CHESS

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.

```
33  class ChessPiece
34  {
35      protected:
36          string Name;
37          int PositionX;
38          int PositionY;
39          char color;
40          int NumberMovement;
41          bool State;
42          string Type;
43      public:
44          ChessPiece(){Name = "     ";}
45          ChessPiece(string n , const int x , const int y) : Name(n) , PositionX(x) , PositionY(y) , State(ALIVE){color = n[1];NumberMovement = 0;}
46          ~ChessPiece(){}
47          string getName()
48          {
54          const int getNumberMovement(){return NumberMovement;}
55          const int getPositionX(){return PositionX;}
56          const int getPositionY(){return PositionY;}
57          const bool getState(){return State;}
58          const string getType(){return Type;}
59          const char getColor(){return color;}
60          ChessPiece& setPositionX(const int x){this->PositionX = x; return *this;}
61          ChessPiece& setPositionY(const int y){this->PositionY = y; return *this;}
62          void setState(int s){this->State = s;}
63          void IncrementNumberofMovement(){NumberMovement++;}
64          static ChessPiece EmptyPiece()
65          {
75          bool operator == (ChessPiece x)
76          {
81          bool operator != (ChessPiece x) {return(!(*this == x));}
82          friend class ChessBoard;
83          friend ostream& operator << (ostream& display , const ChessPiece &x);
84  };
85  ostream& operator << (ostream& display , const ChessPiece &x) {display << x.Name;}
```

(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:

```
94  class Pawn : public ChessPiece
95  {
96      public:
97          Pawn(){}
98          Pawn(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "PAWN";}
99          ~Pawn(){}
100 };
```

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:

```
1167                if(a.Now->getType() == "PAWN")
1168                {
1169                    a.Now = &Wp[((int)s[2]) - 49];
1170                    if (a.Now->getState() == DEAD)
1171                    {
1176                    cout << "Enter which position to move: "; cin >> s1;
1177                    //Initial 2 Space Forward Movement
1178                    if(flag == 0)
1179                    {
1199                    //Normal 1 Space Forward Movement
1200                    if(flag == 0)
1201                    {
1221                    //Kill Forward Left
1222                    if(flag == 0)
1223                    {
1247                    //Kill Forward Right
1248                    if(flag == 0)
1249                    {
1273                    //Puts down Piece
1274                    if(flag == 0)
1275                    {
1283                    //Incorrect Move Entered
1284                    if (flag == 0) {cout << "Invalid Move!" << endl << endl;getch();}
1285                }
```

Rook

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

```
100   class Rook : public ChessPiece
101   {
102       public:
103           Rook(){}
104           Rook(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "ROOK";}
105           ~Rook(){}
106   };
```

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.

```
263       //Function to check if there is an obstacle between a Rook and the entered position
264       bool RookObstacleCheck(int d1 , int d2 , bool Horizontal)
265       {
310       //Overloaded Function to check if there is an obstacle between a Rook and the entered position with ignoring a specific piece
311       bool RookObstacleCheck(int d1 , int d2 , bool Horizontal , ChessPiece& Ignore)
312       {
```

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

As for its movements:

```
1284                if (a.Now->getType() == "ROOK")
1285                {
1286                    a.Now = &Wr[((int)s[2]) - 49];
1287                    if (a.Now->getState() == DEAD)
1288                    {
1293                    cout << "Enter which position to move: "; cin >> s1;
1294                    //Vertical Movements
1295                    if (flag == 0)
1296                    {
1333                    //Horizontal Movements
1334                    if (flag == 0)
1335                    {
1372                    //Puts piece back down
1373                    if(flag == 0)
1374                    {
1382                    //Invalid Move Entered
1383                    if(flag == 0) {cout << "Invalid Move!" << endl; getch();}
1384                }
```

Bishop

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

```
114  class Bishop : public ChessPiece
115  {
116      public:
117          Bishop(){}
118          Bishop(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "BISHOP";}
119          ~Bishop(){}
120  };
```

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.

```
362      //Function to check if the entered position is actually a viable position for the bishop to move to
363      bool BishopViableDiagonal(int d1 , int d2)
364      {
374      //Function to check if there is an obstacle between a Bishop and the entered position
375      bool BishopObstacleCheck(int d1 , int d2 , bool Up , bool Right)
376      {
423      //Overloaded Function to check if there is an obstacle between a Bishop and the entered position with ignoring a specific piece
424      bool BishopObstacleCheck(int d1 , int d2 , bool Up , bool Right , ChessPiece& Ignore)
425      {
```

As for its movements:

```
1385                if(a.Now->getType() == "BISHOP")
1386                {
1387                    a.Now = &Wb[((int)s[2]) - 49];
1388                    if (a.Now->getState() == DEAD)
1389                    {
1394                    cout << "Enter which position to move: "; cin >> s1;
1395                    //ALL diagonal movements
1396                    if (flag == 0)
1397                    {
1442                    //Puts piece back down
1443                    if(flag == 0)
1444                    {
1452                    //Invalid Move Entered
1453                    if(flag == 0) {cout << "Invalid Move69!" << endl; getch();}
1454                }
```

Knight

The Knight also inherits from the base class as follows:

```
107  class Knight : public ChessPiece
108  {
109      public:
110          Knight(){}
111          Knight(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "KNIGHT";}
112          ~Knight(){}
113  };
```

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

```
476          //Function to check if the entered position is actually a viable position for the knight to move to
477          bool KnightViableSpot(int d1 , int d2)
478          {
```

```
1455                 if(a.Now->getType() == "KNIGHT")
1456                 {
1457                     a.Now = &wk[((int)s[2]) - 49];
1458                     if (a.Now->getState() == DEAD)
1459                     {
1464                     cout << "Enter which position to move: "; cin >> s1;
1465                     //Movement in all 'L' directions
1466                     if (flag == 0)
1467                     {
1503                     //Puts piece back down
1504                     if(flag == 0)
1505                     {
1513                     //Invalid Move Entered
1514                     if(flag == 0) {cout << "Invalid Move!" << endl; getch();}
1515                 }
```

Queen

The queen is also an inherited class of Chess Piece.

```
121  class Queen : public ChessPiece
122  {
123      public:
124          Queen(){}
125          Queen(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "QUEEN";color = n[2];}
126          ~Queen(){}
127  };
```

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:

```
86  class King : public ChessPiece
87  {
88      public:
89          King(){}
90          King(string n , const int x , const int y) : ChessPiece(n,x,y){Type = "KING";color = n[0];}
91          ~King(){}
92  };
```

As for its movement:

```
581         //Function to check if the entered position is actually a viable position for the king to move to
582         bool KingViableSpot(int d1 , int d2 , King& Return)
583         {
```

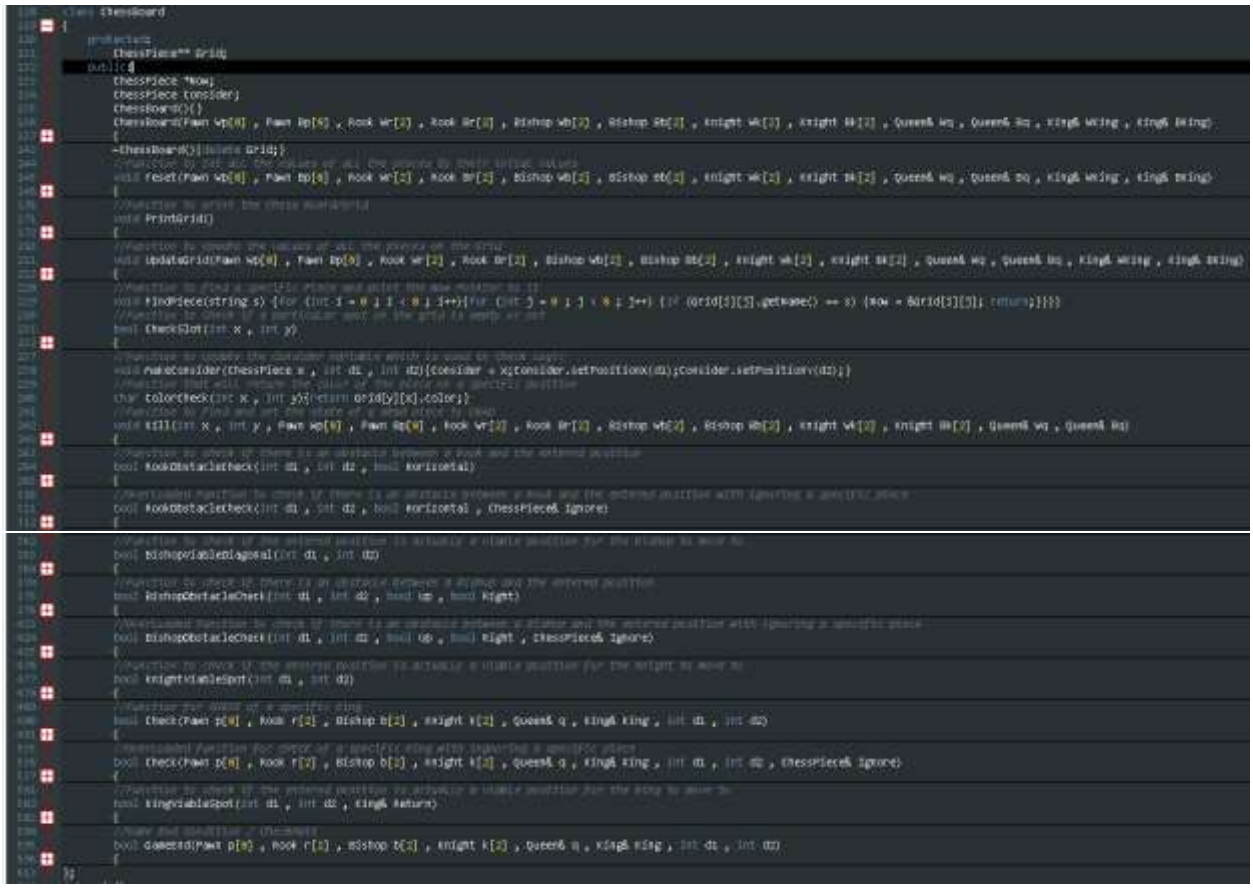
```
1663                if(a.Now->getType() == "KING")
1664                {
1665                    a.Now = &WKing;
1666                    cout << "Enter which position to move: "; cin >> s1;
1667                    //Movement of 1 space in all directions
1668                    if(flag == 0)
1669                    {
1687                    //Puts piece back down
1688                    if(flag == 0)
1689                    {
1697                    //Invalid Move Entered
1698                    if (flag == 0) {cout << "Invalid Move!" << endl << endl;getch();}
1699                }
```

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.

```
489         //Function for CHECK of a specific King
490         bool Check(Pawn p[8] , Rook r[2] , Bishop b[2] , Knight k[2] , Queen& q , King& King , int d1 , int d2)
491         {
535         //Overloaded function for CHECK of a specific King with ingnoring a specific piece
536         bool Check(Pawn p[8] , Rook r[2] , Bishop b[2] , Knight k[2] , Queen& q , King& King , int d1 , int d2 , ChessPiece& Ignore)
537         {
```

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.

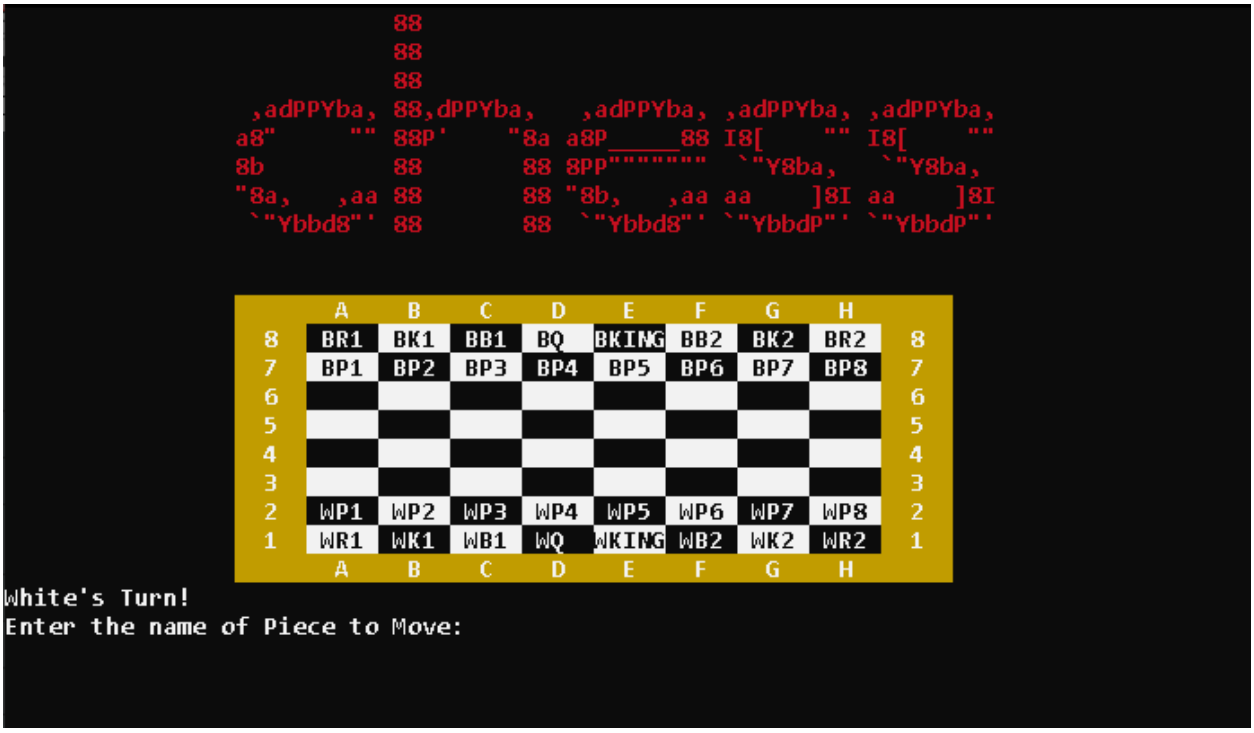
Filing

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.

```
1701        fstream f("MatchStatistics.txt" , ios::out);
1702        if(!f)cout << "Error File <MatchStatistics.txt> Not Found!" << endl;
1703        else
1704        {
1705            for(int i = 0 ; i < 8 ; i++)f<<Wp[i].getName()<<" movement#: "<<Wp[i].getNumberMovement()<<endl;
1706            for(int i = 0 ; i < 8 ; i++)f<<Bp[i].getName()<<" movement#: "<<Bp[i].getNumberMovement()<<endl;
1707            for(int i = 0 ; i < 2 ; i++)f<<Wr[i].getName()<<" movement#: "<<Wr[i].getNumberMovement()<<endl;
1708            for(int i = 0 ; i < 2 ; i++)f<<Br[i].getName()<<" movement#: "<<Br[i].getNumberMovement()<<endl;
1709            for(int i = 0 ; i < 2 ; i++)f<<Wb[i].getName()<<" movement#: "<<Wb[i].getNumberMovement()<<endl;
1710            for(int i = 0 ; i < 2 ; i++)f<<Bb[i].getName()<<" movement#: "<<Bb[i].getNumberMovement()<<endl;
1711            for(int i = 0 ; i < 2 ; i++)f<<Wk[i].getName()<<" movement#: "<<Wk[i].getNumberMovement()<<endl;
1712            for(int i = 0 ; i < 2 ; i++)f<<Bk[i].getName()<<" movement#: "<<Bk[i].getNumberMovement()<<endl;
1713            f<<Wq.getName()<<" movement#: "<<Wq.getNumberMovement()<<endl;
1714            f<<Bq.getName()<<" movement#: "<<Bq.getNumberMovement()<<endl;
1715            f<<WKing.getName()<<" movement#: "<<WKing.getNumberMovement()<<endl;
1716            f<<BKing.getName()<<" movement#: "<<BKing.getNumberMovement()<<endl;
1717        }
1718        f.close();
```

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.