OOPS Project
Two-Player Chess

Prasanna Kumar M

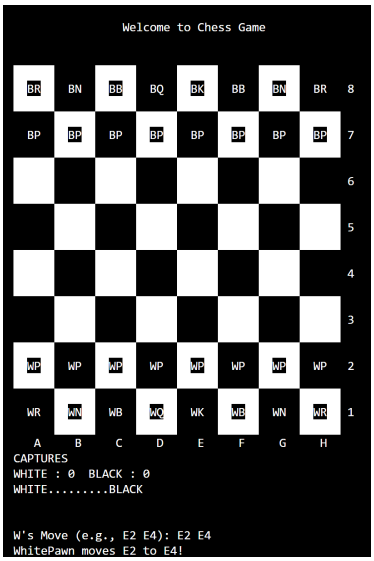
January 12, 2025

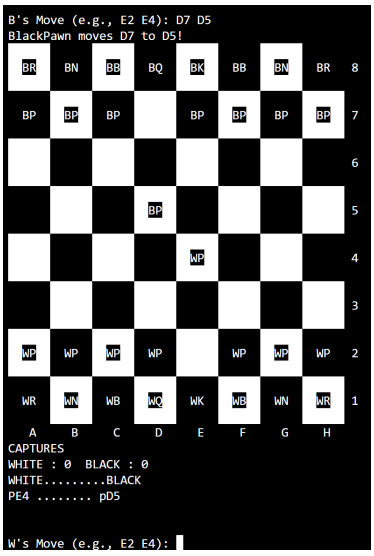
In this project, we have implemented a two-player chess game using the principles of Object-Oriented Programming (OOP). The game models chess pieces, game rules, and player interactions using OOP concepts such as ***abstraction, inheritance, polymorphism, operator overloading***, and ***virtual functions***.

These concepts have helped us structure the code in a modular and maintainable manner, ensuring flexibility and scalability for future modifications.

Game Logic: Move Validation and Execution

The core gameplay logic for our chess game is encapsulated in the `makeMove()` function. This function manages the sequence of events that occur during a player's turn, including move validation, capturing, and special checks for scenarios like putting oneself in check. Here's an overview of the main steps within the `makeMove()` function:





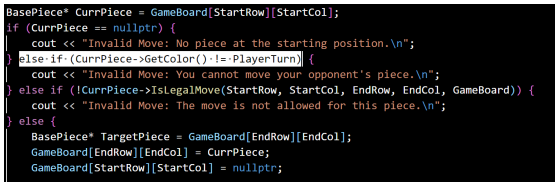
Move Validation and Turn Management

At the beginning of each turn, the game prompts the current player to enter their desired move using standard chess notation (e.g., `E2 E4`). The `makeMove()` function then interprets this input and validates it according to the following rules:

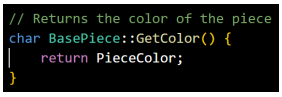
- **Boundary Checks:** Ensure that the coordinates entered are within the boundaries of the 8x8 chessboard.

Figure 1: Boundary Check

- **Piece Ownership:** Check that the piece selected belongs to the current player. The player cannot move the opponent's pieces.



(a) Ownership Check



(b) GetColour Function

Figure 2: Move Validation Checks in Chess Game

- **Legal Moves for Each Piece:** Based on the type of piece selected (e.g., Pawn, Knight, Bishop), the function calls an `IsLegalMove()` function specific to that piece. This function verifies whether the piece's movement is allowed within the rules of chess for that piece type.

Simulating and Checking for Check

To prevent moves that would leave the player's king in check, the function employs a simulation step:

- Before finalizing any move, it temporarily applies the move on the board.
- Using the `IsInCheck()` function, the game verifies if the player's king would be in check as a result of the move.

```cpp
// Checks if the move is legal
bool BasePiece::IsLegalMove(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]) {
    BasePiece* Dest = GameBoard[DestRow][DestCol];
    if ((Dest == nullptr) || (PieceColor != Dest->GetColor())) {
        return AreSquaresLegal(SrcRow, SrcCol, DestRow, DestCol, GameBoard);
    }
    return false;
}
```

Figure 3: Legal Move Check

- If the move places the player's king in check, it is deemed invalid, and the board reverts to its previous state.

```cpp
if (GameBoard[row][col]->IsLegalMove(row, col, toMoveRow, toMoveCol, GameBoard))
{
    //momentarily moving a piece
    BasePiece* Temp = GameBoard[toMoveRow][toMoveCol];
    GameBoard[toMoveRow][toMoveCol] = GameBoard[row][col];
    GameBoard[row][col] = nullptr;

    bool canMove = !IsInCheck(PieceColor);

    //revert to old stage
    GameBoard[row][col] = GameBoard[toMoveRow][toMoveCol];
    GameBoard[toMoveRow][toMoveCol] = Temp;
    if (canMove) {  //move exist stop
        return true;
    }
}
```

Figure 4: Ownership Check

```cpp
bool game::IsInCheck(char PieceColor) {

    int KingRow, KingCol;
    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 8; col++) {
            if (GameBoard[row][col] != nullptr && GameBoard[row][col]->GetColor() == PieceColor && GameBoard[row][col]->GetPiece() == 'K') {
                KingRow = row;
                KingCol = col;
                // cout<<row<<col;
            }
        }
    }
    int row;
    int col;
    for (row = 0; row < 8; row++) {
        for (col = 0; col < 8; col++) {
            if (GameBoard[row][col] != nullptr && GameBoard[row][col]->GetColor() != PieceColor && GameBoard[row][col]->IsLegalMove(row, col, KingRow, KingCol, GameBoa
                cout<<row<<col;
                return true;
            }
        }
    }
    return false;
}
```

Figure 5: Is Check Function

Capturing Opponent Pieces

If the move is valid and a capture is involved (i.e., the target square is occupied by an opponent piece), the captured piece is removed from the board, and the appropriate counters for the capturing player are incremented. The capturing action is also logged in the move history, enabling a clear record of moves for both players.

```
if(TargetPiece != nullptr){
    (*this)++;  //piece -strikes tracker
}
++(*this);  //moves tracker

delete TargetPiece;
ValidMove = true;
```

Figure 6: Capturing pieces

```
void game::operator++(){
    if(PlayerTurn=='W'){
        w_moves++;
    }
    else{
        b_moves++;
    }
}

void game::operator++(int){
    if(PlayerTurn=='W'){
        w_captures++;
    }
    else{
        b_captures++;
    }
}
```

Figure 7: Capturing/moves action incremented

Move Logging and Notation

Each move is logged in two formats:

- **Coordinate Logging:** The exact coordinates of the start and end squares are stored, enabling features like undo and redo of moves.
- **Chess Notation Logging:** The move is recorded in standard chess notation, facilitating clear display of moves in a traditional format. For instance, a move by a white pawn from `E2` to `E4` would be logged as `Pe4`.

Overall, the `makeMove()` function integrates key chess mechanics while enforcing the rules to ensure valid and fair gameplay. This function serves as a fundamental part of the game, regulating each player's turn, validating their moves, and updating the game state accordingly.

```
bool ValidMove = false;

while (!ValidMove) {
    // Display logs of previous moves in notation format
    Print();

    // cout<<"NO OF MOVES PLAYED "<<endl;
    // cout<<"WHITE : "<<w_moves<<"  BLACK : "<<b_moves<<endl;

    cout<<"CAPTURES"<<endl;
    cout<<"WHITE : "<<w_captures<<"  BLACK : "<<b_captures<<endl;

    cout << "WHITE.........BLACK\n";
    for (int i = 0; i < whiteNotation.size(); ++i) {
        cout << whiteNotation[i] << " ........ ";
        if (i < blackNotation.size()) {
            cout << blackNotation[i] << "\n";
        } else {
            cout << "\n";
        }
    }
    cout << endl;
    cout << endl;
```

Figure 8: Logged Moves for User

```
if (PlayerTurn == 'W') {
    whiteMoves.push_back(moveCoords);
    whiteNotation.push_back(moveNotation);
} else {
    blackMoves.push_back(moveCoords);
    blackNotation.push_back(moveNotation);
}
```

Figure 9: Moves logger with notation of Piece

Special moves implementation

1.Castling

First we need to check some conditions in order for castling to be valid:

1. Castling must be the first move of the king and the rook.
2. The king should not be in check in its current position and in the position on which it will be after castling.
3. There should no pieces between the rook and the king.

Once all these conditions are met, castling will be done in our game.These conditions will be checked by `IsInCheck()`,`CanMove()`,`would_be_in_check()` functions.

```
if (CurrPiece->GetPiece() == 'K' && abs(StartCol - EndCol) == 2) {
        if (PlayerTurn == 'W' && !whiteKingMoved) {
            if (EndCol == 6 && !whiteRookKingsideMoved && GameBoard[0][7] != nullptr && GameBoard[0][7]->GetPiece() == 'R') {
                // Kingside castling for white

                if (GameBoard[0][5] == nullptr && GameBoard[0][6] == nullptr) {
                    if (!IsInCheck(PlayerTurn) && !would_be_in_check(PlayerTurn, 0, 5) && !would_be_in_check(PlayerTurn, 0, 6)) {
                        GameBoard[0][6] = GameBoard[0][4];
                        GameBoard[0][4] = nullptr;
                        GameBoard[0][5] = GameBoard[0][7];
                        GameBoard[0][7] = nullptr;
                        whiteKingMoved = true;
                        whiteRookKingsideMoved = true;
                        ValidMove = true;
                        white_castling_possible =false;
                        whiteMoves.push_back({{StartRow, StartCol}, {EndRow, EndCol}});
                        whiteNotation.push_back("O-O");
                        ++(*this);
                        return;
                    }
                }
```

Figure 10: Kingside castling for white

```
else if (EndCol == 2 && !whiteRookQueensideMoved && GameBoard[0][0] != nullptr && GameBoard[0][0]->GetPiece() == 'R') {
  // Queenside castling for white
  if (GameBoard[0][1] == nullptr && GameBoard[0][2] == nullptr && GameBoard[0][3] == nullptr) {
      if (!IsInCheck(PlayerTurn) && !would_be_in_check(PlayerTurn, 0, 3) && !would_be_in_check(PlayerTurn, 0, 2)) {
          GameBoard[0][2] = GameBoard[0][4];
          GameBoard[0][4] = nullptr;
          GameBoard[0][3] = GameBoard[0][0];
          GameBoard[0][0] = nullptr;
          whiteKingMoved = true;
          whiteRookQueensideMoved = true;
          ValidMove = true;
          whiteMoves.push_back({{StartRow, StartCol}, {EndRow, EndCol}});
          whiteNotation.push_back("O-O-O");
          white_castling_possible = false;
          ++(*this);
          return;
      }
  }
```

Figure 11: Queenside castling for white

```
else if (PlayerTurn == 'B' && !blackKingMoved) {
  if (EndCol == 6 && !blackRookKingsideMoved && GameBoard[7][7] != nullptr && GameBoard[7][7]->GetPiece() == 'R') {
      // Kingside castling for black
      if (GameBoard[7][5] == nullptr && GameBoard[7][6] == nullptr) {
          if (!IsInCheck(PlayerTurn) && !would_be_in_check(PlayerTurn, 7, 5) && !would_be_in_check(PlayerTurn, 7, 6)) {

              GameBoard[7][6] = GameBoard[7][4];
              GameBoard[7][4] = nullptr;
              GameBoard[7][5] = GameBoard[7][7];
              GameBoard[7][7] = nullptr;
              blackKingMoved = true;
              blackRookKingsideMoved = true;
              ValidMove = true;
              blackNotation.push_back("O-O");
              blackMoves.push_back({{StartRow, StartCol}, {EndRow, EndCol}});
              black_castling_possible =false;
              ++(*this);
              return;
          }
      }
```

Figure 12: Kingside castling for black

```
            } else if (EndCol == 2 && !blackRookQueensideMoved && GameBoard[7][0] != nullptr && GameBoard[7][0]->GetPiece() == 'R') {
                // Queenside castling for black
                if (GameBoard[7][1] == nullptr && GameBoard[7][2] == nullptr && GameBoard[7][3] == nullptr) {
                    if (!IsInCheck(PlayerTurn) && !would_be_in_check(PlayerTurn, 7, 3) && !would_be_in_check(PlayerTurn, 7, 2)) {

                        GameBoard[7][2] = GameBoard[7][4];
                        GameBoard[7][4] = nullptr;
                        GameBoard[7][3] = GameBoard[7][0];
                        GameBoard[7][0] = nullptr;
                        blackKingMoved = true;
                        blackRookQueensideMoved = true;
                        ValidMove = true;
                        blackNotation.push_back("O-O-O");
                        black_castling_possible =false;
                        blackMoves.push_back({{StartRow, StartCol}, {EndRow, EndCol}});
                        ++(*this);
                        return;
                    }
                }
            }
        }
        continue;
```

Figure 13: Queenside castling for black

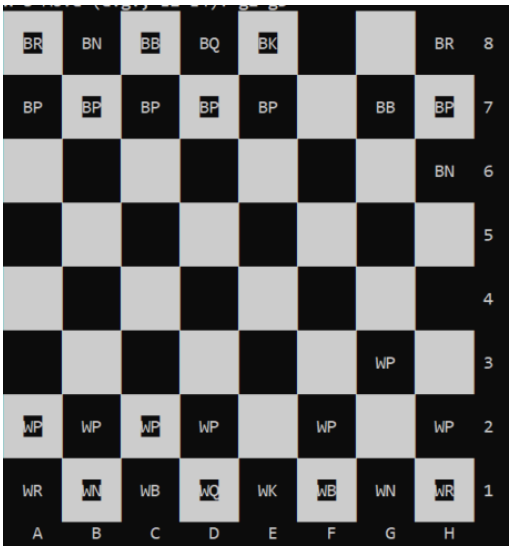


Figure 14: Castling process (Figure 1)

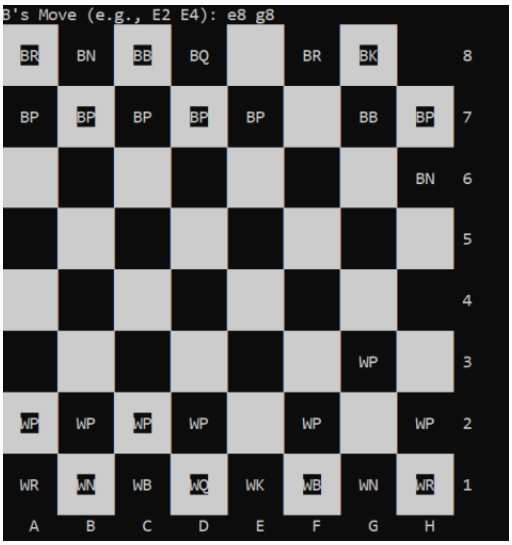


Figure 15: Castling process (Figure 2)

2.En Passant

Conditions for En Passant to be valid:

1. The opposing pawn must have just advanced two squares forward from its starting position in a single move.
2. The capturing pawn must be positioned on an adjacent column and must capture on the move immediately following the opponent's two-square pawn advance.
3. The capturing pawn moves diagonally to the square that the opposing pawn has passed over, as if it had advanced only one square.

```
if (CurrPiece!=nullptr&&  enPassantPossible && dynamic_cast<PawnPiece*>(CurrPiece)) {
        if (StartCol != EndCol && GameBoard[EndRow][EndCol] == nullptr && enPassantTarget == make_pair(EndRow, EndCol)) {
            int capturedRow = (CurrPiece->GetColor() == 'W') ? EndRow - 1 : EndRow + 1;
            BasePiece* temp = GameBoard[capturedRow][EndCol] ;
            BasePiece*temp2 = GameBoard[StartRow][StartCol];

            GameBoard[EndRow][EndCol] = GameBoard[StartRow][StartCol];
            GameBoard[StartRow][StartCol] = nullptr;
            GameBoard[capturedRow][EndCol] = nullptr;
                if (!IsInCheck(PlayerTurn)){
                    (*this)++;  // Update capture tracker
                    ++(*this);
                    delete GameBoard[capturedRow][EndCol] ;

                    string moveNotation = "Px" + EndMove;
                    if (PlayerTurn == 'W') {
                            whiteMoves.push_back({{StartRow, StartCol}, {-1*EndRow, EndCol}});
                            whiteNotation.push_back(moveNotation);
                    } else {
                            blackMoves.push_back({{StartRow, StartCol}, {EndRow, -1*EndCol}});
                            blackNotation.push_back(moveNotation);
                            return ;
                    }
                    else{
                         GameBoard[EndRow][EndCol] = nullptr;
                        GameBoard[StartRow][StartCol] = temp2;
                        GameBoard[capturedRow][EndCol] = temp;
                        cout<<" Moving current piece puts you in check.\n";
                    }
            }
        }
```

Figure 16: En passant implementation

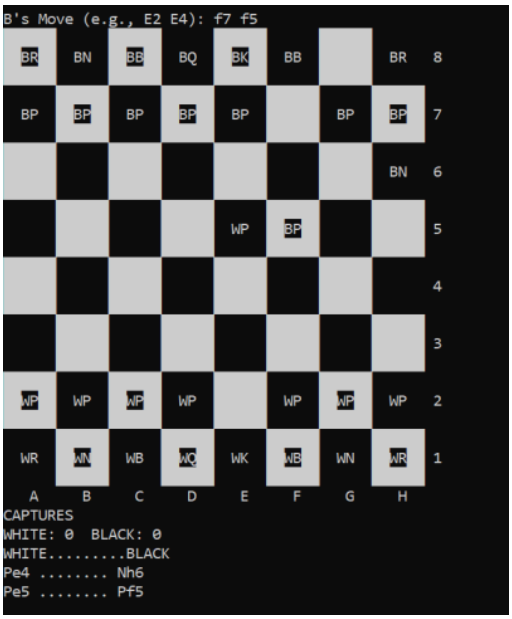


Figure 17: En Passant process (Figure 1)

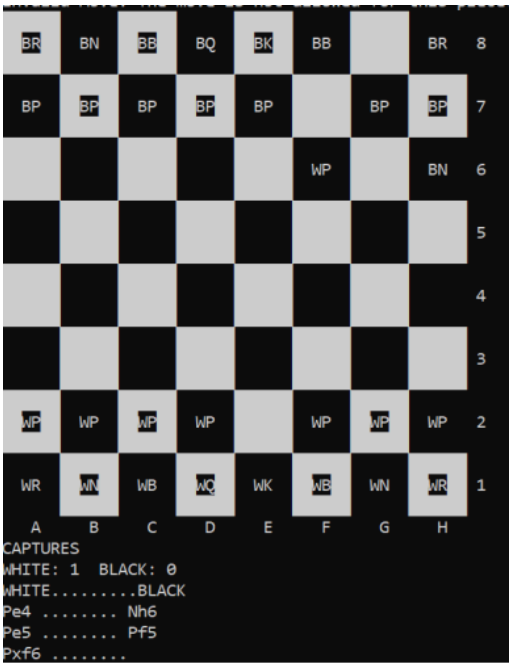


Figure 18: En Passant process (Figure 2)

OOP Concepts in the Project

Abstraction

Abstraction is the process of hiding implementation details and exposing only essential functionalities to the user. In this project, we abstracted the behavior of chess pieces using a base class `BasePiece`. The `BasePiece` class provides common methods, such as `GetPiece()` and `GetPieceName()`, which are overridden in derived classes for specific piece behavior.

basepiece.cpp > ...

```cpp
BasePiece::BasePiece(char PieceColor) : PieceColor(PieceColor) {}
BasePiece::~BasePiece() {}

// Returns the color of the piece
char BasePiece::GetColor() {
    return PieceColor;
}
```

Figure 19: Base Class

king.cpp > ...

```cpp
KingPiece::KingPiece(char PieceColor) : BasePiece(PieceColor) {}

KingPiece::~KingPiece() {}

char KingPiece::GetPiece() {
    return 'K';
}
string KingPiece::GetPieceName() {
    return "King";
}
```

(a) King Class

queen.cpp > ...

```cpp
#include"includes.h"

// ****************************************   QueenPiece   ***

QueenPiece::QueenPiece(char PieceColor) : BasePiece(PieceColor) {}

QueenPiece::~QueenPiece() {}

char QueenPiece::GetPiece() {
    return 'Q';
}
string QueenPiece::GetPieceName() {
    return "Queen";
}
```

(b) Queen class

Figure 20: Piece classes

chess.h > ...

```cpp
class QueenPiece : public BasePiece {
public:
    QueenPiece(char PieceColor);
    ~QueenPiece();
private:
    virtual char GetPiece();
    virtual string GetPieceName();
    bool AreSquaresLegal(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]);
};


// ************************************     King     ********************************************

class KingPiece : public BasePiece {
public:
    KingPiece(char PieceColor);
    ~KingPiece();
private:
    virtual char GetPiece();
    virtual string GetPieceName();
    bool AreSquaresLegal(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]);
};
```

Figure 21: Abstraction And Inheritance

Inheritance

Inheritance allows each chess piece, such as `PawnPiece`, `RookPiece`, and `KnightPiece`, to inherit from the `Piece` base class. This design enables code reuse, as each derived class can share common attributes and methods, while also defining piece-specific behaviors.

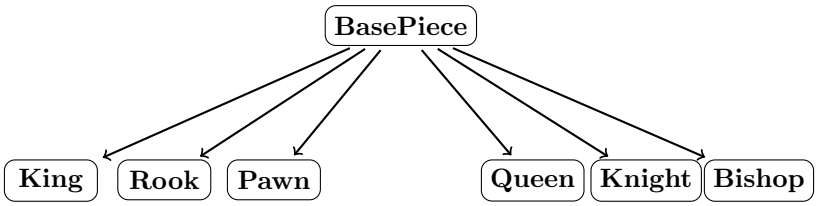


Figure 22: Class Hierarchy and Multilevel Inheritance Diagram for Chess Game

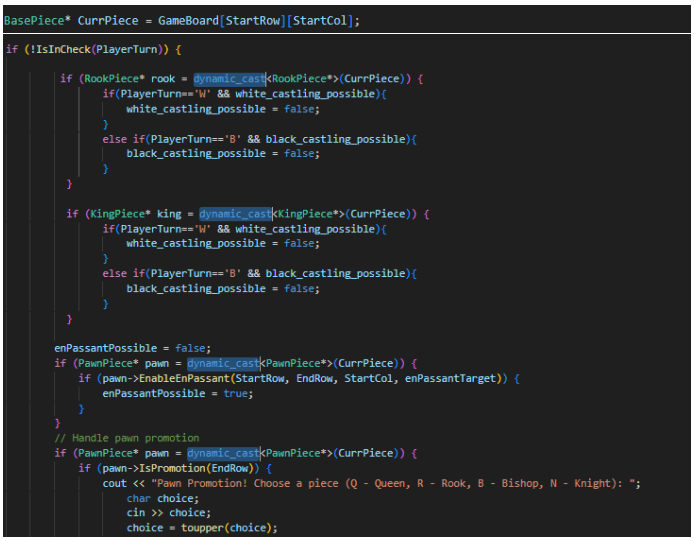
Polymorphism

Polymorphism allows different classes to be treated as instances of the same class through a common interface. In our chess game, polymorphism is used to handle various chess pieces uniformly.
For instance, in `PromotePawn()` function the base class (BasePiece) pointer is assigned to derived class objects (such as QueenPiece, RookPiece etc.)

```
int game::PromotePawn(int row, int col ,char choice) {

    BasePiece* newPiece = nullptr;
    char color = GameBoard[row][col]->GetColor();

    switch (choice) {
        case 'Q':
            newPiece = new QueenPiece(color);
            return 1;
            break;
        case 'R':
            newPiece = new RookPiece(color);
            return 2;
            break;
        case 'B':
            newPiece = new BishopPiece(color);
            return 3;
            break;
        case 'N':
            newPiece = new KnightPiece(color);
            return 4;
            break;
        default:
            cout << "Invalid choice, defaulting to Queen." << endl;
            newPiece = new QueenPiece(color);
         return 1;
    }

    delete GameBoard[row][col]; // Remove pawn from the board
    GameBoard[row][col] = newPiece; // Replace pawn with the new piece
    cout << "Pawn promoted to " << newPiece->GetPieceName() << "!" << endl;

}
```

Figure 23: Up Casting in PromotePawn()

Also, in the `makeMove()` function, upcasting is performed first. The `dynamic_cast` operator then checks the runtime type of the object to ensure it matches the target derived type. If successful, the base class pointer is safely downcasted to access derived class-specific functionality. If the cast fails, `dynamic_cast` returns a nullptr.



```
BasePiece* CurrPiece = GameBoard[StartRow][StartCol];

if (!IsInCheck(PlayerTurn)) {

        if (RookPiece* rook = dynamic_cast<RookPiece*>(CurrPiece)) {
            if(PlayerTurn=='W' && white_castling_possible){
                white_castling_possible = false;
            }
            else if(PlayerTurn=='B' && black_castling_possible){
                black_castling_possible = false;
            }
        }

        if (KingPiece* king = dynamic_cast<KingPiece*>(CurrPiece)) {
            if(PlayerTurn=='W' && white_castling_possible){
                white_castling_possible = false;
            }
            else if(PlayerTurn=='B' && black_castling_possible){
                black_castling_possible = false;
            }
        }

       enPassantPossible = false;
       if (PawnPiece* pawn = dynamic_cast<PawnPiece*>(CurrPiece)) {
           if (pawn->EnableEnPassant(StartRow, EndRow, StartCol, enPassantTarget)) {
               enPassantPossible = true;
           }
       }
       // Handle pawn promotion
       if (PawnPiece* pawn = dynamic_cast<PawnPiece*>(CurrPiece)) {
           if (pawn->IsPromotion(EndRow)) {
               cout << "Pawn Promotion! Choose a piece (Q - Queen, R - Rook, B - Bishop, N - Knight): ";
                   char choice;
                   cin >> choice;
                   choice = toupper(choice);
```

Figure 24: Up and Downcasting in makeMove()

Operator Overloading

Operator overloading is a feature in C++ that allows us to redefine the behavior of operators for user-defined types. In our project, we have overloaded the ++ operator to handle the counting of moves and captures.

```
void game::operator++(){
    if(PlayerTurn=='W'){
        w_moves++;
    }
    else{
        b_moves++;
    }
}

void game::operator++(int){
    if(PlayerTurn=='W'){
        w_captures++;
    }
    else{
        b_captures++;
    }
}
```

Figure 25: Operator Overloading in Incrementing Moves and Captures

Virtual Functions

Virtual functions are used in the base class to allow derived classes to implement their own version of the function. In our project, many functions, such as `IsLegalMove()` and `GetPieceName()`, are declared as virtual in the `BasePiece` class.

```
class BasePiece {
public:
    BasePiece(char PieceColor);
    virtual ~BasePiece();

    virtual char GetPiece() = 0;
    virtual string GetPieceName() =0;
    char GetColor();
    bool IsLegalMove(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]);
private:
    virtual bool AreSquaresLegal(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]) = 0;
    char PieceColor;
};
```

Figure 26: Virtual function

Exception Handling

Exception handling is implemented to manage scenarios where chess rules are violated or when the specified position falls outside the boundaries of the chessboard.

```
bool BasePiece::IsLegalMove(int SrcRow, int SrcCol, int DestRow, int DestCol, BasePiece* GameBoard[8][8]) {
    try {
        BasePiece* Dest = GameBoard[DestRow][DestCol];
        if ((Dest == nullptr) || (PieceColor != Dest->GetColor())) {
            return AreSquaresLegal(SrcRow, SrcCol, DestRow, DestCol, GameBoard);
        }
        return false;
    } catch (const out_of_range& e) {
        cerr << "Error: " << e.what() << endl;
        return false; // Return false to indicate an illegal move
    } catch (const invalid_argument& e) {
        cerr << "Error: " << e.what() << endl;
        return false; // Return false to indicate an illegal move
    } catch (const exception& e) {
        cerr << "An unexpected error occurred: " << e.what() << endl;
        return false; // Return false to indicate an illegal move
    } catch (...) {
        cerr << "An unknown error occurred." << endl;
        return false; // Return false to indicate an illegal move
    }
}
```

Conclusion

In conclusion, our project successfully implements a two-player chess game using Object-Oriented Programming principles. We utilized abstraction, inheritance, polymorphism, operator overloading, and virtual functions, which ensured a modular and maintainable codebase.