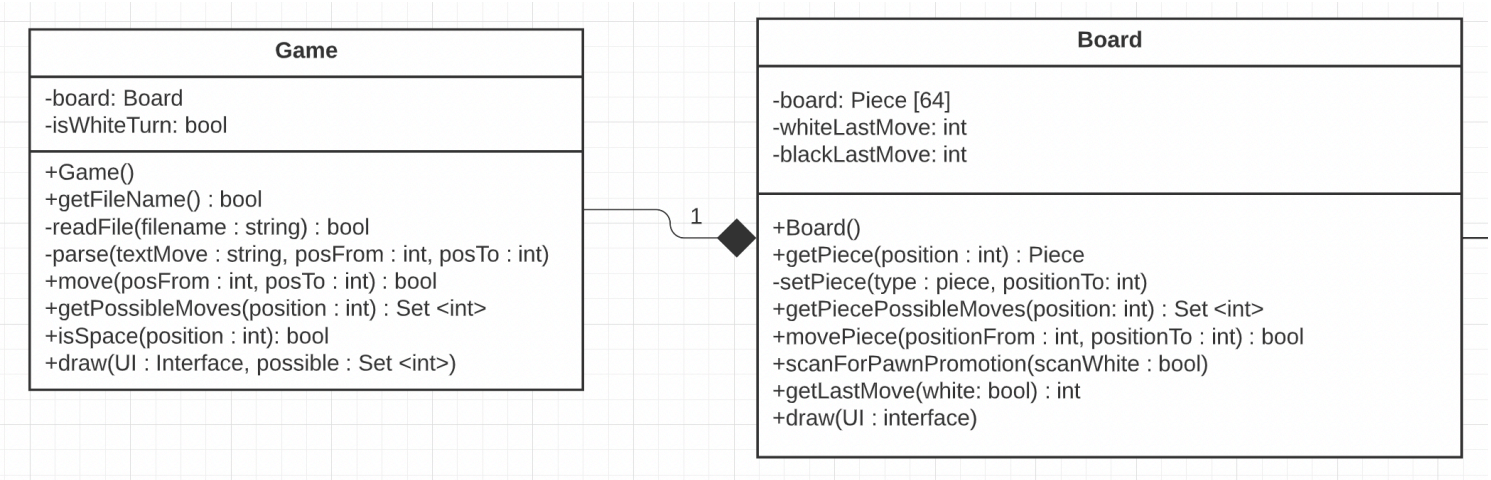
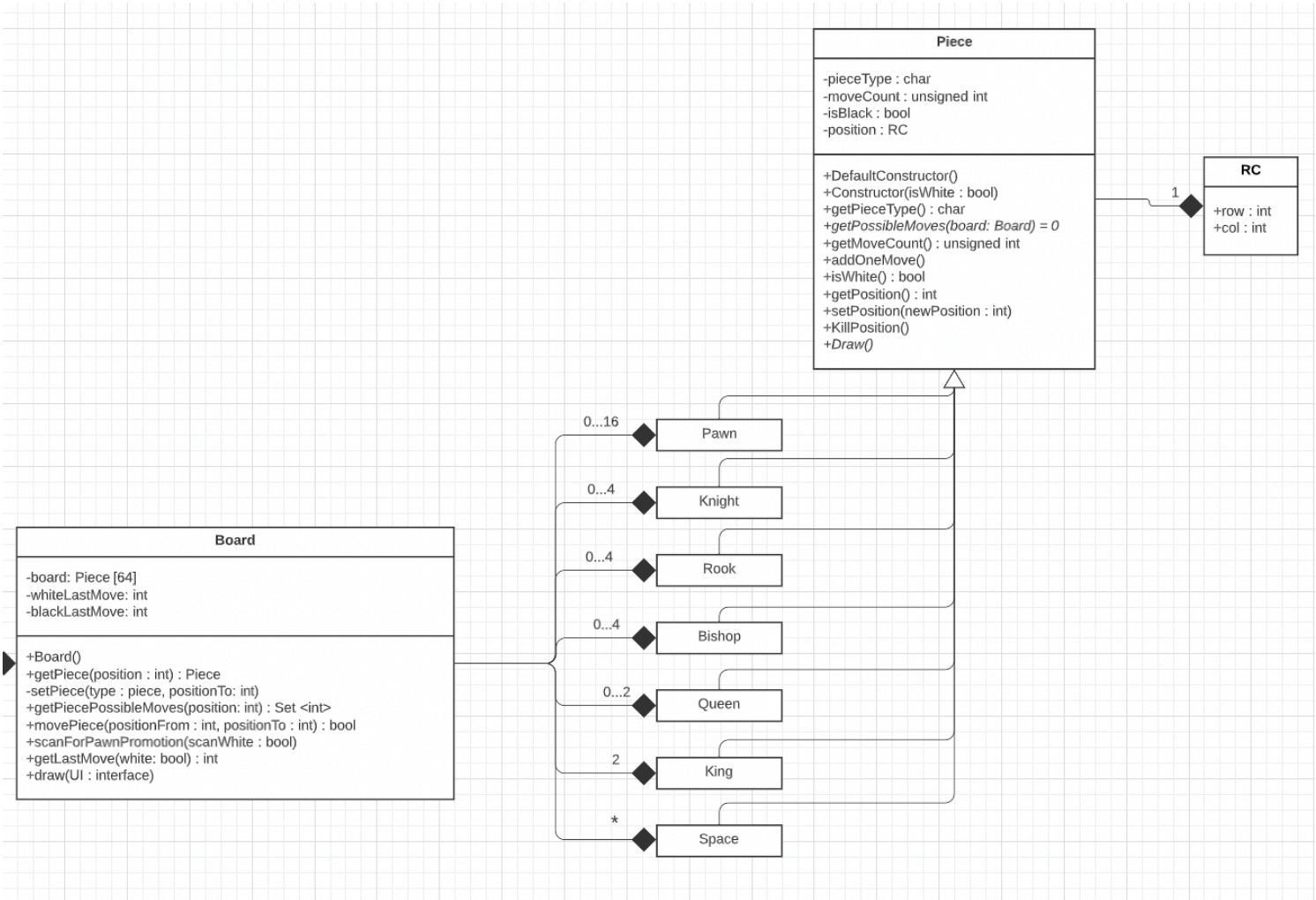
# LAB 02: Chess Design

## Class Diagrams:

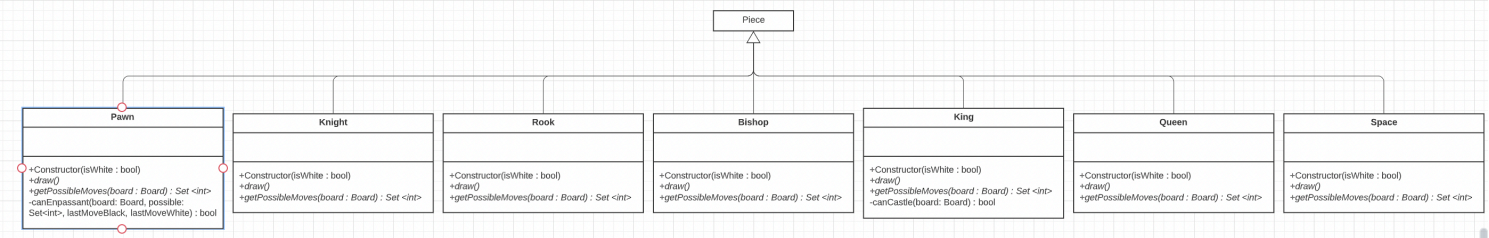
The Game and Board class relation.



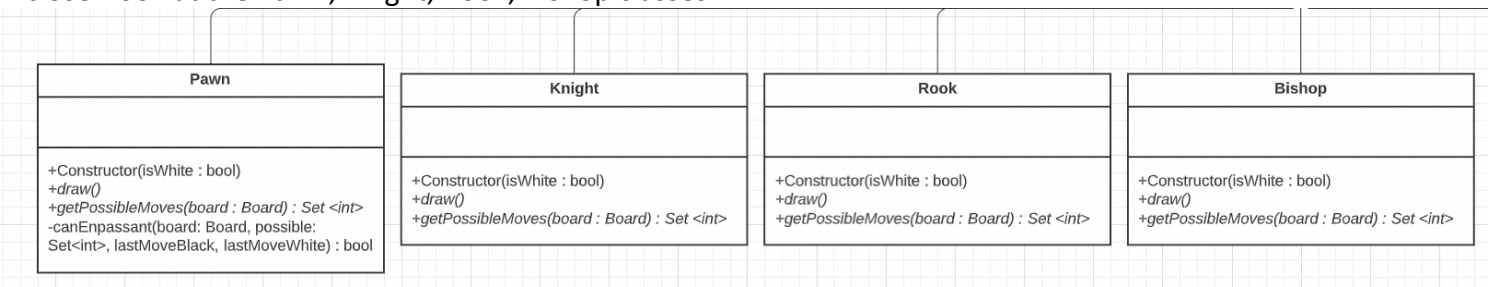

The Board and Piece class relation.



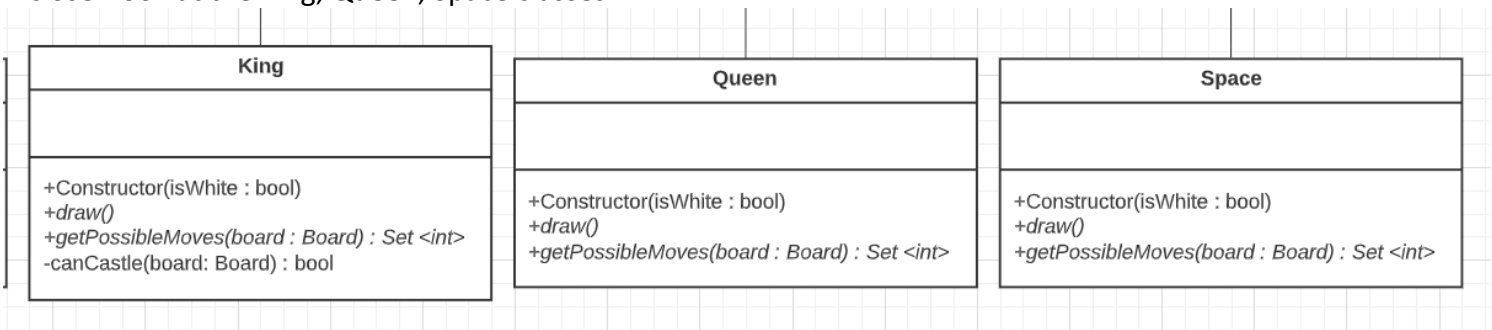
The Derived Piece Classes: Pawn, Knight, Rook, Bishop, King, Queen, Space.




A closer look at the Pawn, Knight, Rook, Bishop classes.

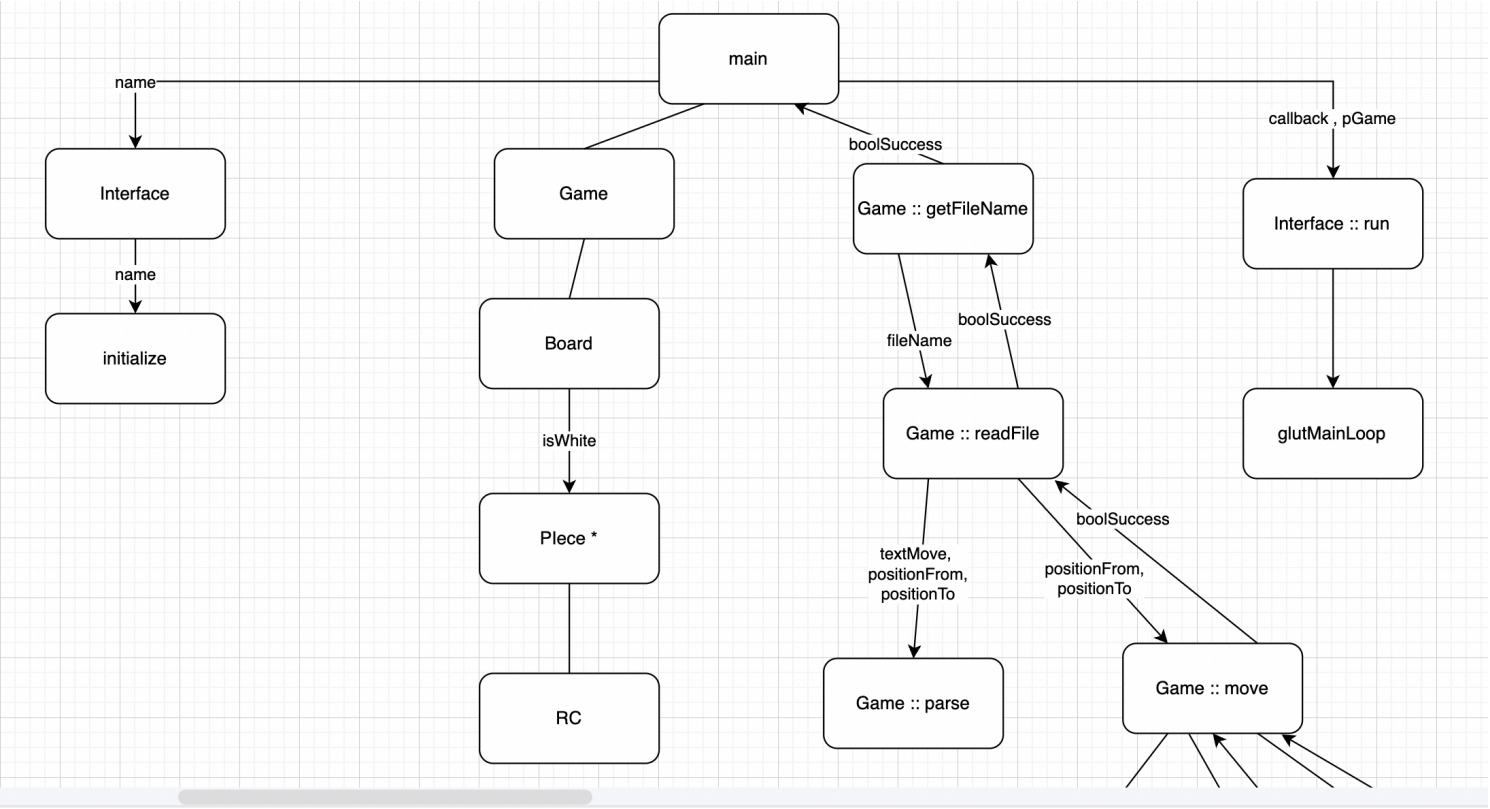



A closer look at the King, Queen, Space classes.

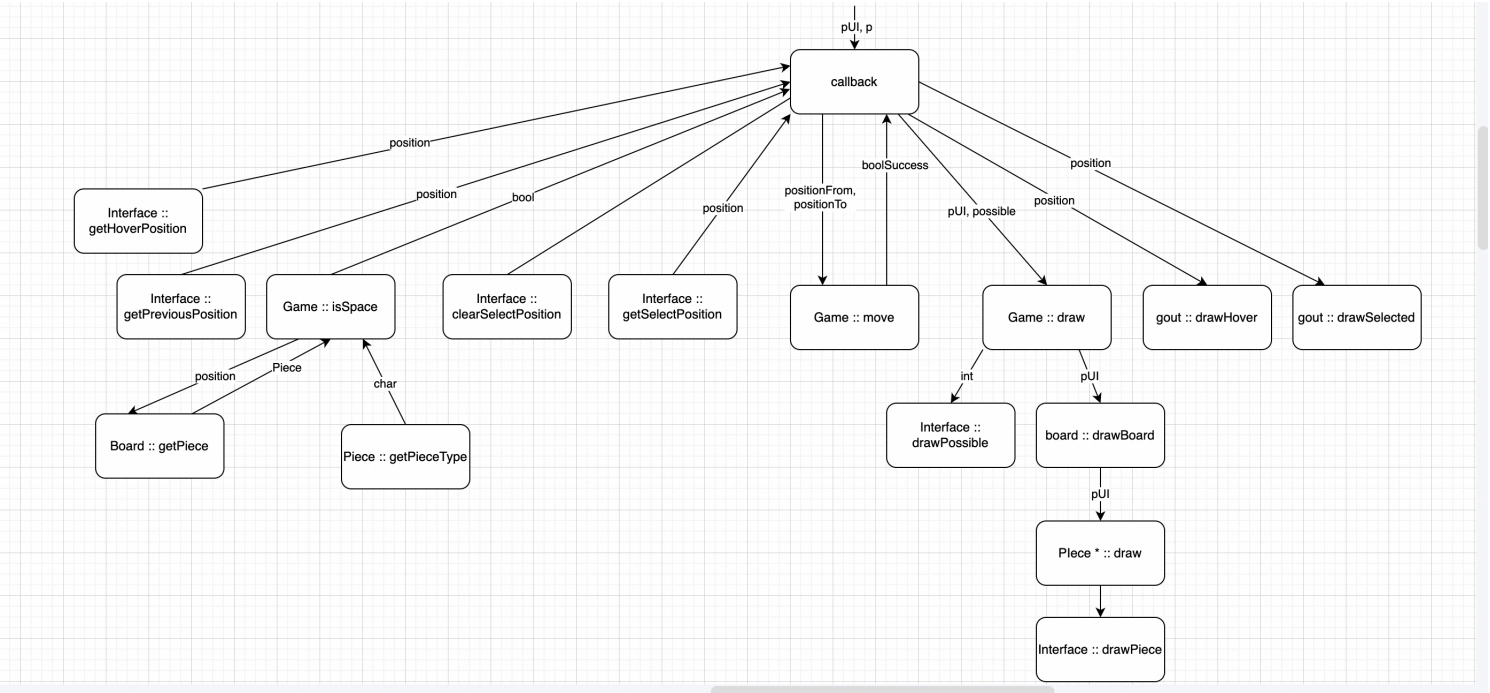



## Structure Chart:

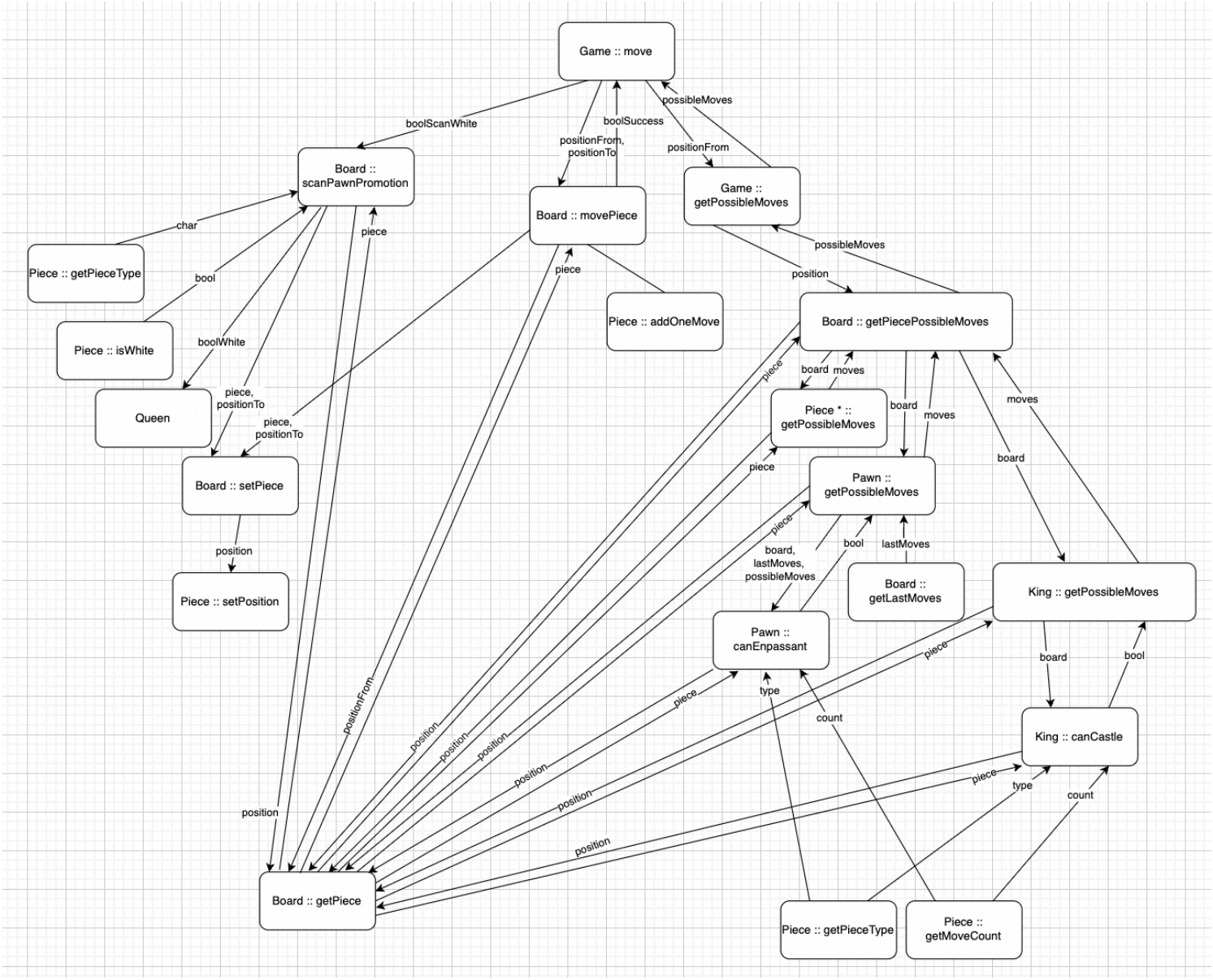
Everything that happens before OpenGL is called.




`glutMainLoop()` runs the OpenGL graphics loop; we do not get to see how it works. It does periodically call `callback()`.



A closer look at `Game :: move()`



# Pseudocode:

## Castling:

```
King :: canCastle(board)

    // Set the correct positions to check based on King color
    IF isBlack
        // Black King
        correctKingPosition <- 60
        correctRookPosition <- 63
        rookShouldBeWhite <- FALSE
    ELSE
        // White King
        correctKingPosition <- 4
        correctRookPosition <- 7
        rookShouldBeWhite <- TRUE

    // First check if the King is in the correct position and has not moved
    IF getPosition() = correctKingPosition AND moveCount = 0

        // Next check that the next two positions are spaces
        nextSquare <- board.getPiece(correctKingPosition + 1)
        twoSquaresOver <- board.getPiece(correctKingPosition + 2)
        IF nextSquare.getPieceType() = ‘s‘ AND twoSquaresOver.getPieceType() = ‘s‘

            // Lastly check that the rook is in the correct position and has not moved
            piece <- board.getPiece(correctRookPosition)
            IF piece.getPieceType() = 'r' AND piece.getMoveCount() = 0 AND piece.isWhite() = rookShouldBeWhite

                // return true to the client; the king can castle.
                // In this case the only client is King :: getPossbileMoves()
                // and having recieved the return value true, getPossibleMoves()
                // will add the castle move to the set of possible moves for the King.
                RETURN TRUE

            ELSE
                RETURN FALSE
        ELSE
            RETURN FALSE
    ELSE
        RETURN FALSE

```

## En-Passant:

```
Pawn :: canEnPassant(*possibleMoves, lastWhiteMove, lastBlackMove ,Board)
    // Set values for the current pawn
    SET startPos <- piece.getPosition()

    // Checking if piece is on the sides
    switch(startPos)
    case: 24
        SET left = FALSE
        SET RIGHT = TRUE
    case: 31
        SET left = TRUE
        SET RIGHT = FALSE
    case: 40
        SET left = FALSE
        SET RIGHT = TRUE
    case: 47
        SET left = TRUE
        SET RIGHT = FALSE
    Default:
        SET left = TRUE
        SET RIGHT = TRUE

    // Checking for black side
    IF isBlack
        IF left
            board.getPieceType(position + 1) = 'p' && piece.getPiece(position + 1).getMoveCount() = 1;
            IF (startPos + 1) == lastWhiteMove
                RETURN lastWhiteMove - 8
        IF RIGHT
            piece.getPieceType(position + 1) = 'p' && piece.getMoveCount() = 1;
            IF (startPos + 1) == lastWhiteMove
                RETURN lastWhiteMove - 8
    // Checking for White side
    ELSE
        IF left
            board.getPieceType(position - 1) = 'P' && board.getPiece(position - 1).getMoveCount() = 1;
            IF (startPos + 1) == lastBlackMove
                RETURN lastBlackMove + 8
        IF RIGHT
            board.getPieceType(position - 1) = 'P' && board.getPiece(position - 1).getMoveCount() = 1;
            IF (startPos + 1) == lastBlackMove
                RETURN lastBlackMove + 8

    RETURN FALSE
```

## Pawn Promotion:

```
Board :: scanForPawnPromotion(scanWhite)
    IF scanWhite
        start <- 56
        stop <- 63
    ELSE
        start <- 0
        stop <- 7

    FOR position <- start...stop
        IF board.getPiece(position) = 'p' and board.getPiece(position).isWhite() = scanWhite
            q <- Queen(scanWhite)
            board.setPiece(Q, position)
            RETURN

```

## Test Cases:

```
#include <iostream>
#include <cassert>

using namespace std;

void testBoardMovePiece()
{
    // Setup
    Board board;
    Pawn pawn(true);
    // Exercise
    board.movePiece(pawn, 16);
    // Verify
    assert(pawn.position == 16);
    assert(board[16].getPieceType() = 'p');
}   // Teardown

void testBoardScanPawnPromotion()
{
    // Setup
    Pawn pawn(true);
    pawn.position = 58;
    Board board;
    // Exercise
    board.scanPawnPromotion(true);
    // Verify
    assert(board[58].getPieceType() = 'q');
}   // Teardown

void testBoardScanPawnNoPromotion()
{
    // Setup
    Pawn pawn(true);
    pawn.position = 32;
    // Exercise
    board.scanPawnPromotion(true);
    // Verify
    assert(board[32].getPieceType() = 'p');
}   // Teardown

void testBoardGetPieceType()
{
    // Setup
    Knight knight(true);
    knight.position = 20;
    // Exercise
    // Verify
    assert(getPieceType(20) == 'n');
}   // Teardown

void testBoardGetPossibleMoves()
{
    // Setup
    Pawn pawn(true);
    pawn.position = 8;
    // Exercise
    // Verify
    assert(pawn.getPossibleMoves() == {16});
}   // Teardown

void testBoardDefault()
{
    // Setup
    Board board;
    // Exercise
    // Verify
    assert(board == {
      'r', 'n', 'b', 'q', 'k', 'b', 'n', 'r',
      'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p',
      ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
      ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
      ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
      ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
      // ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
      'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P',
      'R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'
    });
}   // Teardown
```

```cpp
#include <iostream>
#include <cassert>

using namespace std;


void kingTestPossibleMoves()
{
    // Setup
    King king(true);
    king.position = 35;
    // Exercise
    // Verify
    assert(king.getPossibleMoves() ==
    {26, 27, 28, 34, 36, 42, 43, 44});
}   // Teardown

void kingTestDefaultWhite()
{
    // Setup
    King king(true);
    // Exercise
    // Verify
    assert(king.position == 4);
}   // Teardown

void kingTestDefaultBlack()
{
    // Setup
    King king(true);
    // Exercise
    // Verify
    assert(king.position == 60);
}   // Teardown

```

```cpp
void kingTestCanCastle()
{
    // Setup
    King king(true);
    Rook rook(true);
    // Exercise
    // Verify
    assert(king.moves == 0);
    assert(rook.moves == 0);
    assert(king.position == 60);
    assert(rook.position == 63);
    assert(twoSquaresOver.getPieceType() == "s");
    assert(nextSquare.getPieceType() == "s");
}   // Teardown

void kingTestCantCastle()
{
    // Setup
    King king(true);
    Rook rook(true);
    // Exercise
    // Verify
    assert(king.moves == 0);
    assert(rook.moves == 0);
    assert(king.position == 60);
    assert(rook.position == 63);
    assert(twoSquaresOver.getPieceType() != "s");
    assert(nextSquare.getPieceType() != "s");
}   // Teardown

void kingTestAddOneForward()
{
    // Setup
    King king(true);
    // Exercise
    king.addOneForward();
    // Verify
    assert(king.position == 12);
}   // Teardown

```

```cpp
void kingTestGetMoveCount()
{
    // Setup
    King king(true);
    // Exercise
    king.addOneForward();
    // Verify
    assert(king.moves == 1);
}   // Teardown

void kingTestKillPositionForward()
{
    // Setup
    King king(true);
    king.position = 20;
    Pawn pawn(false);
    pawn.position = 28;
    // Exercise
    king.killPosition(28);
    // Verify
    assert(king.position == 28);
}   // Teardown
```

```cpp
void kingTestKillDiagonalLeft()
{
    // Setup
    King king(true);
    pawn.position = 20;
    Pawn pawn(false);
    pawn.position = 27;
    // Exercise
    king.killPosition(27);
    // Verify
    assert(king.position == 27);
}   // Teardown
```