Project 1

Ruiz, Juan – 48130

Introduction

For this project I programmed up the game Minesweeper. Minesweeper is played on a rows x columns grid. The game randomly places mines on the grid and their location is kept hidden. The goal of the game is to try to clear the empty spaces, i.e. the spaces without mines, leaving only the undetonated mines when the game is finished. A mine is detonated when a player selects a space with a mine underneath. When a player selects a space with mines adjacent, that space reveals how many mines are adjacent. If the space has no mines adjacent, it is clear then every adjacent and clear space is also revealed.

This project is important because I relied on a structure in order to implement this game. The reason that this is important is because structures are a precursor to classes, as part of object-oriented programming. In learning the syntax for manipulating structures, I have learned most of the syntax that will be required for dealing with objects in the future.

Summary

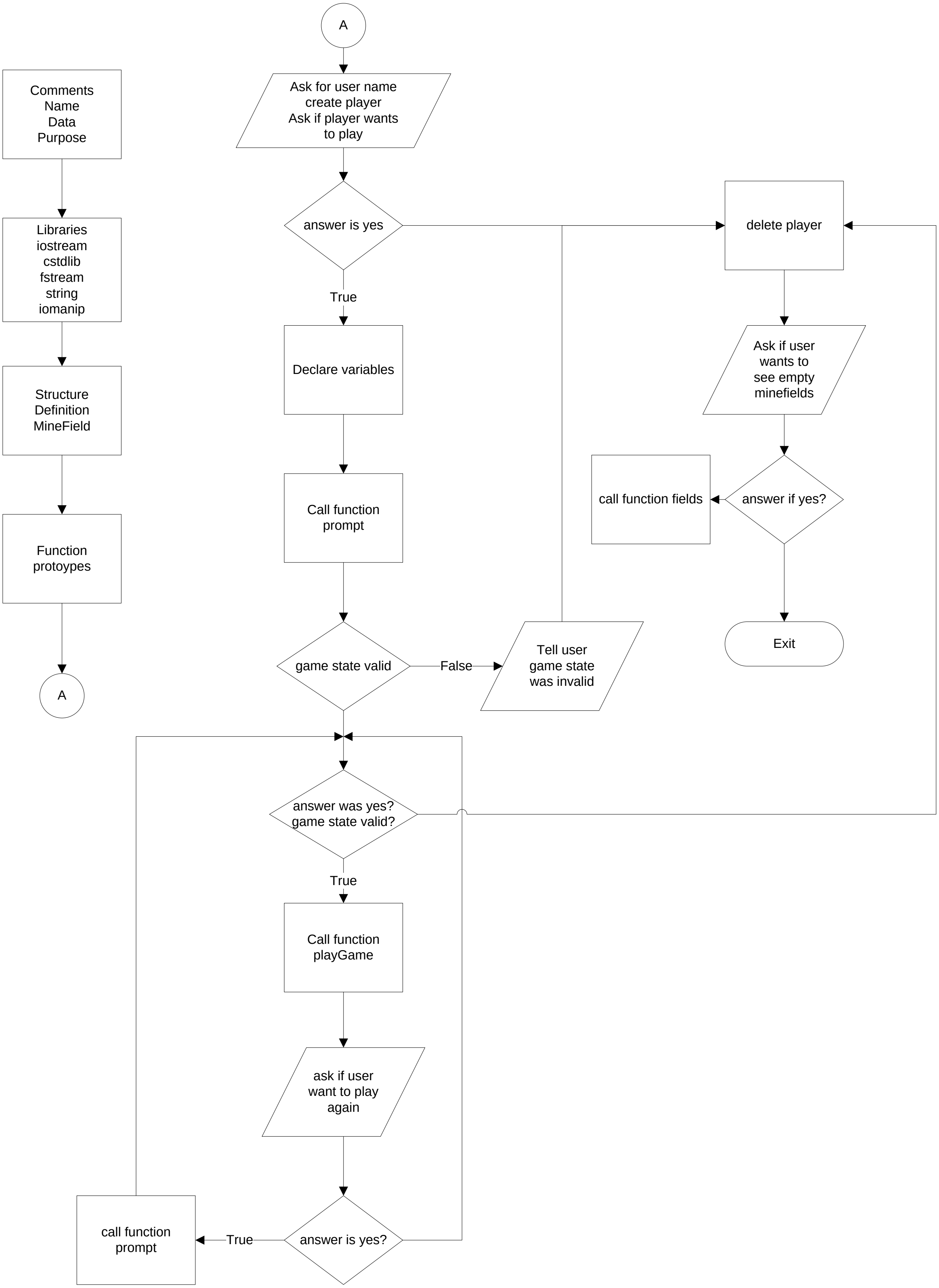
The project comes in at about 575 lines of code.

The major variables are part the MineField structure. The structure consisted of five variables that represented the game variables, as well as two enumerations. One represented various status conditions while playing the game. The other was to determine what difficulty the player was selected, with higher difficulty placing more mines, thus making the game more difficult.

It took about two weeks to finish programming this game. The most challenging part of writing this was changing the internal structure of the MineField structure. These changes cascaded out to all the functions that I had written, therefore it would require extra work to make sure everything worked properly. On the other hand changing the structure, usually meant maintenance became a bit easier.

Description

I wrote this program so that it always asks the user what input it needs. For example, when playing the game, the program will ask the user to input the row, and it will only accept a valid row. It will repeat the process for the column. It then calculates what is behind the selected space and outputs the appropriate message or update game area. As long as the game is not over, this process continues.


A
Comments
Name
Data
Purpose
Ask for user name
create player
Ask if player wants
to play
answer is yes
delete player
Libraries
iostream
cstdlib
fstream
string
iomanip
True
Declare variables
Ask if user
wants to
see empty
minefields
Structure
Definition
MineField
call function fields
answer if yes?
Call function
prompt
Function
protoypes
Exit
game state valid
False
Tell user
game state
was invalid
A
answer was yes?
game state valid?
True
Call function
playGame
ask if user
want to play
again
call function
prompt
True
answer is yes?

Pseudo Code

```
Ask user for their name
Create the player
Ask if user wants to play
If yes
    Initialize the MineField variables
    Prompt user to enter size of minefield and difficulty
    Check if data is valid
    If data is valid
        Begin playing the game
        Continue playing until users has won or lost
        If the game is over ask the user if they would like to play again
        If yes
            Re- prompt user the enter size of minefield and difficulty
            Play again
    Else
        Tell the user that data is invalid
Delete the player
Ask user if they would like to see the result of the last game
If yes
    Print the result of the last game
```

Major Variables

Type	Variable Name	Description	Location
short	**data	This variable holds the game data	Line 28 In struct MineField
short	rows	This variable holds the number of rows	Line 29 In struct MineField
short	cols	This variable holds the number of columns	Line 30 In struct MineField
Enum	Difficulty	This variable determines how many mines to set	Line 24 In struct MineField
enum	Flags	This variable holds status conditions for the spaces in the minefield	Line 26 In struct MineField
char *	Player	This variable holds the name of the player	Line 74 In main()

Program

```
/*
 * Ruiz, Juan - Project 1 - 48130
 *
 * Project allows user to play Minesweeper
 *
 * Structures
 *      Functions with structure input: Most of the functions
 *      Functions returning structures: Function create()      Line 235
 *      Function with array of structures: Function fields()   Line 584
 *
 * Memory allocation:
 *      dynamic 2D array in function create()                Line 242
 *      dynamic 1D array in function userName()              Line 225
 *
 * Binary Files
 *      Writing to: function writeBin()                      Line 553
 *      Reading from: function readBin()                     Line 562
 * Strings
 *      Function:
 *         writeBin(), readBin(), userName()             Line 553, 562, 220
 *
 * Pointers
 *      Structure passed as pointer in most functions
 *     2D array pointer notation in prntClear()            Line 288, 290
 *     1D array pointer notation in userName()              Line 227, 229
 *     Returning pointer in create()                        Line 235
 */

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

/***************************************************
 *
 *               Structure
 *
 **************************************************/
/// This is the structure that holds the minefield
/// as well as the associated flags that occur when
/// a user selects a square
struct MineField {
   /// Determines how many mines to set
   enum Difficulty {EASY, NORMAL, HARD};
   /// Flags representing various square possibilities
   enum Flags {EMPTY=10, MINE, CLEAR, LOSER};
   /// This is the minefield
```

```
    short **data;
    /// The total number of rows
    short rows;
    /// The total number of columns
    short cols;
    /// number of mines
    short mines;
};

using namespace std;

/*************************************************
 *
 *          Function Prototypes
 *
 ************************************************/
MineField *create(short, short);
void destroy(MineField *);
void prntClr(MineField *);
void prntObscr(MineField *);
MineField::Difficulty shortToDiff(short);
bool isValidIn(short, short, MineField::Difficulty);
short nMines(MineField::Difficulty);
void setMines(MineField *);
void setFlags(MineField *);
```

```
short nAdjacent(MineField *, short, short, short = MineField::MINE);
bool isClear(MineField *, short, short);
void clrArea(MineField *, short, short);
void setPerim(MineField *);
void showZeros(MineField *, short, short);
bool hasWon(MineField *);
void fields();
bool cont(MineField *, short, short);
void playGame(short, short, MineField::Difficulty, char*);
void prompt(short&, MineField::Difficulty&);
char *userName();
void writeBin(MineField *, string);
void readBin(string);

/**************************************************
 *
 *                    Main
 *
 *************************************************/
int main(int argc, const char * argv[]) {
   /// Get the user name
   char *player = userName();
   /// ask user if they want to play
   cout << "Hello " << player
       << ", Would you like to play a game of minesweeper?\n"
```

```
        "Hit 'y' if yes\n";
char ans;
cin >> ans;

if (ans == 'y') {
    /// create minefield variables
    short nrows;
    MineField::Difficulty d;
    /// Get game information from user
    prompt(nrows, d);
    /// Check that data is valid before creating the array
    /// that holds the results of previous games
    if (isValidIn(nrows, nrows, d)) {
        while (ans == 'y' && isValidIn(nrows, nrows, d)) {
            playGame(nrows, nrows, d, player);
            cout << endl;
            cout << "Would you like to play again " << player << "? ";
            cin >> ans;
            cout << endl;
            /// Get new data only if user wants to continue
            if (ans =='y')
                prompt(nrows, d);
        }
    }
    /// User information was invalid
```

```
        else
            cout << "Minefield too small. Goodbye: ";
    }
    cout << "Game is Over.\n";

    /// Cleanup
    delete player;

    readBin("result");

    cout << "Would you like to see some empty minefields "
            "stored in a structure?\n"
            "Hit 'y' for yes: ";
    cin >> ans;
    if (ans == 'y')
        fields();

    cout << endl;
    cout << "Goodbye\n";

    return 0;
}

/**************************************************************
 *
 *              Function definitions
 *
 **************************************************************/

void prompt(short &rows, MineField::Difficulty &d) {
   cout << "Enter the number of rows\n"
         "Minefield will be NxN in size: ";
   cin >> rows;

   short diff;
   cout << "Enter the difficulty\n"
         "0=Easy\t 1=Normal\t 2=Hard\n";
   cin >> diff;
   d = shortToDiff(diff);
}

/// Function returns true if input was valid
bool isValidIn(short rows, short cols, MineField::Difficulty diff) {
   /// make sure that the number of mines does not exceed
   /// the number of spots available
   return (rows * cols) > nMines(diff);

}

/// Play a game of minesweeper
/// User inputs how many rows and columns and the dicculty
void playGame(short nrows, short ncols, MineField::Difficulty diff, char *p) {
    srand(static_cast<unsigned int>(time(0)));
    MineField *mf = create(nrows, ncols);
    mf->mines=nMines(diff);
    setMines(mf);
    prntObscr(mf);
    short row, col;
    do {
        /// Select the row
        do {
            cout << "Enter the row: ";
            cin >> row;
            /// check bounds
        } while (row < 0 || row >= mf->rows);
        do {
            cout << "Enter the column: ";
            cin >> col;
            /// check bounds
        } while (col < 0 || col >= mf->cols);
        cout << endl;
    } while (cont(mf, row, col) && !hasWon(mf));

    /// Prepare to print completed minefield
    if (hasWon(mf)) {
        cout << p << "You win\n";

        setFlags(mf);

    }

    else{

        cout << p << " you have lost\n";

        setFlags(mf);

        mf->data[row][col]= MineField::LOSER;

    }

    /// Print the complete minefield

    prntClr(mf);


    /// write result to binary file

    writeBin(mf, "result");

    /// deallocate the game area

    destroy(mf);

}


/// Function gets the user name as a string converts it to a char array

/// for the 1d dynamic array requirement

char *userName() {

    cout << "Enter your name: ";

    string in;

    cin >> in;


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

    return name;
}

/// Function that creates the grid on which game will be played
MineField* create(short rows, short cols) {
    /// dinamically create a minefield
    MineField *out = new MineField;
    out->rows=rows;
    out->cols = cols;

    /// Create the 2D game minefield
    out->data = new short *[rows];

    /// Create each row
    for (short row = 0; row != rows; ++row)
        out->data[row] = new short [cols];

    /// Make sure each square is empty
```

```
  for (short i = 0; i != rows; ++i)
    for (short j = 0; j != rows; ++j)
      out->data[i][j] = MineField::EMPTY;
  return out;
}

/// Function return the MineField::Difficulty type from
/// the short variable
MineField::Difficulty shortToDiff(short choice) {
  switch (choice) {
    case (0):
      return MineField::Difficulty::EASY;
      break;
    case (1):
      return MineField::Difficulty::NORMAL;
      break;
    case (2):
      return MineField::Difficulty::HARD;
    default:
      return MineField::Difficulty::EASY;
      break;
  }
}

/// Function deallocates memory
```

```
void destroy(MineField *mf) {
    /// delete each dynamically allocated row
    for (short i = 0; i != mf->rows; ++i)
        delete[] mf->data[i];
    /// delete the dynamically allocated structure
    delete mf;
}


/// Functions prints the minefield with all the squares revealed.
/// used mostly after player loses
void prntClr(MineField* mf) {
    for (short row = 0; row != mf->rows; ++row){
        for (short col = 0; col != mf->cols; ++col) {
            ///
            if ( *(*(mf->data+row) + col) == MineField::LOSER)
                cout << "T  ";
            else if (*(*(mf->data+row) + col) == MineField::MINE)
                cout << "x  ";
            else if (!isClear(mf, row, col))
                     cout << nAdjacent(mf, row, col) << "  ";
            else
                cout << "0  ";
        }
        cout << endl;
    }
    cout << endl;
}


/// Function prints the minefield with spaces hidden
void prntObscr(MineField* mf) {
    /// Print the column index
    for (short i = 0; i != mf->cols; ++i){
        /// Pad initial output of column indicator
        if (i==0)
            cout << "  ";
        cout << setw(3) << i;
    }
    cout << endl;
    for (short row = 0; row != mf->rows; ++row){
        for (short col = 0; col != mf->cols; ++col){
            if(col == 0 && row < 10) cout << row << "  ";
            if (col == 0 && row >= 10) cout << row << " ";
            /// KEEP EMPTY spaces and MINEs hidden
            if (mf->data[row][col] == MineField::EMPTY ||
                mf->data[row][col] == MineField::MINE)
                cout << setw(3) << right  << "* ";
            /// print out the CLEARed area
            else if (mf->data[row][col] == MineField::CLEAR)
                cout << setw(2)<< 0 << " ";
            /// Print out the actual value of the square
            else
                cout << setw(2)<< mf->data[row][col] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/// Function returns the number of mines to set based on Difficulty
short nMines(MineField::Difficulty d) {
    if (d==MineField::EASY)
        return 15;
    else if (d==MineField::NORMAL)
        return 30;
    else
         return 45;
}

/// Function places mines in grid
void setMines(MineField *mf) {
    /// holds how many mines will be used
    short mines = mf->mines;

    /// keep looping through minefield until all mines are set
    while (mines) {
        for (short i = 0; i != mf->rows; ++i) {
            for (short j = 0; j != mf->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 is empty
                    if (mines && mf->data[i][j] == MineField::EMPTY) {
                        /// set the mine
                        mf->data[i][j] = MineField::MINE;
                        --mines;
                    }
                }
            }
        }
    }
}

/// Function returns how  many 'flag' elements surround a given square
short nAdjacent(MineField *mf, short row, short col, short FLAG) {
    short nAd=0;              /// the number of adjacent mines

    /// not on first or last row or first or last column
    /// most of the searches take place in this area
    if ( row > 0 && col > 0 && row < mf->rows-1 && col < mf->cols-1) {
        /// search the 3x3 grid surrounding a cell
        for (short i = row-1; i <= row+1; ++i) {
            for (short j = col-1; j <= col+1; ++j)
                if (mf->data[i][j] == FLAG)
                    ++nAd;
        }
    }
    /// on the first row, not on first or last column
    else if ( row == 0 && col > 0 && col < mf->cols - 1) {
        for (short i = row; i <= row+1; ++i) {
            for (short j = col-1; j <= col+1; ++j)
                if (mf->data[i][j] == MineField::MINE)
                    ++nAd;
        }
    }
    /// on the last row, not on first or last column
    else if ( row == mf->rows-1 && col > 0 && col < mf->cols - 1) {
        for (short i = row-1; i <= row; ++i) {
            for (short j = col-1; j <= col+1; ++j)
                if (mf->data[i][j] == MineField::MINE)
                    ++nAd;
        }
    }
    /// on the first column, not on first or last row
    /// search to the right
    else if ( col == 0 && row > 0 && row < mf->rows - 1) {
        for (short i = row-1; i <= row+1; ++i) {
            for (short j = col; j <= col+1; ++j)
                if (mf->data[i][j] == MineField::MINE)
                    ++nAd;
        }
    }
    /// on the last column, not on first or last row
    /// search to the left
    else if ( col == mf->cols-1 && row > 0 && row < mf->rows - 1) {
        for (short i = row-1; i <= row+1; ++i) {
            for (short j = col-1; j <= col; ++j)
                if (mf->data[i][j] == MineField::MINE)
                    ++nAd;
        }
    }
    /// top left corner
    else if (row == 0 && col == 0) {
        if (mf->data[row][col+1] == MineField::MINE) ++nAd;
        if (mf->data[row+1][col] == MineField::MINE) ++nAd;
        if (mf->data[row+1][col+1] == MineField::MINE) ++nAd;
    }
    /// top right corner
    else if (row == 0 && col == mf->cols-1) {
        if (mf->data[row][col-1] == MineField::MINE) ++nAd;
        if (mf->data[row+1][col] == MineField::MINE) ++nAd;
```

```
        if (mf->data[row+1][col-1] == MineField::MINE) ++nAd;
    }
    /// bottom left corner
    else if (row == mf->rows-1 && col == 0) {
        if (mf->data[row-1][col] == MineField::MINE) ++nAd;
        if (mf->data[row-1][col+1] == MineField::MINE) ++nAd;
        if (mf->data[row][col+1] == MineField::MINE) ++nAd;
    }
    /// bottom right corner
    else if (row == mf->rows-1 && col == mf->cols-1) {
        if (mf->data[row-1][col-1] == MineField::MINE) ++nAd;
        if (mf->data[row-1][col] == MineField::MINE) ++nAd;
        if (mf->data[row][col-1] == MineField::MINE) ++nAd;
    }
    /// return number of mines from appropriate if statement
    return nAd;
}

/// Function is true if there 0 landmines adjacent to
/// selected square
bool isClear(MineField * mf, short row, short col) {
    if (nAdjacent(mf, row, col))
        return false;           /// there was at least one mine adjacent
    return true;                /// area was clear
}
```

```
/// Clear an area whose values are clear
/// i.e 0 adjacent  mines
void showZeros(MineField *mf, short row, short col) {
   /// check bounds
   if ( row >= mf->rows || row < 0 || col >= mf->cols || col < 0)
      return;
   if (isClear(mf, row, col) && mf->data[row][col] != MineField::CLEAR){
      mf->data[row][col] = MineField::CLEAR;
      /// go up one row
      showZeros(mf, row+1, col);
      /// go down one row
      showZeros(mf, row-1, col);
      /// go right one col
      showZeros(mf, row, col+1);
      /// go left one col
      showZeros(mf, 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 setFlags(MineField *mf) {
    for (short i = 0; i != mf->rows; ++i)
        for (short j = 0; j != mf->cols; ++j)
            /// don't look for adjacent mines in areas where
            /// mine is already located
            if (mf->data[i][j] != MineField::MINE)
                mf->data[i][j] = nAdjacent(mf, i, j);
}

/// 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 cont(MineField * mf, short row, short col) {
    /// check if user selected a losing square
    if (mf->data[row][col] == MineField::MINE)
        return false;

    /// Square is a zero, clear the surrounding area if necessary
    else if (isClear(mf, row, col) ){
        showZeros(mf, row, col); /// show cleared area
        setPerim(mf);
        prntObscr(mf);
        return true;
    }
    /// Square had adjacent mine
    /// reveal the number to the user
    else {
        mf->data[row][col] = nAdjacent(mf, row, col);
        prntObscr(mf);
        return true;
    }
}

/// Function checks whether the player has won
bool hasWon(MineField *mf) {
        for (short i = 0; i != mf->rows; ++i)
            for (short j = 0; j != mf->cols; ++j)
                /// if there are empty spaces player has not won
                if (mf->data[i][j] == MineField::EMPTY)
                    return false;
        /// there were no empty spaces left. Player has won
        return true;
    }

/// Function find the perimeter of the cleared areas
void setPerim(MineField *mf) {
    for (short row = 0; row != mf->rows; ++row ) {
        /// avoid search at left and right edge of array
        for (short col = 0; col != mf->cols; ++col) {
            /// when you're not on the bounds of the array
            }
            if (row > 0 && row < mf->rows-1
                && col > 0 &&  col <mf->cols-1)
                if (mf->data[row][col] == MineField::CLEAR) {
                    /// check that the previous number has mines adjacent
                    if (mf->data[row][col-1] != MineField::CLEAR)
                        mf->data[row][col-1] = nAdjacent(mf, row, col-1);
                    /// check if the next number has mines adjacent
                    if (mf->data[row][col+1] != MineField::CLEAR)
                        mf->data[row][col+1] = nAdjacent(mf,row, col+1);
                    if (mf->data[row-1][col] != MineField::CLEAR)
                        mf->data[row-1][col] = nAdjacent(mf, row-1, col);
                    /// check if the next number has mines adjacent
                    if (mf->data[row+1][col] != MineField::CLEAR)
                        mf->data[row+1][col] = nAdjacent(mf,row+1, col);
                    /// check the adjacent corners
                    if (mf->data[row+1][col-1] != MineField::CLEAR)
                        mf->data[row-1][col-1] = nAdjacent(mf,row-1, col-1);
                    if (mf->data[row-1][col+1] != MineField::CLEAR)
                        mf->data[row-1][col+1] = nAdjacent(mf,row-1, col+1);
                    if (mf->data[row+1][col-1] != MineField::CLEAR)
                        mf->data[row+1][col-1] = nAdjacent(mf,row+1, col-1);
                    if (mf->data[row+1][col+1] != MineField::CLEAR)
                        mf->data[row+1][col+1] = nAdjacent(mf,row+1, col+1);
                }
        }
```

```
    }
}

/// Function writes the minefield structure to a binary file
void writeBin(MineField *mf, string fileName) {
    /// Write the result to a binary file
    fstream out(fileName.c_str(), ios::out | ios::binary);    /// open the file
    out.write(reinterpret_cast<char *>(&mf),sizeof(*mf)); /// write to the file
    out.close();
}

/// Function prints the data variable from the Minefield structure
/// writen to a binary file
void readBin(string fileName) {
    /// Ask user if they want to see the result of the last game
    char response;
    cout << "Would you like to see the result of the last game as "
    "read from a binary file?\n"
    "Hit 'y' if yes: ";
    cin >> response;
    if (response == 'y') {
        cout << "\nResult of your last game:\n";
        /// Create space to hold the file read
        MineField *result;
        fstream in(fileName.c_str(), ios::in | ios::binary);
        in.read(reinterpret_cast<char *>(&result), sizeof(*result));
        prntClr(result);
        in.close();
    }

}

/// This function creates an array of the Minefield structure
/// as part of the requirments to be able to write to and read
/// from an array of structures
void fields() {
    cout << "How many mine fields do you want to see: ";
    int n;
    cin >> n;

    MineField **mf = new MineField*[n];
    const int row = 10;
    const int col = 10;
    /// create the fields
    for (int i = 0; i != n; ++i) {
        /// Create each field
        mf[i] = create(row, col);
        /// get number of mines
        mf[i]->mines = nMines(MineField::EASY);
        /// set the mines
        setMines(*(mf+i));

        /// set the flags

        setFlags(*(mf+i));

        /// print the field

        prntClr(*(mf+i));

        cout << endl;

    }

    cout << endl;


    /// deallocate memory

    for (int i = 0; i != n; ++i) {

        destroy(*(mf+i));

    }

    delete []mf;

}
```