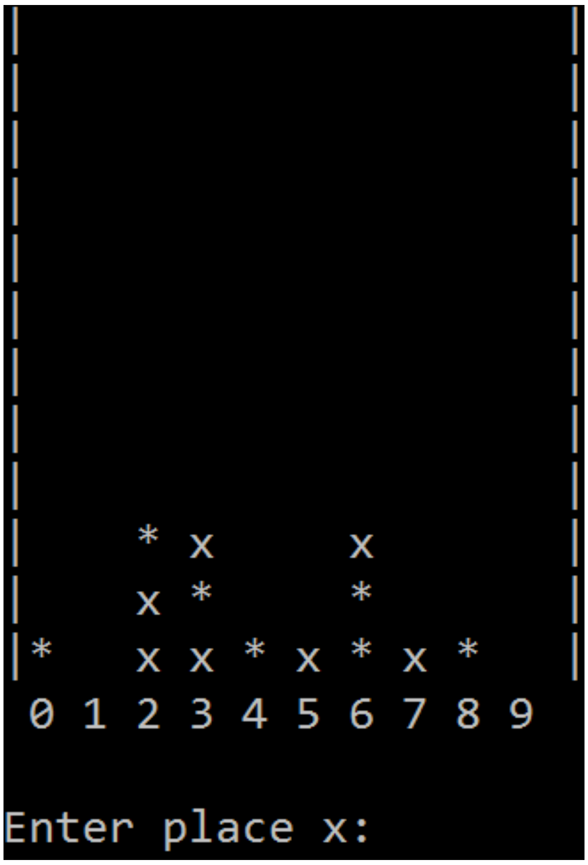
CS2E03 GAME PROJECT REPORT



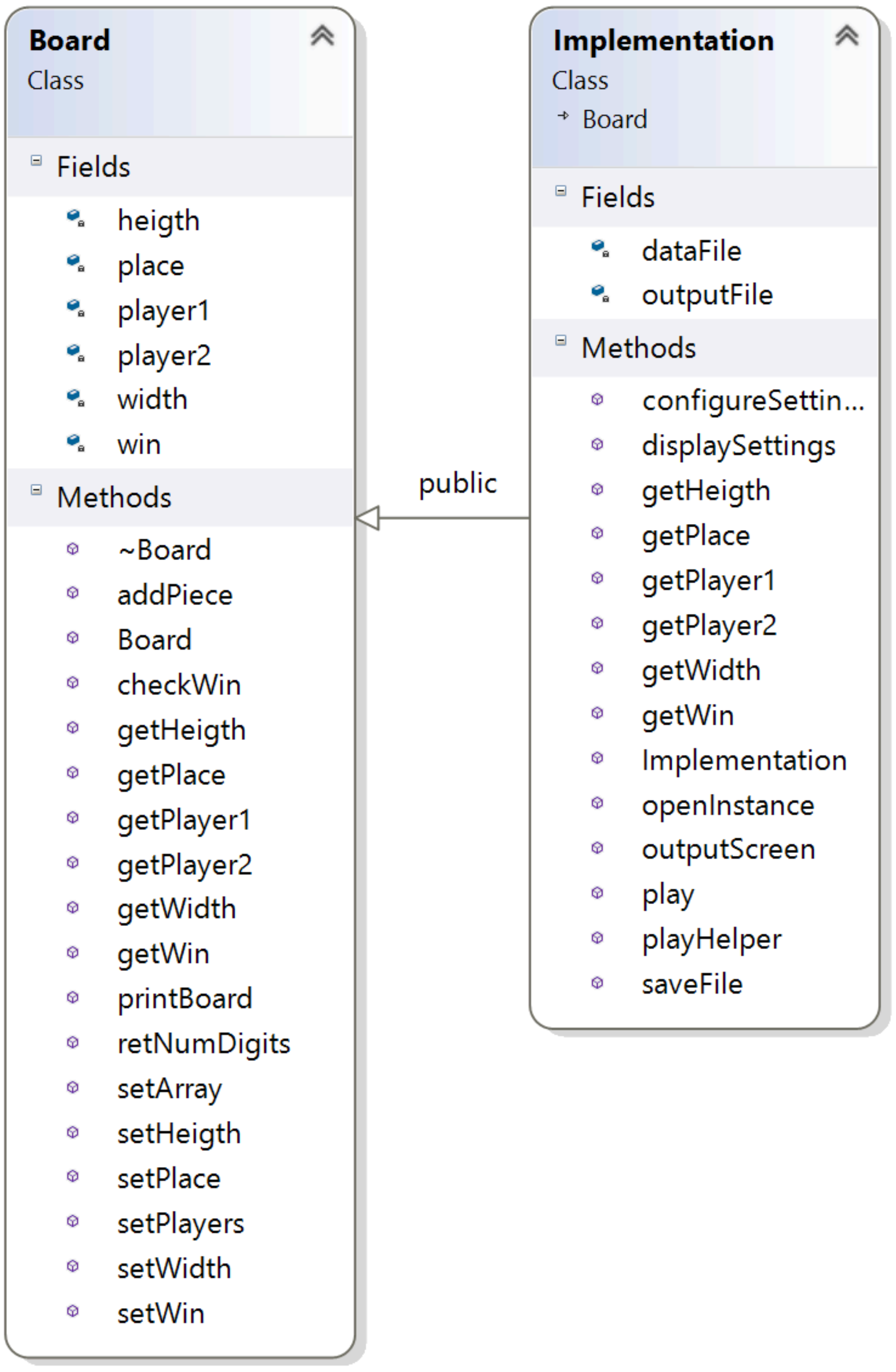
```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|    * x     x       |
|    x *     *       |
|*   x x * x * x *   |
 0 1 2 3 4 5 6 7 8 9

Enter place x:
```

DANIEL DENNIS
15316947

TABLE OF CONTENTS

SECTION A

PART 1



PART 2

The game is a two-player game. By default, the game is a two-dimensional board with 10x12 places where pieces can be placed. Players take turns in putting a piece on the board by entering a number, from 0-9 for where they want to put their piece. In the physical game, the pieces are affected by gravity and will slot in to the lowest place on the board, for example, if a player enters *3*, (as stated before, there are 10x12 places), the piece will then occupy the lowest available space that has not already been occupied by a previous entry. The objective of the game is for one of the players to get four concurrent pieces in a row, which can be horizontal, vertical, or diagonal. When this occurs, that player wins and the game ends.

```
void Board::setArray(int inWidth, int inHeight)
{
     for (int i = 0; i < width; i++)
          delete[] place[i];
     delete[] place;

     height = inHeight;
     width = inWidth;

     place = new char*[inWidth];
     for (int i = 0; i < inWidth; i++)
          place[i] = new char[inHeight];
     for (int i = 0; i < inHeight; i++)
     {
          for (int j = 0; j <inWidth; j++)
               place[j][i] = ' ';
     }

}
```

The above code is the function that sets up or changes the 2d array that holds the data for the board.

While the default setup is a 10x12 board, with ‘x’ and ‘*’ as the default pieces, and four the amount of pieces in a row required to win, the game is customisable before a new game is started and all of these characteristics can be changed and is only practically limited by the size of the output screen and the ASCII character set. When a new game is opened, an option appears to configure the settings of the game, these settings cannot be changed however during the gameplay.

The game also has the option of saving the current game. If either user inputs -1 while playing the game, the program saves the current instance to a file called ‘data.c4x’ (which is created if it doesn’t already exist). In the file, it contains the current status of the board, as well as the current settings of the board, namely the dimensions, the length of row required to win as well as the character used for each player. When the game is initially opened, the user is prompted whether they want to open a previous instance, if a previous instance does exist, the program opens the file and inputs all of the settings to memory and the gameplay is resumed from the time before -1 was entered, but if no file exists, or the data file is empty, the program prompts the user to open a new instance instead.

The programme has two classes, with two files per class containing their respective .h and .cpp files as well as a file called main.cpp containing the main function. The two classes are *Board* and *Implementation*; the characteristics of *Implementation* are inherited from *Board*.

Board contains the basic elements required for playing the game, including a dynamic 2d array which contains the data of the board, two int variables holding the respective height and width of the board, two char variables holding the value for what piece is used for each player, and another int variable for the size of the row required to win. It has methods to print the current board from the 2d array, a method to add a piece to the board, a method for checking to see if one of the players has won, a method for changing the size of the 2d array, a method for returning the amount of digits contained in an int value, as well as a set of get and set values for all of the private members that are accessed from *Implementation*. It is import to note that although *Board* contains a constructor, as is required because *Implementation* inherits *Board*, it is empty and if any other class is to use *Board*, it has to set all of the values in order to play the game.

```
//Left-Right diagonal check (going down)
     for (int i = 0; i < width - win; i++)
     {
          for (int j = height - 1; j >= win - 1; j--)
          {
               if (place[i][j] == checker)
               {
                    for (int k = 0; k < win; k++)
                    {
                         if (place[i + k][j - k] == checker)
                              count++;
                         else
                         {
                              count = 0;
                              break;
                         }
                    }
                    if (count == win)
                         return true;
               }
          }
     }
```

Above code is an extract from the function that checks if a the player (passed in with a char variable, checker) has won, this checks in left-right diagonal direction going down.

Implementation uses the *Board* to facilitate playing the game. Since *Board* does not contain a proper constructor, the constructor of *Implementation* initialises all of the values, setting the game to a 10x12 board, setting the player pieces to ‘x’ and ‘*’, and setting the default win to 4. The constructor also calls another method called *void openInstance* which allows the user to open up a previous instance from a file using <fstream>. It contains a method for changing the settings if a new game is opened, it also calls a helper method for displaying the current settings. Finally, it contains a method called *saveFile* for outputting the current state of the game to the 'data.c4x' file.

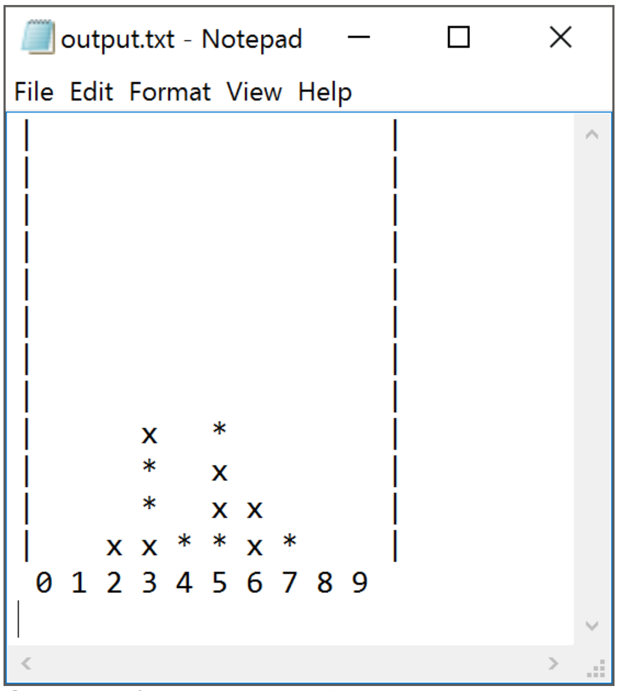
The save file is structured into two parts, the first line has the settings, and the rest contains the values of the board. If the file is opened in a text editor, the board can be seen upside-down because the 2d array is arranged in the tradition Cartesian format (ie 0,0 coordinate system), but it makes more sense to output and input the values in reverse because the *for* loops used are simpler. While the actual board represents an empty space as ' ' in the 2d array, for the purpose of inputting and outputting the board from a file, it has to be represented by a character other than a space so it uses either 'x', 'y', or 'z' depending on whether either of the players are using none, one, or two of the aforementioned characters. In the line dedicated to the settings, the first entry is the filler character used to represent an empty space, followed by the width of the board, the height of the board, the size of the row required for a win, the character for player 1, and the character for player 2.

```
void Implementation::saveFile()
{
	char deadCharacter = 'x';
	if ((getPlayer1() == deadCharacter) || (getPlayer2() ==
deadCharacter))
	{
		deadCharacter = 'y';
		if ((getPlayer1() == deadCharacter) || (getPlayer2()
== deadCharacter))
			deadCharacter = 'z';
	}
	dataFile.open("data.c4x", ios::out | ios::trunc);
	dataFile << deadCharacter << " ";
	dataFile << getWidth() << " " << getHeight() << " " <<
getWin() << " ";
	dataFile << getPlayer1() << " " << getPlayer2() << " " <<
endl;
	for (int i = 0; i < getHeight(); i++)
	{
		for (int j = 0; j < getWidth(); j++)
		{
			if (getPlace()[j][i] != ' ')
				dataFile << getPlace()[j][i];
			else
				dataFile << deadCharacter;
			dataFile << " ";
		}
		dataFile << endl;
	}
}
```

This is the function that saves the game to a file, not that deadCharacter is the character used to denote empty space in the file.

PART 3

Since the output of the screen is reset every time a move is made, I designed the function so that it has to be explicitly called to work. It outputs the current state of the board the same way it shows on the screen.



```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|      x   *         |
|      *   x         |
|      *   x x       |
|    x x * * x *     |
 0 1 2 3 4 5 6 7 8 9
```

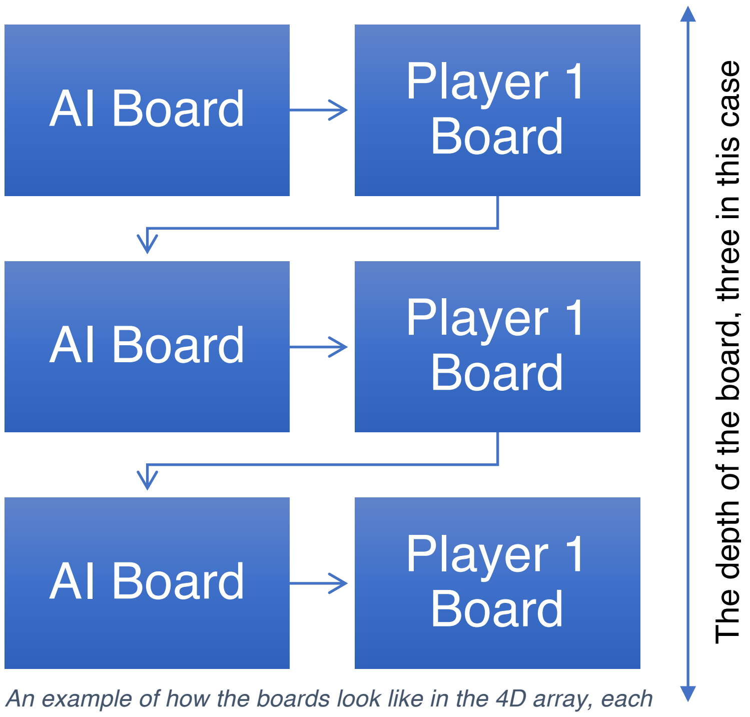
Screenshot from notepad in Windows.

```
void Implementation::outputScreen()
{
     outputFile.open("output.txt", ios::out | ios::trunc);

     for (int j = getHeight() - 1; j > -1; j--)
     {
          outputFile << "|";
          for (int i = 0; i < getWidth(); i++)
          {
               outputFile << getPlace()[i][j];
               for (int k = 0; k < retNumDigits(i); k++)//Used
if the width is greater than 10 to increase spacing
                    outputFile << " ";
          }
          outputFile << "|";
          outputFile << endl;
     }
     outputFile << " ";
     for (int i = 0; i < getWidth(); i++)
          outputFile << i << " ";
     outputFile << endl;

     outputFile.close();
}
```

SECTION B

PART 1

To extend the features of my program, I decided to program an Artificial Intelligence player so that one can play this game by themselves, playing against the computer. This requires a new class to be made. In short, how the class works by looking at the current board state and examining all of the possible future moves. First it looks at a particular input, and then it looks at all of the possible moves after that up to a limit and it ranks that first move with an integer value based on how likely it is to make the AI player win. To conserve memory, the potential future boards are stored in a 4D array. How it works is that there is a hierarchy of boards, the board at the top is passed onto the next board and a move is made, when it moves onto the next position, it overwrites the board with the board above in the hierarchy, thus, for the default 10*12 board and a depth of three, it only takes 480 entries.



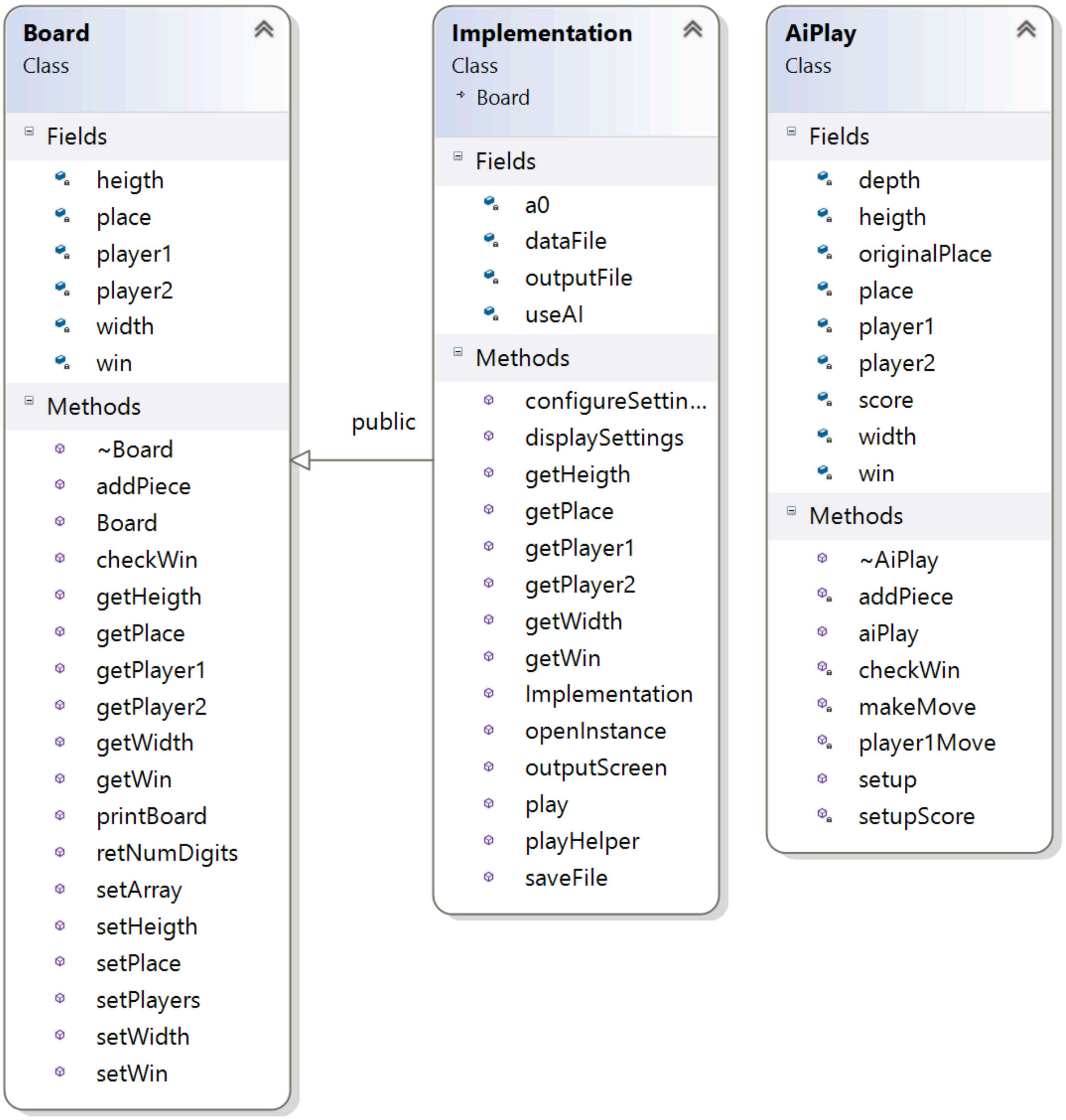
An example of how the boards look like in the 4D array, each square represents a board, by default, it is 10x12, but that can be changed by the player.

First, the function `int aiPlay();` is called, in that, there is a for loop that makes a hypothetical move on a separate board which is copied from the original 2d array to a new 4d array~~.~~ The for loop loops for the width of the board. In each loop, it then calls another function called `void player1Move(int currDepth, int currPosition);` which makes a subsequent move for player 1 in a for loop on another board which is copied from the previous board, it then enters a mutually recursive loop with another function called `void makeMove(int currDepth, int currPosition);` which makes another move for player 2 again, again in a for loop. The length of the loop is determined by a variable called *depth*, so the number of moves made for each increment in depth increases exponentially, that is elaborated upon below. The structure of the 4D array is as follows, the first coordinate is for the *depth* of moves made, the second coordinate is for which player is making the move, the AI player has entry 0, and the user has entry 1. The third coordinate is for the x axis and fourth coordinate is for the y axis of the board (*width* and *height* respectively). The 4D array is this arranged in a hierarchy, instead of making what could be quite a substantial binary tree, as the two mutually recursive functions go through each iteration, they copy the board from a higher ranking one, rewriting the old one and bring it to a state before a move was made in the previous loop (see the above diagram). The hierarchy works in the following order; [0][0][width][height] is copied from the actual board where the game is being played on, the `int aiPlay();` function makes a move for the AI on that board, it then calls `void player1Move(int currDepth, int currPosition);` which then copied the board onto [0][1][width][length]. When that function enters the mutually recersive loop with `void makeMove(int currDepth, int currPosition);`, it then copies the board to [1][0][width][length] copies from [0][1][width][length], then when it loops around, [1][1][width][length] copies from [1][0][width][length], and so on, until the loop has been exhausted, as determined by the *depth* variable. Every time a hypothetical move is made, `int checkWin(char checker, int playerNum, int position)` is called. This function makes a score for that move, so every time the AI gets closer to winning (having a number closer to the *win* variable, it inscreases the score, but if the user gets closer, it decreases the score, and if the next move is a winning move, it gives a score substantially larger score than it normally would, this biasing the algorithm into choosing that move. This function is very similar to the function in Board that checks if a player has won, it has just been modified so that instead of counting if there are four in a row, it counts the amount of times there are two or more pieces in a row and increments the score each time that occurs.

One problem with this method however is that it puts a heavy burden on the CPU, I've found that my own computer can only examine moves up to a depth of three before it takes too long to process an additional move. This is because as the depth of moves is increased, the amount of calculations required moves exponentially. By default, the board has a width of ten, there are also, of course two players, with a depth of just two, the program makes over three million calculations and comparisons to find the right move (I put an int value which increments every time the *checkWin* function checks for concurrent pieces). When the depth is changed to three, that number increases to over three hundred million. If it looks at three sets of moves in advance, it first has to make 10 moves per player, so it makes one million moves in total, it then has to check the board for consecutive pieces, which is takes about 300 checks per move, which explains why it takes 300 million calculations. In my design therefore, it is theoretically possible to have a very difficult AI player, by having a large depth, but that isn't practical in the real world because for a normal computer, while it is capable of doing the calculations because the program doesn't use that much memory, it still takes too long to process and since this is supposed to be a game, it wouldn't be that enjoyable. When I changed the depth to four, I started to time how long it would take the computer to determine its next move, it took 43 minutes to calculate the next move, and thirty-one billion calculations and comparisons, clearly this is not practical for an actual game. If the depth was switched five, I suspect it would take my computer around a day to calculate the next move, also the score counters would likely run into overflow and a system to store them separately would have to be made. There was a certain scope for optimisation therefore to reduce the burden on the CPU and make the game run faster. In the checkWin function, when it is checking sideways and vertically, when it is no longer finding any pieces on a row (going up), it breaks the loop. Expecially when there are not a lot of pieces on the board, it can save quite a bit of CPU time, however, as the board fills with pieces, it means that the response time from the AI gets increasingly slower. I know that I could be more sophisticated with optimising the function even further, but it already goes outside the scope of this project anyway.

Despite everything, the AI gamer still works well at a depth of three, and I've found that the computer gives a reasonable challenge to the player.

PART 2



PART 3

```
class AiPlay
{
private:
     char ****place;
     long long **score;
     int depth;
     int height, width, win;
     char player1, player2;
     char **originalPlace;


     //Private Methods
     bool addPiece(int inPlace, int playerNum, char player, 
int position);
     int checkWin(char checker, int playerNum, int position);
     void setupScore();
     void makeMove(int currDepth, int currPosition);
     void player1Move(int currDepth, int currPosition);


public:
     //Public Methods
     void setup(int inWidth, int inHeight, char inPlayer1, 
char inPlayer2, char **inPlace, int inWin);
     int aiPlay();

     //Constructor and destructor
     ///AiPlay();
     ~AiPlay();
};
```

PART 4

Below is a pseudocode explanation of how each method in the AiPlay class works. It is worth noting that everything is in the same format as the actual code that was written and the actual code works as intended. There is no constructor because the values and the 4D array can only be initialised after the game has started because the user may have modified a setting such as the width of the board.

```
void AiPlay::setup(int inWidth, int inHeight, char inPlayer1, 
char inPlayer2, char **inPlace, int inWin)
{
     width = inWidth;
     height = inHeight;
     player1 = inPlayer1;
     player2 = inPlayer2;
     originalPlace = inPlace;
     win = inWin;
```

```
/* Above values are given in from the Implementation class
after the settings have been set and the game is about to
start*/

	/*
	Structure of 4D array:

	[Depth of moves] [Player Board] [Width] [Height]

	Player 0 is the AI player
	player 1 is the user
	*/
	depth = 3;/*That means that there are 2x3 boards, so
there are 6 hypothetical boards in total*/
	place points to an array of size depth with ***char
values;
	for (run as long as depth with int i)
		place[i] points to an array of size 2 (because there
are two players, so there are two board for each level of
boards) with **char values;
	for (run as long as depth with int i)
	{
		for (run two times with int j)
		{
			place[i][j] points to an array of size width
with *char values;
		}
	}
	for (run as long as depth with int i)
	{
		for (run two times with int j)
		{
			for (run as long as width with int j)
			{
				place[i][j][k] points to an array of size
height with char values;
			}
		}
	}
	setupScore();
}

bool AiPlay::addPiece(int inPlace, int playerNum, char player,
int position)
{
	for (run for the height of the board starting at the
bottom)
	{
		if (if the position at that particular height is
empty)
		{
			that place is equal to player (variable above)
```

```
                return true;
            }
        }

        return false; /*Only happens if that particular move is
full*/
}



int AiPlay::checkWin(char checker, int playerNum, int
position)
{
        int count = 0; /*Used to count how many concurrent pieces
there are in a row (for determining whether a win has
occurred)*/
        int grandCount = 0; /*Used to calculate the overall
score*/
        int cutoff = 0; /*Used to optimise the function so it can
stop searching for pieces when it knows that there are no more
to check*/
        int winReturn; /*Used to biase the AI into choosing a
particular move because it will return a larger score than
normal*/

        if (position == 0)
            winReturn = -1000000;
        else
            winReturn = 100;

        //Right-left diagonal check (going down)
        for (run for width of the board minus win)
        {
            for (run for height of the board minus win)
            {
                if (if position being checked matches the
checker value)
                {
                    grandCount--;
                    for (run win times)
                    {
                        if (using the three for loops, check
if position being checked matches checker variable diagonally
going left-right downwards)
                        {
                            count++;
                            grandCount++;
                        }
                        else
                        {
                            count = 0;
                            break;
                        }
                    }
                    if (if a win has occurred)
                        return winReturn;
                }
            }
        }

        //Left-Right diagonal check (going down)
        for (run for width of the board minus win)
        {
            for (run for height of the board minus win)
            {
                if (if position being checked matches the
checker value)
                {
                    grandCount--;
                    for (run win times)
                    {
                        if (using the three for loops, check
if position being checked matches checker variable going
right-left downwards)
                        {
                            count++;
                            grandCount++;
                        }
                        else
                        {
                            count = 0;
                            break;
                        }
                    }
                    if (if a win has occurred)
                        return winReturn;
                }
            }
        }

        //Left-Right Horizontal check
        for (run for height of the board minus win)
        {
            for (run for width of the board)
            {
                if (if position being checked matches the
checker value)
                {
                    grandCount--;
                    cutoff = 0;
                    for (run win times)
                    {
```

```
                         if (using the three for loops, check
if position being checked matches checker variable going
sideways)
                         {
                              count++;
                              grandCount++;
                         }
                         else
                         {
                              count = 0;
                              break;
                         }
                    }
                    if (if a win has occurred)
                         return winReturn;
               }
               else
                    cutoff++;
          }


          for (run for width - win of the board (i.e. the
parts missed in the other for loop)) /*The other for loop
above this one only checks up to win places from the edge,
this checks the missed spots*/
          {
               if (if position being checked matches the
checker value)
               {
                    for (run until there are no spaces left to
check on the edge at the particular height of the board)
                    {
                         if (using the three for loops, check
if position being checked matches checker variable going
sideways)
                         {
                              grandCount++;
                              /*There is no count++ here
because there is no need to check for a win here since it is
impossible*/
                         }
                    }
               }
               else
                    cutoff++;
          }

          if (cutoff == width)
               goto end3;

          cutoff = 0;
     }
```

```
	end3:

	//Up-Down Check
	for (run for height of the board minus win)
	{
		for (run for width of the board minus win)
		{
			if (if position being checked matches the 
checker value)
			{
				grandCount--;
				cutoff = 0;
				for (run win times)
				{
					if (using the three for loops, check 
if position being checked matches checker variable going 
upwards)
					{
						count++;
						grandCount++;
					}
					else
					{
						count = 0;
						break;
					}
				}
				if (if a win has occurred)
					return winReturn;
			}
			else
				cutoff++;
			if (if the for loop has gone through an empty 
row (i.e. there are no more pieces above))
				goto end4;
		}
	}
	end4:
	return grandCount;
}

void AiPlay::setupScore()
{
	Set the variable score to a new array of long long 
pointers of size 2;
	for (run for two iterations)
		score at the current iteration is a new long long 
variable of size width which is a variable passed in from the 
Implementation class;

	for (run as long as the width of the board)
```

```
          Set the scores (which are kept on the right column)
to zero;
     for (run as long as the width of the board)
          Set the positions from 0 to whatever the width of
the board is;

}

int AiPlay::aiPlay()
{
     bool success;
     int position = 0;
     long long max;
     for (run as long as the width of the board)
     {
          for (run as long as the width of the board)
          {
               for (run as long as the height of the board0
                    copy the actual board from the game onto
the AiPlay's own board onto the top left board;
          }

          Add a piece onto the board as determined by the for
loop from the AI's perspective and check if it is successfully
placed;
          if (if the piece is not placed successfully)
               End this instance of the for loop;
          Check how close the AI is to winning and add that to
the score;

          for (run as long as the width of the board)
          {
               for (run as long as the width of the board
                    Copy the top left hand board onto the top
right hand board where the computer makes a hypothetical move
for player 1;
          }

          Call function to make move for player 1 at the
zeroth depth, also pass on which move where the AI made its
first move (which is used to add onto the score);
     }

     Set the max variable to the first score in the score
array;

     for (run as long as the width of the board)
     {
          if (if the score in the array at the current
instance of the for loop is greater than the current value of
max)
          {
                Set the position variable to the same level
where that higher score is located;
                Set the max variable to the same value as were
that higher value was found;
            }
        }

        return position;
}

void AiPlay::makeMove(int currDepth, int currPosition)
{
        bool success;

        for (run as long as the width of the board)
        {
            for (run as long as the width of the board)
            {
                for (run as long as the height of the board)
                    copy the board from the player 1's
hypothetical board which is on the right hand side, one lower
than the current depth;
            }

            Add a piece onto the board as determined by the for
loop from the AI's perspective and check if it is successfully
placed;
            if (if the piece is not placed successfully)
                End this instance of the for loop;
            Check how close the AI is to winning and add that to
the score;
            Call function to make a hypothetical move for Player
1 at the current depth, also pass on which move where the AI
made its first move at the zeroth depth (which is used to add
onto the score);
        }
}

void AiPlay::player1Move(int currDepth, int currPosition)
{
        bool success;

        for (run as long as the width of the board)
        {
            for (run as long as the width of the board)
            {
                for (run as long as the height of the board)
                    copy the board from the AI's board which
is on the left hand side of the current depth, as determined
by the variable currDepth;
            }
```

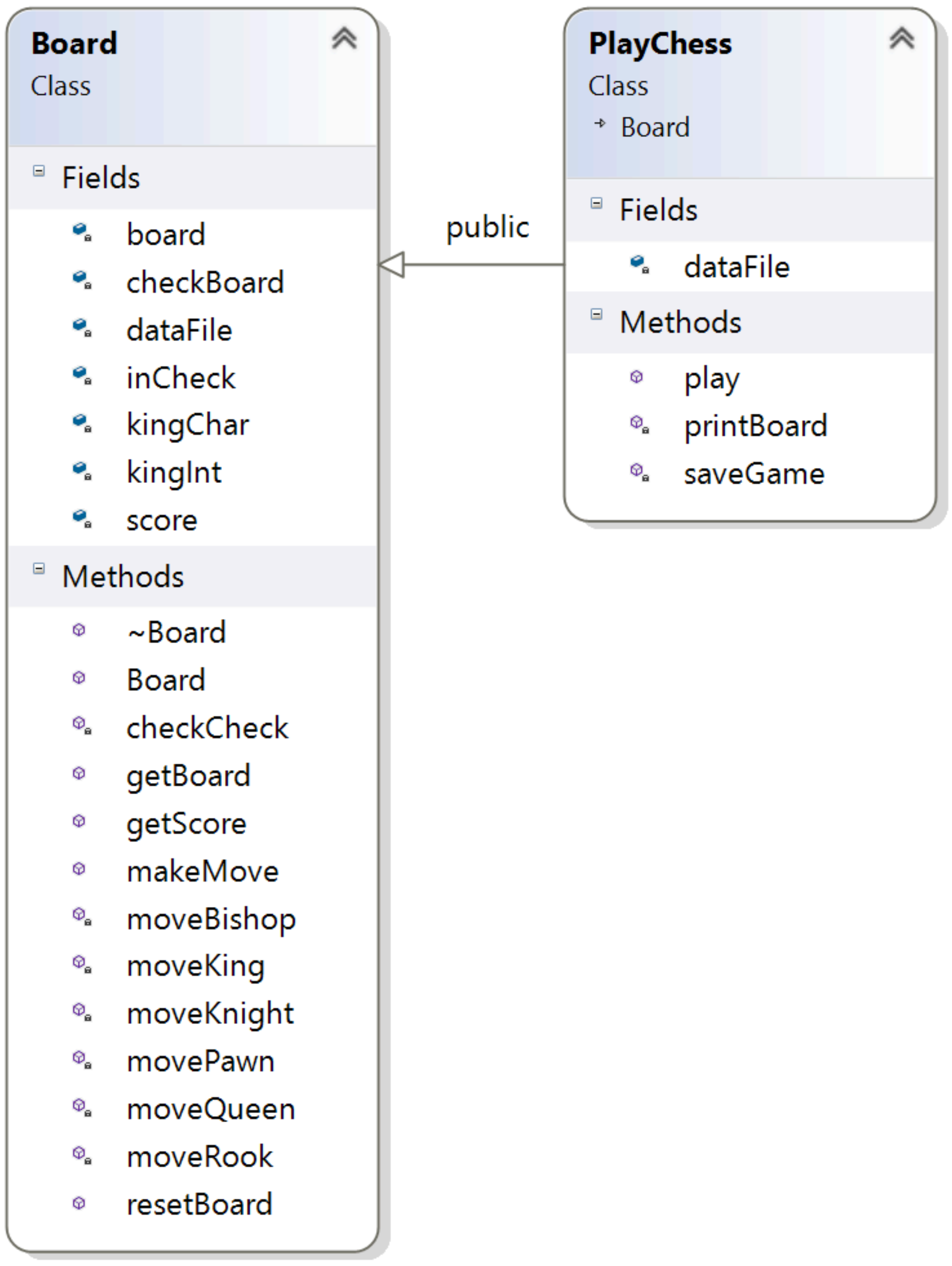
```
          Add a piece onto the board as determined by the for
loop from the Player 1's perspective and check if it is
successfully placed;
          if (if the piece is not placed successfully)
               End this instance of the for loop;

          Check how close Player 1 is to winning and subtract
that to the score at a depth determined by currPosition;


          if (if not currently at the maximum depth (currDepth
is compared with the depth variable))
               makeMove(currDepth + 1, currPosition);
               Call function to make move for the AI at the
current depth plus one, also pass on which move where the AI
made its first move at the zeroth depth (which is used to add
onto the score);

     }
}
```

SECTION C

PART 1

If I was to programme a different game, I would make Chess. The underlying structure would be similar to how I made Connect4. The programme would consist of two classes, one, called Board would store the basic functions for the game such as having a 2D char array to store the board, a function to move a piece which would in itself call another function depending on what piece is being moved, so if, for example, a knight was being moved, the makeMove function is first called, then when it recognises that it is a knight that is being moved, it would call moveKnight which does all of the checks necessary to see if it is a valid move. Every time a move is made as well, a check needs to be made to determine whether the king is in check which is done with checkCheck, if that is found to be true, the bool array inCheck would be set to true and that would make the relevant restrictions on the next move for the other player, it is an array so the makeMove function can determine which player is in check. resetBoard resets the current game and sets up a new one. kingChar and kingInt store the positions of each king to help the checkCheck function determine whether the king is in check.

The other class is PlayChess which actually prints the board onto the screen. It inherits Board. For the sake of practicality, the char values would be different for each piece on the board, but the printBoard function would print the same char value that corresponds to the piece, eg a knight would have k printed out to represent it, and a different colour would be used to distinguish from player 1 and 2. Print board can save the current game to a text file which is then read in by the constructor when the game starts if the user opts to. To make a move, the user would input a char and int value, like how a real chess board is referenced, with a-h on the horizontal axis and 1-8 on the vertical axis.


Board
Class
Fields
board
checkBoard
dataFile
inCheck
kingChar
kingInt
score
Methods
~Board
Board
checkCheck
getBoard
getScore
makeMove
moveBishop
moveKing
moveKnight
movePawn
moveQueen
moveRook
resetBoard
public
PlayChess
Class
Board
Fields
dataFile
Methods
play
printBoard
saveGame