Project 1
<MINESWEEPER>

CIS 17A
46721

July 15, 2020

Gabrielle Ante

Introduction

Game: Minesweeper

Minesweeper is a game where a set number of mines are secretly scattered around the board. The player's goal is to reveal or clear out all the cells of the boards except for the cells where the mines are. To accomplish this, the player picks a cell at a time and determines whether that cell has a mine. If the player thinks the chosen cell has a mine underneath, then the player flags it (or just leaves it alone). If the player thinks the cell doesn't have a mine, then the player can clear it. To help the player solve the puzzle, cells that don't have a mine have numbers that tell the player how many mines surround that particular cell. The player automatically loses if they clear a cell containing a mine.

This game was chosen for the project because I am familiar with the rules of the game and I thought it would be simple enough to implement in code. Not to mention, I thought it would be fun to implement a game board that essentially reveals what it is hidden underneath, be it a value or a mine. This feature of revealing what is inside of a cell is important because it provided a way to apply the concepts learned.

Summary

Project size: 748 lines of code
Number of Variables: 50+ identifiers

This project meets the criteria for a first project as it does not contain classes, incorporates the lessons from Chapter 9-12 of the Gaddis textbook and has more than 250 lines of code. In general, this was a challenging project as it took time to plan the constructs to apply to the game. The approach on building this code was to create a working code first, or rather build the game, and then from there apply the concepts and integrate it into the code. Based on the time stamps from Version 1 of the code, creating the program took 9 days to complete. Perhaps the most challenging aspect was in reading and writing the binary files because there were times when I could not see that the data was read or written and sometimes I didn't know which of the two wasn't working. I was essentially "mirroring" the read and write codes to troubleshoot it.

Description

I first started by getting the display board to work (Version 1). Getting started, I knew that the minesweeper board was going to be displayed as a 2D array and therefore it can be represented with pointer to pointer variables (**board as equivalent to board[i][j]). With each cell containing either a mine or a value and also being hidden, displayed or flagged; a struct was used to track each cell's description. To help display the board, this code involved initializing the board variables through init() and then printing out the board to the console through dspBrd(). The symbols mean as follows:

- * = hidden, the cell has not yet revealed whether it has a mine

- % = the mine, if the player sees this then the cell was revealed to have a mine and the game is over
- - = empty, the cell does not have a mine nor does it have a mine in the cells around it
- (a number) = value, if the player sees this then the player has chosen a cell with this many mines in the cells around it
- P = flag, the player has determined that this cell has a mine and has protected it from being revealed

Since the creation of the minesweeper board involved input from the user for the number of rows, columns, and mines, input validation was added to limit the number of the mines (Version 2) This is based on what the user had already inputted for the number of rows and columns. The destroy function was also added to counter the usage of *new*, which allows to dynamically allocate the array. The function contains *delete* to prevent a memory leak that would have been caused by the memory allocation for the pointer to pointer board. On top of this, another struct InProp was created to implement the concept of nested structures.

With the code made to scatter the mines randomly around the board, sweep() was then added to calculate the number of mines adjacent to the cell and assign that value to the cell (Version 3). Then the code asks for player input to select a cell by choosing a row and column (Version 4) The row and column is then checked if the corresponding cell has already been revealed.

Moving forward with building the code, I included a game over function so that I could have the program ask for the player's move continuously until a mine was revealed or until the player clears all the cells not containing a mine (Version 5). At this time, there was a flaw in the logic where if the cell was correctly flagged (flagged on a cell containing a mine), the game would be over. There was also an issue in checking if the player had won. In that time, this logic issue was not identified and troubleshooted.

This issue was still not addressed but the code was polished so that when the game is over, every cell is revealed and displayed (Version 6). On top of that, the display board was fixed to include column headers. A feature to clear surrounding cells if there were no mines nearby was implemented. This was accomplished via a recursive function where the clear function was called inside of its own definition.

From there, the clear function was fixed: Previously, when the cells were cleared, the only cells revealed were the empty spaces (with a value of 0) (Version 7). This fix now includes showing the hints or numbers that surround the selected cell. Then the feature was added to let the player know how many more flags to place (Version 8). If the game was won, then the winning board was saved to a text file, along with the date and time of the win.

After that, the code was extended to allow the player to play multiple times in the same run (Version 9). In this way, reRun() was integrated into the code. Not only that, the code was updated to recognize a win. The player does not have to flag all the places where the mines are to win; instead, they just have to clear all the cells where the mines are not.

Because it did not make sense to keep track of the winning boards, only to have it be updated and replaced by the next winning board; the code was modified with ios::app so that the next winning board would be added below the previously saved board (Version 10). Furthermore, to apply the concept from 11.8 Function Return for Structured Data, the data function of type Cell (which is a struct) [re: Cell data()] is used to initialize each variable/characteristic of each space on the board. This function essentially returns an initialized struct from a function.

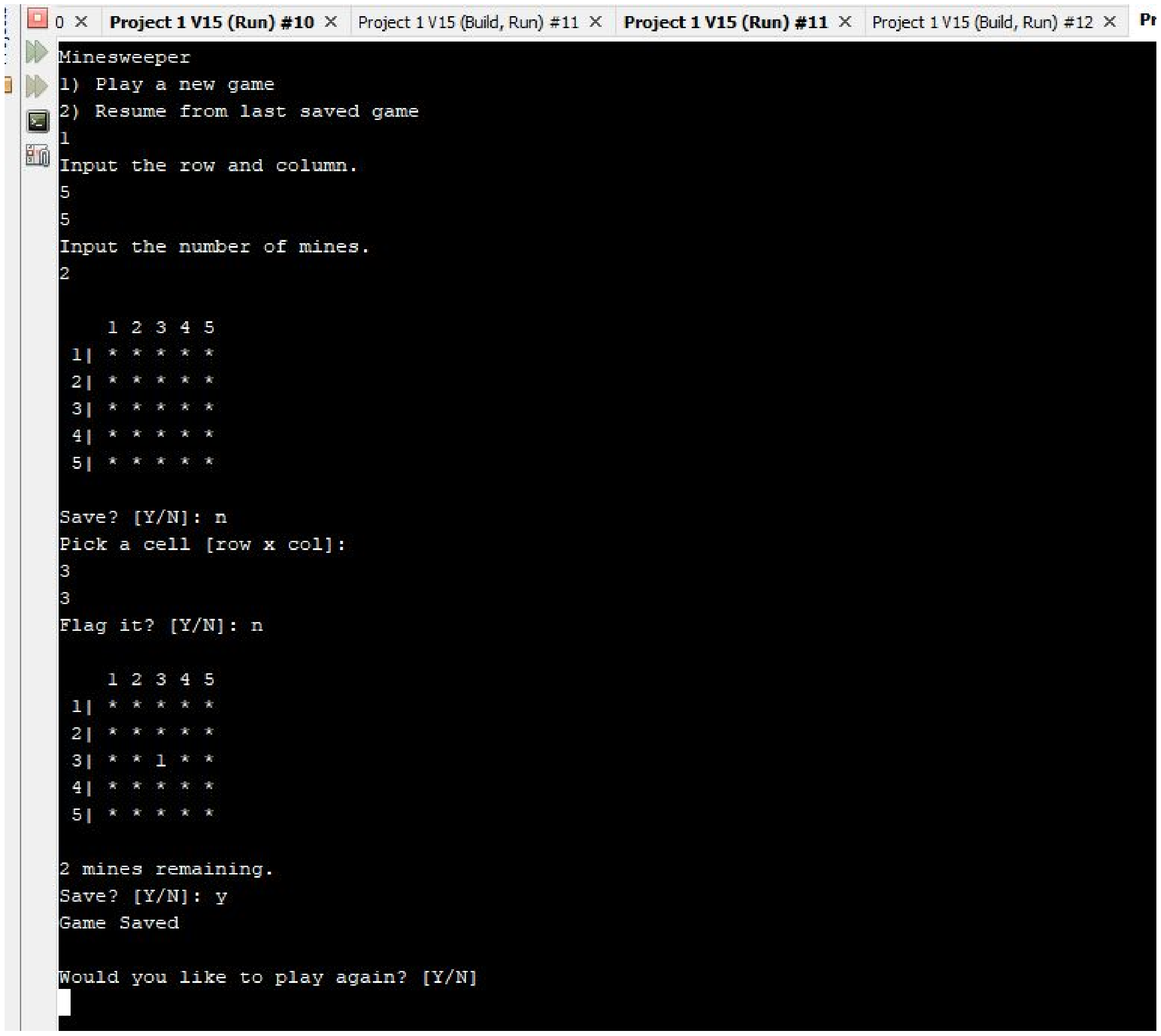
To incorporate binary files into the project, a save mode feature was integrated such that the board could be saved and written to a binary file (Version 11). Then either within the same program run or at a different time, the last saved board could be read from the binary file to be played later on. This particular aspect of the project was difficult to apply because the struct to be written and read was a pointer-to-pointer, a concept that needed a little more research to incorporate into the vision of the project. Not to mention, the board to be saved at any given time is variable so I essentially cannot anticipate how much to read from the file. I used math and divided the entire length of the file by the size of the struct to know how often to read the struct elements. I had to manually add the size of each struct element rather than using sizeof(Cell) because the total of the former was 17 whereas sizeof(Cell) was 20. Part of the challenge was also figuring out if the issue was in writing the file or in reading the file and so my approach was to mirror the code between the read and write. Along the way, a struct element of char type display was added to keep track of what the player can see, and display board and board file functions were modified to account for this change. On top of that, nrow and ncol were also added in the struct to keep track of the cell's location and later identify the number of rows and cols of the saved board.

After saving this version, the next version was used to remove unnecessary comments, or code that was commented out to prevent it from executing during trial and error (Version 12). The main function was then updated to incorporate the save mode. Moving the code around main, had actually made a previously working code, non-working. Because the display board function was called inside the read file function when it was working but would not work in main, troubleshooting lead me to conclude that the read board was not passed back successfully through the function, more specifically that it was not successfully assigned to the pointer to be read by the display board function. That said, I ended up splitting up the read file function such that one function returns the number of rows and columns to dynamically allocate the board array, and the other assigns the read data to the board array.

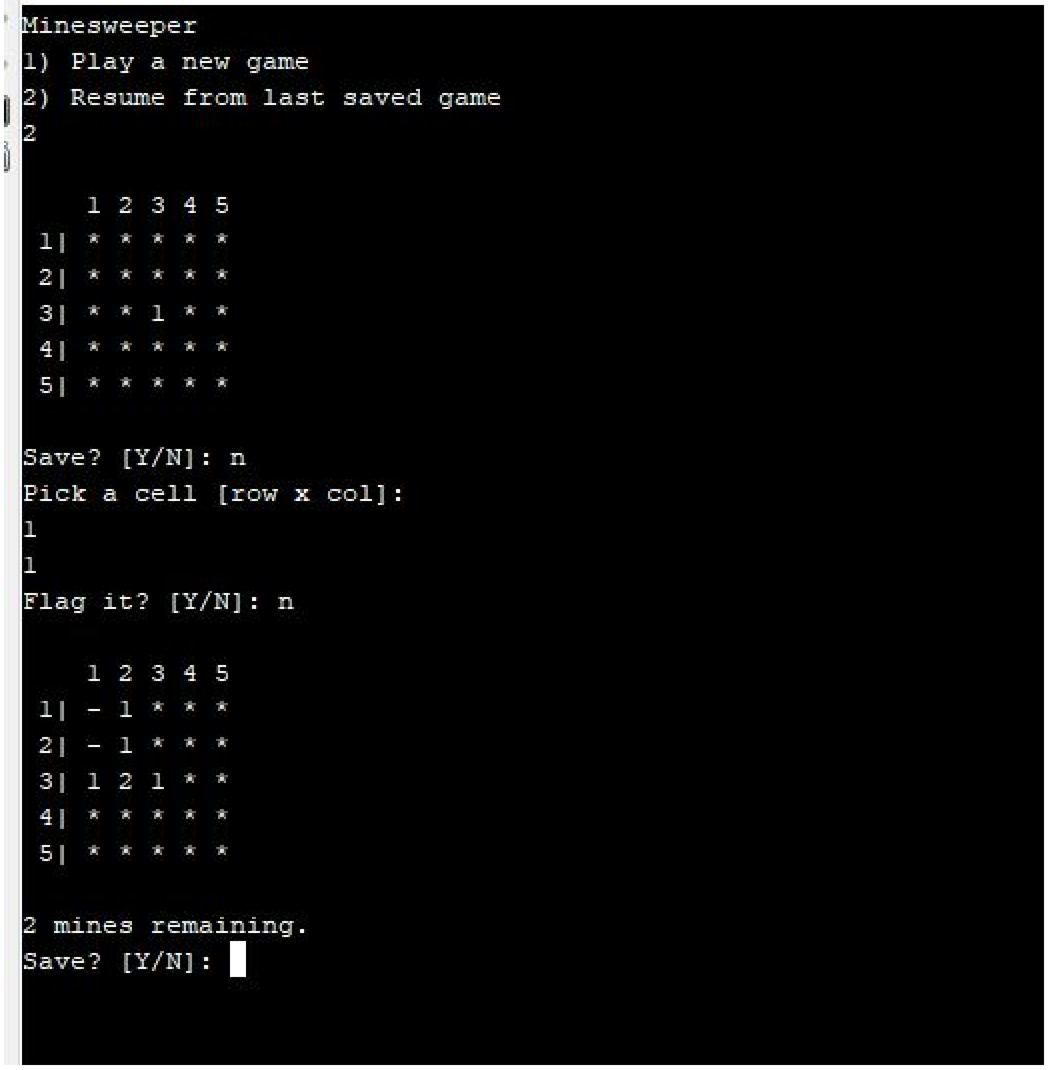
After fixing the save mode, the mine function was then created to count the number of mines in the saved board (Version 13). That way, the program can still tell the player how many more flags to place. Next, input validation was incorporated with cin.fail() and strcpy() was used to integrate the concept using C-Strings (Version 14).

At last, the stats tracking was implemented through reading and writing files (Version 15). This took a while to debug as there were many runtime errors where the whole program runs as though successful but the console determined that the runs have failed. I figured that because the whole program was executed, the problem was located at the end of the code. Further investigation revealed a segmentation fault so I had to play around with the pointers and make sure there weren't any memory leaks or undefined behaviors.

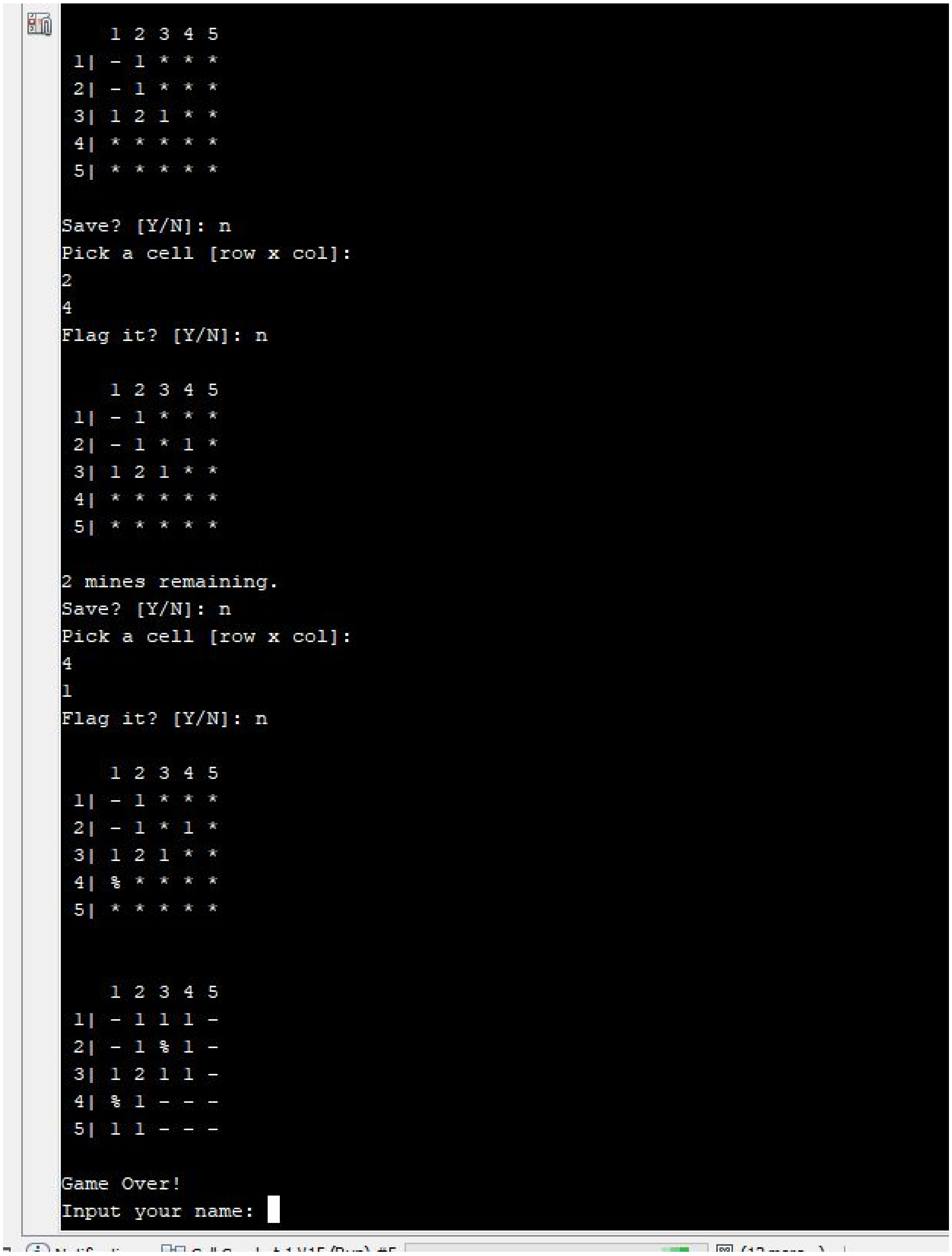
Sample Input/Output


0 ×
Project 1 V15 (Run) #10 ×
Project 1 V15 (Build, Run) #11 ×
Project 1 V15 (Run) #11 ×
Project 1 V15 (Build, Run) #12 ×
Pr

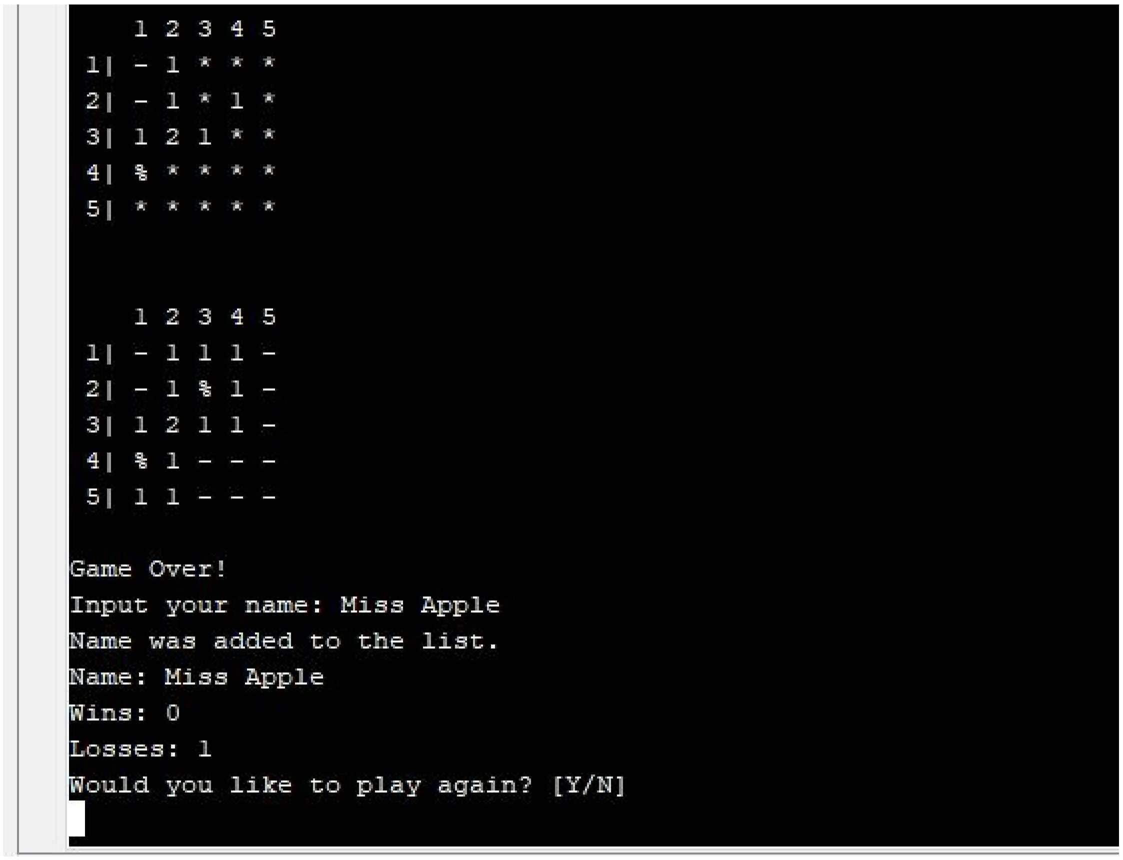
```
Minesweeper
1) Play a new game
2) Resume from last saved game
1
Input the row and column.
5
5
Input the number of mines.
2

    1 2 3 4 5
 1| * * * * *
 2| * * * * *
 3| * * * * *
 4| * * * * *
 5| * * * * *

Save? [Y/N]: n
Pick a cell [row x col]:
3
3
Flag it? [Y/N]: n

    1 2 3 4 5
 1| * * * * *
 2| * * * * *
 3| * * 1 * *
 4| * * * * *
 5| * * * * *

2 mines remaining.
Save? [Y/N]: y
Game Saved

Would you like to play again? [Y/N]
```



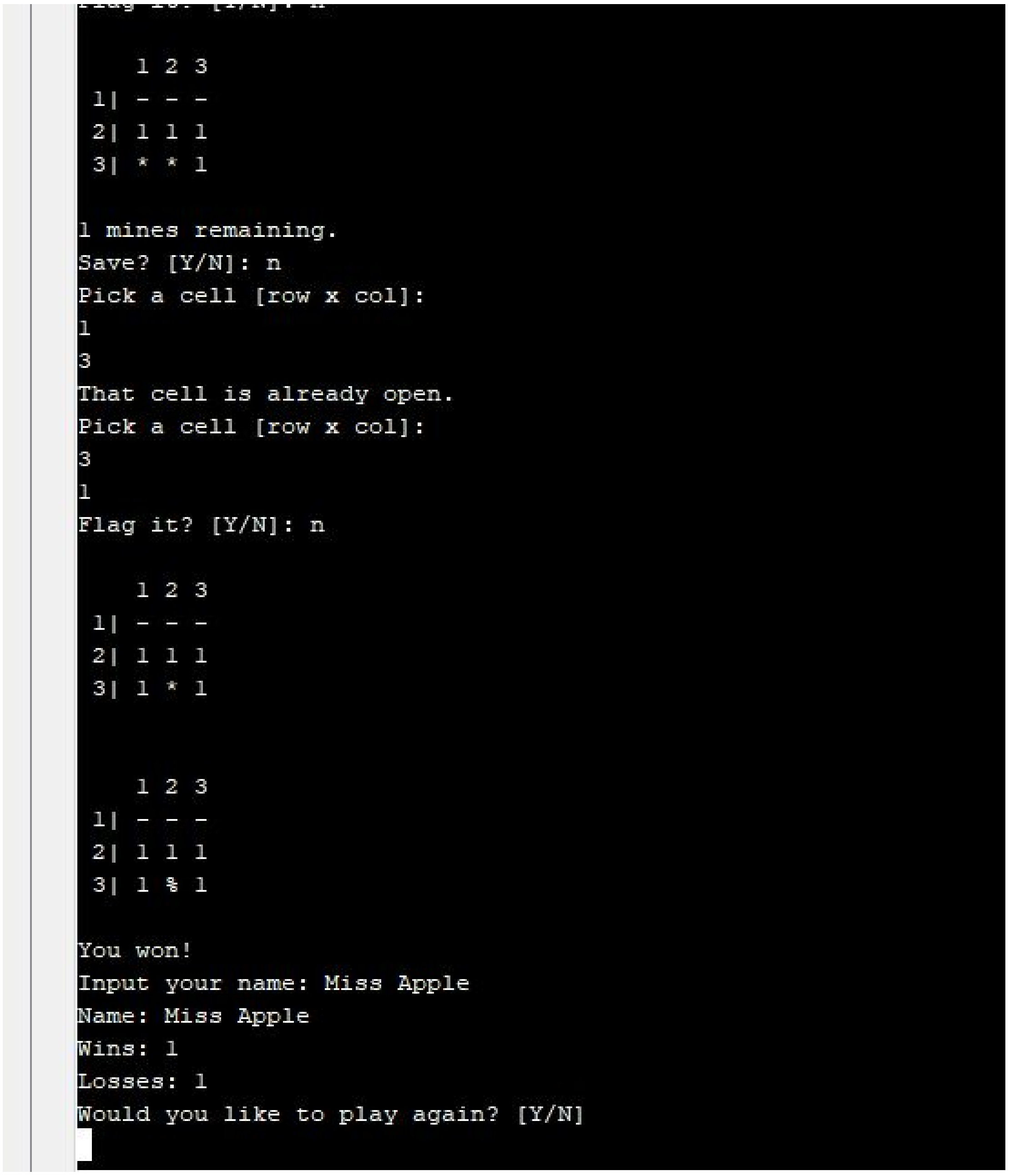
```
Minesweeper
1) Play a new game
2) Resume from last saved game
2

    1 2 3 4 5
 1| * * * * *
 2| * * * * *
 3| * * 1 * *
 4| * * * * *
 5| * * * * *

Save? [Y/N]: n
Pick a cell [row x col]:
1
1
Flag it? [Y/N]: n

    1 2 3 4 5
 1| - 1 * * *
 2| - 1 * * *
 3| 1 2 1 * *
 4| * * * * *
 5| * * * * *

2 mines remaining.
Save? [Y/N]:
```



```
    1 2 3 4 5
 1| - 1 * * *
 2| - 1 * * *
 3| 1 2 1 * *
 4| * * * * *
 5| * * * * *

Save? [Y/N]: n
Pick a cell [row x col]:
2
4
Flag it? [Y/N]: n

    1 2 3 4 5
 1| - 1 * * *
 2| - 1 * 1 *
 3| 1 2 1 * *
 4| * * * * *
 5| * * * * *

2 mines remaining.
Save? [Y/N]: n
Pick a cell [row x col]:
4
1
Flag it? [Y/N]: n

    1 2 3 4 5
 1| - 1 * * *
 2| - 1 * 1 *
 3| 1 2 1 * *
 4| % * * * *
 5| * * * * *



    1 2 3 4 5
 1| - 1 1 1 -
 2| - 1 % 1 -
 3| 1 2 1 1 -
 4| % 1 - - -
 5| 1 1 - - -

Game Over!
Input your name: 
```

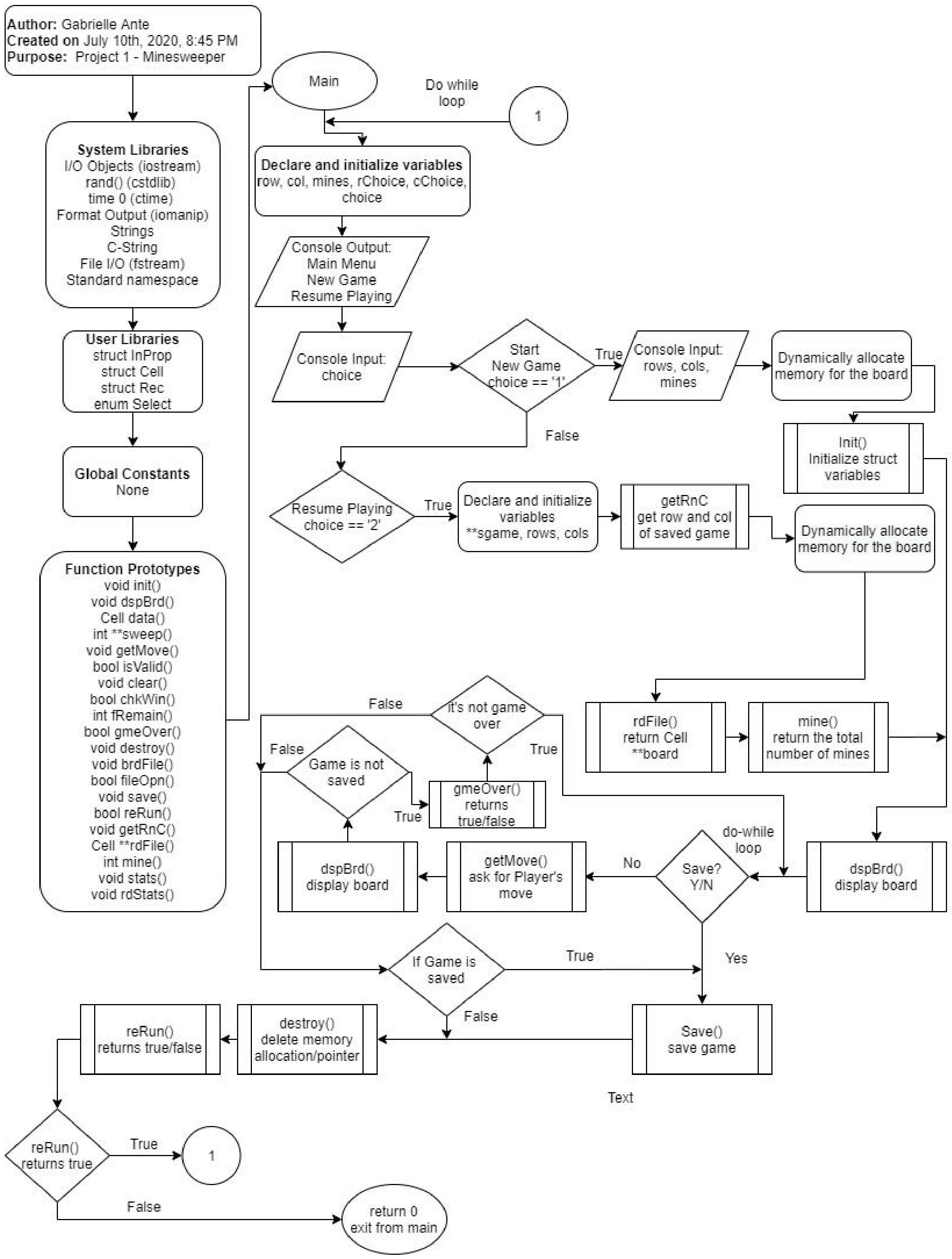


```
    1 2 3 4 5
 1| - 1 * * *
 2| - 1 * 1 *
 3| 1 2 1 * *
 4| % * * * *
 5| * * * * *



    1 2 3 4 5
 1| - 1 1 1 -
 2| - 1 % 1 -
 3| 1 2 1 1 -
 4| % 1 - - -
 5| 1 1 - - -

Game Over!
Input your name: Miss Apple
Name was added to the list.
Name: Miss Apple
Wins: 0
Losses: 1
Would you like to play again? [Y/N]
```



```
Flag it? [Y/N]: n

    1 2 3
 1| - - -
 2| 1 1 1
 3| * * 1

1 mines remaining.
Save? [Y/N]: n
Pick a cell [row x col]:
1
3
That cell is already open.
Pick a cell [row x col]:
3
1
Flag it? [Y/N]: n

    1 2 3
 1| - - -
 2| 1 1 1
 3| 1 * 1


    1 2 3
 1| - - -
 2| 1 1 1
 3| 1 % 1

You won!
Input your name: Miss Apple
Name: Miss Apple
Wins: 1
Losses: 1
Would you like to play again? [Y/N]
```

Flowcharts/Pseudocode/UML


Author: Gabrielle Ante
Created on July 10th, 2020, 8:45 PM
Purpose: Project 1 - Minesweeper
System Libraries
I/O Objects (iostream)
rand() (cstdlib)
time 0 (ctime)
Format Output (iomanip)
Strings
C-String
File I/O (fstream)
Standard namespace
User Libraries
struct InProp
struct Cell
struct Rec
enum Select
Global Constants
None
Function Prototypes
void init()
void dspBrd()
Cell data()
int **sweep()
void getMove()
bool isValid()
void clear()
bool chkWin()
int fRemain()
bool gmeOver()
void destroy()
void brdFile()
bool fileOpn()
void save()
bool reRun()
void getRnC()
Cell **rdFile()
int mine()
void stats()
void rdStats()
Main
Do while loop
1
Declare and initialize variables
row, col, mines, rChoice, cChoice, choice
Console Output: Main Menu New Game Resume Playing
Console Input: choice
Start New Game choice == '1'
True
Console Input: rows, cols, mines
Dynamically allocate memory for the board
Init() Initialize struct variables
False
Resume Playing choice == '2'
True
Declare and initialize variables **sgame, rows, cols
getRnC get row and col of saved game
Dynamically allocate memory for the board
rdFile() return Cell **board
mine() return the total number of mines
it's not game over
False
True
Game is not saved
False
True
gmeOver() returns true/false
dspBrd() display board
getMove() ask for Player's move
No
Save? Y/N
do-while loop
dspBrd() display board
Yes
If Game is saved
True
False
Save() save game
Text
destroy() delete memory allocation/pointer
reRun() returns true/false
reRun() returns true
True
1
False
return 0 exit from main

Pseudocode

- Main Menu (Start New Game/Resume Playing)
    - Start New Game
        - Input the number of rows, columns and mines
            - Validate input for the number of mines
        - Initialize each cell
            - Initialize each element of the struct
                - Display, which the player can see, starts as * (hidden)
                - Symbol, which there are no mines placed yet starts as (-) (empty)
                - Hidden starts as true because all cells start as hidden
                - Mine starts as false as there are no mines yet
                - Flagged starts as false because player has not chosen to flag a cell yet
                - Value starts as 0 because there are no mines surrounding the adjacent cells yet
            - Initialize nrow and ncol, so that each cell contains its own location in the struct
            - Randomly place where the mines are
                - Validate if space already has a mine
            - Determine the values of the cells in the board
                - Count the # of mines around the cell for each cell
            - Assign values to board
        - Display Board
        - Ask to Save
            - Validate user input
            - If no, Ask for Player Move
                - Ask Player to select a row and col
                    - Validate if space has already been revealed
                - If flagged, unflag cell
                - If not flagged, Ask Player to Flag
                    - Yes flags the cell
                    - No reveals the cell
                        - If revealed cell has no mines around it, the cell clears the adjacent cells
                        - Repeat for adjacent cells until there is a mine adjacent to the cell (if the selected cell has a value)
                - Display Board
                - Go back to Ask to Save if player had chosen No to save and it's not game over
                    - Gave Over if player has revealed a mine (lose)
                        - Update Stats

                - Game Over if player has cleared all cells not containing a mine (win)
                    - Update Stats
                - Not game over otherwise
            - If yes, out of Ask to Save loop and board is saved
            - Pointer is deleted
    - Resume Playing
        - Read file to get the rows and columns
        - Read File to retrieve the saved board
        - Count the number of mines from the saved board
        - Display Board
        - Ask to Save
            - Validate user input
            - If no, Ask for Player Move
                - Ask Player to select a row and col
                    - Validate if space has already been revealed
                - If flagged, unflag cell
                - If not flagged, Ask Player to Flag
                    - Yes flags the cell
                    - No reveals the cell
                        - If revealed cell has no mines around it, the cell clears the adjacent cells
                        - Repeat for adjacent cells until there is a mine adjacent to the cell (if the selected cell has a value)
                - Display Board
                - Go back to Ask to Save if player had chosen No to save and it's not game over
                    - Game Over if player has revealed a mine (lose)
                        - Update Stats
                    - Game Over if player has cleared all cells not containing a mine (win)
                        - Update stats
                    - Not game over otherwise
                - If yes, out of Ask to Save loop and board is saved
                - Pointer is deleted
- Ask to rerun program
    - Validate user input
    - If yes, go back to Main Menu
    - If no, exit the program

Variables

Data Type	Variable Name	Description	Location
Struct Cell	board**	A pointer to pointer, essentially a 2D array that stores the game board cells and its properties	Init(); dspBrd(); getMove();
Int	row	Stores the number of rows in the board	Init(); dspBrd(); getMove();
Int	col	Stores the number of columns in the board	Init(); dspBrd(); getMove();
int	mines	Stores the number of mines in the board	Main(); Init(); fRemain();
Int	rChoice	Stores the player's row choice for the cell	getMove()
Int	cChoice	Stores the player's column choice for the cell	getMove();
int	num	Stores the location of the statistics for a particular player	stats(); rdStats();

Concepts

Section	Concept	Line	Section	Concept	Line
9.2	Pointer Variables	735	11.9	Pointers	103
9.3	Arrays/Pointers	178-179	11.11	Enumeration	50
9.7	Function Parameters	52	12.2	Formatting	494
9.8	Memory Allocation	103-106	12.3	Function Parameters	64
9.9	Return Parameters	55	12.5	Member Functions	691
10.3	C-Strings	626-627	12.6	Multiple Files	676-679
10.7	Strings	690	12.7	Binary Files	734
11.5	Arrays	134-140	12.8	Records with Structures	700-703

11.6	Nested	41	12.9	Random Access Files	731-748
11.7	Function Arguments	52	12.10	Input/Output Simultaneous	670-729
11.8	Function Return	54			

References

Gaddis, Tony, Judy Walters, and Godfrey Muganda. "Starting Out With C++: Early Objects, Student Value Edition." (2016).

Std::istream::tellg. (n.d.). Retrieved July 16, 2020, from http://www.cplusplus.com/reference/istream/istream/tellg/

Updating records in a binary file with C++. Retrieved July 16, 2020, from https://stackoverflow.com/questions/7100019/updating-records-in-a-binary-file-with-c

Program

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

/*
 * File:   Ante_Gabrielle_Project_1.cpp
 * Author: Gabrielle
 *
 * Created on July 15, 2020, 10:45 AM
 */

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <string>
#include <cstring>
#include <fstream>

using namespace std;

/*
```

```
 * stats tracking included
 * enumerator
 */

struct InProp {
    bool mine; // if the cell has a mine
    int value; // what the cell value is; hint for adjacent cell
};

struct Cell {
    char display; // what is seen
    char symbol; // to be outputted
    bool hidden; // if the player can see cell
    bool flagged; // if the player holds the cell
    int nrow; // which row cell is located
    int ncol; // which col cell is located
    InProp hint; // if the cell has a mine or where it's located
};

struct Rec{
    string name;
    int wins;
    int losses;
};

enum Select {YES, NO};

void init(Cell **board, int mines, int row, int col);
void dspBrd(Cell **board, int row, int col);
Cell data();
int **sweep(Cell **board, int row, int col);
void getMove(Cell ** board, int row, int col, int &rChoice, int &cChoice);
bool isValid(Cell **board, int row, int col, int rMax, int cMax);
void clear(Cell **board, int row, int col, int rMax, int cMax);
bool chkWin(Cell **board, int rMax, int cMax);
int fRemain(Cell **board, int row, int col, int mines);
bool gmeOver(Cell **board, int row, int col, int rMax, int cMax, int mines);
void destroy(Cell **ary, int row);
void brdFile(Cell **board, int row, int col);
bool fileOpn(ofstream &fVar, const char fName[]);
void save(Cell **board, int row, int col);
bool reRun();
void getRnC(int &rows, int &cols);
```

```
Cell **rdFile(int rows, int cols);
int mine(Cell **board, int row, int col);
void stats(int &num, int win, int loss);
void rdStats( int num);

int main(int argc, char** argv) {

   do {
      int row = 0, col = 0, mines = 0;
      int rChoice = 0, cChoice = 0;
      char choice = ' ';

      cout << "Minesweeper" << endl;
      cout << "1) Play a new game" << endl;
      cout << "2) Resume from last saved game" << endl;
      cin >> choice;

      switch (choice) {
         case '1':
         {
            // User input
            cout << "Input the row and column." << endl;
            cin >> row >> col;
            while (cin.fail())
            {
               cin.clear();
               cin.ignore();
               cout << "Input the row and column." << endl;
               cin >> row >> col;
            }
            do {
               cout << "Input the number of mines." << endl;
               cin >> mines;
            } while (mines > row * col);

            Cell**space = new Cell*[row];
            for (int i = 0; i < row; i++) {
               *(space + i) = new Cell [col];
            }

            //Initialize struct variables
            init(space, mines, row, col);
```

```
        //Display board
        dspBrd(space, row, col);
        do {
            do {
                cout << "Save? [Y/N]: ";
                cin >> choice;
            } while (toupper(choice) != 'Y' && toupper(choice) != 'N');

            if (toupper(choice) == 'N') {
                getMove(space, row, col, rChoice, cChoice);
                dspBrd(space, row, col);
            }
        } while (toupper(choice) == 'N'
                && !gmeOver(space, rChoice - 1, cChoice - 1, row, col, mines));

        if (toupper(choice) == 'Y') {
            save(space, row, col);
        }
        destroy(space, row);
        break;
    }
    case '2':
    {
        Cell**sgame;
        int rows = 0, cols = 0;
        getRnC(rows, cols);
        sgame = new Cell*[rows];
        for (int i = 0; i < rows; i++) {
            *(sgame + i) = new Cell [cols];
        }

        sgame = rdFile(rows, cols);
        mines = mine(sgame, rows, cols);
        //Display board
        dspBrd(sgame, rows, cols);
        do {
            do {
                cout << "Save? [Y/N]: ";
                cin >> choice;
            } while (toupper(choice) != 'Y' && toupper(choice) != 'N');

            if (toupper(choice) == 'N') {
                getMove(sgame, rows, cols, rChoice, cChoice);
```

```
                    dspBrd(sgame, rows, cols);
                }
            } while (toupper(choice) == 'N'
                    && !gmeOver(sgame, rChoice - 1, cChoice - 1, rows, cols, mines));

            if (toupper(choice) == 'Y') {
                save(sgame, rows, cols);
            }
            destroy(sgame, rows);
            break;
        }
    }
} while (reRun());

return 0;

}

void init(Cell **board, int mines, int row, int col) {
    srand(time(0));
    int rnum = 0, cnum = 0;
    int min = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            *(*(board + i) + j)  = data();
            board[i][j].nrow = i + 1;
            board[i][j].ncol = j + 1;
        }
    }
    for (int i = 0; i < mines; i++) {
        do {
            rnum = rand() % row;
            cnum = rand() % col;
        } while (board[rnum][cnum].hint.mine == true);
        board[rnum][cnum].hint.mine = true;
        board[rnum][cnum].symbol = '%';
    }

    int **val = sweep(board, row, col);

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            board[i][j].hint.value = val[i][j];
        }
    }
}

Cell data() {
    Cell c;

    c.display = '*';
    c.symbol = '-';
    c.hidden = true;
    c.hint.mine = false;
    c.flagged = false;
    c.hint.value = 0;

    return c;
}

void dspBrd(Cell **board, int row, int col) {
    cout << endl;
    cout << setw(4) << " ";
    for (int i = 0; i < col; i++) {
        cout << i + 1 << " ";
    }
    cout << endl;
    for (int i = 0; i < row; i++) {
        cout << setw(2) << i + 1 << "| ";
        for (int j = 0; j < col; j++) {
            if (board[i][j].flagged == true) {
                board[i][j].display = 'P';
            } else if (board[i][j].hidden == true) {
                board[i][j].display = '*';
            } else if (board[i][j].hint.value == 0) {
                board[i][j].display = board[i][j].symbol;
            } else {
                board[i][j].display = board[i][j].hint.value + 48;
            }
            cout << board[i][j].display << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

```
int **sweep(Cell **board, int row, int col) {
    int **mines = new int*[row];
    for (int i = 0; i < row; i++) {
        *(mines + i) = new int [col];
    };

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            mines[i][j] = 0;
            if (board[i][j].hint.mine == false) {
                if (i - 1 >= 0 && j - 1 >= 0) // top left
                {
                    if (board[i - 1][j - 1].hint.mine == true) {
                        mines[i][j]++;
                    }
                }
                if (i - 1 >= 0 && j >= 0) // top
                {
                    if (board[i - 1][j].hint.mine == true) {
                        mines[i][j]++;
                    }
                }
                if (i - 1 >= 0 && j + 1 < col) // top right
                {
                    if (board[i - 1][j + 1].hint.mine == true) {
                        mines[i][j]++;
                    }
                }
                if (i >= 0 && j + 1 < col) // right
                {
                    if (board[i][j + 1].hint.mine == true) {
                        mines[i][j]++;
                    }
                }
                if (i + 1 < row && j + 1 < col) // bottom right
                {
                    if (board[i + 1][j + 1].hint.mine == true) {
                        mines[i][j]++;
                    }
                }
                if (i + 1 < row && j >= 0) // bottom
                {
                    if (board[i + 1][j].hint.mine == true) {
```

```
                    mines[i][j]++;
                }
            }
            if (i + 1 < row && j - 1 >= 0) // bottom left
            {
                if (board[i + 1][j - 1].hint.mine == true) {
                    mines[i][j]++;
                }
            }
            if (i >= 0 && j - 1 >= 0) // left
            {
                if (board[i][j - 1].hint.mine == true) {
                    mines[i][j]++;
                }
            }
        }
    }
}

return mines;
}

void getMove(Cell ** board, int row, int col, int &rChoice, int &cChoice) {
    char choice = ' ';
    do {
        cout << "Pick a cell [row x col]: " << endl;
        cin >> rChoice >> cChoice;
        while (cin.fail())
            {
                cin.clear();
                cin.ignore();
                cout << "Pick a cell [row x col]: " << endl;
                cin >> rChoice >> cChoice;
            }
    } while (!isValid(board, rChoice - 1, cChoice - 1, row, col));

    if (board[rChoice - 1][cChoice - 1].flagged == true) {
        board[rChoice - 1][cChoice - 1].flagged = false;
    }
    else {
        do {
            cout << "Flag it? [Y/N]: ";
            cin >> choice;
        } while (toupper(choice) != 'Y' && toupper(choice) != 'N');
        if (toupper(choice) == 'Y') {
            board[rChoice - 1][cChoice - 1].flagged = true;
        } else {
            board[rChoice - 1][cChoice - 1].hidden = false;
            if (board[rChoice - 1][cChoice - 1].symbol == '-' && board[rChoice - 1][cChoice -
1].hint.value == 0) {
                clear(board, rChoice - 1, cChoice - 1, row, col);
            }
        }
    }
}

bool isValid(Cell **board, int row, int col, int rMax, int cMax) {
    if (row < 0 || row >= rMax) {
        return false;
    } else if (col < 0 || col >= cMax) {
        return false;
    }
    else if (board[row][col].hidden == false) {
        cout << "That cell is already open." << endl;
        return false;
    } else {
        return true;
    }
}

void clear(Cell **board, int row, int col, int rMax, int cMax) {

    if (row - 1 >= 0 && col - 1 >= 0 && board[row - 1][col - 1].hidden == true) // top left
    {
        board[row - 1][col - 1].hidden = false;
        if (board[row - 1][col - 1].symbol == '-' && board[row - 1][col - 1].hint.value == 0) {
            clear(board, row - 1, col - 1, rMax, cMax);
        }
    }
    if (row - 1 >= 0 && col >= 0 && board[row - 1][col].hidden == true) // top
    {
        board[row - 1][col].hidden = false;
        if (board[row - 1][col].symbol == '-' && board[row - 1][col].hint.value == 0) {
            clear(board, row - 1, col, rMax, cMax);
        }
    }
    if (row - 1 >= 0 && col + 1 < cMax && board[row - 1][col + 1].hidden == true) // top right
    {
        board[row - 1][col + 1].hidden = false;
        if (board[row - 1][col + 1].symbol == '-' && board[row - 1][col + 1].hint.value == 0) {
            clear(board, row - 1, col + 1, rMax, cMax);
        }
    }
    if (row >= 0 && col + 1 < cMax && board[row][col + 1].hidden == true) // right
    {
        board[row][col + 1].hidden = false;
        if (board[row][col + 1].symbol == '-' && board[row][col + 1].hint.value == 0) {
            clear(board, row, col + 1, rMax, cMax);
        }
    }
    if (row + 1 < rMax && col + 1 < cMax && board[row + 1][col + 1].hidden == true) // bottom right
    {
        board[row + 1][col + 1].hidden = false;
        if (board[row + 1][col + 1].symbol == '-' && board[row + 1][col + 1].hint.value == 0) {
            clear(board, row + 1, col + 1, rMax, cMax);
        }
    }
    if (row + 1 < rMax && col >= 0 && board[row + 1][col].hidden == true) // bottom
    {
        board[row + 1][col].hidden = false;
        if (board[row + 1][col].symbol == '-' && board[row + 1][col].hint.value == 0) {
            clear(board, row + 1, col, rMax, cMax);
        }
    }
    if (row + 1 < rMax && col - 1 >= 0 && board[row + 1][col - 1].hidden == true) // bottom left
    {
        board[row + 1][col - 1].hidden = false;
        if (board[row + 1][col - 1].symbol == '-' && board[row + 1][col - 1].hint.value == 0) {
            clear(board, row + 1, col - 1, rMax, cMax);
        }
    }
    if (row >= 0 && col - 1 >= 0 && board[row][col - 1].hidden == true) // left
    {
        board[row][col - 1].hidden = false;
        if (board[row][col - 1].symbol == '-' && board[row][col - 1].hint.value == 0) {
            clear(board, row, col - 1, rMax, cMax);
        }
    }
}
```

```
bool chkWin(Cell **board, int rMax, int cMax) {
    int count = 0;
    for (int i = 0; i < rMax; i++) {
        for (int j = 0; j < cMax; j++) {
            if (board[i][j].hint.mine == true) {
                count++;
            } else if (board[i][j].hint.mine == false && board[i][j].hidden == false) {
                count++;
            }
        }
    }
    if (count == rMax * cMax) {
        return true;
    } else {
        return false;
    }
}

int fRemain(Cell **board, int row, int col, int mines) {
    int flags = 0, diff = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (board[i][j].flagged == true) {
                flags++;
            }
        }
    }
    diff = mines - flags;
    return diff;
}

bool gmeOver(Cell **board, int row, int col, int rMax, int cMax, int mines) {
    int diff = 0;
    int num;

    // Loss
    if (board[row][col].hint.mine == true && board[row][col].hidden == false) {
        for (int i = 0; i < rMax; i++) {
            for (int j = 0; j < cMax; j++) {
                board[i][j].hidden = false;
            }
        }
        dspBrd(board, rMax, cMax);
        cout << "Game Over!" << endl;
        stats(num,0,1);
        rdStats(num);

        return true;
    }
    //Win
    else if (chkWin(board, rMax, cMax)) {
        for (int i = 0; i < rMax; i++) {
            for (int j = 0; j < cMax; j++) {
                board[i][j].hidden = false;
            }
        }
        dspBrd(board, rMax, cMax);
        cout << "You won!" << endl;
        brdFile(board, rMax, cMax);
        stats(num,1,0);
        rdStats(num);
        return true;
    }
    // Not game over
    else {
        diff = fRemain(board, rMax, cMax, mines);
        cout << diff << " mines remaining." << endl;
        return false;
    }
}

void destroy(Cell **ary, int row) {
    for (int i = 0; i < row; i++) {
        delete[] *(ary + i);
    }
    delete[] ary;
}

void brdFile(Cell **board, int row, int col) {
    ofstream oData;
    if (fileOpn(oData, "WinningBoards.txt")) {
        oData << setw(4) << " ";
        for (int i = 0; i < col; i++) {
            oData << i + 1 << " ";
        }
        oData << endl;
        for (int i = 0; i < row; i++) {
            oData << setw(2) << i + 1 << "| ";
            for (int j = 0; j < col; j++) {
                if (board[i][j].flagged == true) {
                    board[i][j].display = 'P';
                } else if (board[i][j].hidden == true) {
                    board[i][j].display = '*';
                } else if (board[i][j].hint.value == 0) {
                    board[i][j].display = board[i][j].symbol;
                } else {
                    board[i][j].display = board[i][j].hint.value + 48;
                }
                oData << board[i][j].display << " ";
            }
            oData << endl;
        }
        // current date/time based on current system
        time_t winTime = time(0);

        // convert now to string form
        char* wTime = ctime(&winTime);

        oData << endl << "Game won on: " << wTime << endl << endl;

        oData.close();
    } else {
        cout << "Error opening file." << endl;
    }
}

bool fileOpn(ofstream &fVar, const char fName[]) {
    fVar.open(fName, ios::app);
    if (fVar.fail()) {
        return false;
    } else {
        return true;
    }
}

void save(Cell **board, int row, int col) {
    ofstream progrss;
    progrss.open("Progress.bin", ios::binary);
    // for each struct/Cell
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            progrss.write(reinterpret_cast<char *> (&board[i][j].display), sizeof (board[i][j].display));
            progrss.write(reinterpret_cast<char *> (&board[i][j].symbol), sizeof (board[i][j].symbol));
            progrss.write(reinterpret_cast<char *> (&board[i][j].hidden), sizeof (board[i][j].hidden));
            progrss.write(reinterpret_cast<char *> (&board[i][j].flagged), sizeof (board[i][j].flagged));
            progrss.write(reinterpret_cast<char *> (&board[i][j].nrow), sizeof (board[i][j].nrow));
            progrss.write(reinterpret_cast<char *> (&board[i][j].ncol), sizeof (board[i][j].ncol));
            progrss.write(reinterpret_cast<char *> (&board[i][j].hint.mine), sizeof
(board[i][j].hint.mine));
            progrss.write(reinterpret_cast<char *> (&board[i][j].hint.value), sizeof
(board[i][j].hint.value));
        }
    }
    cout << "Game Saved" << endl << endl;
    progrss.close();
}

bool reRun() {
    char choice = ' ';
    Select s;
    do {
        cout << "Would you like to play again? [Y/N]" << endl;
        cin >> choice;
        if (toupper(choice) =='Y')
        {
            s = YES;
        }
        else if (toupper(choice) =='N')
        {
            s = NO;
        }
        //s = static_cast<Select>(toupper(choice));
    } while ( s != YES && s != NO);

    if (toupper(choice) == 'Y') {
        return true;
    } else {
        return false;
    }
}
```

```
void getRnC(int &rows, int &cols) {
    ifstream data;
    data.open("Progress.bin", ios::binary);

    //Declare and initialize variables
    Cell *a = new Cell; //Declare the array to return
    rows = 0;
    cols = 0;

    data.seekg(0, data.end);
    long length = data.tellg();
    data.seekg(0, data.beg);

    //Find # of rows and cols
    int size = sizeof (a->display) + sizeof (a->symbol) + sizeof (a->hidden) +
            sizeof (a->flagged) + sizeof (a->nrow) + sizeof (a->ncol) +
            sizeof (a->hint.mine) + sizeof (a->hint.value);

    for (int i = 0; i < length / size; i++) {
        data.read(reinterpret_cast<char *> (&a->display), sizeof (a->display));
        data.read(reinterpret_cast<char *> (&a->symbol), sizeof (a->symbol));
        data.read(reinterpret_cast<char *> (&a->hidden), sizeof (a->hidden));
        data.read(reinterpret_cast<char *> (&a->flagged), sizeof (a->flagged));
        data.read(reinterpret_cast<char *> (&a->nrow), sizeof (a->nrow));
        data.read(reinterpret_cast<char *> (&a->ncol), sizeof (a->ncol));
        data.read(reinterpret_cast<char *> (&a->hint.mine), sizeof (a->hint.mine));
        data.read(reinterpret_cast<char *> (&a->hint.value), sizeof (a->hint.value));
        if (a-> nrow > rows) {
            rows = a-> nrow;
        }
        if (a-> ncol > cols) {
            cols = a->ncol;
        }
    }

    delete a;

    data.close();
}

Cell **rdFile(int rows, int cols) {
    ifstream data;
    const char fName[15] = "Progress.bin";
    char file[15];
    strcpy(file, fName);
    data.open(fName, ios::binary);

    //Declare and initialize variables
    Cell **game;

    //go back to beginning of file
    data.clear();
    data.seekg(0, ios::beg);

    game = new Cell*[rows];
    for (int i = 0; i < rows; i++) {
        *(game + i) = new Cell [cols];
    }

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            data.read(reinterpret_cast<char *> (&game[i][j].display), sizeof (game[i][j].display));
            data.read(reinterpret_cast<char *> (&game[i][j].symbol), sizeof (game[i][j].symbol));
            data.read(reinterpret_cast<char *> (&game[i][j].hidden), sizeof (game[i][j].hidden));
            data.read(reinterpret_cast<char *> (&game[i][j].flagged), sizeof (game[i][j].flagged));
            data.read(reinterpret_cast<char *> (&game[i][j].nrow), sizeof (game[i][j].nrow));
            data.read(reinterpret_cast<char *> (&game[i][j].ncol), sizeof (game[i][j].ncol));
            data.read(reinterpret_cast<char *> (&game[i][j].hint.mine), sizeof (game[i][j].hint.mine));
            data.read(reinterpret_cast<char *> (&game[i][j].hint.value), sizeof (game[i][j].hint.value));
        }
    }

    data.close();
    return game;
}

int mine(Cell **board, int row, int col) {
    int nmines = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            if (board[i][j].hint.mine == true) {
                nmines++;
            }
        }
    }
    return nmines;
}

void stats( int &num, int win, int loss)
{
    ifstream s, list;
    ofstream a, out;
    string name = " ";
    string x = " ";
    s.open("Stats.bin", ios::in|ios::out|ios::binary);
    a.open("Stats.bin", ios::in|ios::out|ios::binary);
    list.open("NameList.txt");
    out.open("NameList.txt", ios::app);
    string temp = " ";
    num = 1;
    long cursor = 0L;
    Rec *r = new Rec;
    Rec *n = new Rec;

    cout << "Input your name: ";
    cin.ignore();
    getline(cin,name);

    x.assign(name.c_str(), 0,name.length());
    while (getline(list, temp) && temp != x)
    {
        num++;
    }

    if (temp == name)
    {
        cursor = num*sizeof(Rec);

        s.seekg(cursor, ios::beg);
        s.read(reinterpret_cast<char *> (&n->name),sizeof (n->name));
        s.read(reinterpret_cast<char *> (&n->wins),sizeof (n->wins));
        s.read(reinterpret_cast<char *> (&n->losses),sizeof (n->losses));

    }
    else
    {
        out << name << endl;
        n->name = name;
        n->wins =0;
        n->losses = 0;
        cout << "Name was added to the list. " << endl;
    }
    r->name = name;
    r->wins = n->wins + win;
    r->losses = n->losses + loss;

    a.clear();
    cursor = num*sizeof(Rec);
    a.seekp(cursor, ios::beg);
    a.write(reinterpret_cast<char *> (&r->name),sizeof (r->name));
    a.write(reinterpret_cast<char *> (&r->wins),sizeof (r->wins));
    a.write(reinterpret_cast<char *> (&r->losses),sizeof (r->losses));

    s.close();
    a.close();
    list.close();
    out.close();
}

void rdStats(int num)
{
    fstream s;
    s.open("Stats.bin", ios::in|ios::binary);

    long cursor = 0L;
    Rec *r = new Rec;
    cursor = num*sizeof(Rec);
    s.seekg(cursor, ios::beg);
    s.read(reinterpret_cast<char *> (&r->name),sizeof (r->name));
    s.read(reinterpret_cast<char *> (&r->wins),sizeof (r->wins));
    s.read(reinterpret_cast<char *> (&r->losses),sizeof (r->losses));
    cout << "Name: " << r->name << endl;
    cout << "Wins: " << r->wins << endl;
    cout << "Losses: " << r->losses << endl;

    s.close();
}
```