Introduction

There are lots of implementations of Chess' games available on the internet, most of them richer in features than this one. Nevertheless, there's no demerit on developing a simpler, lightweight piece of software, specially aiming for didatic purposes.

What this game is (or tries to be):

Lightweight. The size of version 1.0 of the application is 155 KB.

Implemented fully on console

What this game is not/does not have:

Does not have a GUI

Does not have artificial intelligence (AI)

Background

This game runs in a console, i.e., that means no GUI is available to the user. All the input is taken from the keyboard, and for that, it uses the Coordinate Notation.

The white pieces are represented by capital letters and the black pieces are represented in lowercase letters. They are all represented by the first letter of their names, the only exception being the Knight, which is represented by an N, leaving the K for the king):

Pawn

Rook

Knight

Bishop

Queen

King

I will try to explain some of the concepts I used when developing the game, if anything is not clear or if I missed an important point, please let me know in the discussion.

Drawing the Board

We can use the ASCII characters 0xDB and 0xFF to draw white and black cells, respectively.

C++

Copy Code

#define WHITE\_SQUARE 0xDB

#define BLACK\_SQUARE 0xFF

First, we have to decide how big we want the squares do be. Speaking about the height, should one square on the board be as big as one single character? Or maybe two or three?

I ended up choosing the third option, which means the height of one square equals to three characters. Now, we face another problem. The mentioned characters (0xDB and 0xFF) are not squared; they are actually rectangular with one side being twice as big as the other. This means that, in order to form a square, we have to use six characters in a row.

Structure of the Code

The code consists of three .cpp files:

main.cpp: Entry-point of the application. Prompts the user for an action (new game, move, undo, save, load, quit) and, depending on the action to be performed, prompts for more information and call the functions from the other files.

chess.cpp: consists of two classes. The first one is named Chess and contains enums, structs and simple functions to describe chess pieces, colors and the board. The second one is called Game, it inherits from Chess. It stores all the information that a single game has, like the position of every piece in the board, list of moves made, list of pieces captured. It also contained functions to determine if the king is in check, if castling is allowed, if a square is occupied, everything that is necessary to verify if a move is valid or not.

user\_interface.cpp: Basically consists of functions printing information to the console, like printing the board, last moves, menu, messages for the user, etc.

I have designed the application in a way that, if the user interface is to be improved (for example, if someone decides to fork this code and develop a GUI), no changes should be made in the chess.cpp file. Needed changes would be basically to replace the user\_interface.cpp file with a new interface and replace the calls to that interface in the main.cpp file.

Validating a Move

C++

Copy Code

bool isMoveValid(Chess::Position present, Chess::Position future,

Chess::EnPassant\* S\_enPassant, Chess::Castling\* S\_castling)

[See main.cpp, line 19]

After the user has entered the command to move a piece, several things must be checked to verify if it is a valid move.

Is the desired piece allowed to move in that direction? Here, we have to create a switch with cases for all types of pieces. Knights, rooks, bishops and queen are less complicated because they always move in the same fashion. Pawns, in the other hand, move vertically but are allowed to move diagonally to capture a piece. Also, they have the option of advancing two squares, but only if it is its first move. And there is even the "en passant" move, when the pawn moves forward and yet captures a piece. The King can move one square to every direction, but when castling is applied, it can move two squares (but only if it's the first move for both the king and the rook involved in the move).

Is there another piece of the same color on the destination square? If positive, then the move must be invalidated. If there is a piece, but from the other color, then this piece will be captured.

Would this move put the king in check? Either if the king was already in check or not, we need to check if, after that move, the king would be under immediate attack by any opponent piece.

Storing the Moves and Captured Pieces

We're taking advantage of some containers provided by C++ to store the game information. But first, I created a simple structure that stores one white and one black move. Each move is a tring that contains the position of the piece to be moved, followed by a dash, and the destination square, e.g., E2-E4.

C++

Copy Code

struct Round

{

string white\_move;

string black\_move;

};

A double-ended queue was the data structure I chose to store the Rounds. It's a versatile structure, which accepts inserting and deleting elements from both the beginning and the end of the queue. Declaration and examples of use are as follows:

C++

Copy Code

std::deque<Round> rounds;

// How many rounds are stored?

rounds.size()

// Access a round

rounds[i].white\_move.c\_str()

// Clear the container

rounds.clear();

// Insert or remove elements

rounds.pop\_back();

rounds.push\_back(round);

For the captured pieces, these are stored in vectors:

C++

Copy Code

// Save the captured pieces

std::vector<char> white\_captured;

std::vector<char> black\_captured;

And they can't be printed on the screen like this:

C++

Copy Code

cout << "WHITE captured: ";

for (unsigned i = 0; i < game.white\_captured.size(); i++)

{

cout << char(game.white\_captured[i]) << " ";

}

cout << "black captured: ";

for (unsigned i = 0; i < game.black\_captured.size(); i++)

{

cout << char(game.black\_captured[i]) << " ";

}

Playing the Game

Starting a New Game

Start the app and press N, followed by ENTER, to start a new game. The board is shown and it's WHITE turn.

Make a Move

Type M to make a move.

You will be prompted to choose a piece to be moved. Do it by entering two characters (uppercase or lowercase will give the same results) describing first the column, then the row where the piece you want to be moved currently is. For example, the white pawn in front of the king is the E2 square.

Next, you'll be prompted for the destination square. One of the most common moves is moving the pawn from E2 to E4.

You will be warned if the move is invalid.

Undo a Move

Simply type U followed by ENTER to undo the last move. It is possible to undo only the very last move.

Checkmate

C++

Copy Code

bool Game::isCheckMate()

[See chess.cpp, line 1394.]

After every move, we must check if a checkmate has taken place. These are the steps to be followed:

Is the king in check? If not, no need to check any further.

Can the king move to another square? If the king can move to another square and not be under attack anymore, than it's not checkmate.

Can the attacker be taken or another piece get in the way? If the attacker can be taken, then it's not a checkmate. If it can't, there's still the possibility for another piece to get in the middle of the way between the attacker and the king.

If the answers to questions two and three are NO, then it's a checkmate and the game is over!

Saving / Loading a Game

Save a game is useful if you want to finish it later, but also it is an incredibly useful debugging tool. It was tedious to begin every time with all the pieces in their original positions if I'm testing a checkmate, a castling or even an 'en passant' move. Being able to save the game on a particular position, correcting the code and testing again from the same point proved to be an extraordinary tool.

How was it done? When the user types 'S' on the menu to save the game, he's prompted for a name. The application will create (or override) a file called 'name\_entered.dat' on the same directory as the executable. You can open the file with notepad++ and have a look, if you are curious. The first couple lines of the file could look like this:

Copy Code

[Chess console] Saved at: Fri Feb 9 00:07:43 2018

E2-E4 | C7-C5

C2-C3 | D7-D5

Time and date were included for debugging purposes.

Above the header lines, all moves are printed, one round per line, always starting with the white player. So, in that case, White started advancing the pawn on E2 to E4 and Black advanced the pawn from C7 to C5. (If you're wondering if this is a good move from Black, well, it was made by Gary Kasparov against Deep Blue).

Since all the moves are stored in a double-ended queue, it is easy to print that information to a file, as you can see below

When the user wants to load a saved game, the application prompts the user for the name of the file (again, without the .dat extension). After that, the steps are: first, check if the file exists and open. After skipping the first line (header), every line should be read, split into White and Black moves, and every move must be verified for validity.

Is that really necessary? Well, we sure validated all the moves before saving, but we cannot guarantee that the file hasn't been tampered with, so it's better to be on the safe side and verify again.

You can find on the github project page a bunch of saved games that helped me test and debug the game. Pay special attention to the KasparovVSdeepblue\_game\_1.dat. It was a lot of fun for me to recreate every move from Game 1 between Deep Blue versus Kasparov, 1996. It is an important game because it was the first game to be won by a chess-playing computer against a reigning world champion under normal chess tournament conditions and classical time controls.

Bugs

This application is certainly not bug-free. If you encounter an error, a crash, an invalid situation in the game, etc., please email me a screenshot or (even better) save the game and send me the .dat file. Your help is much appreciated!

Improvements / Future Steps

Chess Symbols in Unicode

Not everyone knows that there are chess symbols in Unicode. Nevertheless, it's not that straightforward to output them to a console. Two caveats are:

The console must output text in Unicode

The font used by the console must implement the glyphs for the chess pieces (not true for all fonts)

With the following source code and ConEmu terminal emulator, I managed to print the pieces in Unicode.

However, after pondering about this matter for a while, I decided only a few users would be able to display the pieces correctly, so it was not worth the effort. Nevertheless, I'm curious to see if anyone reading this will feel challenged to draw the board and the pieces with chess glyphs.

Graphical User Interface

One of the obvious improvements this game could benefit is a beautiful GUI. Plenty of options here: wxWidgets, Windows Forms and Windows Presentation Foundation (WPF), to name a few.

Since all the logic of the chess game is implemented in two classes in the Chess.cpp file, it can be built into a DLL which can be accessed by other programming languages.

If you feel compelled to address any of the improvements I suggested, you're welcome to fork the project on GitHub and let's discuss it further!

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)