CS246 Spring 2024 Project – Chess

C. Kierstead, J. Schmitz

- DO NOT EVER SUBMIT TO MARMOSET WITHOUT COMPILING AND TESTING FIRST. If your final submission doesn't compile, or otherwise doesn't work, you will have nothing to show during your demo. Resist the temptation to make last-minute changes. They probably aren't worth it.

- This project is intended to be doable by three people in two weeks. Because the breadth of students' abilities in this course is quite wide, exactly what constitutes two weeks' worth of work for three students is difficult to nail down. Some groups will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.

- Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

- **Note:** If you have not already done so for Assignment 4, make sure you are able to use graphical applications from your Unix session. If you are using Linux you should be fine (if making an ssh connection to a campus machine, be sure to pass the `-Y` option). If you are using Windows and PuTTY, you should download and run an X server such as XMing, and be sure that PuTTY is configured to forward X connections. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).

  Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. Try executing the following:

  ```
  g++20i window.cc graphicsdemo.cc -o graphicsdemo -lX11
  ```

The Game of Chess

In this project, you and your partner(s) will work together to produce a chess game.

The rules of chess are readily available online, so we outline them only briefly here.

Chess is played on an 8x8 checkerboard, arranged so that there is a white square at the bottom right. Players take turns making one move at a time. The player controlling the white pieces makes the first move.

There are six types of pieces:

- **King (K)** Moves one square in any direction.

- **Queen (Q)** Moves in any of the eight possible directions, any distance, but cannot move past any piece that blocks its path.

- **Bishop (B)** Moves in any of the four diagonal directions, any distance, but cannot move past any piece that blocks its path.

- **Rook (R)** Moves in any of the four vertical/horizontal directions, any distance, but cannot move past any piece that blocks its path.

- **Knight (N)** If it sits on square (x, y), it can move to square $(x \pm 2, y \pm 1)$ or $(x \pm 1, y \pm 2)$. Can "jump over" any piece that blocks its path.

- **Pawn (P)** Moves one square forward.

A piece *captures* another piece by moving onto the square occupied by that piece. The captured piece is then permanently removed from the board. A piece that could capture another piece is said to attack that piece. A piece may only capture a piece of the opposite colour.

The object of the game is to place your opponent's king under attack, such that your opponent's king cannot escape in one move. This is known as *checkmate*. An attack on the king, whether it can escape or not, is known as *check*.

The following additional rules govern the movement of pieces:

- The pawn is the only piece whose standard move is different from its capturing move. A pawn moves only forward, but it captures on the forward diagonals (one square). Thus, on capturing, a pawn must move diagonally forward, one square, to take over a square occupied by another piece.

- A pawn, on its first move, may move either one square forward or two squares forward.

- If a pawn, by moving two squares forward, avoids capture by another pawn (i.e, if moving one square forward would have put it under attack by another pawn), the would-be attacking pawn may still capture it by moving one square diagonally forward to the square the other pawn skipped over. This is known as pawn capture *en passant*. This option is only available immediately following the two-square move by the opposing pawn. If you wait, you can't do it.

- A pawn, upon reaching the other end of the board is replaced by either a rook, knight, bishop, or queen (your choice).

- A move known as *castling* helps to move the king to a safer square while simultaneously mobilizing a rook. To execute it, the king moves two squares towards one of the rooks, and that rook then occupies the square "skipped over" by the king. This happens in one move. For castling to be legal, the king and rook used must not previously have moved in the game; there must be no pieces between the king and rook used; and the king must not be in check on either its starting position, its final position, or the position in between (to be occupied by the rook).

- It is not legal to make any move that puts your king in check.

- If any player ever has no legal moves available, but is not in check, it is known as *stalemate*, and the game is a draw.

Display

You need to provide both a text-based display and a graphical display of your game board. A sample text display follows:

```
8 rnbqkbnr
7 pppppppp
6 _ _ _ _
5 _ _ _ _
4 _ _ _ _
3 _ _ _ _
2 PPPPPPPP
1 RNBQKBNR

  abcdefgh
```

In this display, capital letters denote white pieces, and lower case letters denote black pieces. Unoccupied squares are denoted by a blank space for white squares, and an underscore character for dark squares. The above board also represents the initial configuration of the game.

After every move, the board must be redisplayed. If one player or the other is in check, additionally display `White is in check.` or `Black is in check.`, as appropriate. If one player has won the game, display `Checkmate! White wins!` or `Checkmate! Black wins!` If the game is stalemated, output `Stalemate!` If the game is resigned, output `White wins!` or `Black wins!`, as appropriate.

Your graphical display should be set up in a similar way. Do your best to make it visually pleasing. It is permitted to represent the pieces as letters in your graphical display.

Players

Your system should accommodate both human and computer players. In particular, human vs. human, human vs. computer, and computer vs. computer should all be possible. When both players are human, the role of your program is to provide the game board, to detect checkmate and stalemate, and to prohibit illegal moves. Computer players should operate at one of several difficulty levels:

- Level 1: random legal moves.

- Level 2: prefers capturing moves and checks over other moves.

- Level 3: prefers avoiding capture, capturing moves, and checks.

- Levels 4+: something more sophisticated.

For levels 4 and above, try to come up with more sophisticated strategies for your player to use. Attempt level 4 (or any higher levels) only when the rest of the project is done.

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example `https://www.chess.com/explorer` which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)

Command Interpreter

You interact with the system by issuing text-based commands. The following commands are to be supported:

- `game` *white-player black-player* starts a new game. The parameters *white-player* and *black-player* can be either `human` or `computer[1-4]`.

- `resign` concedes the game to your opponent. This is the only way, outside of winning or drawing the game, to end a game.

- A move consists of the command `move`, followed by the starting and ending coordinates of the piece to be moved. For example: `move e2 e4`. Castling would specified by the two-square move for the king: `move e1 g1` or `move e1 c1` for white. Pawn promotion would additionally specify the piece type to which the pawn is promoted: `move e7 e8 Q`. In the case of a computer player, the command `move` (without arguments) makes the computer player make a move.

- `setup` enters setup mode, within which you can set up your own initial board configurations. This can only be done when a game is not currently running. Within setup mode, the following language is used:

  - `+ K e1` places the piece `K` (i.e., white king in this case) on the square `e1`. If a piece is already on that square, it is replaced. The board should be redisplayed.

  - `- e1` removes the piece from the square `e1` and then redisplays the board. If there is no piece at that square, take no action.

  - `= ` *colour* makes it *colour*'s turn to go next.

      – `done` leaves setup mode.

Upon completion of setup mode, you must verify that the board contains exactly one white king and exactly one black king; that no pawns are on the first or last row of the board; and that neither king is in check. The user cannot leave setup mode until these conditions are satisfied. We recommend doing setup mode early, as it may facilitate testing.

You are not required to handle the question of whether a player has the right to castle or make an *en passant* capture after board setup. You are free to simply assume that these are not valid options, or if you choose, you can find a way to incorporate them into the setup phase.

  It would be in your best interest (and will help during your demo) to make sure that your program does not break down if a command is misspelled.

The board should be redrawn, both in text and graphically, each time a move command is issued. For the graphic display, redraw as little of the screen as is necessary to make the needed changes (within reason).

Scoring

A win awards one point to the winner and zero points to the loser. A draw awards half a point to each team. When the program ends (Ctrl-D is pressed), it should print the final score to the screen. For example:

```
Final Score:
White: 2
Black: 1
```

Grading

Your project will be graded as following the project guidelines document. Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire game, please do your best to submit something that works, even if it doesn't solve the entire game requirements. For example:

- can't handle castling, en passant, or pawn promotion

- program only produces text output; no graphics

- only one level of difficulty implemented

- can't detect check or checkmate

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. We suggest: save the graphics for last, and first implement the game in pure text. One of the first things you should probably do is write a routine to draw the game board (probably a `Board` class with an overloaded friend `operator<<`). It can start out blank, and become more sophisticated as you add features. You should also write the command interpreter early, so that you can interact with your program. You can then add commands one-by-one, and separately work on supporting the full command syntax. Work on your pieces one at a time. Once you figure out the right interface and implementation for one piece type, the remaining piece types will be easier to build. Take the time to work on a test suite at the same time as you are writing your project. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.

You will be asked to submit a plan, with projected completion dates and divided responsibilities, as part of your documentation for Due Date 1.

If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features; see the proejct guidelines document for details.

Due Dates and Deliverables

See the project guidelines for due dates and submission requirements.