תיעוד שחמט – Logic

תוכנת הלוגיקה של השחמט מתקשרת עם תוכנת עיבוד התמונה דרך sockets. הלוגיקה בנויה מכמה מחלקות מרכזיות: Board שאחראית על שמירת המצב של הלוח והצגתו, HandleGame שאחראית על הרצת המשחק וקבלת הפקודות מה-frontend ו- Piece – מחלקה אבסטרקטית שממנה יוצאות כל מחלקות החלקים (לדוגמא: Queen, Pawn…). כל מחלקה של חלק אחראית על התזוזה של חלק זה.

אחרי שה-frontend שולח בקשה של תזוזה ל-backend, ה-backend יחזיר לו את אחד מהקודים הבאים:

* 0 – מהלך תקין
* 1 – מהלך תקין, התבצעה תזוזה שגרמה שח על היריב
* 2 – מהלך לא תקין, במשבצת המקור אין כלי של השחקן הנוכחי
* 3 – מהלך לא תקין, במשבצת היעד קיים כלי של השחקן הנוכחי
* 4 – מהלך לא תקין, בעקבות התזוזה יגרם שח על השחקן הנוכחי
* 5 – מהלך לא תקין, אינדקסים של המשבצות אינם חוקיים
* 6 – מהלך לא תקין, תזוזה לא חוקית של כלי
* 7 – מהלך לא תקין, משבצת המקור ומשבצת היעד זהות
* 8 – מהלך תקין, התבצע שחמט!
* 9 – הצרחה

isInCheck:

הפונקציה עוברת על כל הלוח ובודקת האם יש חלק שיכול לאכול את המלך של השחקן הנגדי, אם כן: אז הוא נמצא בשח.

isInMate:

הפונקציה בודקת האם המלך נמצא בשח והאם אין לו מהלכים חוקיים, אם שני התנאים האלה מתקיימים, המלך נמצא בשחמט והמשחק נגמר.

Piece.h:

1. #ifndef \_\_PIECE\_H\_\_
2. #define \_\_PIECE\_H\_\_
4. #include <iostream>
5. #include "HandleGame.h"
7. #define BLACK 1
8. #define WHITE 0
10. class Board;
12. class Piece
13. {
14. protected:
15. char \_type;
16. bool \_hasMoved;
18. /// <summary>
19. /// This function handles the checking of codes SELF\_CHECK and CHECK.
20. /// </summary>
21. /// <param name="board"> The game board </param>
22. /// <param name="src"> The position of the piece that the player wants to move </param>
23. /// <param name="dest"> Where the player wants to move the piece to </param>
24. /// <returns></returns>
25. static int HandleCheckCodes(Board& board, const int\* src, const int\* dest);
27. public:
28. /// <summary>
29. /// This function is the constructor of the class Piece.
30. /// </summary>
31. /// <param name="type"> The type of the piece </param>
32. Piece(const char type);
34. /// <summary>
35. /// This function sets a new type to \_type.
36. /// </summary>
37. /// <param name="type"> The new type </param>
38. void setType(char type);
40. /// <summary>
41. /// This function gets the type of the piece.
42. /// </summary>
43. /// <returns> The piece type </returns>
44. char getType() const;
46. /// <summary>
47. /// This function checks if a move that the player wants to play is valid.
48. /// </summary>
49. /// <param name="src"> The position of the piece that the player wants to move </param>
50. /// <param name="dest"> Where the player wants to move the piece to </param>
51. /// <param name="board"> The game board </param>
52. /// <returns> A number code between 0 - 8 that indicates a certain outcome of the checking </returns>
53. static int isValidMove(const int\* src, const int\* dest, Board& board);
55. /// <summary>
56. /// This function checks if a move that is proformed on a piece matches the way that the piece moves.
57. /// </summary>
58. /// <param name="src"> The position of the piece that the player wants to move </param>
59. /// <param name="dest"> Where the player wants to move the piece to </param>
60. /// <param name="board"> The game board </param>
61. /// <returns> If the move is valid or not </returns>
62. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const = 0;
64. bool getHasMoved() const;
65. void setHasMoved(const bool hasMoved);
66. };
68. #endif // !\_\_PIECE\_H\_\_

 Piece.cpp:

1. #include "Piece.h"
3. Piece::Piece(const char type)
4. {
5. this->\_type = type;
6. this->\_hasMoved = false;
7. }
9. void Piece::setType(char type)
10. {
11. this->\_type = type;
12. }
14. char Piece::getType() const
15. {
16. return this->\_type;
17. }
19. int Piece::isValidMove(const int\* src, const int\* dest, Board& board)
20. {
21. Piece\* srcPiece = board.getBoard()[src[0]][src[1]];
22. Piece\* destPiece = board.getBoard()[dest[0]][dest[1]];
23. bool srcColor = (isupper(srcPiece->getType())) ? WHITE : BLACK;
24. bool destColor = (isupper(destPiece->getType())) ? WHITE : BLACK;
26. if (srcColor != HandleGame::getCurrentPlayer() || srcPiece->getType() == EMPTY\_SQUARE)
27. {
28. return INVALID\_SRC;
29. }
31. if (destPiece->getType() != EMPTY\_SQUARE && destColor == HandleGame::getCurrentPlayer())
32. {
33. return INVALID\_DEST;
34. }
36. if (!srcPiece->isValidPieceMove(src, dest, board))
37. {
38. return INVALID\_MOVE;
39. }
41. return Piece::HandleCheckCodes(board, src, dest);
42. }
44. int Piece::HandleCheckCodes(Board& board, const int\* src, const int\* dest)
45. {
46. Piece\* destPiece = board.getBoard()[dest[0]][dest[1]];
47. int code = VALID;
48. int checkValue = NO\_CHECK;
49. int currentPlayer = HandleGame::getCurrentPlayer();
51. board.movePiece(src, dest, false);
52. checkValue = board.isInCheck();
54. if (currentPlayer != checkValue && checkValue != NO\_CHECK)
55. {
56. code = CHECK;
57. }
58. if (currentPlayer == checkValue && checkValue != NO\_CHECK)
59. {
60. code = SELF\_CHECK;
61. }
62. if (checkValue == 2)
63. {
64. code = SELF\_CHECK;
65. }
67. board.movePiece(dest, src, true);
68. board.getBoard()[dest[0]][dest[1]] = destPiece;
70. return code;
71. }
73. bool Piece::getHasMoved() const
74. {
75. return this->\_hasMoved;
76. }
78. void Piece::setHasMoved(const bool hasMoved)
79. {
80. this->\_hasMoved = hasMoved;
81. }

HandleGame.h:

1. #pragma once
2. #include <iostream>
3. #include "Board.h"
4. #include <WinSock2.h>
5. #include <string>
6. #include <vector>
7. class Board;
8. class Piece;
10. #define BLACK 1
11. #define WHITE 0
12. #define INIT\_STR "rnbqkbnrpppppppp################################PPPPPPPPRNBQKBNR0"
14. enum returnCodes {VALID, CHECK, INVALID\_SRC, INVALID\_DEST, SELF\_CHECK, INVALID\_INDEX, INVALID\_MOVE, SAME\_SRC\_AND\_DEST, CHECKMATE, CASTLE};
16. class HandleGame
17. {
18. private:
19. static bool \_currentPlayer; // black - true, white - false
21. public:
22. /// <summary>
23. /// This function returns the current player.
24. /// </summary>
25. /// <returns> The current player (black or white) </returns>
26. static bool getCurrentPlayer();
28. /// <summary>
29. /// This function sets the current player.
30. /// </summary>
31. /// <param name="color"> The current player (black or white) </param>
32. static void setCurrentPlayer(int color);
34. /// <summary>
35. /// This function runs the main loop of the game.
36. /// </summary>
37. /// <param name="board"> The game board </param>
38. /// <param name="pipe"> A connection between the backend and the frontend </param>
39. static void startGame(Board\* board, SOCKET\* pipe);
41. /// <summary>
42. /// This function handles a player's turn:
43. /// Gets his move, checks if it's valid and returns the fitting code.
44. /// </summary>
45. /// <param name="instruction"> The instruction from the graphics </param>
46. /// <param name="board"></param>
47. /// <returns> The code to send to the graphics </returns>
48. static int handleTurn(std::string instruction, Board& board);
50. /// <summary>
51. /// This function changes the current player according to the number of rounds.
52. /// </summary>
53. /// <param name="code"> A code that was sent to the graphics at the end of a turn </param>
54. /// <param name="roundCounter"> A pointer to the amount of rounds that have been played </param>
55. static void changeCurrentPlayer(const int code, int\* roundCounter);
57. static std::string receiveMsg(SOCKET sock, int len, int offset);
58. static void sendMsg(SOCKET sock, const char\*\* buffer, int len);
59. };

HandleGame.cpp:

1. #include "HandleGame.h"
2. #include "Rook.h"
3. #include "Pawn.h"
4. #include "Queen.h"
5. #include "King.h"
6. #include "Bishop.h"
7. #include "Knight.h"
8. #include "Piece.h"
10. const std::string castleCodes[4] = { "e1g1", "e1c1", "e8g8", "e8c8" };
11. bool HandleGame::\_currentPlayer;
13. bool HandleGame::getCurrentPlayer()
14. {
15. return \_currentPlayer;
16. }
18. void HandleGame::setCurrentPlayer(int color)
19. {
20. HandleGame::\_currentPlayer = color;
21. }
23. void HandleGame::startGame(Board\* board, SOCKET\* pipe)
24. {
25. int roundCounter = 0;
26. int code = 0;
27. char strCode[2] = { 0 };
28. std::string msgToGraphics = INIT\_STR;
29. const char\* pStr = msgToGraphics.c\_str();
30. std::string msgFromGraphics = "";
32. HandleGame::setCurrentPlayer(WHITE);
33. board->printBoard();
35. while (msgFromGraphics != "quit")
36. {
37. if (msgFromGraphics == "reset")
38. {
39. roundCounter = 0;
40. delete board;
41. board = new Board(INIT\_STR);
42. HandleGame::setCurrentPlayer(WHITE);
43. board->printBoard();
44. msgToGraphics = INIT\_STR;
45. pStr = msgToGraphics.c\_str();
46. sendMsg(\*pipe, &pStr, msgToGraphics.length());
47. msgFromGraphics = receiveMsg(\*pipe, 4, 0);
48. continue;
49. }
50. msgFromGraphics = receiveMsg(\*pipe, 4, 0);
51. code = HandleGame::handleTurn(msgFromGraphics, \*board);
53. strCode[0] = code + '0';
54. strCode[1] = '\0';
56. msgToGraphics = strCode;
57. pStr = msgToGraphics.c\_str();
58. sendMsg(\*pipe, &pStr, msgToGraphics.length());
60. HandleGame::changeCurrentPlayer(code, &roundCounter);
61. }
62. }
64. int HandleGame::handleTurn(std::string instruction, Board& board)
65. {
66. int\* src = board.convertIndex(instruction.substr(0, 2));
67. int\* dest = board.convertIndex(instruction.substr(2, 2));
68. char srcChar = board.getBoard()[src[0]][src[1]]->getType();
69. char destChar = board.getBoard()[dest[0]][dest[1]]->getType();
70. const std::string\* str = std::find(std::begin(castleCodes), std::end(castleCodes), instruction);
71. int code = 0;
73. if (tolower(srcChar) == B\_KING && str != std::end(castleCodes))
74. {
75. code = board.castle(src, dest);
76. }
77. else
78. {
79. code = board.getBoard()[src[0]][src[1]]->isValidMove(src, dest, board);
80. }
82. if (code == VALID || code == CHECK || code == CASTLE)
83. {
84. board.getBoard()[src[0]][src[1]]->setHasMoved(true);
85. board.movePiece(src, dest, true);
87. if (srcChar == B\_PAWN || srcChar == W\_PAWN)
88. {
89. if (dest[0] == 7 || dest[0] == 0)
90. {
91. delete board.getBoard()[dest[0]][dest[1]];
92. board.getBoard()[dest[0]][dest[1]] = (dest[0] == 7) ? new Queen(B\_QUEEN) : new Queen(W\_QUEEN);
93. }
94. }
96. if (code == CHECK)
97. {
98. code = board.isInMate(!HandleGame::\_currentPlayer);
99. }
100. }
101. board.printBoard();
103. delete[] src;
104. delete[] dest;
105. return code;
106. }
108. void HandleGame::changeCurrentPlayer(const int code, int\* roundCounter)
109. {
110. if (code == VALID || code == CHECK || code == CASTLE)
111. {
112. (\*roundCounter)++;
113. }
114. if ((\*roundCounter) % 2 == 0 && (code == VALID || code == CHECK || code == CASTLE))
115. {
116. HandleGame::setCurrentPlayer(WHITE);
117. }
118. else if (code == VALID || code == CHECK || code == CASTLE)
119. {
120. HandleGame::setCurrentPlayer(BLACK);
121. }
122. }
124. std::string HandleGame::receiveMsg(SOCKET sock, int len, int offset)
125. {
126. if (len == 0)
127. {
128. return (char\*)"";
129. }
131. char\* data = new char[len + 1];
132. int res = recv(sock, data, len, 0);
134. if (res == INVALID\_SOCKET)
135. {
136. std::string s = "Error while recieving from socket: ";
137. s += std::to\_string(sock);
138. throw std::exception(s.c\_str());
139. }
141. data[len] = 0;
142. return data;
143. }
145. void HandleGame::sendMsg(SOCKET sock, const char\*\* buffer, int len)
146. {
147. if (send(sock, \*buffer, len, 0) == INVALID\_SOCKET)
148. {
149. throw std::exception("Error while sending message to client");
150. }
151. }

 EmptySquare.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
4. #include "Board.h"
5. class Board;
7. class EmptySquare : public Piece
8. {
9. public:
10. /// <summary>
11. /// This function is the constructor of the class EmptySquare.
12. /// </summary>
13. /// <param name="type"> An empty square char - '#' </param>
14. EmptySquare(char type);
16. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
17. };

EmptySquare.cpp:

1. #include "EmptySquare.h"
3. EmptySquare::EmptySquare(char type) : Piece(type)
4. {
5. }
7. bool EmptySquare::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
8. {
9. return false;
10. }

 Board.h:

1. #ifndef \_\_BOARD\_H\_\_
2. #define \_\_BOARD\_H\_\_
4. #include <iostream>
6. #define BOARD\_SIDE 8
7. #define BOARD\_INDEX 2
8. #define FIRST\_LETTER 'a'
9. #define EMPTY\_SQUARE '#'
10. #define WHITE\_IN\_CHECK 0
11. #define BLACK\_IN\_CHECK 1
12. #define NO\_CHECK -1
14. #define B\_KING 'k'
15. #define W\_KING 'K'
16. #define B\_ROOK 'r'
17. #define W\_ROOK 'R'
18. #define B\_BISHOP 'b'
19. #define W\_BISHOP 'B'
20. #define B\_QUEEN 'q'
21. #define W\_QUEEN 'Q'
22. #define B\_PAWN 'p'
23. #define W\_PAWN 'P'
24. #define B\_KNIGHT 'n'
25. #define W\_KNIGHT 'N'
27. class Piece;
28. class King;
30. class Board
31. {
32. private:
33. Piece\*\* \_board[BOARD\_SIDE];
34. King\* \_blackKing;
35. King\* \_whiteKing;
37. /// <summary>
38. /// This function checks if a piece can move to anywhere on the board.
39. /// </summary>
40. /// <param name="piece"> The piece to check </param>
41. /// <param name="srcX"> it's row </param>
42. /// <param name="srcY"> in's col </param>
43. /// <returns> If the piece can move </returns>
44. bool canMove(Piece& piece, int srcX, int srcY);
46. public:
47. /// <summary>
48. /// This function is the constructor of the class Board.
49. /// </summary>
50. /// <param name="strBoard"> A string that contains the board content </param>
51. Board(std::string strBoard);
53. /// <summary>
54. /// This function is the destructor of the class Board.
55. /// </summary>
56. ~Board();
58. /// <summary>
59. /// This function prints the board.
60. /// </summary>
61. void printBoard() const;
63. /// <summary>
64. /// This function returns the board.
65. /// </summary>
66. /// <returns> The board </returns>
67. Piece\*\*\* getBoard() const;
69. /// <summary>
70. /// This function gets the black king.
71. /// </summary>
72. /// <returns> The black king </returns>
73. King\* getBlackKing() const;
75. /// <summary>
76. /// This function gets the white king.
77. /// </summary>
78. /// <returns> The white king </returns>
79. King\* getWhiteKing() const;
81. /// <summary>
82. /// This function moves a piece of the board.
83. /// </summary>
84. /// <param name="src"> The position of the piece that the player wants to move </param>
85. /// <param name="dest"> Where the player wants to move the piece to </param>
86. /// <param name="toDelete"> If to delete the dest piece </param>
87. void movePiece(const int\* src, const int\* dest, bool toDelete);
89. /// <summary>
90. /// This function converts a position on the graphics board to match the board in the code.
91. /// </summary>
92. /// <param name="strIndex"> A position on the graphics board </param>
93. /// <returns> A position on the code board </returns>
94. int\* convertIndex(std::string strIndex) const;
96. /// <summary>
97. /// This function checks if one of the players is in check.
98. /// </summary>
99. /// <returns> If a player is in check and which player it is </returns>
100. int isInCheck() const;
102. /// <summary>
103. /// This function checks if one of the players is in mate.
104. /// </summary>
105. /// <param name="playerInCheck"> The player that is in check </param>
106. /// <returns> If the player is in mate </returns>
107. int isInMate(bool playerInCheck);
109. /// <summary>
110. /// This function checks if a castling move is valid.
111. /// </summary>
112. /// <param name="src"> The position of the piece that the player wants to move </param>
113. /// <param name="dest"> Where the player wants to move the piece to </param>
114. /// <returns> 9 - if castle is valid, 6 - if not valid move and 2 if the dest is not free </returns>
115. int castle(const int\* src, const int\* dest);
117. /// <summary>
118. /// These function all check if the squares between the king and the rook are empty and not under attack.
119. /// </summary>
120. /// <returns> All function return if the result of the checking was true or false </returns>
121. bool checkBottomLeft();
122. bool checkBottomRight();
123. bool checkTopLeft();
124. bool checkTopRight();
126. /// <summary>
127. /// This function checks if a square on the board is being attacked.
128. /// </summary>
129. /// <param name="pos"> A position on the poard </param>
130. /// <param name="isWhite"> If white is the attacker or not </param>
131. /// <returns> If the position is under attack </returns>
132. bool isUnderAttack(const int\* pos, bool isWhite);
133. };
135. #endif // !\_\_BOARD\_H\_\_

  Board.cpp:

1. #include "Board.h"
2. #include "Rook.h"
3. #include "Pawn.h"
4. #include "Queen.h"
5. #include "King.h"
6. #include "Bishop.h"
7. #include "Knight.h"
8. #include "Piece.h"
9. #include "EmptySquare.h"
10. #include <string>
12. Board::Board(std::string strBoard)
13. {
14. int counter = 0;
15. char piece = 0;
16. this->\_blackKing = 0;
17. this->\_whiteKing = 0;
19. for (int i = 0; i < BOARD\_SIDE; i++)
20. {
21. this->\_board[i] = new Piece\*[BOARD\_SIDE];
22. }
24. for (int i = 0; i < BOARD\_SIDE; i++)
25. {
26. for (int j = 0; j < BOARD\_SIDE; j++)
27. {
28. piece = strBoard[counter];
29. switch (piece)
30. {
31. case B\_KING:
32. case W\_KING:
33. {
34. King\* king = new King(piece, i, j);
35. if (piece == B\_KING)
36. {
37. this->\_blackKing = king;
38. }
39. else
40. {
41. this->\_whiteKing = king;
42. }
44. this->\_board[i][j] = king;
45. break;
46. }
48. case B\_ROOK:
49. case W\_ROOK:
50. this->\_board[i][j] = new Rook(piece);
51. break;
53. case B\_BISHOP:
54. case W\_BISHOP:
55. this->\_board[i][j] = new Bishop(piece);
56. break;
58. case B\_QUEEN:
59. case W\_QUEEN:
60. this->\_board[i][j] = new Queen(piece);
61. break;
63. case B\_PAWN:
64. case W\_PAWN:
65. this->\_board[i][j] = new Pawn(piece);
66. break;
68. case B\_KNIGHT:
69. case W\_KNIGHT:
70. this->\_board[i][j] = new Knight(piece);
71. break;
73. default:
74. this->\_board[i][j] = new EmptySquare(EMPTY\_SQUARE);
75. break;
76. }
78. counter++;
79. }
80. }
81. }
83. Board::~Board()
84. {
85. for (int i = 0; i < BOARD\_SIDE; i++)
86. {
87. for (int j = 0; j < BOARD\_SIDE; j++)
88. {
89. delete this->\_board[i][j];
90. }
91. delete[] this->\_board[i];
92. }
93. }
95. void Board::printBoard() const
96. {
97. std::cout << "The board:" << std::endl;
98. for (int i = 0; i < BOARD\_SIDE; i++)
99. {
100. for (int j = 0; j < BOARD\_SIDE; j++)
101. {
102. std::cout << this->\_board[i][j]->getType() << " ";
103. }
104. std::cout << "\n";
105. }
106. }
108. Piece\*\*\* Board::getBoard() const
109. {
110. return (Piece\*\*\*)this->\_board;
111. }
113. King\* Board::getBlackKing() const
114. {
115. return this->\_blackKing;
116. }
118. King\* Board::getWhiteKing() const
119. {
120. return this->\_whiteKing;
121. }
123. void Board::movePiece(const int\* src, const int\* dest, bool toDelete)
124. {
125. char srcChar = this->\_board[src[0]][src[1]]->getType();
127. if (toDelete)
128. {
129. delete this->\_board[dest[0]][dest[1]];
130. }
131. this->\_board[dest[0]][dest[1]] = this->\_board[src[0]][src[1]];
132. this->\_board[src[0]][src[1]] = new EmptySquare(EMPTY\_SQUARE);
134. if (srcChar == B\_KING)
135. {
136. this->\_blackKing->setPosition(dest);
137. }
138. else if (srcChar == W\_KING)
139. {
140. this->\_whiteKing->setPosition(dest);
141. }
142. }
144. int\* Board::convertIndex(std::string strIndex) const
145. {
146. int\* index = new int[BOARD\_INDEX];
148. index[0] = (int)strIndex[1] - '0';
149. index[0] = index[0] + (BOARD\_SIDE - 2 \* index[0]);
150. index[1] = (int)strIndex[0] - FIRST\_LETTER;
152. return index;
153. }
155. int Board::isInCheck() const
156. {
157. int i = 0, j = 0;
158. bool blackInCheck = false, whiteInCheck = false;
159. int currPos[BOARD\_INDEX] = { 0 };
160. int returnValue = 0;
161. int\* kingToCheck = nullptr;
162. bool checkWhiteKing = false;
163. Piece\* piece;
165. for (i = 0; i < BOARD\_SIDE; i++)
166. {
167. for (j = 0; j < BOARD\_SIDE; j++)
168. {
169. piece = this->\_board[i][j];
170. if (piece->getType() != EMPTY\_SQUARE)
171. {
172. currPos[0] = i;
173. currPos[1] = j;
174. if (islower(piece->getType()))
175. {
176. checkWhiteKing = true;
177. kingToCheck = this->\_whiteKing->getPosition();
178. }
179. else
180. {
181. checkWhiteKing = false;
182. kingToCheck = this->\_blackKing->getPosition();
183. }
185. if (piece->isValidPieceMove(currPos, kingToCheck, \*this))
186. {
187. if (checkWhiteKing)
188. {
189. whiteInCheck = true;
190. }
191. else
192. {
193. blackInCheck = true;
194. }
195. }
196. }
197. }
198. }
200. if (blackInCheck && whiteInCheck)
201. {
202. return 2;
203. }
204. else if (whiteInCheck)
205. {
206. return WHITE\_IN\_CHECK;
207. }
208. else if (blackInCheck)
209. {
210. return BLACK\_IN\_CHECK;
211. }
212. return -1;
213. }
215. int Board::isInMate(bool playerInCheck)
216. {
217. int i = 0, j = 0;
218. HandleGame::setCurrentPlayer(!HandleGame::getCurrentPlayer());
220. for (i = 0; i < BOARD\_SIDE; i++)
221. {
222. for (j = 0; j < BOARD\_SIDE; j++)
223. {
224. if (playerInCheck == WHITE && isupper(this->\_board[i][j]->getType()))
225. {
226. if (canMove(\*this->\_board[i][j], i, j))
227. {
228. HandleGame::setCurrentPlayer(!HandleGame::getCurrentPlayer());
229. return CHECK;
230. }
231. }
233. if (playerInCheck == BLACK && islower(this->\_board[i][j]->getType()))
234. {
235. if (canMove(\*this->\_board[i][j], i, j))
236. {
237. HandleGame::setCurrentPlayer(!HandleGame::getCurrentPlayer());
238. return CHECK;
239. }
240. }
241. }
242. }
243. return CHECKMATE;
244. }
246. int Board::castle(const int\* src, const int\* dest)
247. {
248. Piece\* rook = this->\_board[0][0];
249. int rookSrc[BOARD\_INDEX] = { 0 };
250. int rookDest[BOARD\_INDEX] = { 0 };
251. bool srcColor = (isupper(this->\_board[src[0]][src[1]]->getType())) ? WHITE : BLACK;
253. if (HandleGame::getCurrentPlayer() != srcColor || this->\_board[src[0]][src[1]]->getType() == EMPTY\_SQUARE)
254. {
255. return INVALID\_SRC;
256. }

259. if (dest[0] == 7 && dest[1] == 2) // white queen side
260. {
261. if (this->\_board[7][0]->getType() == EMPTY\_SQUARE || !checkBottomLeft())
262. {
263. return INVALID\_MOVE;
264. }
266. rook = this->\_board[7][0];
267. rookSrc[0] = 7;
268. rookSrc[1] = 0;
269. rookDest[0] = 7;
270. rookDest[1] = 3;
271. }
272. else if (dest[0] == 7 && dest[1] == 6) // white king side
273. {
274. if (this->\_board[7][7]->getType() == EMPTY\_SQUARE || !checkBottomRight())
275. {
276. return INVALID\_MOVE;
277. }
279. rook = this->\_board[7][7];
280. rookSrc[0] = 7;
281. rookSrc[1] = 7;
282. rookDest[0] = 7;
283. rookDest[1] = 5;
284. }
285. else if (dest[0] == 0 && dest[1] == 2) // black queen side
286. {
287. if (this->\_board[0][0]->getType() == EMPTY\_SQUARE || !checkTopLeft())
288. {
289. return INVALID\_MOVE;
290. }
292. rook = this->\_board[0][0];
293. rookSrc[0] = 0;
294. rookSrc[1] = 0;
295. rookDest[0] = 0;
296. rookDest[1] = 3;
297. }
298. else if (dest[0] == 0 && dest[1] == 6) // black king side
299. {
300. if (this->\_board[0][7]->getType() == EMPTY\_SQUARE || !checkTopRight())
301. {
302. return INVALID\_MOVE;
303. }
305. rook = this->\_board[0][7];
306. rookSrc[0] = 0;
307. rookSrc[1] = 7;
308. rookDest[0] = 0;
309. rookDest[1] = 5;
310. }
312. if (this->\_board[src[0]][src[1]]->getHasMoved() || rook->getHasMoved() || this->isInCheck() != -1)
313. {
314. return INVALID\_MOVE;
315. }
317. movePiece(rookSrc, rookDest, true);
318. return CASTLE;
319. }
321. bool Board::checkBottomLeft()
322. {
323. int square1[BOARD\_INDEX] = { 7, 1 };
324. int square2[BOARD\_INDEX] = { 7, 2 };
325. int square3[BOARD\_INDEX] = { 7, 3 };
327. if (this->\_board[7][1]->getType() != EMPTY\_SQUARE || this->\_board[7][2]->getType() != EMPTY\_SQUARE || this->\_board[7][3]->getType() != EMPTY\_SQUARE)
328. {
329. return false;
330. }
331. if (isUnderAttack(square1, false) || isUnderAttack(square2, false) || isUnderAttack(square3, false))
332. {
333. return false;
334. }
335. return true;
336. }
338. bool Board::checkBottomRight()
339. {
340. int square1[BOARD\_INDEX] = { 7, 5 };
341. int square2[BOARD\_INDEX] = { 7, 6 };
343. if (this->\_board[7][5]->getType() != EMPTY\_SQUARE || this->\_board[7][6]->getType() != EMPTY\_SQUARE)
344. {
345. return false;
346. }
347. if (isUnderAttack(square1, false) || isUnderAttack(square2, false))
348. {
349. return false;
350. }
351. return true;
352. }
354. bool Board::checkTopLeft()
355. {
356. int square1[BOARD\_INDEX] = { 0, 1 };
357. int square2[BOARD\_INDEX] = { 0, 2 };
358. int square3[BOARD\_INDEX] = { 0, 3 };
360. if (this->\_board[0][1]->getType() != EMPTY\_SQUARE || this->\_board[0][2]->getType() != EMPTY\_SQUARE || this->\_board[0][3]->getType() != EMPTY\_SQUARE)
361. {
362. return false;
363. }
364. if (isUnderAttack(square1, true) || isUnderAttack(square2, true) || isUnderAttack(square3, true))
365. {
366. return false;
367. }
368. return true;
369. }
371. bool Board::checkTopRight()
372. {
373. int square1[BOARD\_INDEX] = { 0, 5 };
374. int square2[BOARD\_INDEX] = { 0, 6 };
376. if (this->\_board[0][5]->getType() != EMPTY\_SQUARE || this->\_board[0][6]->getType() != EMPTY\_SQUARE)
377. {
378. return false;
379. }
380. if (isUnderAttack(square1, true) || isUnderAttack(square2, true))
381. {
382. return false;
383. }
384. return true;
385. }
387. bool Board::isUnderAttack(const int\* pos, bool isWhite)
388. {
389. int i = 0, j = 0;
390. int src[BOARD\_INDEX] = { 0 };
392. for (i = 0; i < BOARD\_SIDE; i++)
393. {
394. for (j = 0; j < BOARD\_SIDE; j++)
395. {
396. src[0] = i;
397. src[1] = j;
399. if (isWhite && isupper(this->getBoard()[i][j]->getType()))
400. {
401. if (this->\_board[i][j]->isValidPieceMove(src, pos, \*this))
402. {
403. return true;
404. }
405. }
406. else if (!isWhite && islower(this->getBoard()[i][j]->getType()))
407. {
408. if (this->\_board[i][j]->isValidPieceMove(src, pos, \*this))
409. {
410. return true;
411. }
412. }
413. }
414. }
415. return false;
416. }
418. bool Board::canMove(Piece& piece, int srcX, int srcY)
419. {
420. int i = 0, j = 0;
421. int src[BOARD\_INDEX] = { srcX, srcY };
422. int dest[BOARD\_INDEX] = { 0 };
424. for (i = 0; i < BOARD\_SIDE; i++)
425. {
426. for (j = 0; j < BOARD\_SIDE; j++)
427. {
428. dest[0] = i;
429. dest[1] = j;
431. if (piece.isValidMove(src, dest, \*this) == VALID || piece.isValidMove(src, dest, \*this) == CHECK)
432. {
433. return true;
434. }
435. }
436. }
438. return false;
439. }

 Bishop.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
5. class Bishop : public Piece
6. {
7. public:
8. /// <summary>
9. /// This function is the constructor of the class bishop.
10. /// </summary>
11. /// <param name="type"> If the bishop is black or white </param>
12. Bishop(char type);
14. /// <summary>
15. /// This function checks if a move that is proformed on a bishop matches the way that a bishop moves.
16. /// </summary>
17. /// <param name="src"> The position of the bishop that the player wants to move </param>
18. /// <param name="dest"> Where the player wants to move the bishop to </param>
19. /// <param name="board"> The game board </param>
20. /// <returns> If the move is valid or not </returns>
21. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
22. };

Bishop.cpp:

1. #include "Bishop.h"
3. Bishop::Bishop(char type) : Piece(type)
4. {
5. }
7. bool Bishop::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
8. {
9. int subX = src[0] - dest[0];
10. int subY = src[1] - dest[1];
12. if (src[0] == dest[0] || src[1] == dest[1] || abs(subX) != abs(subY)) // If the bishop is not going diagonally
13. {
14. return false;
15. }
17. for (int i = 1; i < abs(subX); i++)
18. {
19. if (subX > 0 && subY > 0)
20. {
21. if (board.getBoard()[src[0] - i][src[1] - i]->getType() != EMPTY\_SQUARE) // If path is not clear
22. {
23. return false;
24. }
25. }
26. else if (subX > 0 && subY < 0)
27. {
28. if (board.getBoard()[src[0] - i][src[1] + i]->getType() != EMPTY\_SQUARE) // If path is not clear
29. {
30. return false;
31. }
32. }
33. else if (subX < 0 && subY > 0)
34. {
35. if (board.getBoard()[src[0] + i][src[1] - i]->getType() != EMPTY\_SQUARE) // If path is not clear
36. {
37. return false;
38. }
39. }
40. else if (subX < 0 && subY < 0)
41. {
42. if (board.getBoard()[src[0] + i][src[1] + i]->getType() != EMPTY\_SQUARE) // If path is not clear
43. {
44. return false;
45. }
46. }
47. }
49. return true;
50. }

 King.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
5. class Board;
7. class King : public Piece
8. {
9. private:
10. int \_position[BOARD\_INDEX];
12. public:
13. /// <summary>
14. /// This function is the constructor of the class king.
15. /// </summary>
16. /// <param name="type"> If the king is black or white </param>
17. King(char type, int pos1, int pos2);
19. /// <summary>
20. /// This function returns the position of the king.
21. /// </summary>
22. /// <returns> The position of the king. </returns>
23. int\* getPosition() const;
25. /// <summary>
26. /// This function sets the position of the king.
27. /// </summary>
28. /// <param name="position"> The new position for the king </param>
29. void setPosition(const int\* position);
31. /// <summary>
32. /// This function checks if a move that is proformed on a king matches the way that a king moves.
33. /// </summary>
34. /// <param name="src"> The position of the king that the player wants to move </param>
35. /// <param name="dest"> Where the player wants to move the king to </param>
36. /// <param name="board"> The game board </param>
37. /// <returns> If the move is valid or not </returns>
38. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
39. };

 King.cpp:

1. #include "King.h"
2. #include "HandleGame.h"
3. class Piece;
5. King::King(char type, int pos1, int pos2) : Piece(type)
6. {
7. this->\_position[0] = pos1;
8. this->\_position[1] = pos2;
9. }
11. int\* King::getPosition() const
12. {
13. return (int\*)this->\_position;
14. }
16. void King::setPosition(const int\* position)
17. {
18. this->\_position[0] = position[0];
19. this->\_position[1] = position[1];
20. }
22. bool King::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
23. {
24. int subX = src[0] - dest[0];
25. int subY = src[1] - dest[1];
27. if (abs(subX) > 1 || abs(subY) > 1)
28. {
29. return false;
30. }
32. return true;
33. }

 Knight.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
4. #include "Board.h"
5. class Board;
7. class Knight : public Piece
8. {
9. public:
10. /// <summary>
11. /// This function is the constructor of the class knight.
12. /// </summary>
13. /// <param name="type"> If the knight is black or white </param>
14. Knight(char type);
16. /// <summary>
17. /// This function checks if a move that is proformed on a knight matches the way that a knight moves.
18. /// </summary>
19. /// <param name="src"> The position of the knight that the player wants to move </param>
20. /// <param name="dest"> Where the player wants to move the knight to </param>
21. /// <param name="board"> The game board </param>
22. /// <returns> If the move is valid or not </returns>
23. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
24. };

 Knight.cpp:

1. #include "Knight.h"
3. Knight::Knight(char type) : Piece(type)
4. {
5. }
7. bool Knight::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
8. {
9. int subX = abs(src[0] - dest[0]);
10. int subY = abs(src[1] - dest[1]);
12. if ((subX == 2 && subY == 1) || (subX == 1 && subY == 2))
13. {
14. return true;
15. }
17. return false;
18. }

 Pawn.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
4. #include "Board.h"
5. class Board;
7. #define BLACK\_PAWN\_ROW 1
8. #define WHITE\_PAWN\_ROW 6
10. class Pawn : public Piece
11. {
12. public:
13. /// <summary>
14. /// This function is the constructor of the class Pawn.
15. /// </summary>
16. /// <param name="type"> If the pawn is black or white </param>
17. Pawn(char type);
19. /// <summary>
20. /// This function checks if a move that is proformed on a pawn matches the way that a pawn moves.
21. /// </summary>
22. /// <param name="src"> The position of the pawn that the player wants to move </param>
23. /// <param name="dest"> Where the player wants to move the pawn to </param>
24. /// <param name="board"> The game board </param>
25. /// <returns> If the move is valid or not </returns>
26. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
28. private:
29. /// <summary>
30. /// This function gets information about the pawn:
31. /// It gets the char after the pawn, gets the initial row of the pawn
32. /// and checks if the pawn is going backwards.
33. /// </summary>
34. /// <param name="board"> The game board </param>
35. /// <param name="initialRow"> A pointer to the initial row of the pawns </param>
36. /// <param name="afterSrc"> A pointer to the char after the src char </param>
37. /// <param name="src"> The position of the pawn to be moved </param>
38. /// <param name="dest"> Where to move the pawn to </param>
39. /// <returns> If the pawn tried to move backwards </returns>
40. bool getPawnInfo(const Board& board, int\* initialRow, Piece\*\* afterSrc, const int\* src, const int\* dest) const;
41. };

 Pawn.cpp:

1. #include "Pawn.h"
3. Pawn::Pawn(char type) : Piece(type)
4. {
5. }
7. bool Pawn::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
8. {
9. int subX = src[0] - dest[0];
10. int subY = src[1] - dest[1];
11. int pawnRow = src[0];
12. int initialRow = 0;
13. Piece\* destChar = board.getBoard()[dest[0]][dest[1]];
14. Piece\* afterSrc = 0;
16. if (!this->getPawnInfo(board, &initialRow, &afterSrc, src, dest)) // If the pawn is going backwards
17. {
18. return false;
19. }
21. if (destChar->getType() != EMPTY\_SQUARE) // If the pawn is eating
22. {
23. if (subY == 0) // If the pawn is going forward
24. {
25. return false;
26. }
27. else if (!(abs(subX) == 1 && abs(subY) == 1)) // If the pawn went more than one step
28. {
29. return false;
30. }
31. }
32. else
33. {
34. if (subY != 0) // If the pawn is not going forward
35. {
36. return false;
37. }
38. else if (abs(subX) > 1 && pawnRow != initialRow) // If the pawn went more than one step and moved already
39. {
40. return false;
41. }
42. else if (abs(subX) > 2) // If the pawn went more that two steps
43. {
44. return false;
45. }
46. else if (afterSrc)
47. {
48. if (afterSrc->getType() != EMPTY\_SQUARE) // If the pawn jumped over a piece
49. {
50. return false;
51. }
52. }
53. }
55. return true;
56. }
58. bool Pawn::getPawnInfo(const Board& board, int\* initialRow, Piece\*\* afterSrc, const int\* src, const int\* dest) const
59. {
60. int subX = src[0] - dest[0];
62. if (isupper(this->\_type))
63. {
64. \*initialRow = WHITE\_PAWN\_ROW;
65. if (src[0] - 1 > 0 && src[0] - 1 < BOARD\_SIDE)
66. {
67. \*afterSrc = board.getBoard()[src[0] - 1][src[1]];
68. }
70. if (subX < 0)
71. {
72. return false;
73. }
74. return true;
75. }
76. else
77. {
78. \*initialRow = BLACK\_PAWN\_ROW;
79. if (src[0] + 1 > 0 && src[0] + 1 < BOARD\_SIDE)
80. {
81. \*afterSrc = board.getBoard()[src[0] + 1][src[1]];
82. }
84. if (subX > 0)
85. {
86. return false;
87. }
88. return true;
89. }
90. }

 Queen.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
4. #include "Board.h"
5. class Board;
7. class Queen : public Piece
8. {
9. public:
10. /// <summary>
11. /// This function is the constructor of the class queen.
12. /// </summary>
13. /// <param name="type"> If the queen is black or white </param>
14. Queen(char type);
16. /// <summary>
17. /// This function checks if a move that is proformed on a queen matches the way that a queen moves.
18. /// </summary>
19. /// <param name="src"> The position of the queen that the player wants to move </param>
20. /// <param name="dest"> Where the player wants to move the queen to </param>
21. /// <param name="board"> The game board </param>
22. /// <returns> If the move is valid or not </returns>
23. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
24. };

 Queen.cpp:

1. #include "Queen.h"
2. #include "Bishop.h"
3. #include "Rook.h"
5. Queen::Queen(char type) : Piece(type)
6. {
7. }
9. bool Queen::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
10. {
11. Rook rook('r');
12. Bishop bishop('b');
13. return rook.isValidPieceMove(src, dest, board) || bishop.isValidPieceMove(src, dest, board);
14. }

 Rook.h:

1. #pragma once
2. #include <iostream>
3. #include "Piece.h"
4. #include "Board.h"
5. class Piece;
6. class Board;
8. class Rook : public Piece
9. {
10. public:
11. /// <summary>
12. /// This function is the constructor of the class rook.
13. /// </summary>
14. /// <param name="type"> If the rook is black or white </param>
15. Rook(char type);
17. /// <summary>
18. /// This function checks if a move that is proformed on a rook matches the way that a rook moves.
19. /// </summary>
20. /// <param name="src"> The position of the rook that the player wants to move </param>
21. /// <param name="dest"> Where the player wants to move the rook to </param>
22. /// <param name="board"> The game board </param>
23. /// <returns> If the move is valid or not </returns>
24. virtual bool isValidPieceMove(const int\* src, const int\* dest, const Board& board) const;
25. };

 Rook.cpp:

1. #include "Rook.h"
2. #include "HandleGame.h"
3. class HandleGame;
5. Rook::Rook(char type) : Piece(type)
6. {
7. }
9. bool Rook::isValidPieceMove(const int\* src, const int\* dest, const Board& board) const
10. {
11. int tempSrc[BOARD\_INDEX] = { src[0], src[1] };
12. int tempDest[BOARD\_INDEX] = { dest[0], dest[1] };
14. if (tempSrc[0] != tempDest[0] && tempSrc[1] != tempDest[1]) // If the rook is not going straight
15. {
16. return false;
17. }
19. if (tempSrc[0] == tempDest[0]) // If the rook is going right or left
20. {
21. if (tempDest[1] < tempSrc[1])
22. {
23. std::swap(tempSrc[1], tempDest[1]);
24. }
26. for (int i = tempSrc[1] + 1; i < tempDest[1]; i++)
27. {
28. if (board.getBoard()[tempSrc[0]][i]->getType() != EMPTY\_SQUARE) // If the rook is trying to pass a piece
29. {
30. return false;
31. }
32. }
33. }
34. else // If the rook is going up or down
35. {
36. if (tempDest[0] < tempSrc[0])
37. {
38. std::swap(tempSrc[0], tempDest[0]);
39. }
41. for (int i = tempSrc[0] + 1; i < tempDest[0]; i++)
42. {
43. if (board.getBoard()[i][tempSrc[1]]->getType() != EMPTY\_SQUARE) // If the rook is trying to pass a piece
44. {
45. return false;
46. }
47. }
48. }
50. return true;
51. }

 Main.cpp:

1. #include <iostream>
2. #include <thread>
3. #include "Board.h"
4. #include "WSAInitializer.h"
5. #include <WinSock2.h>
6. #include "HandleGame.h"
7. #include <string>
9. #define PORT\_ADDRESS 3000
11. void bindAndListen(SOCKET& m\_socket)
12. {
13. struct sockaddr\_in sa { 0 };
14. sa.sin\_port = htons(PORT\_ADDRESS);
15. sa.sin\_family = AF\_INET;
16. sa.sin\_addr.s\_addr = INADDR\_ANY;
18. if (bind(m\_socket, (struct sockaddr\*)&sa, sizeof(sa)) == SOCKET\_ERROR)
19. {
20. throw std::exception(\_\_FUNCTION\_\_ " - bind");
21. }
23. if (listen(m\_socket, SOMAXCONN) == SOCKET\_ERROR)
24. {
25. throw std::exception(\_\_FUNCTION\_\_ " - listen");
26. }
27. std::cout << "Listening on port " << PORT\_ADDRESS << std::endl;
28. }
30. int main()
31. {
32. WSAInitializer wsaInit;
33. srand(time\_t(NULL));
34. SOCKET m\_socket;
35. Board\* board = new Board(INIT\_STR);
36. m\_socket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);
38. if (m\_socket == INVALID\_SOCKET)
39. throw std::exception(\_\_FUNCTION\_\_ " - socket");
40. try
41. {
42. bindAndListen(m\_socket);
43. }
44. catch (const std::exception& e)
45. {
46. std::cerr << "Something bad happened here... " << e.what() << std::endl;
47. }
48. SOCKET client\_socket = accept(m\_socket, NULL, NULL);
50. if (client\_socket == INVALID\_SOCKET)
51. throw std::exception(\_\_FUNCTION\_\_);
52. HandleGame::startGame(board, &client\_socket);
53. closesocket(m\_socket);
54. return 0;
55. }

 WSAInitializer.h:

1. #pragma once
3. #include <WinSock2.h>
4. #include <Windows.h>
5. #pragma comment(lib,"WS2\_32")

8. class WSAInitializer
9. {
10. public:
11. WSAInitializer();
12. ~WSAInitializer();
13. };

 WSAInitializer.cpp:

1. #include "WSAInitializer.h"
2. #include <exception>

5. WSAInitializer::WSAInitializer()
6. {
7. WSADATA wsa\_data = { };
8. if (WSAStartup(MAKEWORD(2, 2), &wsa\_data) != 0)
9. throw std::exception("WSAStartup Failed");
10. }
12. WSAInitializer::~WSAInitializer()
13. {
14. // Q: why is this try necessary ?
15. // A: to avoid throwing exceptions in d-tors !
16. // if we do throw think what will happened in regular exception, our object
17. // will be destroyed because an exception occurred and then the d-tor will be
18. // called and if we throw now there is no one to handle the exception because
19. // we are already in the flow of exception handling !! (inception...)
20. // please read more about exception handling and why it's forbidden to throw
21. // exception from the destructor.
22. try
23. {
24. WSACleanup();
25. }
26. catch (...) {}
27. }