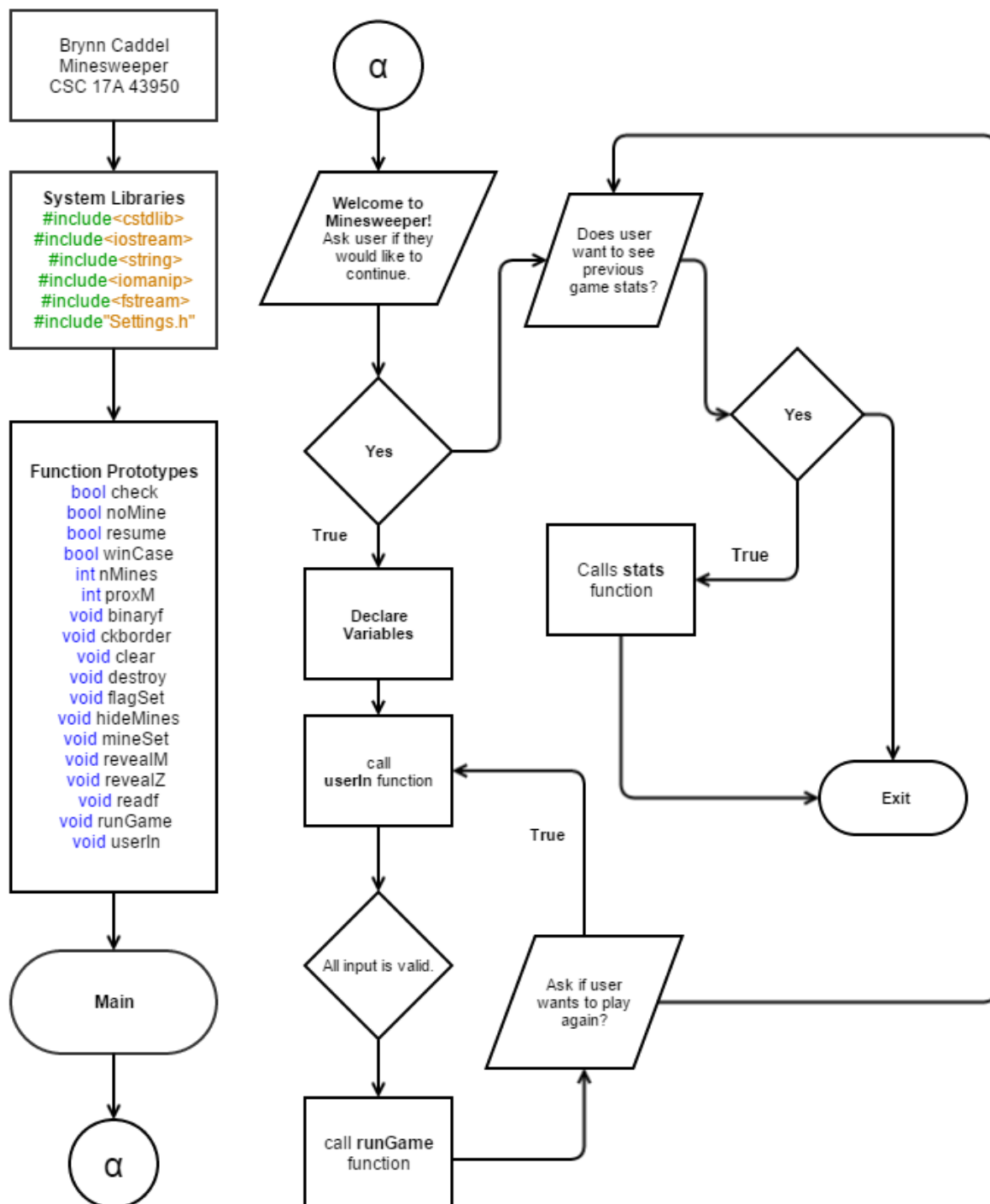
X			X
X			X
X			X
X			X

X	X	X	X
X	X	X	X

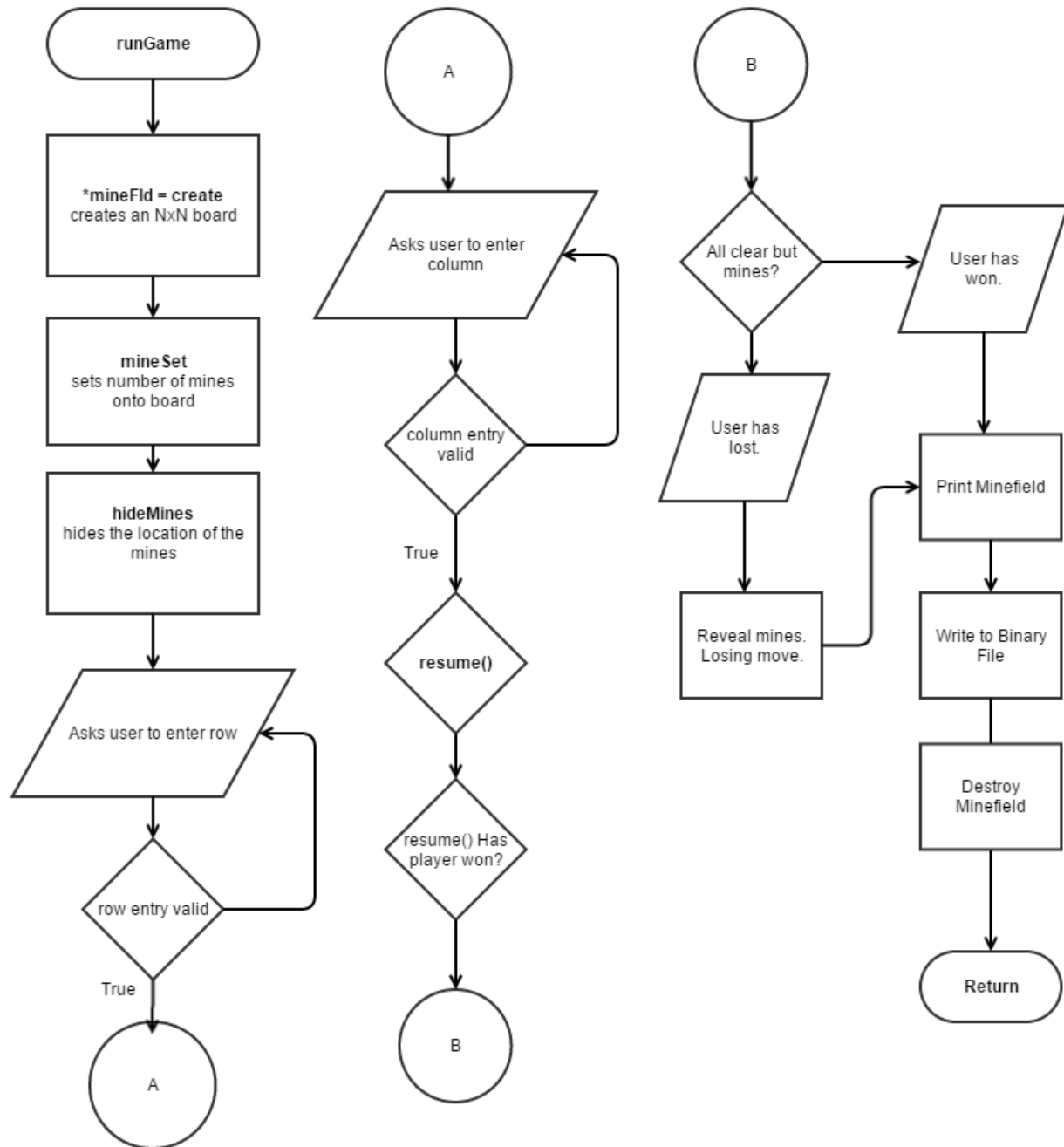
Within this function I added additionally comments that describe each area of the board the function inspects.

- 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

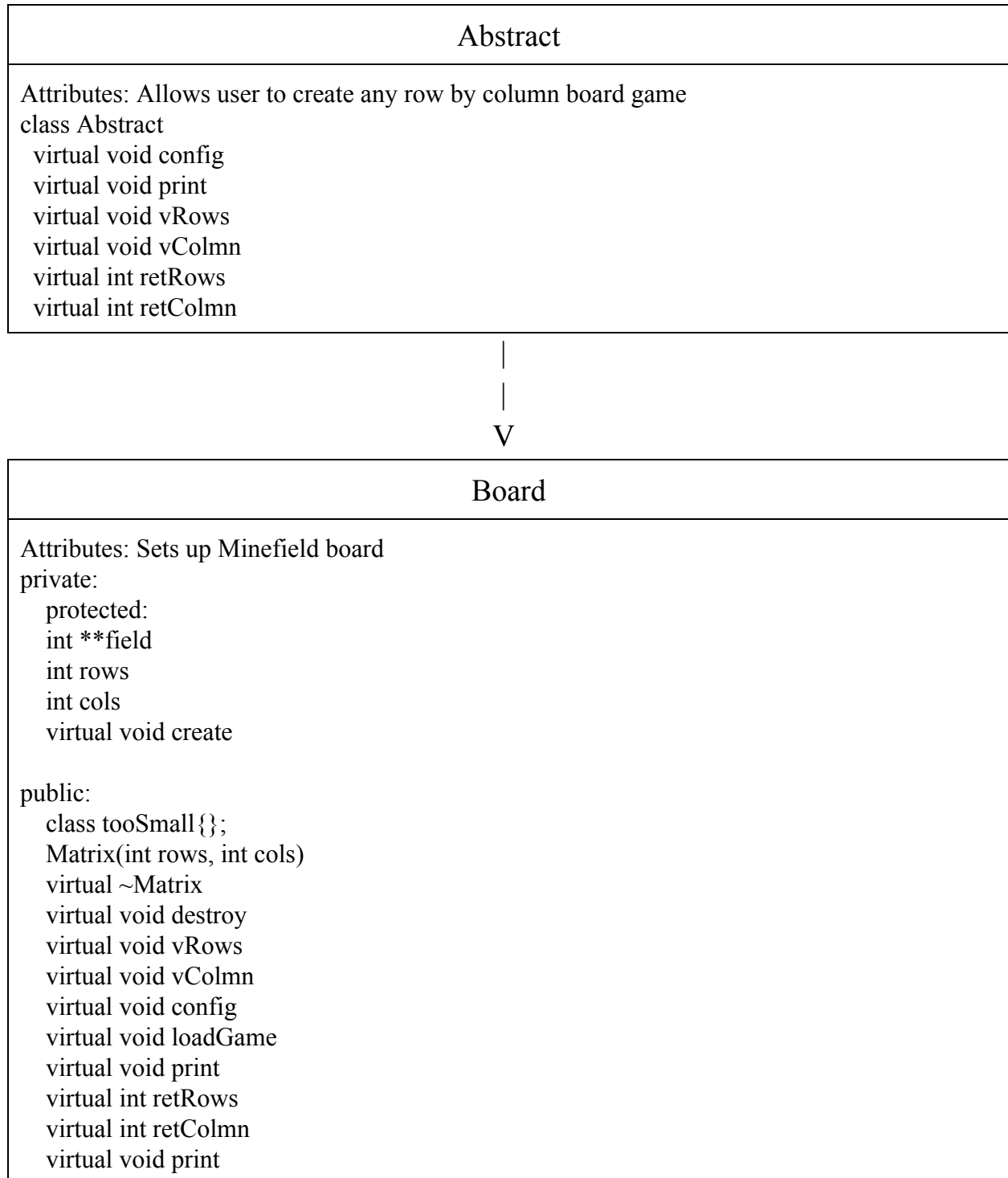
Main (Before I added my Menu):



runGame Function:



UML:



|
|
V

Game Data

Attributes: Determines placement and existence of mines on board

```
class Settings : public Matrix {
```

```
private:
```

```
    enum Difficulty
    enum Squares
    int mines
    Settings::Difficulty convert
    bool check
    bool cont
    bool noMine
    bool winCase
    char *playerN
    int nMines
    void create
    void ckborder
    void flagSet
    void mineSet
    int proxM
    void revealZ
    void userIn
```

```
public:
```

```
    int retColmn
    int retRow
    int getMines
    void config
    void instructions
    void loadGame
    void run
    void print
    void revealM
    void saveGame
    void vColmn
    void vRows
    void wipeB
```

MAIN:

```
/*
 * File:  main.cpp
 * Author: Brynn Caddel
 * Project 2 - MineSweeper
 * CSC17A - 43950
 */

///System Libraries
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <string>
#include <iomanip>
#include <vector>
#include "GameData.h"
#include "Board.h"
#include "Template.h"
#include "Abstract.h"

using namespace std;

//Execution Begins Here

int main(int argc, const char * argv[]) {
    ///Used when recording the length of a game
    srand(static_cast<unsigned int> (time(0)));

    ///Used for Template
    BoardGame<Matrix> m(new Settings(10, 10));

    try {
        m->config();
    }
    //Error Checking:
    //catch used to determine if board is too large or too small
    catch (Settings::tooSmall) {
        cout << "Error: Minefield was either too small or too large."
              << "If you've entered a board size larger than 10x10,"
              << "try entering a smaller size. Or your board may be too"
              << "small to be compatible with your selected difficulty level."
              << "Try selecting a larger board if you would like to play on"
```

```

        "Difficulty: EXPERT. ";
    } catch (const char* ord) {
        cout << ord << endl;
    }

    return 0;
}

//Function Definitions

// Function returns true if input is valid

bool Settings::check() const {
    ///Makes sure that the number of mines does not exceed spots in
    ///the minefield
    return (((rows * cols) > mines) && (mines > 0));
}

// Function reveals what is underneath the square that the user has selected
// and whether to continue based on what is revealed
// i.e selecting a mine means you lost, game over
bool Settings::cont(int row, int col) {
    /// check if user selected a losing square
    if (field[row][col] == Settings::MINE)
        return false;

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

// Function returns true if
// there are 0 mines adjacent to selected square
bool Settings::noMine(int row, int col) const {
    if (proxM(row, col))
        return false; /// nAdjacent returned 1 or more
    return true; /// nAdjacent returned 0
}

```

```

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

///This creates the board based off user selections of size
void Settings::create(int row, int col) {
    ///Used when creating dynamic board
    rows = row;
    cols = col;

    /// Creates each of the rows
    field = new int *[rows];

    /// Creates each of the columns
    for (int row = 0; row != rows; ++row)
        field[row] = new int [cols];
    ///iterates to create board
}

///If input is less than or equal to 0, it throws input
void Settings::vRows(int row) {
    if (row <= 0)
        throw tooSmall();
    rows = row;
}

///If input is less than or equal to 0, it throws input
void Settings::vColumn(int col) {
    if (col <= 0)
        throw tooSmall();
    cols = col;
}

///Function prompts user to enter difficulty level and minefield size
void Settings::userIn() {
    ///Allows user to input the size of their board
    cout << "          " << endl;
    cout << " ----- " << endl;
    cout << "|      Minefield Size:      |" << endl;
}

```



```

cout << " ----- " << endl;
cout << "Note: Board will be of size nxn." << endl;
cout << "Enter the value of n: ";
int row;
cin >> row;
/// Invalid sizes. Restricted between 1 and 10 to ensure it can be saved in
/// a binary file with out blowing things up
if (row > 10 || row < 1)
    throw tooSmall();
rows = row;
cols = row;
int rigour;
///Allows user to enter difficulty of their game
cout << "                " << endl;
cout << " ----- " << endl;
cout << "|    Difficulty Levels:    |" << endl;
cout << " ----- " << endl;
cout << "| Beginner    = 0        |" << endl;
cout << "| Intermediate = 1        |" << endl;
cout << "| Expert      = 2        |" << endl;
cout << " ----- " << endl;
cout << "Please enter your difficulty level: ";
cin >> rigour;
cout << "                " << endl;
cout << "                " << endl;
mines = nMines(convert(rigour));
}

void Settings::config() {
    int choice;
    bool exitMenu = true;
    char *player = playerN();
    ///Loop: Implemented Until Exit
    cout << "    " << endl;
    do {
        ///Output Menu
        cout << " ----- " << endl;
        cout << "|    Select From The Menu    |" << endl;
        cout << " ----- " << endl;
        cout << "| 1. Play MineSweeper        |" << endl;
        cout << "| 2. Instructions and Gameplay |" << endl;
        cout << "| 3. Press Anything Else to Exit |" << endl;
        cout << " ----- " << endl;
        cout << "Selection: ";
        cin >> choice;
        cout << "    " << endl;
        cout << "    " << endl;
        switch (choice) {

```

```

case 1:
    /// ask user if they want to play
    cout << "Hello, " << player << ", welcome to Minesweeper!" << endl;
    cout << "Would you like to continue? " << endl;
    cout << "| y for yes | n for no |" << endl;
    char input;
    cin >> input;
    cout << "      " << endl;

    /// If yes, then it prompts the user to play the game
    if (input == 'y') {
        cout << "Would you like to load a previous game? " << endl;
        cout << "To continue:" << endl;
        cout << "| y for yes | n for no |" << endl;
        char inputdeux;
        cin >> inputdeux;
        cout << "      " << endl;

        ///Loads the previous game
        if (inputdeux == 'y') {
            loadGame();
        } else
            /// Retrieves row and column information from user
            userIn();
        ///Verifies input
        if (check()) {
            while (input == 'y' && check()) {
                ///Runs the actual program
                run();
                cout << endl;
                cin.ignore();
                cout << "Would you like to play again?" << endl;
                cout << "To continue:" << endl;
                cout << "| y for yes | n for no |" << endl;
                cin >> input;
                cout << endl;
                /// Retrieves data if user wants to continue
                if (input == 'y') {
                    userIn();
                    ///Wipes the boards
                    wipeB();
                }
            }
        } ///Error Checking: information was invalid
        else
            throw tooSmall();
    }
    cout << "Game Over." << endl;

```

```

        /// Taking out the trash
        delete player;
        break;

    case 2:
        instructions();
        break;

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

} while (exitMenu);

}

///Function allows user to enter rows, columns, and difficulty
void Settings::run() {
    ///Calls function that sets mines
    mineSet();
    ///Calls function that reveals the mines
    revealM();
    int row, col;
    ///Initializes turn at 0 to start turn count
    int turn = 0;
    ///Implemented to record length of game
    int initialTime = static_cast<unsigned int> (time(0));
    do {
        int tNought = static_cast<unsigned int> (time(0));
        cout << "Turn: " << turn++ << endl;
        /// Row Selection
        do {
            ///Allows user to save and exit
            cout << "To save and exit: enter -1" << endl;
            cout << "Row: ";
            cin >> row;
            /// If user wants to save game and exit, they can press 0
            if (row == -1) {
                ///Calls function to save game board information
                saveGame();
                return;
            }
            ///Error checking: checks validity of bounds for rows
        } while (row < 0 || row >= rows);
        /// Column Selection

```

```

do {
    cout << "Column: ";
    cin >> col;
    ///Error Checking: checks validity of bounds for columns
} while (col < 0 || col >= cols);

/// End Process
int tFinal = static_cast<unsigned int> (time(0));
cout << "Turn took: " << tFinal - tNought << " seconds.\n";
cout << endl;
} while (cont(row, col) && !winCase());

///Once the game has either been won or lost
///Win Case
if (winCase()) {
    cout << "You win! " << endl;
    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();
}    ///Lose Case
else {
    cout << "*boom* You have lost the game..." << endl;
    flagSet();
    field[row][col] = Settings::LOSER;
}

///This displays the length of time it took for user to complete game
int finalTime = static_cast<unsigned int> (time(0));
cout << "You've completed the game in"
    " " << finalTime - initialTime << " seconds." << endl;

///This prints the final board
print();
}

///Character array with strings to satisfy rubric
char* Settings::playerN() {
    cout << "Enter your first name: ";
    string in;
    cin >> in;
    typedef string::size_type sType;
    sType size = in.size();
    char *name = new char[size + 1];
    for (sType i = 0; i != size; ++i) {
        *(name + i) = in[i];
    }
    *(name + size + 1) = '\0';
}

```

```

    return name;
}

/// Function that clears the grid on which game will be played
void Settings::wipeB() {
    /// Make sure each square is empty
    for (int i = 0; i != rows; ++i)
        for (int j = 0; j != rows; ++j)
            field[i][j] = Settings::NONE;
}

///This function takes the int variable and returns it as Settings::Difficulty
Settings::Difficulty Settings::convert(int choice) {
    ///The purpose of these cases is to determine how many mines to set
    switch (choice) {
        case (0):
            return Settings::BEGINNER;
            break;
        case (1):
            return Settings::INTERMEDIATE;
            break;
        case (2):
            return Settings::EXPERT;
        default:
            return Settings::BEGINNER;
            break;
    }
}

///Once the game is won/lost, this function prints the location of the mines
///revealed
void Settings::print() const {
    cout << "Board: ";
    for (int row = 0; row != rows; ++row) {
        for (int col = 0; col != cols; ++col) {
            ///Used to identify mines/empty spots/proximity to mines
            if (*(field + row) + col == Settings::LOSER)
                cout << "t ";
            else if (*(field + row) + col == Settings::MINE)
                cout << "x ";
            else if (!noMine(row, col))
                cout << proxM(row, col) << " ";
            else
                cout << "0 ";
        }
        cout << endl;
    }
}

```

```

    cout << endl;
}

///Once the game is won/lost, this function prints the location of the mines
///revealed
void Settings::revealM() const {
    /// Print the column index
    for (int i = 0; i != cols; ++i) {
        if (i == 0)
            cout << " ";
        cout << setw(3) << i;
    }
    cout << endl;
    for (int row = 0; row != rows; ++row) {
        for (int col = 0; col != cols; ++col) {
            if (col == 0 && row < 10) cout << row << " ";
            if (col == 0 && row >= 10) cout << row << " ";
            ///Keeps empty squares and mines hidden
            if (field[row][col] == Settings::NONE ||
                field[row][col] == Settings::MINE)
                cout << setw(3) << right << "+ ";
            ///Reveals cleared areas
            else if (field[row][col] == Settings::NOMINE)
                cout << setw(2) << 0 << " ";
            /// Print out the actual value of the square
            else
                cout << setw(2) << field[row][col] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

///There are a varying number of mines laid out depending on the user inputted
///difficulty level
int Settings::nMines(Settings::Difficulty userSet) const {
    if (userSet == Settings::BEGINNER)
        return (rows * cols) / 10;
    else if (userSet == Settings::INTERMEDIATE)
        return (rows * cols) / 6;
    else
        return (rows * cols) / 4;
}

///Function randomly places mines in the field
void Settings::mineSet() {
    ///Determines how many mines will be used
    int copyM = mines;

```

```

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

///Determine how many mines are adjacent to selected square
int Settings::proxM(int row, int col, int FLAG) const {
    ///The number of adjacent mines
    int adjMine = 0;
    ///Ignore first/last row/column
    /// Search interior
    if (row > 0 && col > 0 && row < rows - 1 && col < cols - 1) {
        for (int i = row - 1; i <= row + 1; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (field[i][j] == FLAG)
                    ++adjMine;
        }
    }
    /// checks first row, doesn't check first or last column
    else if (row == 0 && col > 0 && col < cols - 1) {
        for (int i = row; i <= row + 1; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (field[i][j] == Settings::MINE)
                    ++adjMine;
        }
    }
    /// checks last row, doesn't check first or last column
    else if (row == rows - 1 && col > 0 && col < cols - 1) {
        for (int i = row - 1; i <= row; ++i) {
            for (int j = col - 1; j <= col + 1; ++j)
                if (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 < rows - 1) {
    for (int i = row - 1; i <= row + 1; ++i) {
        for (int j = col; j <= col + 1; ++j)
            if (field[i][j] == Settings::MINE)
                ++adjMine;
    }
} // checks last column, doesn't check first or last row
// sweeps to the left
else if (col == cols - 1 && row > 0 && row < rows - 1) {
    for (int i = row - 1; i <= row + 1; ++i) {
        for (int j = col - 1; j <= col; ++j)
            if (field[i][j] == Settings::MINE)
                ++adjMine;
    }
} // checks top left corner
else if (row == 0 && col == 0) {
    if (field[row][col + 1] == Settings::MINE) ++adjMine;
    if (field[row + 1][col] == Settings::MINE) ++adjMine;
    if (field[row + 1][col + 1] == Settings::MINE) ++adjMine;
} // checks top right corner
else if (row == 0 && col == cols - 1) {
    if (field[row][col - 1] == Settings::MINE) ++adjMine;
    if (field[row + 1][col] == Settings::MINE) ++adjMine;
    if (field[row + 1][col - 1] == Settings::MINE) ++adjMine;
} // checks bottom left corner
else if (row == rows - 1 && col == 0) {
    if (field[row - 1][col] == Settings::MINE) ++adjMine;
    if (field[row - 1][col + 1] == Settings::MINE) ++adjMine;
    if (field[row][col + 1] == Settings::MINE) ++adjMine;
} // checks bottom right corner
else if (row == rows - 1 && col == cols - 1) {
    if (field[row - 1][col - 1] == Settings::MINE) ++adjMine;
    if (field[row - 1][col] == Settings::MINE) ++adjMine;
    if (field[row][col - 1] == Settings::MINE) ++adjMine;
}
// return number of mines from appropriate if statement

return adjMine;
}

// Clear an area whose values are CLEAR
// i.e 0 adjacent mines
void Settings::revealZ(int row, int col) {
    // check bounds
    if (row >= rows || row < 0 || col >= cols || col < 0)
        return;
    if (noMine(row, col) && field[row][col] != Settings::NOMINE) {
        field[row][col] = Settings::NOMINE;
    }
}

```



```

    /// go up one row
    revealZ(row + 1, col);
    /// go down one row
    revealZ(row - 1, col);
    /// go right one col
    revealZ(row, col + 1);
    /// go left one col
    revealZ(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 Minesweeper
void Settings::flagSet() {
    for (int i = 0; i != rows; ++i)
        for (int j = 0; j != cols; ++j)
            /// don't look for adjacent mines in areas where
            /// mine is already located
            if (field[i][j] != Settings::MINE)
                field[i][j] = proxM(i, j);
}

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

```

```

        field[row - 1][col + 1] = proxM(row - 1, col + 1);
    if (field[row + 1][col - 1] != Settings::NOMINE)
        field[row + 1][col - 1] = proxM(row + 1, col - 1);
    if (field[row + 1][col + 1] != Settings::NOMINE)
        field[row + 1][col + 1] = proxM(row + 1, col + 1);
    }
    }
    }
}

void Settings::saveGame() {
    fstream saveFile("gameSave", ios::out | ios::binary);
    saveFile.write(reinterpret_cast<char*> (this), sizeof (*this));
    saveFile.close();
}

/// Function prints the data variable from the Minesweeper structure
/// written to a binary file
void Settings::loadGame() {
    fstream saveFile("gameSave", ios::in | ios::binary);
    if (!saveFile.is_open())
        throw "No previous settings found\n";

    saveFile.read(reinterpret_cast<char*> (this), sizeof (*this));
    //print();
    saveFile.close();
}

Settings& Settings::operator=(const Settings &right) {
    create(right.retRows(), right.retColmn());

    for (int i = 0; i != right.retRows(); ++i) {
        for (int j = 0; j != right.retColmn(); ++j)
            field[i][j] = right[i][j];
    }
    return *this;
}

///This creates the board based off user selections of size
void Matrix::create(int row, int col) {
    ///Used when creating dynamic board
    rows = row;
    cols = col;

    /// Creates each of the rows
    field = new int *[rows];

```

```
    /// Creates each of the columns
    for (int row = 0; row != rows; ++row)
        field[row] = new int [cols];
    ///iterates to create board
}

/// Function that clears the grid on which game will be played
///Allows user to play game again
void Matrix::wipeB() {
    for (int i = 0; i != rows; ++i)
        for (int j = 0; j != cols; ++j)
            field[i][j] = 0;
}

void Matrix::loadGame() {
}

void Matrix::config() {
}

/// Function deallocates memory
void Matrix::destroy() {
    ///Deletes each row
    for (int i = 0; i != rows; ++i)
        delete[] field[i];
    ///Deletes the dynamically allocated structures
    delete field;
}

void Matrix::vRows(int row) {
    ///Checks whether the input value for rows is less than or equal to 0
    if (row <= 0)
        throw tooSmall();
    rows = row;
}

void Matrix::vColmn(int col) {
    ///Checks whether the input value for columns is less than or equal to 0
    if (col <= 0)
        throw tooSmall();
    cols = col;
}

void Matrix::print() const {
    for (int i = 0; i != rows; ++i) {
        for (int j = 0; j != cols; ++j) {
            std::cout << field[i][j] << " ";
        }
    }
}
```

```

    }
    std::cout << std::endl;
}
}

///Function that displays the rules of the game
void Settings::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;
};

```

ABSTRACT CLASS:

```

#ifndef AbstractClass
#define AbstractClass

class Abstract {
    virtual void config()=0;
    virtual void print() const = 0;
    virtual void vRows(int)=0;
    virtual void vColmn(int)=0;
    virtual int retRows() const =0;
    virtual int retColmn() const =0;
};

#endif

```

GAME DATA CLASS:

```

#ifndef GameData
#define GameData

#include <string>
#include "Board.h"

class Settings : public Matrix {
private:
    //Used in determining how many mines are placed on board
    enum Difficulty {
        BEGINNER, INTERMEDIATE, EXPERT
    };

    //Used to keep track of empty spaces and spaces with mines
    enum Squares {
        NONE = 10, MINE, NOMINE, LOSER
    };
    int mines;
    //Descriptions of each function can be found in Function Definitions in main
    Settings::Difficulty convert(int);
    bool check() const;
    bool cont(int, int);
    bool noMine(int, int) const;
    bool winCase() const;
    char *playerN();
    int nMines(Settings::Difficulty) const;
    void create(int, int);
    void ckborder();
    void flagSet();

```

```

void mineSet();
int proxM(int, int, int = Settings::MINE) const;
void revealZ(int, int);
void userIn();

public:
    ///Constructors and Destructors

    Settings(int row, int col) : Matrix(row, col) {
        wipeB();
    }
    ///Descriptions of each function can be found in Function Definitions in main

    int retColumn() const {
        return cols;
    }

    int retRows() const {
        return rows;
    }

    int getMines() const {
        return mines;
    }
    void config();
    void instructions();
    void loadGame();
    void run();
    void print() const;
    void revealM() const;
    void saveGame();
    void vColumn(int);
    void vRows(int);
    void wipeB();

    Settings& operator=(const Settings&);
};

#endif /* defined(__Project_2__Minesweeper__) */

```

BOARD CLASS:

```

#ifndef BoardHeader
#define BoardHeader

```

```
#include "Abstract.h"

class Matrix: public Abstract{
private:

protected:
    ///Holds the game's data
    int **field;
    ///Rows on minefield
    int rows;
    ///columns on minefield
    int cols;
    ///Used to create the gameboard
    virtual void create(int, int);

public:
    /// Throw this if user tries to set negative row or column
    class tooSmall{};

    Matrix(int rows, int cols) {create(rows,cols);wipeB();}
    virtual ~Matrix(){destroy();}
    virtual void destroy();
    virtual void vRows(int);
    virtual void vColmn(int);
    virtual int retRows() const {return rows;}
    virtual int retColmn() const {return cols;}
    virtual void wipeB();
    virtual void config();
    virtual void loadGame();
    virtual void print() const;
    int* operator[](int index) { return field[index];}
    int* operator[](int index) const { return field[index];}
};

#endif
```

