Minesweeper

By: John Haller

**Introduction:**

The purpose of my program is to create a game of minesweeper. I chose this game because i felt that it was a game that was doable for me, while still maintaining a complex enough theme to challenge me. I also wanted to choose this game because it is one that i used to play a lot, whenever the internet went out, and one that I had a lot of fun playing.

**Summary:**

My program is 1011 lines of code

There are 17 major variables

As well as many counter variables used for loops and writing to files.

This program took me approximately a week and spread out over the last month to make it.

The most challenging part of the project was figuring out how to convert my class to be derived from an abstract class after I originally got it working without it. But eventually I figured it out, and that was the most rewarding part of the project.

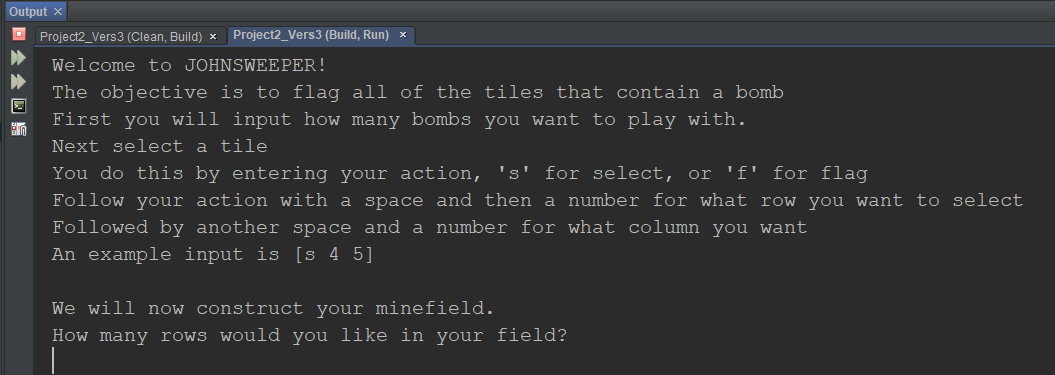
**Description:**

The way this program runs is by creating objects for 3 classes that run most of the program. The Player class, which handles the field that the player sees and interracts with. The Bomb class which is the field that contains bombs and is used to be compared agaisnt with the player field. And finally the object for the FileOut class which handles all of the writing to files.

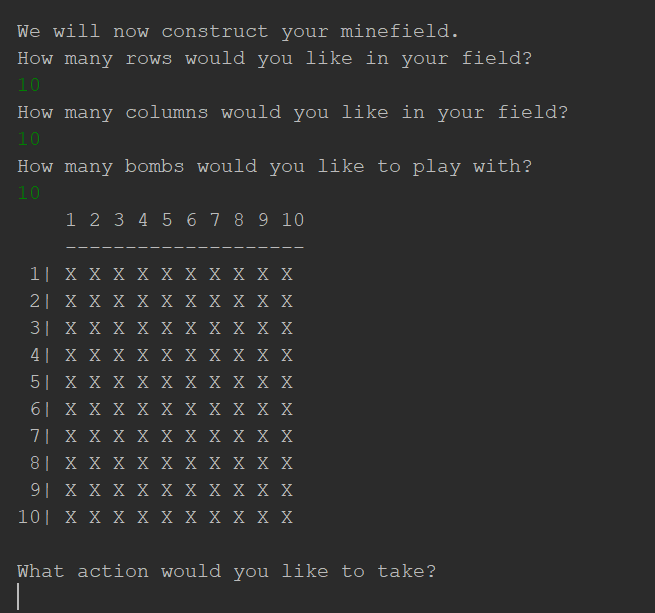
It then creates the the field and fills them. Then moves on to inputting the players choice and checking for bombs. If none are hit then it will continue until the player flags all bombs or dies.

It will also right the turn data to a file after every move, and at the end of game it will write the result to a seperate file for each game played in a single program run.

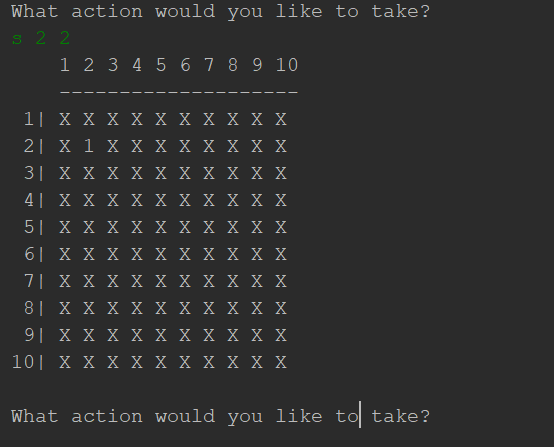
**Sample I/O**

****

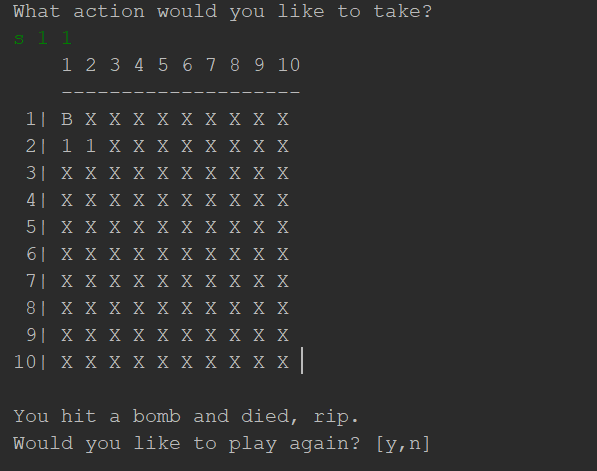
This is the introduction screen and prompt for the size of field



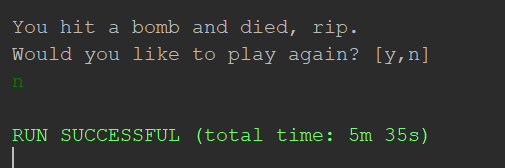
This is the input for field size and the first time the field is displayed



This is after selecting tile 2 2 and the field after your selection has been processed.



This is after hitting a bomb



After saying no to playing again the program ends.

**Pseudocode:**

Declare Variables

Display introduction

Start loop for multiple games

set alive to true

declare counters

input desired rows,cols, and bombs

Test if bombs input are <0, if true throw exception

Catch if exception thrown

Increment games

Create pointers to classes

Print player field

initialize data in FileOut class

Start loop for turns

Start loop for input validation

Input player move

If valid end loop

increment turn

Send turn data to be written to file

write turn data to file

if player selected tile

check tile

if hit bomb

dead

end turn loop

check surrounding tiles

If player flagged tile

set tile to ‘F’

if all bombs flagged

end turn loop

if alive

output to file

if dead

output to file

promt to play again

if yes restart loop

if no end game loop

**Doxygen:**

The doxygen html file will be included in the Project2 folder that is turned in.

**Major Variables:**

int nRows - The number of rows the player wants to play with

int nCols - The number of columns the player wants to play with

int bombs - The number of bombs the player wants to play with

int chosenR - The row the player has chosen for their turn

int chosenC - The col the player has chosen for their turn

char command - The action the player is going to take

bool alive - The players life status

int games - The current game number

int flagCnt - The count of flags successfully placed on bombs

int nFlags - The number of flags placed

int rows – number of rows in field

int cols – number of cols in field

char \*\*field – the 2d field

int nBombs – number of bombs in the bomb field

int games - Counter for number of games played

int turn - Counter for what turn it is

char action - Action input (f,s)

**Concepts used:**

Instance of a class – 119,122

Private data members – BOMBFLD.h file line 19

Specification vs Implementation – Any of the .h vs .cpp files

Inline – BOMBFLD.h line 32,35

Constructors – PLAYFLD.h line 19

Destructors – PLAYFLD.h line 39

UML – included in project folder

Static – IEXIST.h line 27

Copy Constructors – PLAYFLD.cpp line 17,18

Overloaded Operators – IEXIST.h Lines 37,41,46

Aggregation – IEXIST.h Line 29

Protected Members – FIELD.h Line 20

Polymorphism – line 227

Abstract Classes – FIELD.h Line 21

Exceptions – Line 106

Templates – FILETMP.h – Line 21

STL – Line 216

**References**

StackOverflow.com for various things

Textbook Gaddis\_9thEd for help with Inheritance and STL

**Code for program:**

Main:

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

//Variables

//The number of rows the player wants to play with

int nRows;

//The number of columns the player wants to play with

int nCols;

//The number of bombs the player wants to play with

int bombs;

//The row the player has chosen for their turn

int chosenR;

//The col the player has chosen for their turn

int chosenC;

//The action the player is going to take

char command;

//The players life status

bool alive;

//The current game number

int games=0;

//Intro

intro();

//Character for if you want to replay or not

char replay;

do{

alive=true;

//The count of flags successfully placed on bombs

int flagCnt=0;

//The number of flags placed

int nFlags=0;

//Input size of grid and number of bombs

cout<<"We will now construct your minefield."<<endl;

cout<<"How many rows would you like in your field?"<<endl;

cin>>nRows;

cout<<"How many columns would you like in your field?"<<endl;

cin>>nCols;

cout<<"How many bombs would you like to play with?"<<endl;

cin>>bombs;

//Exception throwing/handling time

try{

//Tests to see if bombs are negative or not, throws if so

tstBomb(bombs);

}

//Catch the exception if thrown

catch(string exception){

cout<<exception<<endl;

exit(EXIT\_FAILURE);

}

//Increment game counter

games++;

//Pointer for the player grid, this will be the field that the player sees,

//and interacts with.

Player \*pPoint=new Player(nRows,nCols);

//Pointer for the bomb grid, this will not be visible to the player, and

//they will not be able to interact with it.

Bomb \*bPoint=new Bomb(nRows,nCols,bombs);

//Create pointer for FileOut class, this is purely used for writing to

//files. not visible to player.

FileOut \*fPoint=new FileOut(games);

//Print bomb field for testing

// bPoint->prtFld();

//Print player field

pPoint->prtFld();

//Set game counter in fPoint object

fPoint->setGame(games);

//set replay so that game doesnt automatically replay

replay='n';

//Loop to play game

//This do-while is to loop the process of taking turns until the game is over

do{

//input players decision

//this do-while is used to loop the input process until appropriate

//values have been inputted(input validation)

do{

//Run the choice function to input their turn

choice(command,chosenR,chosenC);

//If the user inputs values outside of their specified range

//The program will prompt them to try again

if(chosenR>nRows-1||chosenC>nCols-1){

cout<<"Invalid Input, Try again"<<endl;

}

//End of loop for input validation

}while(chosenR>nRows-1||chosenC>nCols-1);

//Increment turn counter in fPoint object

fPoint->incTurn();

//Set members in fPoint object for writing to file

setFile(fPoint,chosenR,chosenC,command);

//Write turn data to file

write(fPoint);

//If player selects a tile

//This is used to drive the program if the player wants to select a tile

if(command=='s'){

//Check to see if the tile is a bomb and drive program depending on result

alive=check(pPoint,bPoint,chosenR,chosenC);

//Print the field so user can see result of their action

if(alive)pPoint->prtFld();

//If the player chooses to flag a tile

}else if(command=='f'){

//Runs the flag function to flag the chosen tile, as well as increment

//flagCnt if necessary

flag(pPoint,bPoint,chosenR,chosenC,flagCnt,nFlags);

}

//If the player is dead break out of game

if(!alive)break;

//End of loop for the game, when this ends the player has won.

//The condition is set this way so that the player needs to have flagged the

//exact amount of bombs, that way they cannot flag every single tile and then win.

}while(flagCnt!=bombs || nFlags!=bombs);

if(alive){

cout<<"Congratulations you have won JOHNSWEEPER!"<<endl;

//Write vitory to Games.txt

writeGme(fPoint,'V');

//Write loss to Games.txt

}else writeGme(fPoint,'L');

//Delete pointers to objects

cleanup(pPoint,bPoint,fPoint);

cout<<"Would you like to play again? [y,n]"<<endl;

cin>>replay;

}while(replay=='y');

//Create an array of objects that exists for a secret reason

int SIZE=5;

//Declaring impObjs(important objects) of the IExist class

IExist impObjs[SIZE];

//I dont even want to explain this in text..

//Just know its very very important and top secret

for(IExist i;i<SIZE;++i){

impObjs[i.getX()].incCnt();

++i;

--i;

}

ofstream out;

out.open("Important.txt");

//output results of above process to "Important.txt"

out<<impObjs[0].getCnt();

out.close();

//STL stuff

vector <int>v;

for(int i=0;i<SIZE;i++){

v.push\_back(i);

}

out.open("VectorsArePartOfTheSTL.txt");

for(int i=0;i<SIZE;i++){

out<<v[i];

}

out.close();

//Polymorphism

Player \*object;

Field \*point=object;

object->poly();

//Exit stage right!

return 0;

}

Functions:

//The choice function takes the players command, the chosen row, and the chosen col

//It returns nothing

//It is used to input the command of the player

void choice(char &command,int &chosenR,int &chosenC){

cout<<"What action would you like to take?"<<endl;

cin>>command>>chosenR>>chosenC;

//This is done to offset the row for ease of use with array indexes

chosenR-=1;

//This is done to offset the col for ease of use with arrat indexes

chosenC-=1;

}

//The check function takes the pointer to Player class, pointer to Bomb class,

// the chosen row, and the chosen column

//It returns a boolean

//This function determines if the chosen tile is a bomb or not, and then further

// drives the program depending on the result

bool check(Player \*pPoint,Bomb \*bPoint,int chosenR,int chosenC){

//Checks to see if the chosen tile is bomb

if(bPoint->getBomb(chosenR,chosenC)=='B'){

//If it is it sets the tile to = 'B'

pPoint->setFld(chosenR,chosenC,'B');

//Calls death function to output result and return false for alive

death(pPoint);

return false;

}

//If the chosen tile is not a bomb

else if(bPoint->getBomb(chosenR,chosenC)=='X'){

//It will then check the surrounding 8 tiles for bombs

fndBomb(pPoint,bPoint,chosenR,chosenC);

return true;

}

}

//The fndBomb function takes the pointer to Player class, pointer to Bomb class,

// the chosen row, and the chosen column

//It returns nothing

//This function checks the surrounding tiles for bombs, and then sets the chosen

// tile to = the amount found. If none are found the program continues onto recursion

void fndBomb(Player \*pPoint,Bomb \*bPoint,int row,int col){

//This if checks if the tile has been checked already to prevent infinite looping

if(pPoint->getFld(row,col)!=' '){

//Counter for bombs found

char bombCnt=0;

//Check top left

if(bPoint->getBomb(row-1,col-1)=='B')bombCnt++;

//Check top middle

if(bPoint->getBomb(row-1,col)=='B')bombCnt++;

//Check top right

if(bPoint->getBomb(row-1,col+1)=='B')bombCnt++;

//Check middle left

if(bPoint->getBomb(row,col-1)=='B')bombCnt++;

//Check middle right

if(bPoint->getBomb(row,col+1)=='B')bombCnt++;

//Check bottom left

if(bPoint->getBomb(row+1,col-1)=='B')bombCnt++;

//Check bottom middle

if(bPoint->getBomb(row+1,col)=='B')bombCnt++;

//Check bottom right

if(bPoint->getBomb(row+1,col+1)=='B')bombCnt++;

//If no bombs are found around the tile

if(bombCnt==0){

//Set the tile equal to ' '

pPoint->setFld(row,col,' ');

//Call the recursion function to check tiles to the N,S,E,W

recurs(pPoint,bPoint,row,col);

}

//If some amount of bombs are found

else{

//Set the chosen tile = to the number of bombs found.(+48 so that it

//displays correctly as a character)

pPoint->setFld(row,col,bombCnt+48);

}

}

}

//The recurs function takes the pointer to Player class, pointer to Bomb class,

// the chosen row, and the chosen column

//It returns nothing

//This function's sole purpose is the recursively check all tiles to the N,S,E,W

// of the chosen tile

void recurs(Player \*pPoint,Bomb \*bPoint,int row,int col){

check(pPoint,bPoint,row-1,col);

check(pPoint,bPoint,row,col-1);

check(pPoint,bPoint,row,col+1);

check(pPoint,bPoint,row+1,col);

}

//The death function takes the pointer to the Player class

//It returns nothing

//This class is used to output a death message as well as exit the program.

// This is only done if the player selects a bomb as their chosen tile.

void death(Player \*pPoint){

//Print the field one final time

pPoint->prtFld();

//Death message, I know its beautiful

cout<<"You hit a bomb and died, rip."<<endl;

}

//The flag function takes the pointer to the Player class, the pointer to the Bomb class,

// the chosen row, the chosen col, the flagCnt by reference, as well as nFlags by reference

//It returns nothing.

//This program is used to both set the chosen tile as a flag, 'F', and to count

// the times the player succesfully marked a flag, and the amount of flags placed

void flag(Player \*pPoint,Bomb \*bPoint,int row,int col,int &flagCnt,int &nFlags){

//If the chosen tile is already a flag

if(pPoint->getFld(row,col)=='F'){

//Then unflag it

pPoint->setFld(row,col,'X');

//If tile was unflagged decrement flag counter

nFlags--;

}

//otherwise do this

else {

//Set the chosen tile to = flag, 'F'

pPoint->setFld(row,col,'F');

//Increment number of flags

nFlags++;

}

//If the flag was placed on a bomb, increment flagCnt;

if(bPoint->getBomb(row,col)=='B')flagCnt++;

//Print field after players move

pPoint->prtFld();

}

/\*The intro function takes nothing and returns nothing.

\* it is used to display the introduction as well as the rules and how to

\* format your input

\*/

void intro(){

cout<<"Welcome to JOHNSWEEPER!"<<endl;

cout<<"The objective is to flag all of the tiles that "

<<"contain a bomb"<<endl;

cout<<"First you will input how many bombs you "

<<"want to play with. "<<endl;

cout<<"Next select a tile"<<endl;

cout<<"You do this by entering your action, 's' for select, "

<<"or 'f' for flag"<<endl;

cout<<"Follow your action with a space and then a number "

<<"for what row you want to select"<<endl;

cout<<"Followed by another space and a number for what "

<<"column you want"<<endl;

cout<<"An example input is [s 4 5]"<<endl<<endl;

}

/\*The setFile function takes a pointer to fileOut class, chosen row,chosen col,

\* and chosen action

\* It returns nothing

\* Its purpose is to set members in the fileOut class in order to prepare for

\* file writing

\*/

void setFile(FileOut \*fPoint,int row,int col,char action){

fPoint->setRow(row+1);

fPoint->setCol(col+1);

fPoint->setAct(action);

}

/\*The write function takes a pointer to fileOut class.

\* It returns nothing

\* It's purpose is to send info to the fileOut template to be written to Moves.txt

\*/

void write(FileOut \*fPoint){

fileOut(" ");

fileOut(fPoint->getGame());

fileOut(" ");

fileOut(fPoint->getTurn());

fileOut(" ");

fileOut(fPoint->getAct());

fileOut(" ");

fileOut(fPoint->getRow());

fileOut(" , ");

fileOut(fPoint->getCol());

fileOut("\r\n");

}

/\*The wrtieGme function takes a pointe to the fileOut class, and the result of the game

\* It returns nothing

\* The purpose of this function is to send info to the fleOutG template, and be

\* written to Games.txt

\*/

void writeGme(FileOut \*fPoint,char result){

fleOutG(" ");

fleOutG(fPoint->getGame());

fleOutG(" ");

fleOutG(result);

fleOutG("\r\n");

}

/\*The cleanup function takes pointers to the player class, the bomb class, and

\* the fileOut class.

\* It returns nothing

\* It's purpose is to delete dynamic pointers to classes

\*/

void cleanup(Player \*pPoint,Bomb \*bPoint, FileOut \*fPoint){

delete pPoint;

delete bPoint;

delete fPoint;

}

/\*The tstBomb function takes the number of bombs in the game

\* It returns nothing

\* It's purpose is to test if the player input a negative number of bombs, and

\* will throw an exception if so.

\*/

void tstBomb(int bombs){

if(bombs<=0){

string exception="Error, cannot have less than 1 bomb";

throw exception;

}

}

Classes

#ifndef IEXIST\_H

#define IEXIST\_H

#include "AGGREGATE.h"

//This class is used to demonstrate that i know how to do things that i didn't

//find a use for in my project

class IExist{

private:

//Variable used for the top secret thing in main

int x;

//Counter for objects used above top secret thing

static int oCount;

//The place where i used aggregation

Aggregate important;

public:

IExist(){

//Initialize members to 0

oCount=0;

x=0;

}

//Overloaded operator for top secret thing

IExist operator++(){

x++;

return \*this;

}

IExist operator--(){

x--;

return \*this;

}

//Another overloaded operator for top secret thing

bool operator<(int y){

if(x<y)return true;

return false;

}

//Setting member for top secret thing

void setX(int z){

x=z;

}

//Returning x for top secret thing

int getX()const{

return x;

}

//Returns counter for objects

int getCnt()const{

return oCount;

}

//Used to increment counter of objects

void incCnt(){

oCount++;

}

};

#endif /\* IEXIST\_H \*/

#ifndef FILEOUT\_H

#define FILEOUT\_H

#include <iostream>

using namespace std;

//This class is used strictly for writing to files

class FileOut{

private:

//Counter for number of games played

int games;

//Counter for what turn it is

int turn;

//Chosen row

int row;

//Chosen col

int col;

//Action input (f,s)

char action;

public:

FileOut(int);

//Sets the number of game being played

void setGame(int game){

games=game;

}

//used to increment turn

void incTurn(){

turn++;

}

//Returns game number

int getGame()const{

return games;

}

//Sets chosen row

void setRow(int x){

row=x;

}

//Sets chosen col

void setCol(int x){

col=x;

}

//Sets selected action

void setAct(char x){

action=x;

}

//Returns current turn

int getTurn()const{

return turn;

}

//Returns chosen row

int getRow()const{

return row;

}

//Returns chosen col

int getCol()const{

return col;

}

//Returns chosen action

char getAct()const{

return action;

}

};

#endif /\* FILEOUT\_H \*/

#include <iomanip>

#include <fstream>

using namespace std;

#ifndef FILETMP\_H

#define FILETMP\_H

//The purpose of this template is to write to "Moves.txt"

template <class T>

T fileOut(T n){

ofstream out;

//open file

out.open("Moves.txt",ios::app);

//write to file

out<<n;

//close file

out.close();

}

#endif /\* FILETMP\_H \*/

#include <iostream>

#include <fstream>

using namespace std;

#ifndef FOUTTMP\_H

#define FOUTTMP\_H

//This template is used for writing to the "Games.txt" file

template <class T>

T fleOutG(T n){

ofstream out;

//open file

out.open("Games.txt",ios::app);

//write to file

out<<n;

//close file

out.close();

}

#endif /\* FOUTTMP\_H \*/

#ifndef FIELD\_H

#define FIELD\_H

#include <iostream>

//This is an abstract class used to inherit from.

//The classes inheriting from this are Player, and Bomb

class Field{

protected:

int rows;

int cols;

char \*\*field;

public:

void poly(){

if(1<0)std::cout<<"HI"<<std::endl;

}

//Virtual functions

virtual int getRow()const=0;

virtual int getCol()const=0;

};

#endif /\* FIELD\_H \*/

class Player :public Field{

public:

Player ();

//Sets members inherited from Field

Player(int,int);

//Returns number of rows

int getRow()const{

return rows;

}

//Returns number of cols

int getCol()const{

return cols;

}

//Sets the value of chosen tile

void setFld(int i,int j,char tile){

field[i][j]=tile;

}

//Gets the value of the chosen tile

char getFld(int i,int j)const{

return field[i][j];

}

void makeF();//Allocates pField

void filFld();//Fills player array with 'X'

void prtFld();//Prints the player field

~Player();

};

#endif /\* PLAYFLD\_H \*/

#include "PLAYFLD.h"

#include "BOMBFLD.h"

#include <iostream>

#include <iomanip>

using namespace std;

Player::Player(){}

//Initializes members inherited from Field class, and drives creation of player field

Player::Player(int nrows,int nCols){

rows=nrows;

cols=nCols;

//Allocates memory for the player field

makeF();

//Fills the player field with 'X's

filFld();

}

//Allocates memory for player field

void Player::makeF(){

char \*\*a=new char\*[rows];

for(int i = 0;i < rows;i++){

a[i]=new char[cols];

}

field=a;

}

//Fills the player field with 'X's

void Player::filFld(){

for(int i = 0; i < rows; i++){

for(int j = 0; j < cols; j++){

field[i][j]='X';

}

}

}

//Prints the Player field and its borders

void Player::prtFld(){

//formatting

cout<<" ";

//Numbers at top of field

for(int i=0;i<cols;i++){

cout<<i+1;

if(i+1<10)cout<<" ";

}

//Lines under numbers

cout<<endl<<" ";

for(int i=0;i<cols;i++){

cout<<"--";

}

cout<<endl;

//Start printing of field

for(int i = 0; i < rows; i++){

//Numbers to the left of field as well as a line

cout<<setw(2)<<i+1<<"| ";

for(int j = 0; j < cols; j++){

std::cout<<field[i][j]<<" ";

}

cout<<endl;

}

cout<<endl;

}

//Destructor deallocates memory used in player field

Player::~Player(){

for(int i=0; i<rows;i++){

delete field[i];

}

delete field;

}