# **Introduction**

Our main motive was to re-create the popular game called CHESS digitally using the core principles of Object Oriented Programming.

Team Members:

1. Mohammad Yehya Hayati ; K21-3309
2. Asad Noor Khan ; K21-4678
3. Deepak Chawla ; K21-4931

# **Background**

The libraries that were included for this project were:

* Iostream
* Windows.h
* Fstream
* Conio.h

Iostream was used for basic console operations.

Windows.h was used for the implementation of color on the console.

Fstream was used to perform filing operations.

Conio.h was used to hold the screen at specific times using the getch () function.

All logic was built/created on our own hence the length of the project exceeds 1700 lines and 67000+ characters.

# **Solution Design**

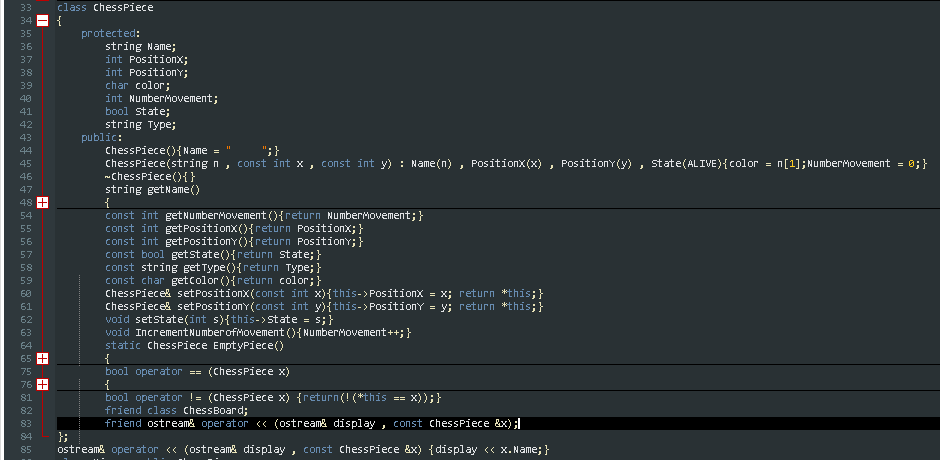
The base solution is fairly complicated as it involves some technical and logically advanced techniques that go beyond our level as a student. The first thing that happens is that we initialize all pieces of the board with their respective positions, names, types, etc. This is fairly important as without setting these pieces of information, the program would not know how to differentiate between two different pieces, especially when those two different pieces have different functionalities. Next, we need to show the grid/board that we will play on. This was achieved by a simple function which included loops and std::cout. Then we ask the user to input any piece of the relevant color and check whether the entered function is a valid piece using a global function to compare the entered string with an array of names of all the pieces. After that is done we need to ask it to move the piece to a specific position and check if the entered position is a valid position that the piece is allowed to move to. This was done by a series of functions that first checks the type of piece and then compares the values entered using a complex function that each piece has. If the position entered in invalid then the computer simply spits a message and alerts the user that an invalid/illegal move has been played and to try again. During this advanced algorithm, we will also check if the piece that is being moved will put their respective KING into the CHECK state. If so then we will again spit a message and say that the entered move was putting the KING in CHECK. Once a move is played then a filing operation is performed in which we write the number of movements of all the pieces. All of this is the same for the opposing player. To make it bit more professional we add colors and clear screen functions to make the illusion of it being similar to a GUI.

# **Implementation & Testing**

(In-depth Code analysis can be performed by viewing the code given with this report)

## **ChessPiece**

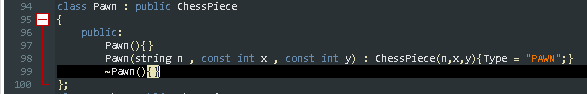
The base class for all the pieces, as they inherit from this class.



(It can be seen that ChessBoard is a friend of ChessPiece)

## **PAWN**

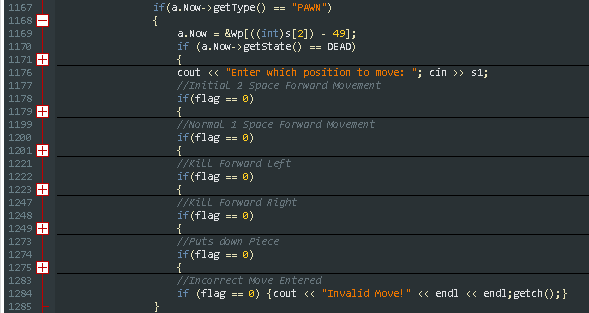
Pawns are the footman of the game. They inherit all the properties of the base Chess Piece as shown below:



They have three types of movement.

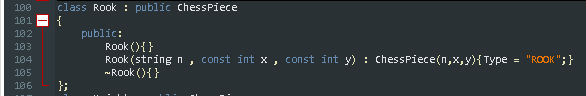
1. They can move forward by 2 spaces as their first move.
2. They can move forward by one when there is no piece in front of it.
3. They can move diagonally when there are opposing pieces on their corners.

This can be seen in the snippet below:

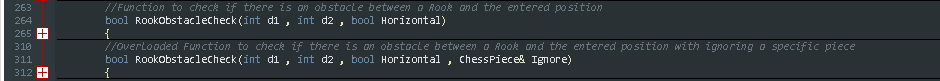


## **Rook**

Rooks are also pieces that inherit from the base Chess Piece class.

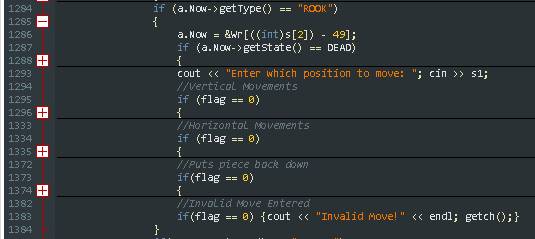


Rooks are in each corner of the board. They are a powerful piece that can move horizontally and vertically for any number of spaces. In order to do that there must be no piece in between the Rook and their desired destination. To do this we use two overloaded functions that can help us do this.



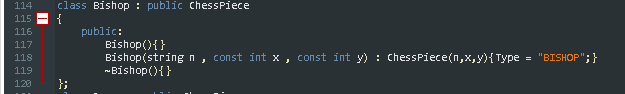
These functions are very important as they play a vital role in the conditions for CHECK.

As for its movements:

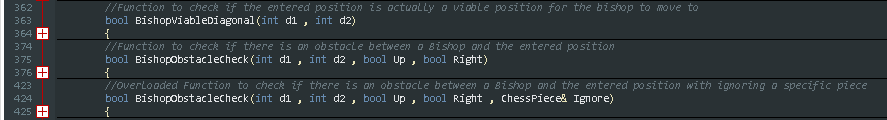


## **Bishop**

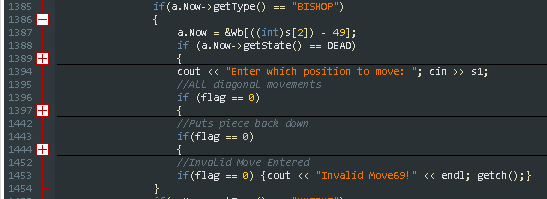
Bishops are also pieces that inherit from the base Chess Piece class.



Bishops reside in the third spot from each vertical side of the board. They are also a powerful piece as they move diagonally for any amount of space, given that there is no piece between it and its desired position. So similar functions were defined for Bishops.

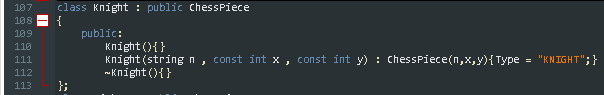


As for its movements:



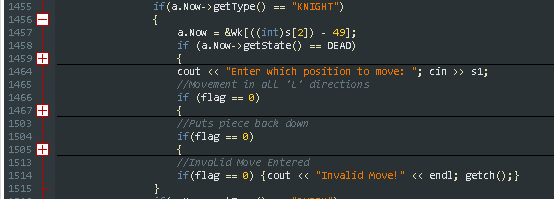
## **Knight**

The Knight also inherits from the base class as follows:



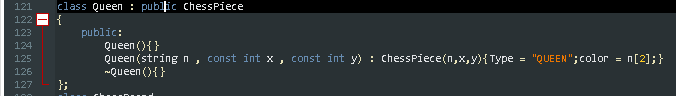
The knight is a comparatively more complex piece to define as its movement is quite eccentric. It moves in an ‘L’ pattern:





## **Queen**

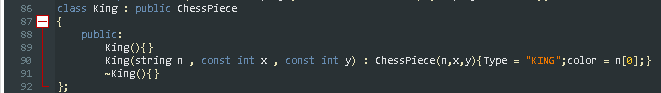
The queen is also an inherited class of Chess Piece.



The queen’s movements were not difficult to implement since they are the same as those of the Rook and the Bishop, so to follow the principle of code reusability, we will use the same functions of Rook and Bishop for the Queen.

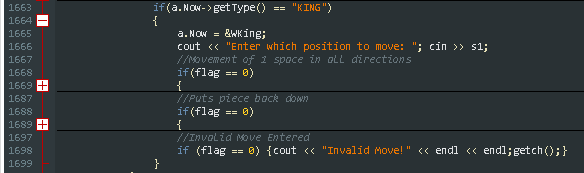
## **King**

The King is essentially the most important piece on the board, so the main function revolves around the King. It also inherits from the base class Chess Piece:



As for its movement:



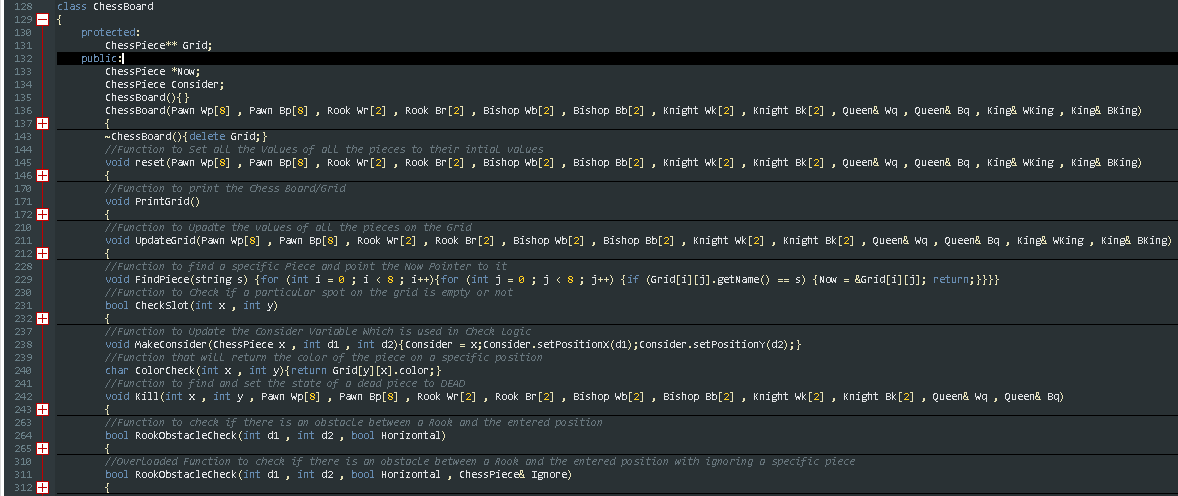


Whenever the King moves we have to make sure it doesn’t put itself into CHECK. So in its movement functions we have to check that if there is any enemy piece that puts that position into check and this is done by using two overloaded functions.



## **ChessBoard**

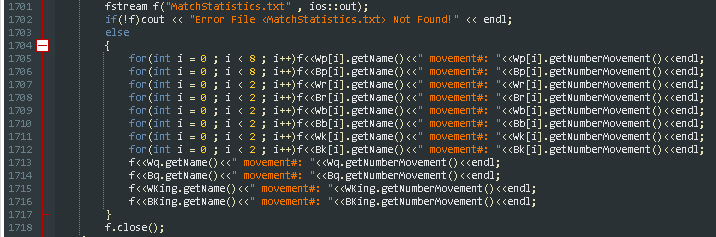
This function basically houses all the pieces on the board as it contains the 8X8 grid. It also contains a pointer of class ChessPiece so whenever we want to move a certain piece the pointer will point to that piece(Up casting) and perform all the actions from within the ChessBoard class. This is allowed as the ChessBoard class is actually a friend class of ChessPiece class. This can be seen in the ChessPiece class snapshot given above.



All the relevant functions are given in the ChessBoard class.

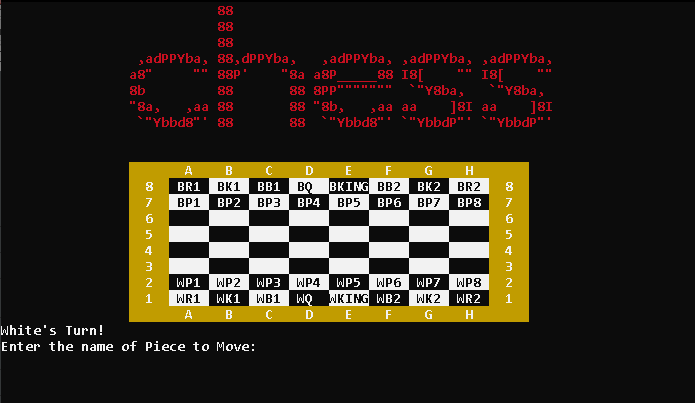
# **Filing**

There really isn’t much that you can do with filing in chess. However, for sake of completing requirements we added a feature which records the number of movements of each piece.



# **Final Result**

In the end, with the culmination of all these different aspects the final product looks like this:



# **Project Breakdown Structure**

We started our project on the 15th of April 2021. We decided to split the project into two phases:

1. Main Game
2. Debugging
3. Clean Up of Program

The main thing that was of difficulty was the main movement and the obstacle detection of each piece and the check condition. So the so called “Game Engine” was implemented at around 15th of May. The debugging stage and clean up went all the way to the 25th of May and that was when this project came to completion. As for the contribution of the participants; all of them equally contributed to this project.

# **Conclusion**

In conclusion we are extremely happy of the finished product and are sure that it will be evaluated and rated highly among the other projects in our batch.