INTRODUCTION TO C++

DOCUMENTATION

CHESS GAME

Submitted By: **Sadhabi Dev**
Level : 4
Student ID : 23085140
Cyber Security and Digital Forensics

1. Game Overview

Chess Master is a C++ console game made using Code::Blocks that lets two players play chess. It follows all the standard chess rules, like castling, en passant, and pawn promotion. However, instead of playing until checkmate, the game ends after four checks, and the player who gives the fourth check wins.
The game has an interactive, menu-based interface that allows players to play, save, and load games. It also shows a colorful board and explains the rules. Behind the scenes, it uses a 2D array to manage the board, handles user input carefully, keeps track of the game state, and checks for valid piece movements.

This document provides a detailed specification of all output generated by the Chess Master program, a console-based C++ chess game for two players. Outputs include menus, board displays, prompts, error messages, game status updates, and saved game files, formatted with ANSI color codes for visual clarity. The documentation describes each output's content, format, context, and purpose, ensuring a complete understanding of the program's visible and persistent behavior.

2. Purpose and Scope

The program offers a complete chess game for two human players, with the following features:

a) All standard chess moves, including special ones like castling, en passant, and pawn promotion
b) The ability to save and load the game using files
c) A user-friendly console interface with colorful text
d) A custom rule where the game ends after four checks
e) A clear visual display of the board, showing captured pieces and move counts

The game does not include computer opponents, complex draw rules (like threefold repetition or not enough material), or any graphical interface. It focuses on being a simple and lightweight text-based chess game.

3. Submission Details

Files Included:

- ***ChessGame.h*** – This header file contains important constants, data structures, and class declarations.
- ***ChessGame.cpp*** – This is the main source file where the actual game logic is written.

Development Environment:

- ***IDE Used:*** *Code::Blocks 20.03* with the *MinGW (GNU GCC Compiler)*
- ***Programming Language:*** *C++*
- ***Libraries Used:*** Only standard C++ libraries – including *<iostream>*, *<vector>*, *<string>*, *<fstream>*, *<thread>*, and *<chrono>*

4. System Architecture

The program is structured into two files:

a) ***ChessGame.h:*** Defines constants, the *ChessPiece* structure, *ChessBoard* and *ChessGame* classes, and function prototypes.

b) ***ChessGame.cpp:*** Implements chess rules, board management, game flow, and user interaction.

Key Components

4.1 ChessPiece Structure

a. Purpose:
This structure is used to represent a chess piece. It stores information about what the piece is, its color, and whether it has moved.

b. ***Attributes:***
c.

- *symbol:* A character that shows the type of piece (for example, 'K' for a white king and 'k' for a black king).

- *isWhite:* A true/false value that tells if the piece is white (true) or black (false).

- *hasMoved:* A true/false value used to check if the piece has moved, which is important for castling.

d. Constructor:
It sets up a piece with its symbol and color. By default, it represents an empty square if no values are given.

4.2 ChessBoard Class

a. Purpose:
This class handles everything related to the chessboard. It keeps track of the 8x8 grid, checks if moves are valid, and manages the current game state using a 2D array.

b. Private Members:

- *board*: A 2D vector (8x8 grid) that stores all the chess pieces.

- *lastMoveFrom*, *lastMoveTo*: Used to track the last move, especially for en passant.

- *whiteCanCastleKingside*, *whiteCanCastleQueenside*, *blackCanCastleKingside*, *blackCanCastleQueenside*: These keep track of whether each side can still castle.

- *halfMoveClock*: Counts moves for the 50-move draw rule.

- *fullMoveNumber*: Tracks the number of full moves played in the game.

c. ***Key Methods:***

- ***ChessBoard()***: Sets up a new board when the game starts.
- ***resetBoard()***: Places all the pieces in their starting positions.
- ***displayBoard****(const std::vector<char>& whiteCaptures, const std::vector<char>& blackCaptures)*: Shows the current board and lists of captured pieces.
- ***movePiece****(std::string move, bool isWhiteTurn, char promotion, std::vector<char>& whiteCaptures, std::vector<char>& blackCaptures)*: Handles a player's move (like "e2 e4"), including promotion and capturing.
- ***isKingInCheck****(bool isWhite)*: Checks if a king is currently in check.
- ***isCheckmate****(bool isWhite)*: Checks if the current player is in checkmate.
- ***isStalemate****(bool isWhite)*: Checks for stalemate (no legal moves but not in check).
- ***isDraw()***: Checks if the 50-move draw rule applies.
- ***hasLegalMoves****(bool isWhite)*: Verifies if the player has any legal moves left.
- ***saveGame****(const std::string& filename)*: Saves the current game to a file.
- ***loadGame****(const std::string& filename)*: Loads a previously saved game from a file.

d. ***Helper Functions:***

isValidPosition, ***findKing***, ***isPathClear***, ***canPieceAttack***, ***isValidMove***: These help check if moves are allowed and follow the rules.

4.3 ChessGame Class

a. Purpose:
This class controls the overall chess game. It handles the gameplay, user input, whose turn it is, keeping track of captured pieces, and how many times a player has been in check.

b. Private Variables (used inside the class only):
- *board:* Holds the chessboard and pieces.
- *whiteTurn:* A true or false value that tells us if it is White's turn.
- *whiteCaptures, blackCaptures:* Lists that store the pieces captured by each side.

- *checkCount:* A number that increases each time a player is in check. The game ends when it reaches 4.

c. ***Main Functions (What the class can do)***:

- *ChessGame():* Sets up the game when it starts.
- *displayWelcomeMessage():* Shows a message with instructions on how to play.
- *displayRules():* Shows the rules of the game.
- *start():* Starts and runs the main part of the game.
- *saveGame(filename):* Saves the game to a file (uses the ChessBoard class to help).
- *loadGame(filename):* Loads a saved game from a file (also uses ChessBoard).

d. ***Global Functions (Used outside the class):***

- *displayMenu():* Shows the main menu options.
- *main():* The starting point of the program, runs the menu and handles the user's choices.

5. Functionality

5.1 Game Rules and Features

Standard Rules:

- All chess pieces move just like in normal chess (for example, knights move in an L shape, bishops move diagonally).
- The game allows special moves like castling, en passant, and pawn promotion.
- Captured pieces are shown and kept track of.

Custom Rule:

- The game ends after a player puts the other player in check four times.
- The player who gives the fourth check wins the game.

Draw Rules:

- 50-move rule: If 50 moves happen without a capture or a pawn move, the game ends in a draw.
- Stalemate: The game also ends in a draw if a player has no legal moves and is not in check.

5.2 2D Array Manipulation

The chessboard is made using a 2D array **(an 8 x 8 grid)**. It is used for:

a. Piece placement *(resetBoard)*.
b. Move execution *(movePiece updates positions)*.
c. Collision detection *(checking target squares for captures or blockages)*.

5.3 Input Handling

a. Handles user inputs via ***std::cin:***
- Menu selections *(1-5)*.
- Move inputs in algebraic notation *(e.g., e2 e4, d7 d8=Q)*.
- File names for *save/load*.

b. Validates inputs, rejecting invalid formats or illegal moves with error messages.

5.4 State Management

a. Tracks game state via:
- ***ChessBoard:*** Board configuration, castling flags, move counters.
- ***ChessGame:*** Turn, captures, check count.

b. The game can save and load all this information so you can continue playing later.

5.5 Basic Collision Detection

a. It checks if a move is allowed by making sure the square is either empty or has an opponent's piece that can be captured.

b. Checks paths for sliding pieces (rooks, bishops, queens) to ensure no obstructions.

c. It also takes care of special moves like en passant and castling.

5.6 User Interface

a. **Menu**: Options to start, save/load, view rules, or exit.

b. **Board** Display: ***8x8 grid*** with ***row (1-8)*** and ***column (a-h)*** labels, captures, and counters.

c. **Input Format**:
- Moves: ***e2 e4.***
- Promotion: ***d7 d8=Q.***
- Castling: ***e1 g1***.

d. **Exit**: ***exit***.

e. **Feedback**: Colorful prompts for moves, errors, checks, and outcomes.

5.7 Persistence

a. ***Save Game:*** Stores board, castling flags, and counters in a text file.

b. ***Load Game:*** Restores game state from a file.

6. Source Code

Below is the complete source code for ChessGame.h and ChessGame.cpp, implementing the described functionality.

HEADER FILE: *ChessGame.h*

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include <thread>
#include <chrono>

const int BOARD_SIZE = 8;
const char EMPTY = '.';
const char WHITE_KING = 'K';
const char WHITE_QUEEN = 'Q';
const char WHITE_ROOK = 'R';
const char WHITE_BISHOP = 'B';
const char WHITE_KNIGHT = 'N';
const char WHITE_PAWN = 'P';
const char BLACK_KING = 'k';
const char BLACK_QUEEN = 'q';
const char BLACK_ROOK = 'r';
const char BLACK_BISHOP = 'b';
const char BLACK_KNIGHT = 'n';
const char BLACK_PAWN = 'p';

struct ChessPiece {
    char symbol;
    bool isWhite;
    bool hasMoved;
    ChessPiece(char s = EMPTY, bool color = true);
};

class ChessBoard {
private:
    std::vector<std::vector<ChessPiece>> board;
    std::pair<int, int> lastMoveFrom, lastMoveTo;
    bool whiteCanCastleKingside, whiteCanCastleQueenside;
    bool blackCanCastleKingside, blackCanCastleQueenside;
    int halfMoveClock;
    int fullMoveNumber;
    bool isValidPosition(int row, int col) const;
    std::pair<int, int> findKing(bool isWhite) const;
    bool isPathClear(int fromRow, int fromCol, int toRow, int toCol) const;
    bool canPieceAttack(const ChessPiece& piece, int fromRow, int fromCol, int
toRow, int toCol) const;
    bool isValidMove(const ChessPiece& piece, int fromRow, int fromCol, int toRow,
int toCol);

public:
    ChessBoard();
    void resetBoard();
    void displayBoard(const std::vector<char>& whiteCaptures, const
std::vector<char>& blackCaptures) const;
    bool isKingInCheck(bool isWhite) const;
    bool movePiece(std::string move, bool isWhiteTurn, char promotion,
                std::vector<char>& whiteCaptures, std::vector<char>& blackCaptures);
    bool isCheckmate(bool isWhite);
    bool isStalemate(bool isWhite);
    bool isDraw() const;
    bool hasLegalMoves(bool isWhite);
    void saveGame(const std::string& filename);
    void loadGame(const std::string& filename);
};

class ChessGame {
private:
    ChessBoard board;
    bool whiteTurn;
    std::vector<char> whiteCaptures;
    std::vector<char> blackCaptures;
    int checkCount;

public:
    ChessGame();
    void displayWelcomeMessage();
    void displayRules();
    void start();
    void saveGame(const std::string& filename);
    void loadGame(const std::string& filename);
};

void displayMenu();
```

SOURCE FILE: *ChessGame.cpp*

```
#include "ChessGame.h"
using namespace std;
```

```
ChessPiece::ChessPiece(char s, bool color) : symbol(s), isWhite(color),
hasMoved(false) {}

ChessBoard::ChessBoard() {
   board.resize(BOARD_SIZE, vector<ChessPiece>(BOARD_SIZE,
ChessPiece(EMPTY)));
   resetBoard();
}

void ChessBoard::resetBoard() {
   lastMoveFrom = lastMoveTo = {-1, -1};
   whiteCanCastleKingside = whiteCanCastleQueenside = true;
   blackCanCastleKingside = blackCanCastleQueenside = true;
   halfMoveClock = 0;
   fullMoveNumber = 1;

   board[0] = {ChessPiece(BLACK_ROOK, false), ChessPiece(BLACK_KNIGHT,
false), ChessPiece(BLACK_BISHOP, false),
            ChessPiece(BLACK_QUEEN, false), ChessPiece(BLACK_KING, false),
ChessPiece(BLACK_BISHOP, false),
            ChessPiece(BLACK_KNIGHT, false), ChessPiece(BLACK_ROOK, false)};
   for (int i = 0; i < BOARD_SIZE; i++) board[1][i] = ChessPiece(BLACK_PAWN,
false);
   for (int row = 2; row < 6; row++) fill(board[row].begin(), board[row].end(),
ChessPiece(EMPTY));
   board[7] = {ChessPiece(WHITE_ROOK, true), ChessPiece(WHITE_KNIGHT,
true), ChessPiece(WHITE_BISHOP, true),
            ChessPiece(WHITE_QUEEN, true), ChessPiece(WHITE_KING, true),
ChessPiece(WHITE_BISHOP, true),
            ChessPiece(WHITE_KNIGHT, true), ChessPiece(WHITE_ROOK, true)};
   for (int i = 0; i < BOARD_SIZE; i++) board[6][i] = ChessPiece(WHITE_PAWN,
true);
}

bool ChessBoard::isValidPosition(int row, int col) const {
   return row >= 0 && row < BOARD_SIZE && col >= 0 && col < BOARD_SIZE;
}

pair<int, int> ChessBoard::findKing(bool isWhite) const {
   for (int row = 0; row < BOARD_SIZE; row++) {
      for (int col = 0; col < BOARD_SIZE; col++) {
         if (board[row][col].symbol == (isWhite ? WHITE_KING : BLACK_KING)) {
            return {row, col};
         }
      }
   }
   return {-1, -1};
}
```

```
bool ChessBoard::isPathClear(int fromRow, int fromCol, int toRow, int toCol) const
{
   int rowStep = (toRow > fromRow) ? 1 : (toRow < fromRow) ? -1 : 0;
   int colStep = (toCol > fromCol) ? 1 : (toCol < fromCol) ? -1 : 0;
   int row = fromRow + rowStep;
   int col = fromCol + colStep;

   while (row != toRow || col != toCol) {
      if (!isValidPosition(row, col) || board[row][col].symbol != EMPTY) return false;
      row += rowStep;
      col += colStep;
   }
   return true;
}

bool ChessBoard::canPieceAttack(const ChessPiece& piece, int fromRow, int
fromCol, int toRow, int toCol) const {
   int rowDiff = toRow - fromRow;
   int colDiff = toCol - fromCol;
   int absRowDiff = abs(rowDiff);
   int absColDiff = abs(colDiff);

   switch (piece.symbol) {
      case WHITE_PAWN:
         return rowDiff == -1 && absColDiff == 1 &&
board[toRow][toCol].symbol != EMPTY && !board[toRow][toCol].isWhite;
      case BLACK_PAWN:
         return rowDiff == 1 && absColDiff == 1 && board[toRow][toCol].symbol !=
EMPTY && board[toRow][toCol].isWhite;
      case WHITE_KING: case BLACK_KING:
         return absRowDiff <= 1 && absColDiff <= 1;
      case WHITE_QUEEN: case BLACK_QUEEN:
         return (absRowDiff == absColDiff || fromRow == toRow || fromCol == toCol)
&&
               isPathClear(fromRow, fromCol, toRow, toCol);
      case WHITE_ROOK: case BLACK_ROOK:
         return (fromRow == toRow || fromCol == toCol) &&
               isPathClear(fromRow, fromCol, toRow, toCol);
      case WHITE_BISHOP: case BLACK_BISHOP:
         return absRowDiff == absColDiff &&
               isPathClear(fromRow, fromCol, toRow, toCol);
      case WHITE_KNIGHT: case BLACK_KNIGHT:
         return (absRowDiff == 2 && absColDiff == 1) || (absRowDiff == 1 &&
absColDiff == 2);
      default:
         return false;
   }
}
```

```
bool ChessBoard::isValidMove(const ChessPiece& piece, int fromRow, int fromCol,
int toRow, int toCol) {
   if (!isValidPosition(fromRow, fromCol) || !isValidPosition(toRow, toCol)) return
false;
   if (fromRow == toRow && fromCol == toCol) return false;

   int rowDiff = toRow - fromRow;
   int colDiff = toCol - fromCol;
   int absRowDiff = abs(rowDiff);
   int absColDiff = abs(colDiff);

   if (board[toRow][toCol].symbol != EMPTY && board[toRow][toCol].isWhite ==
piece.isWhite) {
      return false;
   }

   switch (piece.symbol) {
      case WHITE_PAWN:
         if (fromCol == toCol && rowDiff < 0 && board[toRow][toCol].symbol ==
EMPTY) {
            if (rowDiff == -1) return true;
            if (rowDiff == -2 && fromRow == 6 && board[fromRow -
1][fromCol].symbol == EMPTY) return true;
         }
         if (rowDiff == -1 && absColDiff == 1) {
            if (board[toRow][toCol].symbol != EMPTY
&& !board[toRow][toCol].isWhite) return true;
            if (toRow == 5 && lastMoveTo == make_pair(6, toCol) &&
               lastMoveFrom == make_pair(4, toCol) &&
               board[6][toCol].symbol == BLACK_PAWN) return true;
         }
         return false;
      case BLACK_PAWN:
         if (fromCol == toCol && rowDiff > 0 && board[toRow][toCol].symbol ==
EMPTY) {
            if (rowDiff == 1) return true;
            if (rowDiff == 2 && fromRow == 1 && board[fromRow +
1][fromCol].symbol == EMPTY) return true;
         }
         if (rowDiff == 1 && absColDiff == 1) {
            if (board[toRow][toCol].symbol != EMPTY &&
board[toRow][toCol].isWhite) return true;
            if (toRow == 2 && lastMoveTo == make_pair(1, toCol) &&
               lastMoveFrom == make_pair(3, toCol) &&
               board[1][toCol].symbol == WHITE_PAWN) return true;
         }
         return false;
      case WHITE_KING: case BLACK_KING:
         if (absRowDiff <= 1 && absColDiff <= 1) return true;
         if (!piece.hasMoved && fromRow == toRow && absColDiff == 2) {
            bool kingside = colDiff > 0;
            int rookCol = kingside ? 7 : 0;
            bool& canCastleKingside = piece.isWhite ? whiteCanCastleKingside :
blackCanCastleKingside;
            bool& canCastleQueenside = piece.isWhite ? whiteCanCastleQueenside :
blackCanCastleQueenside;

            if ((kingside && !canCastleKingside) || (!kingside
&& !canCastleQueenside)) return false;
            if (board[fromRow][rookCol].symbol != (piece.isWhite ? WHITE_ROOK :
BLACK_ROOK) ||
                board[fromRow][rookCol].hasMoved || isKingInCheck(piece.isWhite))
return false;

            int step = kingside ? 1 : -1;
            for (int col = fromCol + step; col != rookCol; col += step) {
                if (board[fromRow][col].symbol != EMPTY) return false;
            }

            ChessPiece tempKing = board[fromRow][fromCol];
            for (int i = 1; i <= 2; i++) {
                board[fromRow][fromCol] = ChessPiece(EMPTY);
                board[fromRow][fromCol + i * step] = tempKing;
                if (isKingInCheck(piece.isWhite)) {
                    board[fromRow][fromCol] = tempKing;
                    board[fromRow][fromCol + i * step] = ChessPiece(EMPTY);
                    return false;
                }
                board[fromRow][fromCol + i * step] = ChessPiece(EMPTY);
            }
            board[fromRow][fromCol] = tempKing;
            return true;
        }
        return false;
    default:
        return canPieceAttack(piece, fromRow, fromCol, toRow, toCol);
    }
}

void ChessBoard::displayBoard(const vector<char>& whiteCaptures, const
vector<char>& blackCaptures) const {
    cout << "\033[38;5;117m+-----------------+\033[0m\n";
    cout << "\033[38;5;117m|  a b c d e f g h|\033[0m\n";
    cout << "\033[38;5;117m+-----------------+\033[0m\n";
    for (int row = 0; row < BOARD_SIZE; row++) {
        cout << 8 - row << "|";
        for (int col = 0; col < BOARD_SIZE; col++) {
            cout << " " << board[row][col].symbol;
        }
        cout << " |" << 8 - row << "\n";
    }
    cout << "\033[38;5;216m+-----------------+\033[0m\n";
    cout << "\033[38;5;216m|  a b c d e f g h|\033[0m\n";
    cout << "\033[38;5;216m+-----------------+\033[0m\n";
    cout << "\nHalf-moves: " << halfMoveClock << " Full moves: " <<
fullMoveNumber << "\n";
    cout << "\033[38;5;216mWhite captured: \033[0m";
    if (whiteCaptures.empty()) cout << "None";
    else for (char piece : whiteCaptures) cout << piece << " ";
    cout << "\n\033[38;5;117mBlack captured: \033[0m";
    if (blackCaptures.empty()) cout << "None";
    else for (char piece : blackCaptures) cout << piece << " ";
    cout << "\n\n";
}

bool ChessBoard::isKingInCheck(bool isWhite) const {
    pair<int, int> kingPos = findKing(isWhite);
    if (kingPos.first == -1) return false;

    for (int row = 0; row < BOARD_SIZE; row++) {
        for (int col = 0; col < BOARD_SIZE; col++) {
            ChessPiece piece = board[row][col];
            if (piece.symbol != EMPTY && piece.isWhite != isWhite) {
                if (canPieceAttack(piece, row, col, kingPos.first, kingPos.second)) {
                    return true;
                }
            }
        }
    }
    return false;
}

bool ChessBoard::movePiece(string move, bool isWhiteTurn, char promotion,
vector<char>& whiteCaptures, vector<char>& blackCaptures) {
    if (move.length() != 5 || move[2] != ' ') return false;

    int fromCol = move[0] - 'a';
    int fromRow = 8 - (move[1] - '0');
    int toCol = move[3] - 'a';
    int toRow = 8 - (move[4] - '0');

    if (!isValidPosition(fromRow, fromCol) || !isValidPosition(toRow, toCol)) return
false;

    ChessPiece piece = board[fromRow][fromCol];
    if (piece.symbol == EMPTY || piece.isWhite != isWhiteTurn) return false;

    if (!isValidMove(piece, fromRow, fromCol, toRow, toCol)) return false;

    bool isCapture = board[toRow][toCol].symbol != EMPTY;
    bool isPawnMove = piece.symbol == WHITE_PAWN || piece.symbol ==
BLACK_PAWN;
    bool isCastling = (piece.symbol == WHITE_KING || piece.symbol ==
BLACK_KING) && abs(toCol - fromCol) == 2;
    bool isEnPassant = isPawnMove && abs(toCol - fromCol) == 1 &&
board[toRow][toCol].symbol == EMPTY &&
                    ((piece.isWhite && toRow == 5) || (!piece.isWhite && toRow == 2));

    ChessPiece originalPiece = board[fromRow][fromCol];
    ChessPiece originalToPiece = board[toRow][toCol];
    pair<int, int> originalLastMoveFrom = lastMoveFrom;
    pair<int, int> originalLastMoveTo = lastMoveTo;
    ChessPiece originalRook;
    ChessPiece enPassantCaptured;
    int rookFromCol = 0, rookToCol = 0;

    bool kingInCheckBefore = isKingInCheck(isWhiteTurn);

    board[fromRow][fromCol] = ChessPiece(EMPTY);
    board[toRow][toCol] = ChessPiece(piece.symbol, piece.isWhite);
    board[toRow][toCol].hasMoved = true;
    lastMoveFrom = {fromRow, fromCol};
    lastMoveTo = {toRow, toCol};

    if (isCastling) {
        rookFromCol = (toCol > fromCol) ? 7 : 0;
        rookToCol = (toCol > fromCol) ? fromCol + 1 : fromCol - 1;
        originalRook = board[fromRow][rookFromCol];
        board[fromRow][rookToCol] = board[fromRow][rookFromCol];
        board[fromRow][rookFromCol] = ChessPiece(EMPTY);
        board[fromRow][rookToCol].hasMoved = true;
    }

    if (isEnPassant) {
        int capturedRow = toRow + (piece.isWhite ? 1 : -1);
        enPassantCaptured = board[capturedRow][toCol];
        board[capturedRow][toCol] = ChessPiece(EMPTY);
        isCapture = true;
    }

    if (isPawnMove && (toRow == 0 || toRow == 7)) {
        char promoted = piece.isWhite ? toupper(promotion) : tolower(promotion);
        if (promoted != 'Q' && promoted != 'R' && promoted != 'B' && promoted != 'N')
promoted = 'Q';
        board[toRow][toCol].symbol = promoted;
    }

    bool kingInCheckAfter = isKingInCheck(isWhiteTurn);
    bool invalidMove = (kingInCheckBefore && kingInCheckAfter) ||
(!kingInCheckBefore && kingInCheckAfter);
```

```
    board[fromRow][fromCol] = originalPiece;
    board[toRow][toCol] = originalToPiece;
    lastMoveFrom = originalLastMoveFrom;
    lastMoveTo = originalLastMoveTo;

    if (isCastling) {
        board[fromRow][rookFromCol] = originalRook;
        board[fromRow][rookToCol] = ChessPiece(EMPTY);
    }
    if (isEnPassant) {
        int capturedRow = toRow + (piece.isWhite ? 1 : -1);
        board[capturedRow][toCol] = enPassantCaptured;
    }

    if (invalidMove) return false;

    if (isCapture) {
        (isWhiteTurn ? whiteCaptures :
blackCaptures).push_back(originalToPiece.symbol);
    }

    board[fromRow][fromCol] = ChessPiece(EMPTY);
    board[toRow][toCol] = ChessPiece(piece.symbol, piece.isWhite);
    board[toRow][toCol].hasMoved = true;
    lastMoveFrom = {fromRow, fromCol};
    lastMoveTo = {toRow, toCol};

    if (isCastling) {
        rookFromCol = (toCol > fromCol) ? 7 : 0;
        rookToCol = (toCol > fromCol) ? fromCol + 1 : fromCol - 1;
        board[fromRow][rookToCol] = board[fromRow][rookFromCol];
        board[fromRow][rookFromCol] = ChessPiece(EMPTY);
        board[fromRow][rookToCol].hasMoved = true;
        if (piece.isWhite) {
            whiteCanCastleKingside = whiteCanCastleQueenside = false;
        } else {
            blackCanCastleKingside = blackCanCastleQueenside = false;
        }
    }

    if (isEnPassant) {
        int capturedRow = toRow + (piece.isWhite ? 1 : -1);
        board[capturedRow][toCol] = ChessPiece(EMPTY);
    }

    if (isPawnMove && (toRow == 0 || toRow == 7)) {
        char promoted = piece.isWhite ? toupper(promotion) : tolower(promotion);
        if (promoted != 'Q' && promoted != 'R' && promoted != 'B' && promoted != 'N')
promoted = 'Q';
```

```
        board[toRow][toCol].symbol = promoted;
    }

    if (piece.symbol == WHITE_ROOK && !piece.hasMoved) {
        if (fromCol == 0) whiteCanCastleQueenside = false;
        else if (fromCol == 7) whiteCanCastleKingside = false;
    }
    if (piece.symbol == BLACK_ROOK && !piece.hasMoved) {
        if (fromCol == 0) blackCanCastleQueenside = false;
        else if (fromCol == 7) blackCanCastleKingside = false;
    }
    if (piece.symbol == WHITE_KING) whiteCanCastleKingside =
whiteCanCastleQueenside = false;
    if (piece.symbol == BLACK_KING) blackCanCastleKingside =
blackCanCastleQueenside = false;

    halfMoveClock = (isCapture || isPawnMove) ? 0 : halfMoveClock + 1;
    if (!isWhiteTurn) fullMoveNumber++;
    return true;
}

bool ChessBoard::isCheckmate(bool isWhite) {
    if (!isKingInCheck(isWhite)) return false;
    return !hasLegalMoves(isWhite);
}

bool ChessBoard::isStalemate(bool isWhite) {
    if (isKingInCheck(isWhite)) return false;
    return !hasLegalMoves(isWhite);
}

bool ChessBoard::isDraw() const {
    return halfMoveClock >= 50;
}

bool ChessBoard::hasLegalMoves(bool isWhite) {
    for (int fromRow = 0; fromRow < BOARD_SIZE; fromRow++) {
        for (int fromCol = 0; fromCol < BOARD_SIZE; fromCol++) {
            ChessPiece piece = board[fromRow][fromCol];
            if (piece.symbol != EMPTY && piece.isWhite == isWhite) {
                for (int toRow = 0; toRow < BOARD_SIZE; toRow++) {
                    for (int toCol = 0; toCol < BOARD_SIZE; toCol++) {
                        if (isValidMove(piece, fromRow, fromCol, toRow, toCol)) {
                            ChessPiece originalPiece = board[fromRow][fromCol];
                            ChessPiece originalToPiece = board[toRow][toCol];
                            pair<int, int> originalLastMoveFrom = lastMoveFrom;
                            pair<int, int> originalLastMoveTo = lastMoveTo;

                            board[toRow][toCol] = ChessPiece(piece.symbol, piece.isWhite);
                            board[toRow][toCol].hasMoved = true;
                            board[fromRow][fromCol] = ChessPiece(EMPTY);
                            lastMoveFrom = {fromRow, fromCol};
                            lastMoveTo = {toRow, toCol};

                            bool stillInCheck = isKingInCheck(isWhite);

                            board[fromRow][fromCol] = originalPiece;
                            board[toRow][toCol] = originalToPiece;
                            lastMoveFrom = originalLastMoveFrom;
                            lastMoveTo = originalLastMoveTo;

                            if (!stillInCheck) return true;
                        }
                    }
                }
            }
        }
    }
    return false;
}

void ChessBoard::saveGame(const string& filename) {
    ofstream file(filename);
    if (file.is_open()) {
        file << (whiteCanCastleKingside ? "1" : "0") << " "
             << (whiteCanCastleQueenside ? "1" : "0") << " "
             << (blackCanCastleKingside ? "1" : "0") << " "
             << (blackCanCastleQueenside ? "1" : "0") << " "
             << halfMoveClock << " " << fullMoveNumber << endl;
        for (int row = 0; row < BOARD_SIZE; row++) {
            for (int col = 0; col < BOARD_SIZE; col++) {
                file << board[row][col].symbol;
            }
            file << endl;
        }
        file.close();
    }
}

void ChessBoard::loadGame(const string& filename) {
    ifstream file(filename);
    if (file.is_open()) {
        file >> whiteCanCastleKingside >> whiteCanCastleQueenside
             >> blackCanCastleKingside >> blackCanCastleQueenside
             >> halfMoveClock >> fullMoveNumber;
        file.ignore();
        for (int row = 0; row < BOARD_SIZE; row++) {
            string line;
            if (getline(file, line) && line.length() >= BOARD_SIZE) {
                for (int col = 0; col < BOARD_SIZE; col++) {
                    board[row][col] = ChessPiece(line[col], line[col] >= 'A' && line[col] <=
'Z');
                }
                }
        }
        file.close();
    }
}

ChessGame::ChessGame() : whiteTurn(true), checkCount(0) {
    whiteCaptures.clear();
    blackCaptures.clear();
}

void ChessGame::displayWelcomeMessage() {
    cout << "\n\n\033[38;5;183m*     Welcome to Chess Master     *\033[0m\n";
    cout <<
"\033[38;5;183m*********************************************\033[0m\n
";
    cout << "\033[38;5;30m     Enter moves as 'e2 e4'       \033[0m\n";
    cout << "\033[38;5;30m     Promotion: 'd7 d8=Q'         \033[0m\n";
    cout << "\033[38;5;30m   Castling: 'e1 g1' or 'e8 g8'   \033[0m\n";
    cout << "\033[38;5;30m     Type 'exit' to end game      \033[0m\n";
    cout << "\033[38;5;30m     White moves first            \033[0m\n";
    cout <<
"\033[38;5;183m*********************************************\033[0m\
n";
    this_thread::sleep_for(chrono::seconds(1));
    cout << "\n";
}

void ChessGame::displayRules() {
    cout << "\n\n\033[38;5;30m*------------------------------------------*\033[0m\n";
    cout << "\033[48;5;30m|               Chess Rules               |\033[0m\n";
    cout << "\033[38;5;30m*------------------------------------------*\033[0m\n";
    cout << "\033[38;5;30m|              1. White moves first        |\033[0m\n";
    cout << "\033[38;5;30m|                                          |\033[0m\n";
    cout << "\033[38;5;30m|           2. Game ends after 2 checks    |\033[0m\n";
    cout << "\033[38;5;30m|                                          |\033[0m\n";
    cout << "\033[38;5;30m|----------3. Pieces move as follows:--------|\033[0m\n";
    cout << "\033[38;5;30m|        *   King: 1 square any dir        |\033[0m\n";
    cout << "\033[38;5;30m|        -  Queen: Any dir, any dist       |\033[0m\n";
    cout << "\033[38;5;30m|        *   Rook: Horz/vert any dist      |\033[0m\n";
    cout << "\033[38;5;30m|        -  Bishop: Diag any dist          |\033[0m\n";
    cout << "\033[38;5;30m|        *   Knight: L-shape (2x1)         |\033[0m\n";
    cout << "\033[38;5;30m|        -  Pawn: 1 forward, 2 start       |\033[0m\n";
    cout << "\033[38;5;30m|                                          |\033[0m\n";
    cout << "\033[38;5;30m| 4. Capture by landing on opponent's piece  |\033[0m\n";
    cout << "\033[38;5;30m|                                          |\033[0m\n";
    cout << "\033[38;5;30m|          5. Special moves:               |\033[0m\n";
```

```
   cout << "\033[38;5;30m|                                        |\033[0m\n";
   cout << "\033[38;5;30m|      - Castling (King+Rook)            |\033[0m\n";
   cout << "\033[38;5;30m|      - En passant (Pawn)               |\033[0m\n";
   cout << "\033[38;5;30m|      - Promotion (Pawn)                |\033[0m\n";
   cout << "\033[38;5;30m*----------------------------------------*\033[0m\n";
   cout << "\n\033[45mPress Enter to return to menu...\033[0m";
   cin.get();
}

void ChessGame::start() {
   displayWelcomeMessage();
   string move;
   while (true) {
      board.displayBoard(whiteCaptures, blackCaptures);
      cout << (whiteTurn ? "\033[38;5;216m White" : "\033[38;5;117mBlack") << "'s
turn. Enter move: \033[0m";
      getline(cin, move);

      if (move == "exit") {
         cout << "\033[31m            +-----------------+\033[0m\n";
         cout << "\033[31m            |   Game Over!   |\033[0m\n";
         cout << "\033[31m            +-----------------+\033[0m\n";
         break;
      }

      char promotion = 'Q';
      if (move.length() == 7 && move[5] == '=') {
         promotion = move[6];
         move = move.substr(0, 5);
      }

      if (board.movePiece(move, whiteTurn, promotion, whiteCaptures,
blackCaptures)) {
         bool opponentInCheck = board.isKingInCheck(!whiteTurn);
         if (opponentInCheck) {
            checkCount++;
            board.displayBoard(whiteCaptures, blackCaptures);
            cout << "\033[31m+--------------------------------------------------
+\033[0m\n";
            cout << "\033[31m|****************        Check!
*************|\033[0m\n";
            cout << "\033[31m+--------------------------------------------------
+\033[0m\n";

            if (checkCount >= 4) {
               cout <<
"\033[38;5;30m\n############################################################
##############+\033[0m\n";
               cout << "\033[38;5;30m|                Game Over!
|\033[0m\n";
```

```
                cout << "\033[38;5;30m|                    " << (whiteTurn ? "Black" : "White")
<< "\033[38;5;30m wins after second check!                |\033[0m\n";
                cout <<
"\033[38;5;30m+###########################################################
##############+\033[0m\n";
                break;
            }
        }
        whiteTurn = !whiteTurn;
    } else {
        board.displayBoard(whiteCaptures, blackCaptures);
        cout << "\033[31m|*******************  Invalid move!
**********************|\033[0m\n";
    }
  }
}

void ChessGame::saveGame(const string& filename) {
  board.saveGame(filename);
  cout << "\033[38;5;183m+-----------------+\033[0m\n";
  cout << "\033[45m|   Game saved!   \033[0m|\n";
  cout << "\033[38;5;183m+-----------------+\033[0m\n";
}

void ChessGame::loadGame(const string& filename) {
  board.loadGame(filename);
  cout << "\033[38;5;183m+-----------------+\033[0m\n";
  cout << "\033[45m|   Game loaded!  |\033[0m\n";
  cout << "\033[38;5;183m+-----------------+\033[0m\n";
}

void displayMenu() {
  cout << "\033[38;5;183m\n*-------------------------*\033[0m\n";
  cout << "\033[38;5;183m|      Chess Master       |\033[0m\n";
  cout << "\033[38;5;183m*------------------------*\033[0m\n";
  cout << "\033[45m|     1. Start Game      |\033[0m\n";
  cout << "\033[45m|     2. Save Game       |\033[0m\n";
  cout << "\033[45m|     3. Load Game       |\033[0m\n";
  cout << "\033[45m|     4. View Rules      |\033[0m\n";
  cout << "\033[45m|     5. Exit            |\033[0m\n";
  cout << "\n\033[38;5;183mEnter choice: \033[0m";
}

int main() {
  ChessGame game;
  int choice;
  while (true) {
    displayMenu();
    cin >> choice;
    cin.ignore();
```

```
        switch (choice) {
            case 1: game.start(); break;
            case 2: {
                string filename;
                cout << "Enter filename to save: ";
                getline(cin, filename);
                game.saveGame(filename);
                break;
            }
            case 3: {
                string filename;
                cout << "Enter filename to load: ";
                getline(cin, filename);
                game.loadGame(filename);
                break;
            }
            case 4: game.displayRules(); break;
            case 5:
                cout <<
"\n\033[38;5;183m*++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++*\033[0m\n";
                cout << "\033[45m|     Thank you for playing the Chess! HAVE A NICE
DAY      |\033[0m\n";
                cout <<
"\033[38;5;183m*++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++*\033[0m\n";
                return 0;
            default:
                cout << "\033[38;5;30m    +-----------------+\033[0m\n";
                cout << "\033[38;5;30m    | Invalid choice! |\033[0m\n";
                cout << "\033[38;5;30m    +-----------------+\033[0m\n";
        }
    }
    return 0;
}
```

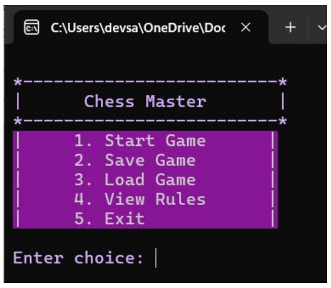
7. Console Output Specification

The game uses std::cout to show messages in the console, and ANSI color codes are used to add color. These colors appear based on what the player does or game events.

a) Main Menu

ANSI color means using special codes in our C++ program to add color or styling (like bold or underline) to text shown in the terminal or console.

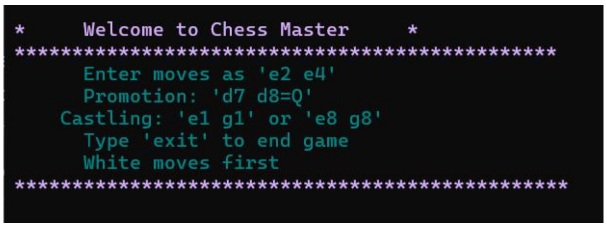
Light purple header (ANSI: \033[38;5;183m), purple background options (\033[45m).



```
C:\Users\devsa\OneDrive\Doc

*-------------------------*
|      Chess Master       |
*-------------------------*
|      1. Start Game     |
|      2. Save Game      |
|      3. Load Game      |
|      4. View Rules     |
|      5. Exit           |

Enter choice: |
```

b) Welcome Message

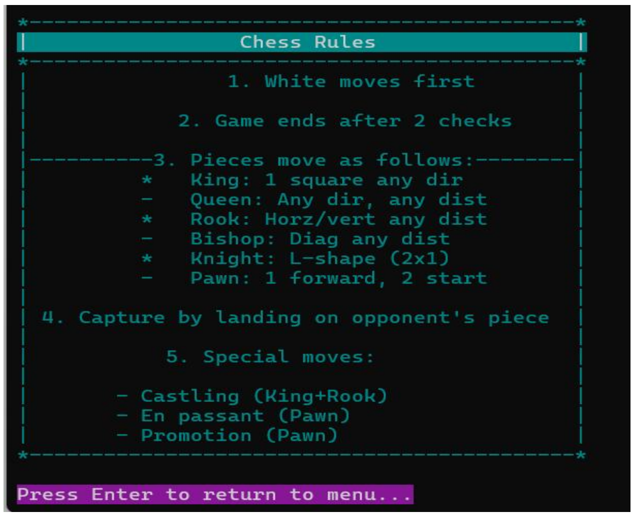
Light purple header (ANSI: \033[38;5;183m), teal instructions (ANSI:\033[38;5;30m), 1-second pause.



```
*     Welcome to Chess Master     *
**************************************************
      Enter moves as 'e2 e4'
      Promotion: 'd7 d8=Q'
    Castling: 'e1 g1' or 'e8 g8'
      Type 'exit' to end game
      White moves first
***************************************************
```

c) Game Rules Display

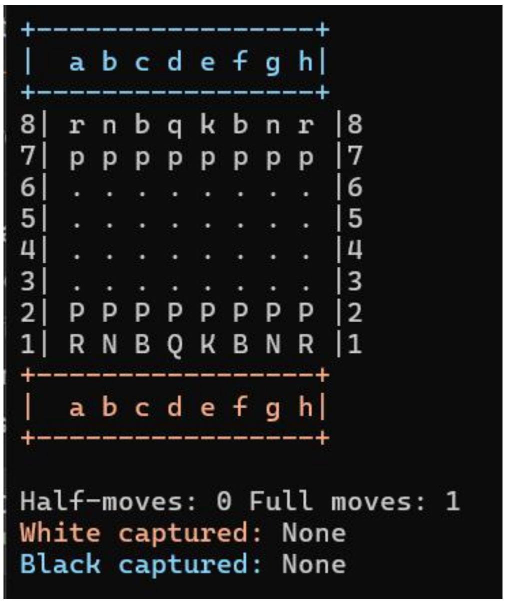
Teal rules (ANSI: \033[38;5;30m), purple prompt (ANSI: \033[45m).



```
*---------------------------------------------*
|                 Chess Rules                 |
*---------------------------------------------*
|                1. White moves first         |
|                                             |
|            2. Game ends after 2 checks      |
|                                             |
|----------3. Pieces move as follows:---------|
|          *   King: 1 square any dir         |
|          -   Queen: Any dir, any dist       |
|          *   Rook: Horz/vert any dist       |
|          -   Bishop: Diag any dist          |
|          *   Knight: L-shape (2x1)          |
|          -   Pawn: 1 forward, 2 start       |
|                                             |
| 4. Capture by landing on opponent's piece   |
|                                             |
|           5. Special moves:                 |
|                                             |
|        - Castling (King+Rook)               |
|        - En passant (Pawn)                  |
|        - Promotion (Pawn)                   |
*---------------------------------------------*

Press Enter to return to menu...
```

d) Chess Board Display

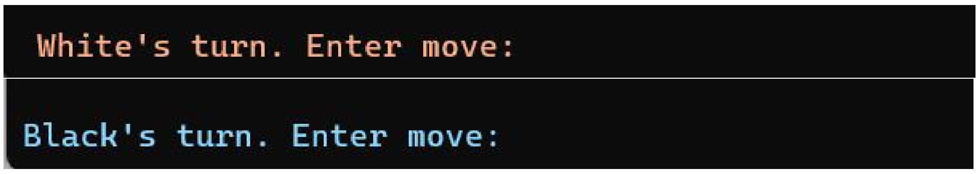
Light Blue and Peach Borders (ANSI: \033[38;5;117m, \033[38;5;216m)



```
+-----------------+
|  a b c d e f g h|
+-----------------+
8| r n b q k b n r |8
7| p p p p p p p p |7
6| . . . . . . . . |6
5| . . . . . . . . |5
4| . . . . . . . . |4
3| . . . . . . . . |3
2| P P P P P P P P |2
1| R N B Q K B N R |1
+-----------------+
|  a b c d e f g h|
+-----------------+

Half-moves: 0 Full moves: 1
White captured: None
Black captured: None
```

e) Move Prompt

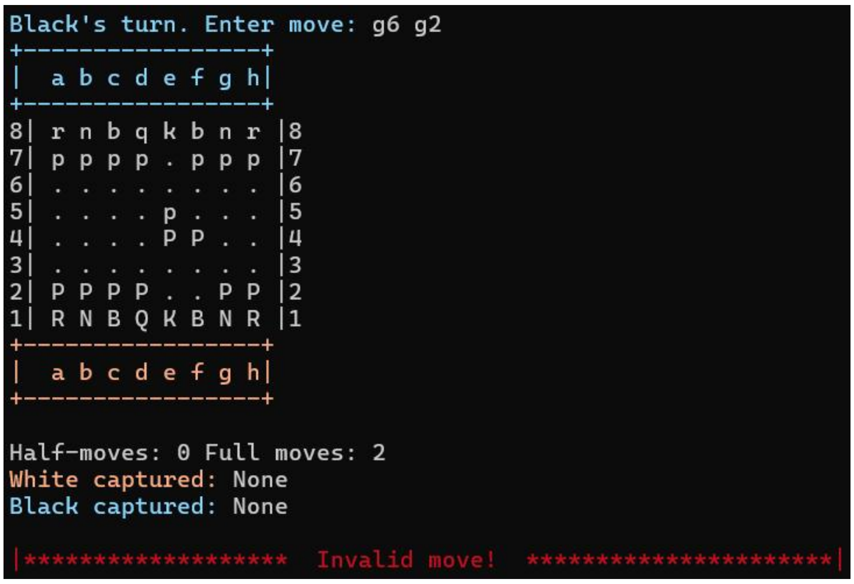
Light Blue and Peach Players (ANSI: \033[38;5;117m, \033[38;5;216m)



```
White's turn. Enter move:
```

```
Black's turn. Enter move:
```

f) Invalid Move Message

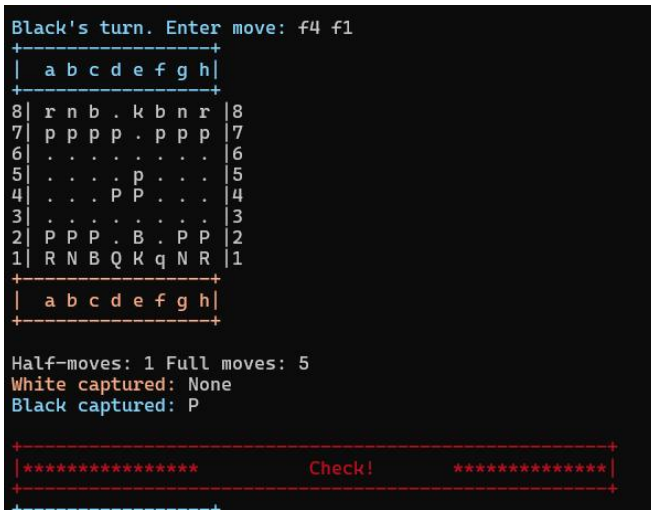


```
Black's turn. Enter move: g6 g2
+-----------------+
|  a b c d e f g h|
+-----------------+
8| r n b q k b n r |8
7| p p p p . p p p |7
6| . . . . . . . . |6
5| . . . . p . . . |5
4| . . . . P P . . |4
3| . . . . . . . . |3
2| P P P P . . P P |2
1| R N B Q K B N R |1
+-----------------+
|  a b c d e f g h|
+-----------------+

Half-moves: 0 Full moves: 2
White captured: None
Black captured: None

|*******************  Invalid move!  **********************|
```

Red (ANSI: \033[31m).

g) Check Message

Red (ANSI: \033[31m).

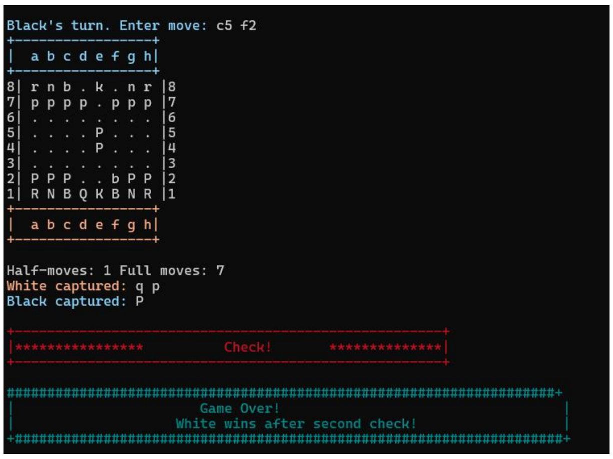


```
Black's turn. Enter move: f4 f1
+-----------------+
|  a b c d e f g h|
+-----------------+
8| r n b . k b n r |8
7| p p p p . p p p |7
6| . . . . . . . . |6
5| . . . . p . . . |5
4| . . . P P . . . |4
3| . . . . . . . . |3
2| P P P . B . P P |2
1| R N B Q K q N R |1
+-----------------+
|  a b c d e f g h|
+-----------------+

Half-moves: 1 Full moves: 5
White captured: None
Black captured: P

+-----------------------------------------------------+
|****************          Check!       **************|
+-----------------------------------------------------+
```

h) Game Over Messages

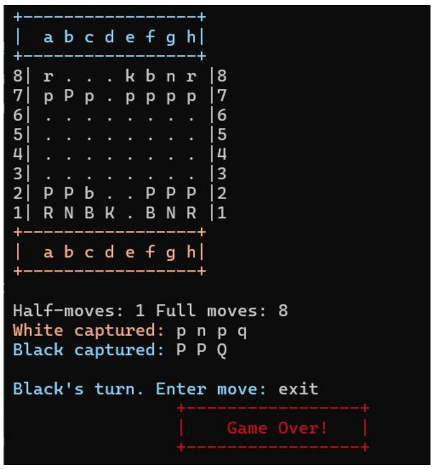
Teal for four checks (ANSI: \033[38;5;30m), red for exit (ANSI: \033[31m).

- ***Four checks***



```
Black's turn. Enter move: c5 f2
+-----------------+
|  a b c d e f g h|
+-----------------+
8| r n b . k . n r |8
7| p p p p . p p p |7
6| . . . . . . . . |6
5| . . . . P . . . |5
4| . . . . P . . . |4
3| . . . . . . . . |3
2| P P P . . b P P |2
1| R N B Q K B N R |1
+-----------------+
|  a b c d e f g h|
+-----------------+

Half-moves: 1 Full moves: 7
White captured: q p
Black captured: P

+-----------------------------------------------------+
|****************          Check!       **************|
+-----------------------------------------------------+

####################################################################+
|                          Game Over!                                |
|                  White wins after second check!                    |
+####################################################################+
```

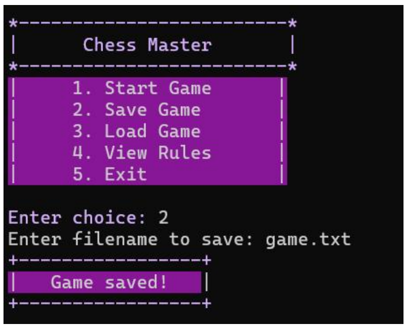
- ***Exit***



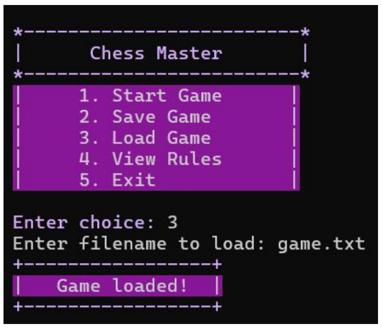
```
+-----------------+
|  a b c d e f g h|
+-----------------+
8| r . . . k b n r |8
7| p P p . p p p p |7
6| . . . . . . . . |6
5| . . . . . . . . |5
4| . . . . . . . . |4
3| . . . . . . . . |3
2| P P b . . P P P |2
1| R N B K . B N R |1
+-----------------+
|  a b c d e f g h|
+-----------------+

Half-moves: 1 Full moves: 8
White captured: p n p q
Black captured: P P Q

Black's turn. Enter move: exit
                +------------------+
                |    Game Over!    |
                +------------------+
```

i) Save Game and Load Game Confirmation

Purple background (ANSI: \033[45m), light purple borders (ANSI: \033[38;5;183m).



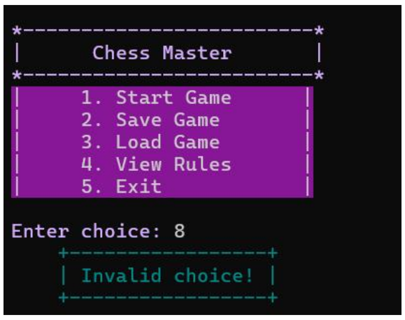
```
*-------------------------*
|       Chess Master      |
*-------------------------*
|       1. Start Game    |
|       2. Save Game     |
|       3. Load Game     |
|       4. View Rules    |
|       5. Exit          |

Enter choice: 2
Enter filename to save: game.txt
+-----------------+
|   Game saved!   |
+-----------------+
```



```
*-------------------------*
|       Chess Master      |
*-------------------------*
|       1. Start Game    |
|       2. Save Game     |
|       3. Load Game     |
|       4. View Rules    |
|       5. Exit          |

Enter choice: 3
Enter filename to load: game.txt
+-----------------+
|   Game loaded!  |
+-----------------+
```

j) Invalid Menu Choice

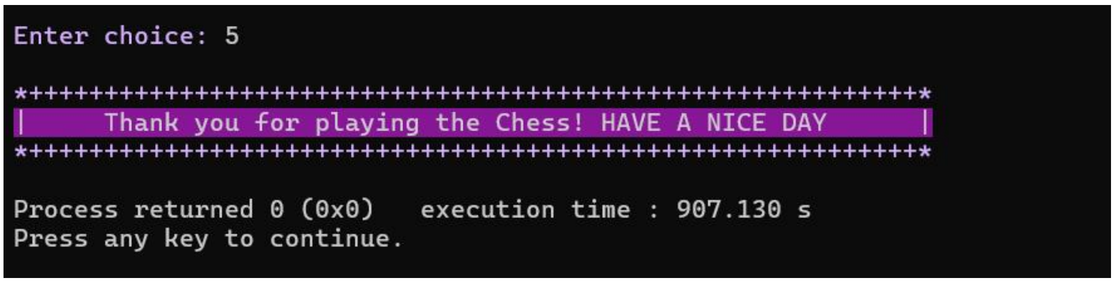
Teal (ANSI: \033[38;5;30m)



```
*-------------------------*
|       Chess Master      |
*-------------------------*
|       1. Start Game    |
|       2. Save Game     |
|       3. Load Game     |
|       4. View Rules    |
|       5. Exit          |

Enter choice: 8
    +-----------------+
    | Invalid choice! |
    +-----------------+
```

k) Exit Message

Purple background (ANSI: \033[45m), light purple borders (ANSI: \033[38;5;183m).



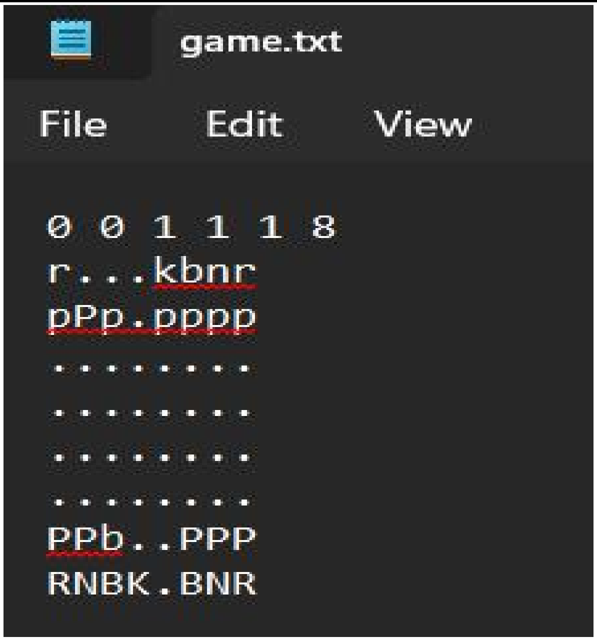
```
Enter choice: 5

************************************************************
|      Thank you for playing the Chess! HAVE A NICE DAY     |
************************************************************

Process returned 0 (0x0)   execution time : 907.130 s
Press any key to continue.
```

8. File Output Specification

Saved Game File:

a. ***First line***: Four binary digits **(0/1)** for castling, integers for half-move clock, full move number **(e.g: 1 1 1 1 0 1)**.

b. ***Board rows***: Piece symbols (**K, Q,** etc. for **white**; k, q, etc. for black; . for empty).



```
0 0 1 1 1 8
r...kbnr
pPp.pppp
........
........
........
........
PPb..PPP
RNBK.BNR
```

9. System Information

a) Dependencies

The program uses standard C++ libraries:

- *<iostream>* – For displaying messages and reading input from the console
- *<vector>* – Used for the 2D chessboard and storing captured pieces
- *<string>* – Handles text input and strings
- *<fstream>* – Reads from and writes to files for saving and loading games
- *<thread>* and *<chrono>* – Adds pauses, like in the welcome message

b) Data Structures

- ***Board:*** A 2D vector ***(std::vector<std::vector<ChessPiece>>)*** is used to store the pieces on the chessboard
- ***Captures:*** Captured pieces are stored in vectors of characters ***(std::vector<char>)***
- ***Move Tracking:*** Moves are represented using pairs of integers ***(std::pair<int, int>)***

c) Error Handling

- Validates move formats, legality, and menu inputs with clear error messages.

10. Setup and Usage in Code::Blocks

a. Setup

- Put *ChessGame.h* and *ChessGame.cpp* in the same folder.
- Open *Code::Blocks* and create a new Console Application project using *C++ (GNU GCC Compiler).*
- Add the *two files* to your project (Project > Add files).
- Press *Ctrl+F9* to build the project, and *Ctrl+F10* to run it.

b. Usage

Main Menu Options:

Choose from the menu:

- ***1*** to start a new game
- ***2*** to save
- ***3*** to load
- ***4*** to view the rules
- ***5*** to exit

During Gameplay:

- Make a move like: *e2 e4*
- Promote a pawn like: *d7 d8=Q*
- Castle like: *e1 g1*
- Type *exit* to leave the game

Saving/Loading:

- Enter the filename when prompted, for example: *game.txt*

11. Limitations

- The game is designed for two human players only with no computer opponent (AI)
- The four check rule takes priority over regular game endings
- Some advanced draw rules like threefold repetition are not included
- Requires ANSI color support.
- File I/O assumes valid file formats.

12. Testing and Verification

a) ***Menu***: Try out every menu option to make sure they work correctly

b) ***Gameplay***: Test normal moves, special moves (like castling and promotion), and game endings (like the four check rule and using exit)

c) ***Save and Load***: Make sure the game saves properly, loads from the file, and keeps all the game info

d) ***Output***: Check that colored text displays correctly in a terminal that supports ANSI colors

13. Conclusion

This documentation details the Chess Master program's architecture, functionality, output, and source code, highlighting 2D array manipulation, input handling, state management, and collision detection. It is designed for evaluators to understand and run the program in Code::Blocks.