# CHESS

★★★★★

## A Digital Chess Challenge

Saad Ahmad 01-134222-130
Sohaib Ahmed 01-134222-142
**Instructor: Mam Afrah Naeem Najmi**

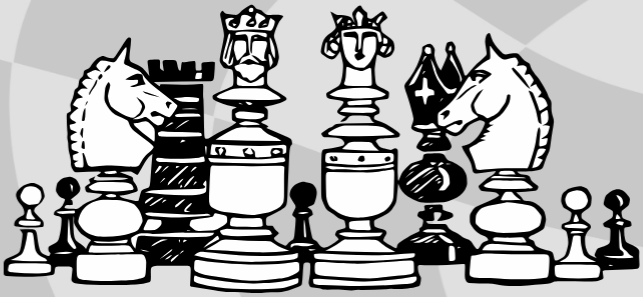
# How to play:

**Objective**: Checkmate the opponent's king.

**Moves:**

- **Pawns:** Move forward, capture diagonally.
- **Rooks:** Move horizontally or vertically.
- **Knights:** Move in an L-shape.
- **Bishops:** Move diagonally.
- **Queen:** Move in any direction.
- **King:** Move one square in any direction.
- **Special Moves:** Castling, pawn protection.

**Capturing:** Replace opponent's pieces when moving onto their square.
**Check:** King under threat; must be resolved.
**Checkmate:** King cannot escape capture.
**Stalemate:** No legal moves without king in check.
**Turns:** Players move one piece per turn, starting with white.
**Endgame:** Checkmate, stalemate, resignation, draw options.



# Table of Contents

**Overview** The project aims to create a basic chess game using the SFML (Simple and Fast Multimedia Library) in C++. The game will feature a graphical user interface allowing two players to play chess on a visual chessboard. The project will include the implementation of chess rules, player input handling, graphical representations of the chess pieces, and basic gameplay functionalities.

## Features

- **Graphical User Interface:** Utilize SFML to create an interactive and visually appealing chessboard interface.
- **Chess Logic Implementation:** Implement the fundamental rules of chess to enable legal moves, captures, and piece interactions.
- **Player Interaction:** Allow players to make moves using mouse clicks or keyboard inputs.
- **Piece Movements:** Enable smooth and valid movement of chess pieces on the board.

- **Highlight Valid Moves:** Visually indicate valid moves for the selected piece to assist players.
- **Game State Management:** Manage game states including start and ongoing gameplay.

## Technologies Used

- C++ programming language for logic and implementation
- SFML library for graphics and user interface

## The Process

### Use of Linked List

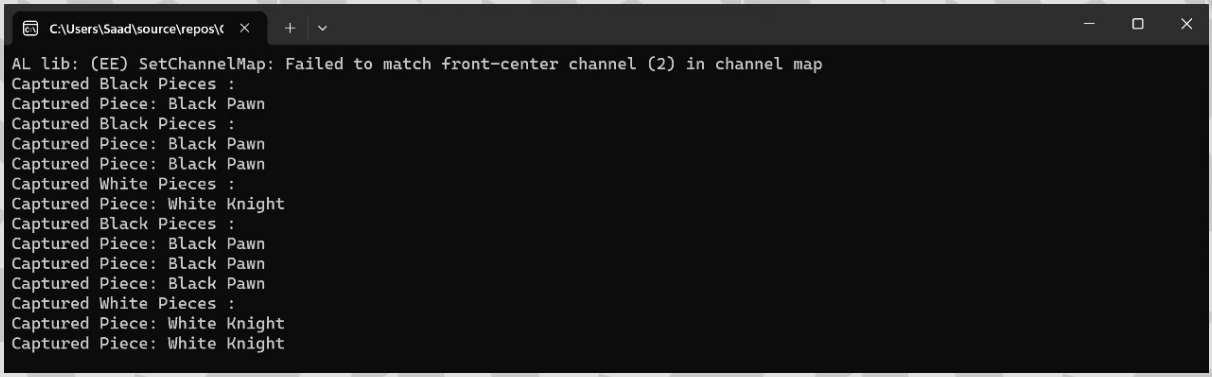
**Code:**

```
struct Node {
public:
    ChessPiece piece;
    Node* next;


    Node(ChessPiece temp) {
        piece = temp;
```

```
        next = nullptr;
    }
};
class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}

    void insert(ChessPiece piece) {
        Node* newNode = new Node(piece);
        newNode->next = head;
        head = newNode;
    }

    void displayWhite() {
        Node* current = head;
        while (current != nullptr) {

            cout << "Captured Piece: " << getWhitePieceName(current->piece) << endl;
```

```
        current = current->next;
    }
}
string getWhitePieceName(ChessPiece piece) {
    switch (piece) {
    case ChessPiece::None: return "None";
    case ChessPiece::WhitePawn: return "White Pawn";
    case ChessPiece::WhiteRook: return "White Rook";
    case ChessPiece::WhiteKnight: return "White Knight";
    case ChessPiece::WhiteBishop: return "White Bishop";
    case ChessPiece::WhiteQueen: return "White Queen";
    case ChessPiece::WhiteKing: return "White King";
    default: return "Unknown Piece";
    }
}
void displayBlack() {
    Node* current = head;
    while (current != nullptr) {


        cout << "Captured Piece: " << getBlackPieceName(current->piece) << endl;
        current = current->next;
```

```
        }
    }
    string getBlackPieceName(ChessPiece piece) {
        switch (piece) {
        case ChessPiece::None: return "None";
        case ChessPiece::BlackPawn: return "Black Pawn";
        case ChessPiece::BlackRook: return "Black Rook";
        case ChessPiece::BlackKnight: return "Black Knight";
        case ChessPiece::BlackBishop: return "Black Bishop";
        case ChessPiece::BlackQueen: return "Black Queen";
        case ChessPiece::BlackKing: return "Black King";
        default: return "Unknown Piece";
        }
    }
};
```



## Chessboard (2D Array)

**Code:**

```
Chessboard::Chessboard() {

    initializeBoard();

    selectedPiece = Vector2i(-1, -1);


    blackPawnTexture.loadFromFile("Images/blackpawn.png");

    whitePawnTexture.loadFromFile("Images/whitepawn.png");

    blackRookTexture.loadFromFile("Images/blackrook.png");

    whiteRookTexture.loadFromFile("Images/whiterook.png");

    blackKnightTexture.loadFromFile("Images/blackknight.png");

    whiteKnightTexture.loadFromFile("Images/whiteknight.png");

    blackBishopTexture.loadFromFile("Images/blackbishop.png");

    whiteBishopTexture.loadFromFile("Images/whitebishop.png");

    blackQueenTexture.loadFromFile("Images/blackqueen.png");

    whiteQueenTexture.loadFromFile("Images/whitequeen.png");

    blackKingTexture.loadFromFile("Images/blackking.png");

    whiteKingTexture.loadFromFile("Images/whiteking.png");


    createPieces();

}
```

```
void Chessboard::initializeBoard() {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            board[i][j] = ChessPiece::None;
        }
    }
}
void Chessboard::createPieces() {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        blackPawnPositions[i] = Vector2i(i, 1);
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        whitePawnPositions[i] = Vector2i(i, 6);
    }
    blackRookPositions[0] = Vector2i(0, 0);
    blackRookPositions[1] = Vector2i(7, 0);
    whiteRookPositions[0] = Vector2i(0, 7);
```

```
whiteRookPositions[1] = Vector2i(7, 7);

blackKnightPositions[0] = Vector2i(1, 0);
blackKnightPositions[1] = Vector2i(6, 0);
whiteKnightPositions[0] = Vector2i(1, 7);
whiteKnightPositions[1] = Vector2i(6, 7);

blackBishopPositions[0] = Vector2i(2, 0);
blackBishopPositions[1] = Vector2i(5, 0);
whiteBishopPositions[0] = Vector2i(2, 7);
whiteBishopPositions[1] = Vector2i(5, 7);

blackQueenPosition = Vector2i(3, 0);
whiteQueenPosition = Vector2i(3, 7);

blackKingPosition = Vector2i(4, 0);
whiteKingPosition = Vector2i(4, 7);

for (int i = 0; i < BOARD_SIZE; ++i) {
    board[blackPawnPositions[i].y][blackPawnPositions[i].x] = ChessPiece::BlackPawn;
```

```
    board[whitePawnPositions[i].y][whitePawnPositions[i].x] = ChessPiece::WhitePawn;
}

board[blackRookPositions[0].y][blackRookPositions[0].x] = ChessPiece::BlackRook;
board[blackRookPositions[1].y][blackRookPositions[1].x] = ChessPiece::BlackRook;
board[whiteRookPositions[0].y][whiteRookPositions[0].x] = ChessPiece::WhiteRook;
board[whiteRookPositions[1].y][whiteRookPositions[1].x] = ChessPiece::WhiteRook;

board[blackKnightPositions[0].y][blackKnightPositions[0].x] = ChessPiece::BlackKnight;
board[blackKnightPositions[1].y][blackKnightPositions[1].x] = ChessPiece::BlackKnight;
board[whiteKnightPositions[0].y][whiteKnightPositions[0].x] = ChessPiece::WhiteKnight;
board[whiteKnightPositions[1].y][whiteKnightPositions[1].x] = ChessPiece::WhiteKnight;

board[blackBishopPositions[0].y][blackBishopPositions[0].x] = ChessPiece::BlackBishop;
board[blackBishopPositions[1].y][blackBishopPositions[1].x] = ChessPiece::BlackBishop;
board[whiteBishopPositions[0].y][whiteBishopPositions[0].x] = ChessPiece::WhiteBishop;
board[whiteBishopPositions[1].y][whiteBishopPositions[1].x] = ChessPiece::WhiteBishop;

board[blackQueenPosition.y][blackQueenPosition.x] = ChessPiece::BlackQueen;
board[whiteQueenPosition.y][whiteQueenPosition.x] = ChessPiece::WhiteQueen;
```

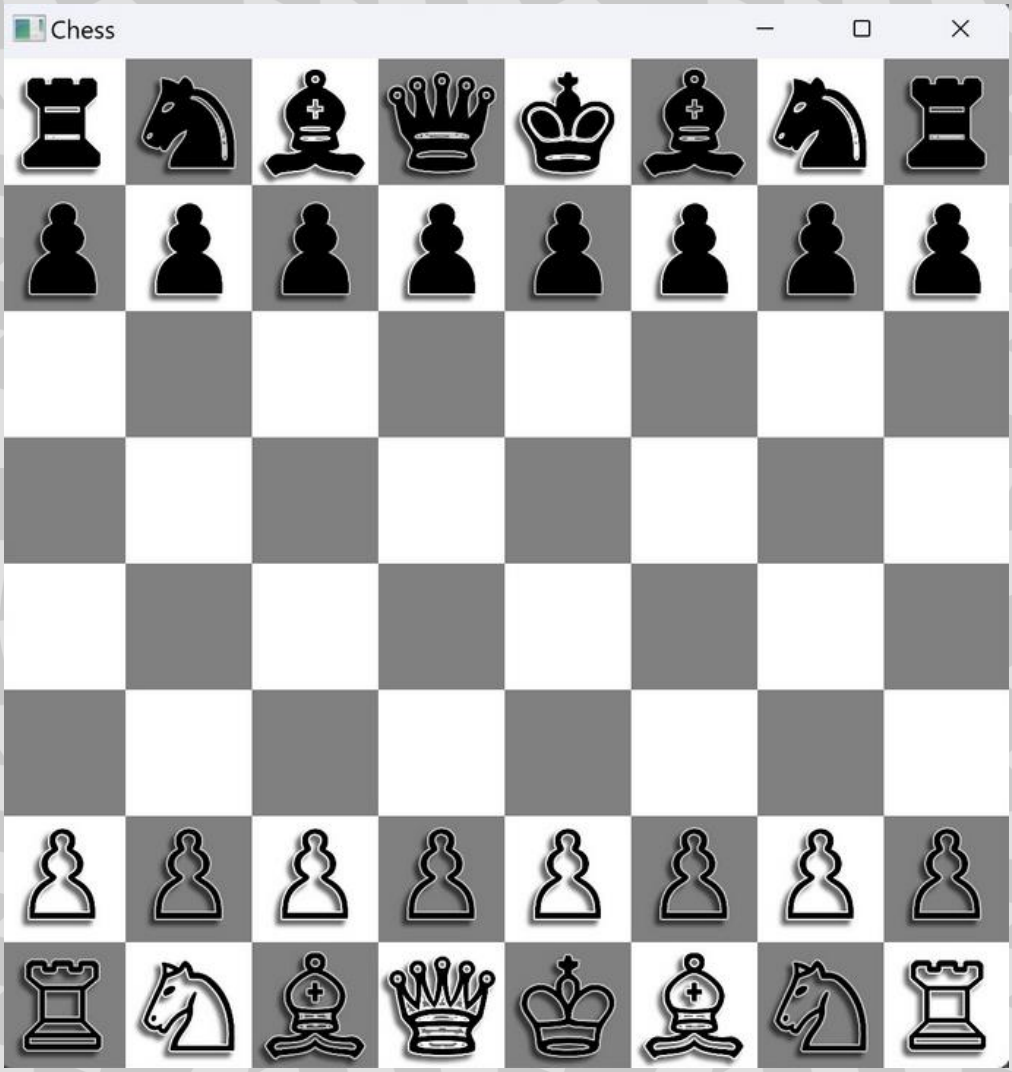
```
    board[blackKingPosition.y][blackKingPosition.x] = ChessPiece::BlackKing;
    board[whiteKingPosition.y][whiteKingPosition.x] = ChessPiece::WhiteKing;

}
void Chessboard::draw(RenderWindow& window) {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            RectangleShape square(Vector2f(SQUARE_SIZE, SQUARE_SIZE));

            if ((i + j) % 2 == 0) {
                square.setFillColor(Color::White);
            }
            else {
                square.setFillColor(Color(128, 128, 128));
            }

            square.setPosition(i * SQUARE_SIZE, j * SQUARE_SIZE);
            window.draw(square);
        }
    }
```

```
    for (int i = 0; i < BOARD_SIZE; ++i) {
        Sprite sprite;
        sprite.setTexture(blackPawnTexture);
        sprite.setPosition(blackPawnPositions[i].x * SQUARE_SIZE, blackPawnPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }


    for (int i = 0; i < BOARD_SIZE; ++i) {
        Sprite sprite;
        sprite.setTexture(whitePawnTexture);
        sprite.setPosition(whitePawnPositions[i].x * SQUARE_SIZE, whitePawnPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteBishopTexture);
```

```
        sprite.setPosition(whiteBishopPositions[i].x * SQUARE_SIZE, whiteBishopPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackRookTexture);
        sprite.setPosition(blackRookPositions[i].x * SQUARE_SIZE, blackRookPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteRookTexture);
        sprite.setPosition(whiteRookPositions[i].x * SQUARE_SIZE, whiteRookPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackKnightTexture);
```

```
        sprite.setPosition(blackKnightPositions[i].x * SQUARE_SIZE, blackKnightPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteKnightTexture);
        sprite.setPosition(whiteKnightPositions[i].x * SQUARE_SIZE, whiteKnightPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackBishopTexture);
        sprite.setPosition(blackBishopPositions[i].x * SQUARE_SIZE, blackBishopPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
    Sprite blackQueenSprite;
    blackQueenSprite.setTexture(blackQueenTexture);
    blackQueenSprite.setPosition(blackQueenPosition.x * SQUARE_SIZE, blackQueenPosition.y *
SQUARE_SIZE);
```

```
    window.draw(blackQueenSprite);
    Sprite whiteQueenSprite;
    whiteQueenSprite.setTexture(whiteQueenTexture);
    whiteQueenSprite.setPosition(whiteQueenPosition.x * SQUARE_SIZE, whiteQueenPosition.y *
SQUARE_SIZE);
    window.draw(whiteQueenSprite);

    Sprite blackKingSprite;
    blackKingSprite.setTexture(blackKingTexture);
    blackKingSprite.setPosition(blackKingPosition.x * SQUARE_SIZE, blackKingPosition.y *
SQUARE_SIZE);
    window.draw(blackKingSprite);

    Sprite whiteKingSprite;
    whiteKingSprite.setTexture(whiteKingTexture);
    whiteKingSprite.setPosition(whiteKingPosition.x * SQUARE_SIZE, whiteKingPosition.y *
SQUARE_SIZE);
    window.draw(whiteKingSprite);
}
```



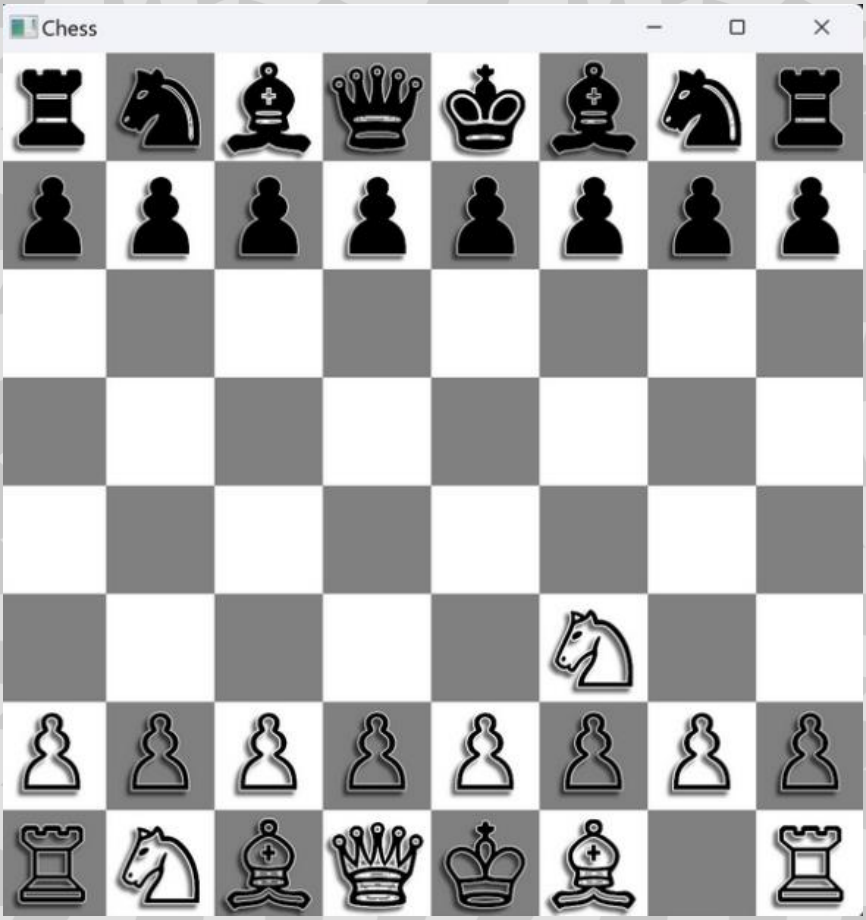

## Movement of the pieces

**Code:**

```
bool Chessboard::handleInput(Event& event) {



    bool check = false;

    if (event.type == Event::MouseButtonPressed) {

        int mouseX = event.mouseButton.x / SQUARE_SIZE;
```

```
int mouseY = event.mouseButton.y / SQUARE_SIZE;

if (selectedPiece.x == -1) {

    if ((isWhiteTurn && isWhitePiece(Vector2i(mouseX, mouseY))) || (!isWhiteTurn && isBlackPiece(Vector2i(mouseX, mouseY)))) {

        selectedPiece = Vector2i(mouseX, mouseY);
    }
}
else {
    check = movePiece(selectedPiece, Vector2i(mouseX, mouseY));
    selectedPiece = Vector2i(-1, -1);

    if (check == true) {
        isWhiteTurn = !isWhiteTurn;
    }
    ChessPiece currentKing = (isWhiteTurn) ? ChessPiece::WhiteKing : ChessPiece::BlackKing;
    if (isKingInCheck(currentKing)) {
        moveSoundBuffer.loadFromFile("Images/check.ogg");
        moveSound.setBuffer(moveSoundBuffer);
```

```
            moveSound.play();

            cout << "Your king is in check!" << endl;

        }

        return true;

    }


    return false;

  }

}
```



## Opponent’s Movement

**Code:**

```
bool Chessboard::isOpponentPiece(const Vector2i& from, const Vector2i& to) const {

    return (isBlackPiece(from) && isWhitePiece(to)) || (isWhitePiece(from) && isBlackPiece(to));
}
bool Chessboard::isBlackPiece(const Vector2i& position) const {

    ChessPiece piece = board[position.y][position.x];
    return (piece == ChessPiece::BlackPawn || piece == ChessPiece::BlackRook || piece == ChessPiece::BlackKnight ||
        piece == ChessPiece::BlackBishop || piece == ChessPiece::BlackQueen || piece == ChessPiece::BlackKing);
}
bool Chessboard::isWhitePiece(const Vector2i& position) const {

    ChessPiece piece = board[position.y][position.x];
    return (piece == ChessPiece::WhitePawn || piece == ChessPiece::WhiteRook || piece == ChessPiece::WhiteKnight ||
        piece == ChessPiece::WhiteBishop || piece == ChessPiece::WhiteQueen || piece == ChessPiece::WhiteKing);
}
```

```
bool Chessboard::isKingInCheck(const ChessPiece kingPiece) const {
    Vector2i kingPosition;
    if (!findPiecePosition(kingPiece, kingPosition)) {
        return false;
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            const Vector2i from(j, i);
            const ChessPiece piece = board[i][j];

            if (isOpponentPiece(from, kingPosition) && canPieceAttack(piece, from, kingPosition)) {

                if (isKingInCheckAfterMove(kingPosition, from)) {
                    return true;
                }
            }
        }
    }
```

```
    return false;
}
bool Chessboard::canPieceAttack(const ChessPiece piece, const Vector2i& from, const Vector2i& to)
const {
    switch (piece) {
    case ChessPiece::WhitePawn:
    case ChessPiece::BlackPawn:
        return canPawnAttack(from, to);
    case ChessPiece::WhiteRook:
    case ChessPiece::BlackRook:
        return canRookAttack(from, to);
    case ChessPiece::WhiteKnight:
    case ChessPiece::BlackKnight:
        return canKnightAttack(from, to);
    case ChessPiece::WhiteBishop:
    case ChessPiece::BlackBishop:
        return canBishopAttack(from, to);
    case ChessPiece::WhiteQueen:
    case ChessPiece::BlackQueen:
        return canQueenAttack(from, to);
    case ChessPiece::WhiteKing:
```

```
    case ChessPiece::BlackKing:
        return canKingAttack(from, to);
    default:
        return false;
    }
}
bool Chessboard::canPawnAttack(const Vector2i& from, const Vector2i& to) const {
    int direction = (board[from.y][from.x] == ChessPiece::BlackPawn) ? 1 : -1;
    return (abs(to.x - from.x) == 1 && to.y - from.y == direction);
}
bool Chessboard::canRookAttack(const Vector2i& from, const Vector2i& to) const {
    return (from.x == to.x || from.y == to.y);
}
bool Chessboard::canKnightAttack(const Vector2i& from, const Vector2i& to) const {
    int dx = abs(to.x - from.x);
    int dy = abs(to.y - from.y);
    return ((dx == 2 && dy == 1) || (dx == 1 && dy == 2));
}
bool Chessboard::canBishopAttack(const Vector2i& from, const Vector2i& to) const {
    return (abs(to.x - from.x) == abs(to.y - from.y));
}
```

```
bool Chessboard::canQueenAttack(const Vector2i& from, const Vector2i& to) const {
    return (canRookAttack(from, to) || canBishopAttack(from, to));
}
bool Chessboard::canKingAttack(const Vector2i& from, const Vector2i& to) const {
    return (abs(to.x - from.x) <= 1 && abs(to.y - from.y) <= 1);
}
bool Chessboard::findPiecePosition(const ChessPiece piece, Vector2i& position) const {
    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            if (board[i][j] == piece) {
                position = Vector2i(j, i);
                return true;
            }
        }
    }
    return false;
}
```


## Pawn’s Movement

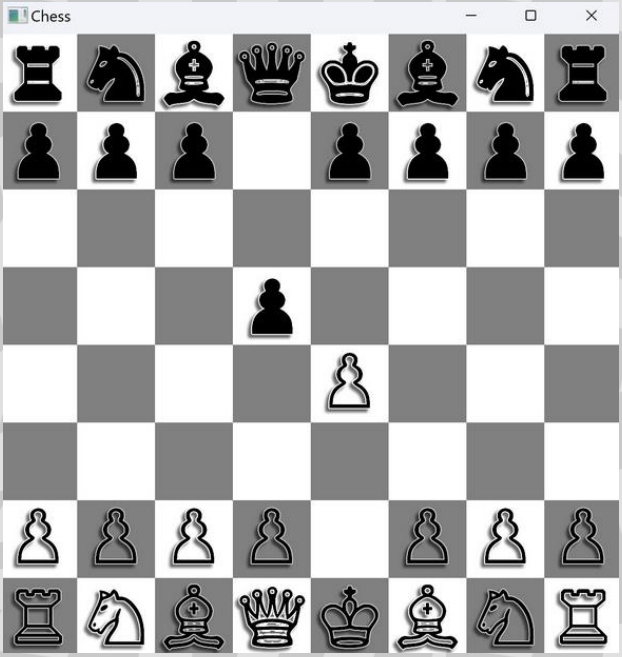
**Code:**

```
bool Chessboard::movePawn(const Vector2i& from, const Vector2i& to) {

    int direction = (board[from.y][from.x] == ChessPiece::BlackPawn) ? 1 : -1;

    if (abs(to.x - from.x) == 1 && to.y - from.y == direction && isOpponentPiece(from, to)) {

        if (board[from.y][from.x] == ChessPiece::WhitePawn && isBlackPiece(to)) {

            CaptureBlackPiece(to);

        }

        else {

            CaptureWhitePiece(to);
```

```
    }
    board[from.y][from.x] = ChessPiece::None;

    board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
    for (int i = 0; i < BOARD_SIZE; ++i) {
      if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
        blackPawnPositions[i] = to;
        break;
      }
      else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
        whitePawnPositions[i] = to;
        break;
      }
    }
    return true;
  }

  if (to.x == from.x && (to.y - from.y == direction && board[to.y][to.x] == ChessPiece::None)) {

    board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
```

```
        board[from.y][from.x] = ChessPiece::None;



        for (int i = 0; i < BOARD_SIZE; ++i) {
            if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
                blackPawnPositions[i] = to;
                break;
            }
            else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
                whitePawnPositions[i] = to;
                break;
            }
        }


        return true;
    }
    if (from.y == (direction == 1 ? 1 : 6) && to.x == from.x && to.y - from.y == 2 * direction &&
board[to.y][to.x] == ChessPiece::None) {


        board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
        board[from.y][from.x] = ChessPiece::None;
```

```
        for (int i = 0; i < BOARD_SIZE; ++i) {
            if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
                blackPawnPositions[i] = to;
                break;
            }
            else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
                whitePawnPositions[i] = to;
                break;
            }
        }
        return true;
    }
    return false;
}
```



## Rook’s Movement

**Code:**

```
bool Chessboard::moveRook(const Vector2i& from, const Vector2i& to) {

    if (from.x == to.x || from.y == to.y) {

        int step = (from.x == to.x) ? ((to.y > from.y) ? 1 : -1) : ((to.x > from.x) ? 1 : -1);
        int currentX = from.x + ((from.x == to.x) ? 0 : step);
        int currentY = from.y + ((from.y == to.y) ? 0 : step);

        bool isValidMove = true;

        while (currentX != to.x || currentY != to.y) {
            if (board[currentY][currentX] != ChessPiece::None) {
                isValidMove = false;
                break;
            }

            currentX += (from.x == to.x) ? 0 : step;
            currentY += (from.y == to.y) ? 0 : step;
        }
```

```
if (isValidMove) {
    if (board[to.y][to.x] == ChessPiece::None) {

        board[from.y][from.x] = ChessPiece::None;
        board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ? ChessPiece::BlackRook : ChessPiece::WhiteRook;

        for (int i = 0; i < 2; ++i) {
            if (blackRookPositions[i] == from) {
                blackRookPositions[i] = to;
                break;
            }
        }

        for (int i = 0; i < 2; ++i) {
            if (whiteRookPositions[i] == from) {
                whiteRookPositions[i] = to;
                break;
            }
```

```
        }
        return true;
    }
    else if (board[from.y][from.x]==ChessPiece::WhiteRook && isBlackPiece(to)) {

        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);


        board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ?
ChessPiece::BlackRook : ChessPiece::WhiteRook;

        for (int i = 0; i < 2; ++i) {
            if (whiteRookPositions[i] == from) {
                whiteRookPositions[i] = to;
                break;
            }
        }
        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::BlackRook && isWhitePiece(to)) {
```

```
            board[from.y][from.x] = ChessPiece::None;

            CcaptureWhitePiece(to);



            board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ?
ChessPiece::BlackRook : ChessPiece::WhiteRook;



            for (int i = 0; i < 2; ++i) {

              if (blackRookPositions[i] == from) {

                blackRookPositions[i] = to;

                break;

              }

            }


            return true;

          }

        }

      }
```

}


## Knight’s Movement

**Code:**

```
bool Chessboard::moveKnight(const Vector2i& from, const Vector2i& to) {

    int dx = abs(to.x - from.x);

    int dy = abs(to.y - from.y);

    if ((dx == 2 && dy == 1) || (dx == 1 && dy == 2)) {
```

```
if (board[to.y][to.x] == ChessPiece::None ) {

    board[from.y][from.x] = ChessPiece::None;

    board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

    for (int i = 0; i < 2; ++i) {

        if (blackKnightPositions[i] == from) {

            blackKnightPositions[i] = to;

            break;

        }

    }

    for (int i = 0; i < 2; ++i) {

        if (whiteKnightPositions[i] == from) {

            whiteKnightPositions[i] = to;

            break;

        }

    }

    return true;
```

```
    }
    else if (board[from.y][from.x]==ChessPiece::WhiteKnight && isBlackPiece(to)) {

        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);

        board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

        for (int i = 0; i < 2; ++i) {
            if (whiteKnightPositions[i] == from) {
                whiteKnightPositions[i] = to;
                break;
            }
        }

        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::BlackKnight && isWhitePiece(to)) {
```

```
            board[from.y][from.x] = ChessPiece::None;
            CaptureWhitePiece(to);

            board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

            for (int i = 0; i < 2; ++i) {
                if (blackKnightPositions[i] == from) {
                    blackKnightPositions[i] = to;
                    break;
                }
            }
            return true;
        }
    }
}
```


## Bishop’s Movement

**Code:**

```
bool Chessboard::moveBishop(const Vector2i& from, const Vector2i& to) {


    int dx = abs(to.x - from.x);

    int dy = abs(to.y - from.y);


    if (dx == dy) {


        int stepX = (to.x > from.x) ? 1 : -1;

        int stepY = (to.y > from.y) ? 1 : -1;
```

```
int currentX = from.x + stepX;
int currentY = from.y + stepY;

bool isValidMove = true;

while (currentX != to.x && currentY != to.y) {
    if (board[currentY][currentX] != ChessPiece::None) {
        isValidMove = false;
        break;
    }

    currentX += stepX;
    currentY += stepY;
}

if (isValidMove) {
    if (board[to.y][to.x] == ChessPiece::None ) {
        board[from.y][from.x] = ChessPiece::None;
```

```
            board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

            for (int i = 0; i < 2; ++i) {
                if (blackBishopPositions[i] == from) {
                    blackBishopPositions[i] = to;
                    break;
                }
            }

            for (int i = 0; i < 2; ++i) {
                if (whiteBishopPositions[i] == from) {
                    whiteBishopPositions[i] = to;
                    break;
                }
            }
            return true;
        }
        else if (board[from.y][from.x] == ChessPiece::WhiteBishop && isBlackPiece(to)) {

            board[from.y][from.x] = ChessPiece::None;
```

```
            CaptureBlackPiece(to);
            board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

            for (int i = 0; i < 2; ++i) {
                if (whiteBishopPositions[i] == from) {
                    whiteBishopPositions[i] = to;
                    break;
                }
            }

            return true;
        }
        else if (board[from.y][from.x] == ChessPiece::BlackBishop && isWhitePiece(to)) {

            board[from.y][from.x] = ChessPiece::None;
            CaptureWhitePiece(to);
            board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

            for (int i = 0; i < 2; ++i) {
                if (blackBishopPositions[i] == from) {
```

```
                blackBishopPositions[i] = to;
                break;
            }
        }
        return true;
    }
}
}
}
```


## Queen’s Movement

**Code:**

```
bool Chessboard::moveQueen(const Vector2i& from, const Vector2i& to) {

    if (from.x == to.x || from.y == to.y || abs(to.x - from.x) == abs(to.y - from.y)) {

        int stepX = (from.x == to.x) ? 0 : ((to.x > from.x) ? 1 : -1);

        int stepY = (from.y == to.y) ? 0 : ((to.y > from.y) ? 1 : -1);
```

```
int currentX = from.x + stepX;
int currentY = from.y + stepY;

bool isValidMove = true;

while ((from.x == to.x && currentY != to.y) || (from.y == to.y && currentX != to.x) || (currentX != to.x && currentY != to.y)) {
    if (board[currentY][currentX] != ChessPiece::None) {
        isValidMove = false;
        break;
    }

    currentX += stepX;
    currentY += stepY;
}

if (isValidMove) {
    if (board[to.y][to.x] == ChessPiece::None ) {
        ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen : ChessPiece::WhiteQueen;
```

```
            board[from.y][from.x] = ChessPiece::None;
            board[to.y][to.x] = movedPiece;

            if (from == blackQueenPosition) {
                blackQueenPosition = to;
            }

            if (from == whiteQueenPosition) {
                whiteQueenPosition = to;
            }
            return true;
        }
        else if (board[from.y][from.x] == ChessPiece::WhiteQueen && isBlackPiece(to)) {
            ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen :
ChessPiece::WhiteQueen;
            board[from.y][from.x] = ChessPiece::None;
            CaptureBlackPiece(to);
            board[to.y][to.x] = movedPiece;

            if (from == whiteQueenPosition) {
```

```
            whiteQueenPosition = to;
        }

        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::BlackQueen && isWhitePiece(to)) {
        ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen :
ChessPiece::WhiteQueen;
        board[from.y][from.x] = ChessPiece::None;
        CaptureWhitePiece(to);
        board[to.y][to.x] = movedPiece;
        if (from == blackQueenPosition) {
            blackQueenPosition = to;
        }

        return true;
    }

}
```

```
    }

}
```


## King’s Movement

**Code:**

```
bool Chessboard::moveKing(const Vector2i& from, const Vector2i& to) {

    if (abs(to.x - from.x) <= 1 && abs(to.y - from.y) <= 1) {

        if (board[to.y][to.x] == ChessPiece::None) {
```

```
        ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing : ChessPiece::WhiteKing;

        board[from.y][from.x] = ChessPiece::None;
        board[to.y][to.x] = movedPiece;

        if (from == blackKingPosition) {
            blackKingPosition = to;

        }

        if (from == whiteKingPosition) {
            whiteKingPosition = to;
        }
        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::WhiteKing && isBlackPiece(to)) {
        ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing : ChessPiece::WhiteKing;
        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);
        board[to.y][to.x] = movedPiece;
```

```
        if (from == whiteKingPosition) {
            whiteKingPosition = to;
        }

        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::BlackKing && isWhitePiece(to)) {
        ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing :
ChessPiece::WhiteKing;
        board[from.y][from.x] = ChessPiece::None;
        CaptureWhitePiece(to);
        board[to.y][to.x] = movedPiece;
        if (from == blackKingPosition) {
            blackKingPosition = to;
        }

        return true;
    }
}
```

}


## Capturing Black

**Code:**

```
void Chessboard::CaptureBlackPiece(const Vector2i& to) {

    ChessPiece piece = board[to.y][to.x];

    cout << "Captured Black Pieces :" << endl;

    capturedBlackPieces.insert(piece);

    capturedBlackPieces.displayBlack();
```

```
switch (piece) {
case ChessPiece::BlackPawn:
    for (int i = 0; i < BOARD_SIZE; ++i) {
        if (blackPawnPositions[i] == to) {
            blackPawnPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::BlackRook:
    for (int i = 0; i < 2; ++i) {
        if (blackRookPositions[i] == to) {
            blackRookPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::BlackKnight:
    for (int i = 0; i < 2; ++i) {
        if (blackKnightPositions[i] == to) {
            blackKnightPositions[i] = Vector2i(-1, -1);
```

```
            break;
        }
    }
    break;
case ChessPiece::BlackBishop:
    for (int i = 0; i < 2; ++i) {
        if (blackBishopPositions[i] == to) {
            blackBishopPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::BlackQueen:
    for (int i = 0; i < 2; ++i) {
        if (blackQueenPosition == to) {
            blackQueenPosition = Vector2i(-1, -1);
            break;
        }
    }
    break;
default:
```

```
        break;
    }
    moveSoundBuffer.loadFromFile("Images/capture.ogg");
    moveSound.setBuffer(moveSoundBuffer);
    moveSound.play();
}
```


## Capturing White

**Code:**

```
void Chessboard::CaptureWhitePiece(const Vector2i& to) {
```

```
ChessPiece piece = board[to.y][to.x];
capturedWhitePieces.insert(piece);
cout << "Captured White Pieces : " << endl;
capturedWhitePieces.displayWhite();
switch (piece) {
case ChessPiece::WhitePawn:
    for (int i = 0; i < BOARD_SIZE; ++i) {
        if (whitePawnPositions[i] == to) {
            whitePawnPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::WhiteRook:
    for (int i = 0; i < 2; ++i) {
        if (whiteRookPositions[i] == to) {
            whiteRookPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
```

```
    break;
case ChessPiece::WhiteKnight:
    for (int i = 0; i < 2; ++i) {
        if (whiteKnightPositions[i] == to) {
            whiteKnightPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::WhiteBishop:
    for (int i = 0; i < 2; ++i) {
        if (whiteBishopPositions[i] == to) {
            whiteBishopPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::WhiteQueen:
    if (whiteQueenPosition == to) {
        whiteQueenPosition = Vector2i(-1, -1);
```

```
        }
        break;
    default:
        break;
    }
    moveSoundBuffer.loadFromFile("Images/capture.ogg");
    moveSound.setBuffer(moveSoundBuffer);
    moveSound.play();
}
```

## Sound Design

**Code:**

```
bool Chessboard::movePiece(const Vector2i& from, const Vector2i& to) {
    bool check = false;
    moveSoundBuffer.loadFromFile("Images/move.ogg");
    moveSound.setBuffer(moveSoundBuffer);
      ChessPiece piece = board[from.y][from.x];
      if ((isWhiteTurn && isWhitePiece(from)) || (!isWhiteTurn && isBlackPiece(from))) {
          switch (piece) {
          case ChessPiece::BlackPawn:
          case ChessPiece::WhitePawn:
              check = movePawn(from, to);
              break;
          case ChessPiece::BlackRook:
          case ChessPiece::WhiteRook:
             check =  moveRook(from, to);
              break;
          case ChessPiece::BlackKnight:
          case ChessPiece::WhiteKnight:
             check =  moveKnight(from, to);
              break;
```

```
            case ChessPiece::BlackBishop:
            case ChessPiece::WhiteBishop:
                check = moveBishop(from, to);
                break;
            case ChessPiece::BlackQueen:
            case ChessPiece::WhiteQueen:
              check =  moveQueen(from, to);
                break;
            case ChessPiece::BlackKing:
            case ChessPiece::WhiteKing:
                check =  moveKing(from, to);
                break;
            default:
                check = false;
                break;
            }
            if (check) {
                moveSound.play();
            }
        }
    return check;
```

}


## Complete Code

```
#include <SFML/Graphics.hpp>

#include <SFML/Audio.hpp>

#include <iostream>


using namespace sf;
```

```
using namespace std;

const int WINDOW_SIZE = 1200;
const int BOARD_SIZE = 8;
const float SQUARE_SIZE = (WINDOW_SIZE) / BOARD_SIZE;
enum class ChessPiece {
    None = 0,
    WhitePawn = 1,
    WhiteRook = 5,
    WhiteKnight = 3,
    WhiteBishop = 4,
    WhiteQueen = 7,
    WhiteKing = 9,
    BlackPawn = -1,
    BlackRook = -5,
    BlackKnight = -3,
    BlackBishop = -4,
    BlackQueen = -7,
    BlackKing = -9
};
```

```
struct Node {
public:
    ChessPiece piece;
    Node* next;

    Node(ChessPiece temp) {
        piece = temp;
        next = nullptr;
    }
};
class LinkedList {
private:
    Node* head;

public:
    LinkedList() : head(nullptr) {}

    void insert(ChessPiece piece) {
        Node* newNode = new Node(piece);
        newNode->next = head;
        head = newNode;
```

```
}

void displayWhite() {
    Node* current = head;
    while (current != nullptr) {

        cout << "Captured Piece: " << getWhitePieceName(current->piece) << endl;
        current = current->next;
    }
}
string getWhitePieceName(ChessPiece piece) {
    switch (piece) {
    case ChessPiece::None: return "None";
    case ChessPiece::WhitePawn: return "White Pawn";
    case ChessPiece::WhiteRook: return "White Rook";
    case ChessPiece::WhiteKnight: return "White Knight";
    case ChessPiece::WhiteBishop: return "White Bishop";
    case ChessPiece::WhiteQueen: return "White Queen";
    case ChessPiece::WhiteKing: return "White King";
    default: return "Unknown Piece";
    }
```

```
}
void displayBlack() {
    Node* current = head;
    while (current != nullptr) {

        cout << "Captured Piece: " << getBlackPieceName(current->piece) << endl;
        current = current->next;
    }
}
string getBlackPieceName(ChessPiece piece) {
    switch (piece) {
    case ChessPiece::None: return "None";
    case ChessPiece::BlackPawn: return "Black Pawn";
    case ChessPiece::BlackRook: return "Black Rook";
    case ChessPiece::BlackKnight: return "Black Knight";
    case ChessPiece::BlackBishop: return "Black Bishop";
    case ChessPiece::BlackQueen: return "Black Queen";
    case ChessPiece::BlackKing: return "Black King";
    default: return "Unknown Piece";
    }
}
```

```
};

class Chessboard {
public:
    Chessboard();
    LinkedList capturedBlackPieces;
    LinkedList capturedWhitePieces;
    void draw(RenderWindow& window);
    bool handleInput(Event& event);

private:
    SoundBuffer moveSoundBuffer;
    Sound moveSound;
    ChessPiece board[BOARD_SIZE][BOARD_SIZE];
    Vector2i selectedPiece;
    Texture blackPawnTexture;
    Texture whitePawnTexture;
    Texture blackRookTexture;
    Texture whiteRookTexture;
    Texture blackKnightTexture;
```

```
Texture whiteKnightTexture;

Texture blackBishopTexture;

Texture whiteBishopTexture;

Texture blackQueenTexture;

Texture whiteQueenTexture;

Texture blackKingTexture;

Texture whiteKingTexture;

Vector2i blackPawnPositions[BOARD_SIZE];

Vector2i whitePawnPositions[BOARD_SIZE];

Vector2i blackRookPositions[2];

Vector2i whiteRookPositions[2];

Vector2i blackKnightPositions[2];

Vector2i whiteKnightPositions[2];

Vector2i blackBishopPositions[2];

Vector2i whiteBishopPositions[2];

Vector2i blackQueenPosition;

Vector2i whiteQueenPosition;

Vector2i blackKingPosition;

Vector2i whiteKingPosition;

bool isWhiteTurn = true;

void initializeBoard();
```

```
void createPieces();

bool movePiece(const Vector2i& from, const Vector2i& to);

bool movePawn(const Vector2i& from, const Vector2i& to);

bool moveRook(const Vector2i& from, const Vector2i& to);

bool moveKnight(const Vector2i& from, const Vector2i& to);

bool isOpponentPiece(const Vector2i& from, const Vector2i& to) const;

bool isWhitePiece(const Vector2i& position) const;

bool isBlackPiece(const Vector2i& positon) const;

bool moveBishop(const Vector2i& from, const Vector2i& to);

bool moveQueen(const Vector2i& from, const Vector2i& to);

bool moveKing(const Vector2i& from, const Vector2i& to);

void CaptureBlackPiece(const Vector2i& from);

void CaptureWhitePiece(const Vector2i& from);

bool isKingInCheck(ChessPiece kingPiece) const;

bool canPieceAttack(ChessPiece piece, const Vector2i& from, const Vector2i& to) const;

bool canPawnAttack(const Vector2i& from, const Vector2i& to) const;

bool canRookAttack(const Vector2i& from, const Vector2i& to) const;

bool canKnightAttack(const Vector2i& from, const Vector2i& to) const;

bool canBishopAttack(const Vector2i& from, const Vector2i& to) const;

bool canQueenAttack(const Vector2i& from, const Vector2i& to) const;

bool canKingAttack(const Vector2i& from, const Vector2i& to) const;
```

```
    bool findPiecePosition(ChessPiece piece, Vector2i& position) const;
    bool isCheckmate() const;
    bool isKingInCheckAfterMove(const Vector2i& from, const Vector2i& to) const;
};
Chessboard::Chessboard() {
    initializeBoard();
    selectedPiece = Vector2i(-1, -1);

    blackPawnTexture.loadFromFile("Images/blackpawn.png");
    whitePawnTexture.loadFromFile("Images/whitepawn.png");
    blackRookTexture.loadFromFile("Images/blackrook.png");
    whiteRookTexture.loadFromFile("Images/whiterook.png");
    blackKnightTexture.loadFromFile("Images/blackknight.png");
    whiteKnightTexture.loadFromFile("Images/whiteknight.png");
    blackBishopTexture.loadFromFile("Images/blackbishop.png");
    whiteBishopTexture.loadFromFile("Images/whitebishop.png");
    blackQueenTexture.loadFromFile("Images/blackqueen.png");
    whiteQueenTexture.loadFromFile("Images/whitequeen.png");
    blackKingTexture.loadFromFile("Images/blackking.png");
    whiteKingTexture.loadFromFile("Images/whiteking.png");
```

```
    createPieces();
}
void Chessboard::initializeBoard() {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            board[i][j] = ChessPiece::None;
        }
    }
}
void Chessboard::createPieces() {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        blackPawnPositions[i] = Vector2i(i, 1);
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        whitePawnPositions[i] = Vector2i(i, 6);
    }
```

```
blackRookPositions[0] = Vector2i(0, 0);
blackRookPositions[1] = Vector2i(7, 0);
whiteRookPositions[0] = Vector2i(0, 7);
whiteRookPositions[1] = Vector2i(7, 7);

blackKnightPositions[0] = Vector2i(1, 0);
blackKnightPositions[1] = Vector2i(6, 0);
whiteKnightPositions[0] = Vector2i(1, 7);
whiteKnightPositions[1] = Vector2i(6, 7);

blackBishopPositions[0] = Vector2i(2, 0);
blackBishopPositions[1] = Vector2i(5, 0);
whiteBishopPositions[0] = Vector2i(2, 7);
whiteBishopPositions[1] = Vector2i(5, 7);

blackQueenPosition = Vector2i(3, 0);
whiteQueenPosition = Vector2i(3, 7);

blackKingPosition = Vector2i(4, 0);
whiteKingPosition = Vector2i(4, 7);
```

```
for (int i = 0; i < BOARD_SIZE; ++i) {
    board[blackPawnPositions[i].y][blackPawnPositions[i].x] = ChessPiece::BlackPawn;
    board[whitePawnPositions[i].y][whitePawnPositions[i].x] = ChessPiece::WhitePawn;
}

board[blackRookPositions[0].y][blackRookPositions[0].x] = ChessPiece::BlackRook;
board[blackRookPositions[1].y][blackRookPositions[1].x] = ChessPiece::BlackRook;
board[whiteRookPositions[0].y][whiteRookPositions[0].x] = ChessPiece::WhiteRook;
board[whiteRookPositions[1].y][whiteRookPositions[1].x] = ChessPiece::WhiteRook;

board[blackKnightPositions[0].y][blackKnightPositions[0].x] = ChessPiece::BlackKnight;
board[blackKnightPositions[1].y][blackKnightPositions[1].x] = ChessPiece::BlackKnight;
board[whiteKnightPositions[0].y][whiteKnightPositions[0].x] = ChessPiece::WhiteKnight;
board[whiteKnightPositions[1].y][whiteKnightPositions[1].x] = ChessPiece::WhiteKnight;

board[blackBishopPositions[0].y][blackBishopPositions[0].x] = ChessPiece::BlackBishop;
board[blackBishopPositions[1].y][blackBishopPositions[1].x] = ChessPiece::BlackBishop;
board[whiteBishopPositions[0].y][whiteBishopPositions[0].x] = ChessPiece::WhiteBishop;
board[whiteBishopPositions[1].y][whiteBishopPositions[1].x] = ChessPiece::WhiteBishop;
```

```
    board[blackQueenPosition.y][blackQueenPosition.x] = ChessPiece::BlackQueen;
    board[whiteQueenPosition.y][whiteQueenPosition.x] = ChessPiece::WhiteQueen;

    board[blackKingPosition.y][blackKingPosition.x] = ChessPiece::BlackKing;
    board[whiteKingPosition.y][whiteKingPosition.x] = ChessPiece::WhiteKing;

}
void Chessboard::draw(RenderWindow& window) {

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            RectangleShape square(Vector2f(SQUARE_SIZE, SQUARE_SIZE));

            if ((i + j) % 2 == 0) {
                square.setFillColor(Color::White);
            }
            else {
                square.setFillColor(Color(128, 128, 128));
            }
```

```
            square.setPosition(i * SQUARE_SIZE, j * SQUARE_SIZE);
            window.draw(square);
        }
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        Sprite sprite;
        sprite.setTexture(blackPawnTexture);
        sprite.setPosition(blackPawnPositions[i].x * SQUARE_SIZE, blackPawnPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        Sprite sprite;
        sprite.setTexture(whitePawnTexture);
        sprite.setPosition(whitePawnPositions[i].x * SQUARE_SIZE, whitePawnPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }
```

```
    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteBishopTexture);
        sprite.setPosition(whiteBishopPositions[i].x * SQUARE_SIZE, whiteBishopPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackRookTexture);
        sprite.setPosition(blackRookPositions[i].x * SQUARE_SIZE, blackRookPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteRookTexture);
        sprite.setPosition(whiteRookPositions[i].x * SQUARE_SIZE, whiteRookPositions[i].y *
SQUARE_SIZE);
```

```
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackKnightTexture);
        sprite.setPosition(blackKnightPositions[i].x * SQUARE_SIZE, blackKnightPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(whiteKnightTexture);
        sprite.setPosition(whiteKnightPositions[i].x * SQUARE_SIZE, whiteKnightPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    for (int i = 0; i < 2; ++i) {
        Sprite sprite;
        sprite.setTexture(blackBishopTexture);
```

```
        sprite.setPosition(blackBishopPositions[i].x * SQUARE_SIZE, blackBishopPositions[i].y *
SQUARE_SIZE);
        window.draw(sprite);
    }

    Sprite blackQueenSprite;
    blackQueenSprite.setTexture(blackQueenTexture);
    blackQueenSprite.setPosition(blackQueenPosition.x * SQUARE_SIZE, blackQueenPosition.y *
SQUARE_SIZE);
    window.draw(blackQueenSprite);

    Sprite whiteQueenSprite;
    whiteQueenSprite.setTexture(whiteQueenTexture);
    whiteQueenSprite.setPosition(whiteQueenPosition.x * SQUARE_SIZE, whiteQueenPosition.y *
SQUARE_SIZE);
    window.draw(whiteQueenSprite);

    Sprite blackKingSprite;
    blackKingSprite.setTexture(blackKingTexture);
    blackKingSprite.setPosition(blackKingPosition.x * SQUARE_SIZE, blackKingPosition.y *
SQUARE_SIZE);
    window.draw(blackKingSprite);
```

```
    Sprite whiteKingSprite;
    whiteKingSprite.setTexture(whiteKingTexture);
    whiteKingSprite.setPosition(whiteKingPosition.x * SQUARE_SIZE, whiteKingPosition.y *
SQUARE_SIZE);
    window.draw(whiteKingSprite);
}
bool Chessboard::handleInput(Event& event) {

    bool check = false;
    if (event.type == Event::MouseButtonPressed) {
        int mouseX = event.mouseButton.x / SQUARE_SIZE;
        int mouseY = event.mouseButton.y / SQUARE_SIZE;

        if (selectedPiece.x == -1) {

            if ((isWhiteTurn && isWhitePiece(Vector2i(mouseX, mouseY))) || (!isWhiteTurn &&
isBlackPiece(Vector2i(mouseX, mouseY)))) {

                selectedPiece = Vector2i(mouseX, mouseY);
            }
        }
```

```
        else {
            check = movePiece(selectedPiece, Vector2i(mouseX, mouseY));
            selectedPiece = Vector2i(-1, -1);

            if (check == true) {
                isWhiteTurn = !isWhiteTurn;
            }
            ChessPiece currentKing = (isWhiteTurn) ? ChessPiece::WhiteKing : ChessPiece::BlackKing;
            if (isKingInCheck(currentKing)) {
                moveSoundBuffer.loadFromFile("Images/check.ogg");
                moveSound.setBuffer(moveSoundBuffer);
                moveSound.play();
                cout << "Your king is in check!" << endl;
            }
            return true;
        }

        return false;
    }
}
```

```
bool Chessboard::isOpponentPiece(const Vector2i& from, const Vector2i& to) const {

    return (isBlackPiece(from) && isWhitePiece(to)) || (isWhitePiece(from) && isBlackPiece(to));
}
bool Chessboard::isBlackPiece(const Vector2i& position) const {

    ChessPiece piece = board[position.y][position.x];
    return (piece == ChessPiece::BlackPawn || piece == ChessPiece::BlackRook || piece ==
ChessPiece::BlackKnight ||
        piece == ChessPiece::BlackBishop || piece == ChessPiece::BlackQueen || piece ==
ChessPiece::BlackKing);
}
bool Chessboard::isWhitePiece(const Vector2i& position) const {

    ChessPiece piece = board[position.y][position.x];
    return (piece == ChessPiece::WhitePawn || piece == ChessPiece::WhiteRook || piece ==
ChessPiece::WhiteKnight ||
        piece == ChessPiece::WhiteBishop || piece == ChessPiece::WhiteQueen || piece ==
ChessPiece::WhiteKing);
}
bool Chessboard::isKingInCheck(const ChessPiece kingPiece) const {
    Vector2i kingPosition;
```

```
    if (!findPiecePosition(kingPiece, kingPosition)) {
        return false;
    }

    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            const Vector2i from(j, i);
            const ChessPiece piece = board[i][j];

            if (isOpponentPiece(from, kingPosition) && canPieceAttack(piece, from, kingPosition)) {

                if (isKingInCheckAfterMove(kingPosition, from)) {
                    return true;
                }
            }
        }
    }

    return false;
}
```

```
bool Chessboard::canPieceAttack(const ChessPiece piece, const Vector2i& from, const Vector2i& to)
const {
    switch (piece) {
    case ChessPiece::WhitePawn:
    case ChessPiece::BlackPawn:
        return canPawnAttack(from, to);
    case ChessPiece::WhiteRook:
    case ChessPiece::BlackRook:
        return canRookAttack(from, to);
    case ChessPiece::WhiteKnight:
    case ChessPiece::BlackKnight:
        return canKnightAttack(from, to);
    case ChessPiece::WhiteBishop:
    case ChessPiece::BlackBishop:
        return canBishopAttack(from, to);
    case ChessPiece::WhiteQueen:
    case ChessPiece::BlackQueen:
        return canQueenAttack(from, to);
    case ChessPiece::WhiteKing:
    case ChessPiece::BlackKing:
        return canKingAttack(from, to);
```

```
    default:
        return false;
    }
}
bool Chessboard::canPawnAttack(const Vector2i& from, const Vector2i& to) const {
    int direction = (board[from.y][from.x] == ChessPiece::BlackPawn) ? 1 : -1;
    return (abs(to.x - from.x) == 1 && to.y - from.y == direction);
}
bool Chessboard::canRookAttack(const Vector2i& from, const Vector2i& to) const {
    return (from.x == to.x || from.y == to.y);
}
bool Chessboard::canKnightAttack(const Vector2i& from, const Vector2i& to) const {
    int dx = abs(to.x - from.x);
    int dy = abs(to.y - from.y);
    return ((dx == 2 && dy == 1) || (dx == 1 && dy == 2));
}
bool Chessboard::canBishopAttack(const Vector2i& from, const Vector2i& to) const {
    return (abs(to.x - from.x) == abs(to.y - from.y));
}
bool Chessboard::canQueenAttack(const Vector2i& from, const Vector2i& to) const {
    return (canRookAttack(from, to) || canBishopAttack(from, to));
```

```
}

bool Chessboard::canKingAttack(const Vector2i& from, const Vector2i& to) const {

    return (abs(to.x - from.x) <= 1 && abs(to.y - from.y) <= 1);

}

bool Chessboard::findPiecePosition(const ChessPiece piece, Vector2i& position) const {

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

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

            if (board[i][j] == piece) {

                position = Vector2i(j, i);

                return true;

            }

        }

    }

    return false;

}

bool Chessboard::movePawn(const Vector2i& from, const Vector2i& to) {

    int direction = (board[from.y][from.x] == ChessPiece::BlackPawn) ? 1 : -1;



    if (abs(to.x - from.x) == 1 && to.y - from.y == direction && isOpponentPiece(from, to)) {
```

```
    if (board[from.y][from.x] == ChessPiece::WhitePawn && isBlackPiece(to)) {
        CaptureBlackPiece(to);
    }
    else {
        CaptureWhitePiece(to);
    }
    board[from.y][from.x] = ChessPiece::None;

    board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
    for (int i = 0; i < BOARD_SIZE; ++i) {
        if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
            blackPawnPositions[i] = to;
            break;
        }
        else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
            whitePawnPositions[i] = to;
            break;
        }
    }
    return true;
}
```

```
if (to.x == from.x && (to.y - from.y == direction && board[to.y][to.x] == ChessPiece::None)) {

    board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
    board[from.y][from.x] = ChessPiece::None;

    for (int i = 0; i < BOARD_SIZE; ++i) {
        if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
            blackPawnPositions[i] = to;
            break;
        }
        else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
            whitePawnPositions[i] = to;
            break;
        }
    }

    return true;
}
```

```
    if (from.y == (direction == 1 ? 1 : 6) && to.x == from.x && to.y - from.y == 2 * direction &&
board[to.y][to.x] == ChessPiece::None) {

        board[to.y][to.x] = (direction == 1) ? ChessPiece::BlackPawn : ChessPiece::WhitePawn;
        board[from.y][from.x] = ChessPiece::None;

        for (int i = 0; i < BOARD_SIZE; ++i) {
            if (board[to.y][to.x] == ChessPiece::BlackPawn && blackPawnPositions[i] == from) {
                blackPawnPositions[i] = to;
                break;
            }
            else if (board[to.y][to.x] == ChessPiece::WhitePawn && whitePawnPositions[i] == from) {
                whitePawnPositions[i] = to;
                break;
            }
        }

        return true;
    }
```

```
    return false;
}
bool Chessboard::moveRook(const Vector2i& from, const Vector2i& to) {

    if (from.x == to.x || from.y == to.y) {

        int step = (from.x == to.x) ? ((to.y > from.y) ? 1 : -1) : ((to.x > from.x) ? 1 : -1);
        int currentX = from.x + ((from.x == to.x) ? 0 : step);
        int currentY = from.y + ((from.y == to.y) ? 0 : step);

        bool isValidMove = true;

        while (currentX != to.x || currentY != to.y) {
            if (board[currentY][currentX] != ChessPiece::None) {
                isValidMove = false;
                break;
            }

            currentX += (from.x == to.x) ? 0 : step;
            currentY += (from.y == to.y) ? 0 : step;
```

```
    }

    if (isValidMove) {
        if (board[to.y][to.x] == ChessPiece::None) {
            board[from.y][from.x] = ChessPiece::None;
            board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ?
ChessPiece::BlackRook : ChessPiece::WhiteRook;

            for (int i = 0; i < 2; ++i) {
                if (blackRookPositions[i] == from) {
                    blackRookPositions[i] = to;
                    break;
                }
            }

            for (int i = 0; i < 2; ++i) {
                if (whiteRookPositions[i] == from) {
                    whiteRookPositions[i] = to;
                    break;
```

```
            }
        }
        return true;
    }
    else if (board[from.y][from.x]==ChessPiece::WhiteRook && isBlackPiece(to)) {

        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);



        board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ?
ChessPiece::BlackRook : ChessPiece::WhiteRook;


        for (int i = 0; i < 2; ++i) {
            if (whiteRookPositions[i] == from) {
                whiteRookPositions[i] = to;
                break;
            }
        }
        return true;
    }
```

```
else if (board[from.y][from.x] == ChessPiece::BlackRook && isWhitePiece(to)) {

    board[from.y][from.x] = ChessPiece::None;
    CaptureWhitePiece(to);

    board[to.y][to.x] = (from == blackRookPositions[0] || from == blackRookPositions[1]) ?
ChessPiece::BlackRook : ChessPiece::WhiteRook;

    for (int i = 0; i < 2; ++i) {
      if (blackRookPositions[i] == from) {
        blackRookPositions[i] = to;
        break;
      }
    }

    return true;
  }
```

```
        }
    }
}
bool Chessboard::moveKnight(const Vector2i& from, const Vector2i& to) {

    int dx = abs(to.x - from.x);
    int dy = abs(to.y - from.y);

    if ((dx == 2 && dy == 1) || (dx == 1 && dy == 2)) {

        if (board[to.y][to.x] == ChessPiece::None ) {

            board[from.y][from.x] = ChessPiece::None;
            board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

            for (int i = 0; i < 2; ++i) {
                if (blackKnightPositions[i] == from) {
                    blackKnightPositions[i] = to;
                    break;
                }
```

```
        }

        for (int i = 0; i < 2; ++i) {
            if (whiteKnightPositions[i] == from) {
                whiteKnightPositions[i] = to;
                break;
            }
        }
        return true;
    }
    else if (board[from.y][from.x]==ChessPiece::WhiteKnight && isBlackPiece(to)) {

        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);

        board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

        for (int i = 0; i < 2; ++i) {
            if (whiteKnightPositions[i] == from) {
```

```
            whiteKnightPositions[i] = to;
            break;
        }
    }

    return true;
}
else if (board[from.y][from.x] == ChessPiece::BlackKnight && isWhitePiece(to)) {

    board[from.y][from.x] = ChessPiece::None;
    CaptureWhitePiece(to);

    board[to.y][to.x] = (from == blackKnightPositions[0] || from == blackKnightPositions[1]) ?
ChessPiece::BlackKnight : ChessPiece::WhiteKnight;

    for (int i = 0; i < 2; ++i) {
        if (blackKnightPositions[i] == from) {
            blackKnightPositions[i] = to;
            break;
        }
```

```
            }

            return true;
        }

    }

}
void Chessboard::CaptureBlackPiece(const Vector2i& to) {
    ChessPiece piece = board[to.y][to.x];
    cout << "Captured Black Pieces :" << endl;
    capturedBlackPieces.insert(piece);
    capturedBlackPieces.displayBlack();
    switch (piece) {
    case ChessPiece::BlackPawn:
        for (int i = 0; i < BOARD_SIZE; ++i) {
            if (blackPawnPositions[i] == to) {
                blackPawnPositions[i] = Vector2i(-1, -1);
                break;
            }
```

```
    }
    break;
case ChessPiece::BlackRook:
    for (int i = 0; i < 2; ++i) {
        if (blackRookPositions[i] == to) {
            blackRookPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::BlackKnight:
    for (int i = 0; i < 2; ++i) {
        if (blackKnightPositions[i] == to) {
            blackKnightPositions[i] = Vector2i(-1, -1);
            break;
        }
    }
    break;
case ChessPiece::BlackBishop:
    for (int i = 0; i < 2; ++i) {
        if (blackBishopPositions[i] == to) {
```

```
                blackBishopPositions[i] = Vector2i(-1, -1);
                break;
            }
        }
        break;
    case ChessPiece::BlackQueen:
        for (int i = 0; i < 2; ++i) {
            if (blackQueenPosition == to) {
                blackQueenPosition = Vector2i(-1, -1);
                break;
            }
        }
        break;
    default:

        break;
    }
    moveSoundBuffer.loadFromFile("Images/capture.ogg");
    moveSound.setBuffer(moveSoundBuffer);
    moveSound.play();
}
```

```
void Chessboard::CaptureWhitePiece(const Vector2i& to) {
    ChessPiece piece = board[to.y][to.x];
    capturedWhitePieces.insert(piece);
    cout << "Captured White Pieces : " << endl;
    capturedWhitePieces.displayWhite();
    switch (piece) {
    case ChessPiece::WhitePawn:
        for (int i = 0; i < BOARD_SIZE; ++i) {
            if (whitePawnPositions[i] == to) {
                whitePawnPositions[i] = Vector2i(-1, -1);
                break;
            }
        }
        break;
    case ChessPiece::WhiteRook:
        for (int i = 0; i < 2; ++i) {
            if (whiteRookPositions[i] == to) {
                whiteRookPositions[i] = Vector2i(-1, -1);
                break;
            }
        }
```

```
    break;
  case ChessPiece::WhiteKnight:
    for (int i = 0; i < 2; ++i) {
      if (whiteKnightPositions[i] == to) {
        whiteKnightPositions[i] = Vector2i(-1, -1);
        break;
      }
    }
    break;
  case ChessPiece::WhiteBishop:
    for (int i = 0; i < 2; ++i) {
      if (whiteBishopPositions[i] == to) {
        whiteBishopPositions[i] = Vector2i(-1, -1);
        break;
      }
    }
    break;
  case ChessPiece::WhiteQueen:
    if (whiteQueenPosition == to) {
      whiteQueenPosition = Vector2i(-1, -1);
    }
```

```
        break;
    default:
        break;
    }
    moveSoundBuffer.loadFromFile("Images/capture.ogg");
    moveSound.setBuffer(moveSoundBuffer);
    moveSound.play();
}
bool Chessboard::moveBishop(const Vector2i& from, const Vector2i& to) {

    int dx = abs(to.x - from.x);
    int dy = abs(to.y - from.y);

    if (dx == dy) {

        int stepX = (to.x > from.x) ? 1 : -1;
        int stepY = (to.y > from.y) ? 1 : -1;

        int currentX = from.x + stepX;
        int currentY = from.y + stepY;
```

```
bool isValidMove = true;

while (currentX != to.x && currentY != to.y) {
    if (board[currentY][currentX] != ChessPiece::None) {
        isValidMove = false;
        break;
    }

    currentX += stepX;
    currentY += stepY;
}

if (isValidMove) {
    if (board[to.y][to.x] == ChessPiece::None ) {
        board[from.y][from.x] = ChessPiece::None;
        board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

        for (int i = 0; i < 2; ++i) {
            if (blackBishopPositions[i] == from) {
```

```
            blackBishopPositions[i] = to;
            break;
        }
    }

    for (int i = 0; i < 2; ++i) {
        if (whiteBishopPositions[i] == from) {
            whiteBishopPositions[i] = to;
            break;
        }
    }
    return true;
}
else if (board[from.y][from.x] == ChessPiece::WhiteBishop && isBlackPiece(to)) {

    board[from.y][from.x] = ChessPiece::None;
    CaptureBlackPiece(to);
    board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

    for (int i = 0; i < 2; ++i) {
```

```
            if (whiteBishopPositions[i] == from) {
                whiteBishopPositions[i] = to;
                break;
            }
        }

        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::BlackBishop && isWhitePiece(to)) {

        board[from.y][from.x] = ChessPiece::None;
        CaptureWhitePiece(to);
        board[to.y][to.x] = (from == blackBishopPositions[0] || from == blackBishopPositions[1]) ?
ChessPiece::BlackBishop : ChessPiece::WhiteBishop;

        for (int i = 0; i < 2; ++i) {
            if (blackBishopPositions[i] == from) {
                blackBishopPositions[i] = to;
                break;
            }
        }
```

```
            return true;
        }

    }

    }
}
bool Chessboard::moveQueen(const Vector2i& from, const Vector2i& to) {

    if (from.x == to.x || from.y == to.y || abs(to.x - from.x) == abs(to.y - from.y)) {

        int stepX = (from.x == to.x) ? 0 : ((to.x > from.x) ? 1 : -1);
        int stepY = (from.y == to.y) ? 0 : ((to.y > from.y) ? 1 : -1);

        int currentX = from.x + stepX;
        int currentY = from.y + stepY;

        bool isValidMove = true;
```

```
        while ((from.x == to.x && currentY != to.y) || (from.y == to.y && currentX != to.x) || (currentX
!= to.x && currentY != to.y)) {
            if (board[currentY][currentX] != ChessPiece::None) {
                isValidMove = false;
                break;
            }

            currentX += stepX;
            currentY += stepY;
        }

        if (isValidMove) {
            if (board[to.y][to.x] == ChessPiece::None ) {
                ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen :
ChessPiece::WhiteQueen;

                board[from.y][from.x] = ChessPiece::None;
                board[to.y][to.x] = movedPiece;

                if (from == blackQueenPosition) {
```

```
            blackQueenPosition = to;
        }

        if (from == whiteQueenPosition) {
            whiteQueenPosition = to;
        }
        return true;
    }
    else if (board[from.y][from.x] == ChessPiece::WhiteQueen && isBlackPiece(to)) {
        ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen :
ChessPiece::WhiteQueen;
        board[from.y][from.x] = ChessPiece::None;
        CaptureBlackPiece(to);
        board[to.y][to.x] = movedPiece;

        if (from == whiteQueenPosition) {
            whiteQueenPosition = to;
        }

        return true;
    }
```

```
        else if (board[from.y][from.x] == ChessPiece::BlackQueen && isWhitePiece(to)) {
            ChessPiece movedPiece = (from == blackQueenPosition) ? ChessPiece::BlackQueen :
ChessPiece::WhiteQueen;
            board[from.y][from.x] = ChessPiece::None;
            CaptureWhitePiece(to);
            board[to.y][to.x] = movedPiece;
            if (from == blackQueenPosition) {
                blackQueenPosition = to;
            }

            return true;
        }

    }

  }
}
bool Chessboard::moveKing(const Vector2i& from, const Vector2i& to) {

    if (abs(to.x - from.x) <= 1 && abs(to.y - from.y) <= 1) {
```

```
        if (board[to.y][to.x] == ChessPiece::None) {

            ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing :
ChessPiece::WhiteKing;

            board[from.y][from.x] = ChessPiece::None;
            board[to.y][to.x] = movedPiece;

            if (from == blackKingPosition) {
                blackKingPosition = to;

            }

            if (from == whiteKingPosition) {
                whiteKingPosition = to;
            }
            return true;
        }
        else if (board[from.y][from.x] == ChessPiece::WhiteKing && isBlackPiece(to)) {
            ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing :
ChessPiece::WhiteKing;
```

```
            board[from.y][from.x] = ChessPiece::None;
            CaptureBlackPiece(to);
            board[to.y][to.x] = movedPiece;

            if (from == whiteKingPosition) {
                whiteKingPosition = to;
            }

            return true;
        }
        else if (board[from.y][from.x] == ChessPiece::BlackKing && isWhitePiece(to)) {
            ChessPiece movedPiece = (from == blackKingPosition) ? ChessPiece::BlackKing :
ChessPiece::WhiteKing;
            board[from.y][from.x] = ChessPiece::None;
            CaptureWhitePiece(to);
            board[to.y][to.x] = movedPiece;
            if (from == blackKingPosition) {
                blackKingPosition = to;
            }
```

```
            return true;
        }
    }
}
bool Chessboard::movePiece(const Vector2i& from, const Vector2i& to) {
    bool check = false;
    moveSoundBuffer.loadFromFile("Images/move.ogg");
    moveSound.setBuffer(moveSoundBuffer);
        ChessPiece piece = board[from.y][from.x];
        if ((isWhiteTurn && isWhitePiece(from)) || (!isWhiteTurn && isBlackPiece(from))) {
            switch (piece) {
            case ChessPiece::BlackPawn:
            case ChessPiece::WhitePawn:
                check = movePawn(from, to);
                break;
            case ChessPiece::BlackRook:
            case ChessPiece::WhiteRook:
                check =  moveRook(from, to);
                break;
            case ChessPiece::BlackKnight:
            case ChessPiece::WhiteKnight:
```

```
    check =  moveKnight(from, to);
    break;
  case ChessPiece::BlackBishop:
  case ChessPiece::WhiteBishop:
    check = moveBishop(from, to);
    break;
  case ChessPiece::BlackQueen:
  case ChessPiece::WhiteQueen:
    check =  moveQueen(from, to);
    break;
  case ChessPiece::BlackKing:
  case ChessPiece::WhiteKing:
    check =  moveKing(from, to);
    break;
  default:
    check = false;
    break;
  }
  if (check) {
    moveSound.play();
  }
```

```
        }

        return check;
    }
    bool Chessboard::isCheckmate() const {

        ChessPiece currentKing = (isWhiteTurn) ? ChessPiece::WhiteKing : ChessPiece::BlackKing;
        Vector2i kingPosition;

        if (!findPiecePosition(currentKing, kingPosition)) {

            cout << "Error: King not found!" << endl;
            return false;
        }

        for (int toY = 0; toY < BOARD_SIZE; ++toY) {
            for (int toX = 0; toX < BOARD_SIZE; ++toX) {
                const Vector2i to(toX, toY);
```

```
        Chessboard* tempBoard = new Chessboard(*this);
        tempBoard->movePiece(kingPosition, to);


        ChessPiece opponentKing = (tempBoard->isWhiteTurn) ? ChessPiece::WhiteKing :
ChessPiece::BlackKing;
        Vector2i opponentKingPosition;


        if (tempBoard->findPiecePosition(opponentKing, opponentKingPosition)) {
            if (tempBoard->isKingInCheckAfterMove(opponentKingPosition, to)) {


                cout << "Move allows the king to escape check. Not checkmate." << endl;
                delete tempBoard;
                return false;
            }
        }
        else {
            cout << "Error: Opponent's king not found!" << endl;
        }
```

```
            delete tempBoard;
        }
    }


    cout << "Checkmate: No moves available to get the king out of check." << endl;
    return true;
}

bool Chessboard::isKingInCheckAfterMove(const Vector2i& from, const Vector2i& to) const {

    Chessboard tempBoard(*this);
    tempBoard.movePiece(from, to);



    ChessPiece opponentKing = (tempBoard.isWhiteTurn) ? ChessPiece::WhiteKing :
ChessPiece::BlackKing;
    Vector2i opponentKingPosition;

    if (!tempBoard.findPiecePosition(opponentKing, opponentKingPosition)) {
```

```
        return false;
    }


    for (int i = 0; i < BOARD_SIZE; ++i) {
        for (int j = 0; j < BOARD_SIZE; ++j) {
            const Vector2i currentPlayerPiece(j, i);

            if ((tempBoard.isWhiteTurn && tempBoard.isWhitePiece(currentPlayerPiece)) ||
(!tempBoard.isWhiteTurn && tempBoard.isBlackPiece(currentPlayerPiece))) {

                if
(tempBoard.canPieceAttack(tempBoard.board[currentPlayerPiece.y][currentPlayerPiece.x],
currentPlayerPiece, opponentKingPosition)) {
                    return true;
                }
            }
        }
    }

    return false;
```

```
}
int main() {

    RenderWindow window(VideoMode(WINDOW_SIZE, WINDOW_SIZE), "Chess");
    window.setFramerateLimit(60);

    Chessboard chessboard;

    while (window.isOpen()) {
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == Event::Closed) {
                window.close();
            }

            chessboard.handleInput(event);
        }

        window.clear();
        chessboard.draw(window);
        window.display();
```

```
    }

    return 0;
}
```

## The Game