Demo Walkthrough

Note: All the user commands are specified to run in the **root** directory.

We have set the seed for the demo to be 2021. If pure randomness is needed, uncomment the code at line 34 in main.cc.

Game Mechanics Functionality

Command: ./chess < basicSetup.in
Demonstrates the following:
- Text-based display works
- Graphical display works (pieces are displayed as letters)
- Proper pieces (white is capital, black is lower case)

For the desired output, please see: **basicSetup.out**

Command: ./chess < kingMoves.in
Demonstrates the following:
- King can move up (from e1 e2)
- King can move down (from e8 e7)
- King can move left (from g7 f7)
- King can move right (from d5 c5)
- King can move diagonally (in all directions)

For the desired output, please see: **kingMoves.out**

Command: ./chess < king_self_check.in
Demonstrates the following:
- It is not legal to make any move that puts your king in check
    - King cannot move to a position that would put itself in check

For the desired output, please see: **king_self_check.out**

Command: ./chess < queenMoves.in
Demonstrates the following:
- Queen can move down (from d8 to d5)
- Queen can move down left diagonally (from d5 a2)
- Queen can move up right diagonally (from a2 d5)
- Queen can move down right diagonally (from d5 e4)
- Queen can move up left diagonally (from e4 c6)
- Queen can move up (from c6 c8)
- Queen cannot move past any piece that blocks its path (from c8 h8)

For the desired output, please see: **queenMoves.out**

Command: ./chess < bishopMoves.in
Demonstrates the following:

- Bishop can move up right (from c1 e3)
- Bishop can move down left (from e3 d2)
- Bishop can move up left (from d2 c3)
- Bishop can move down right (from c3 a1)
- Bishop cannot move past any piece that blocks its path (from a1 f5)

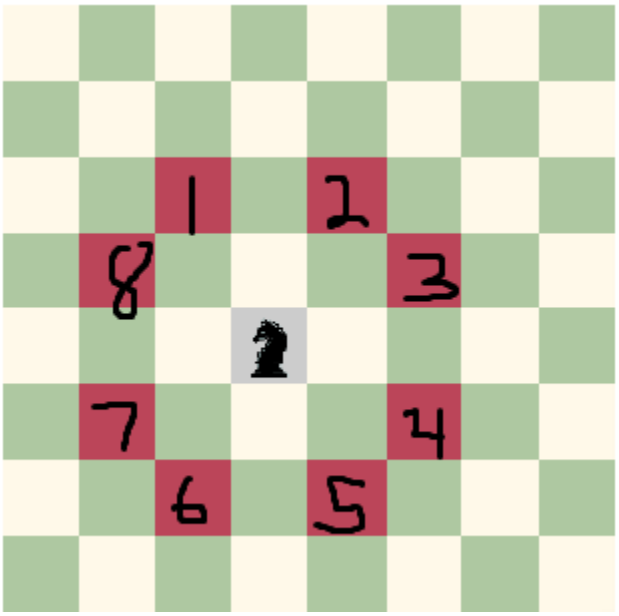
For the desired output, please see: **bishopMoves.out**

Command: ./chess < rookMoves.in
Demonstrates the following:

- Rook can move up (from a1 a6)
- Rook can move down (from c6 c2)
- Rook can move left (from d6 c6)
- Rook can move right (from a6 d6)
- Rook cannot move past any piece that blocks its path (from c2 h2)

For the desired output, please see: **rookMoves.out**

Command: ./chess < knightMoves.in
Given:



Demonstrates the following:

- Knight can move in a 1 form (from b1 a3)
- Knight can move in a 2 form (from b1 c3)
- Knight can move in a 3 form (from b1 d2)

- Knight can move in a 4 form (from b4 d3)
- Knight can move in a 5 form (from a3 b1)
- Knight can move in a 6 form (from c3 b1)
- Knight can move in a 7 form (from c3 a2)
- Knight can move in a 8 form (from d2 b1)

For the desired output, please see: **knightMoves.out**

Command: ./chess < pawnMoves.in
Demonstrates the following:

- Pawn can move 2 squares forward (from e2 e4)
- Pawn can move 1 square forward after (from e4 e5)

For the desired output, please see: **pawnMoves.out**

Movement of Pieces Functionality

Command: ./chess < cannot_eat_same.in
Demonstrates the following:

- A piece may only capture a piece of the opposite color (bishop can't eat pawn from c1 e3)

For the desired output, please see: **cannot_eat_same.out**

Command: ./chess < checkmate1.in
Demonstrates the following:

- An example where the black king is in checkmate by the white rook and queen and has no more moves available

For the desired output, please see: **checkmate1.out**

Command: ./chess < black_checkmate.in
Demonstrates the following:

- Opponent's king cannot escape in one move
- In this case, white is in checkmate, therefore, black wins
- Displays the proper wording: "Checkmate! [color] wins!"

For the desired output, please see: **black_checkmate.out**

Command: ./chess < l2_check_opponent.in
Demonstrates the following:

- An attack on the king, whether it can escape or not, is known as check
- The black king is in check
- It can escape
- Displays the proper wording: "[color] is in check."

For the desired output, please see: **l2_check_opponent.out**

Command: ./chess < capture_with_pawn.in
Demonstrates the following:

- On capturing, a pawn must move diagonally forward, one square, to take over a square occupied by another piece.
    - In this case, the pawn captures the rook from e4 e5
- Pawn cannot capture when moving forward
    - Invalid move happens when Pawn tries to capture Bishop from f5 f6

For the desired output, please see: **capture_with_pawn.out**

Command: ./chess < pawn_promotion.in
Demonstrates the following:

- A pawn, upon reaching the other end of the board is replaced by either a rook, knight, bishop, or queen (user choice)
    - We prompt the user what to promote, in which, we see the user responded with a Queen.
- We can also see that upon promotion, the detection for a check works where the black queen checks the white king.

For the desired output, please see: **pawn_promotion.out**

Command: ./chess < bot_pawn_promotion.in
Demonstrates the following:

- Implements the pawn promotion with a computer player. The computer will pick the piece with the highest value and decrement each time. In this example, the white pawn gets to the other side and promotes to a queen. The next time will be a Rook, then Bishop, then Knight.

For the desired output, please see: **bot_pawn_promotion.out**

Command: ./chess < castle.in
Demonstrates the following:

- Castling without being moved works when the King castles from e1 c1 on White, and e8 g8 on Black
- King was not in check in its starting, final, or in between position
- Rook and king have not been previously moved

For the desired output, please see: **castle.out**

Command: ./chess < invalidCastleWithKing.in
Demonstrates the following:

- White king was previously moved, therefore, the castle is invalid

For the desired output, please see: **invalidCastleWithKing.out**

Command: ./chess < invalidCastleWithRook.in
Demonstrates the following:

- White rook was previously moved, therefore, the castle is invalid

For the desired output, please see: **invalidCastleWithRook.out**

Players Functionality

Note: To move a bot, simply type "move" with no additional parameters.

Command: ./chess < c1vsc2.in
Demonstrates the following:

- Computer 1 playing against Computer 2
- Computer moves are prompted by just "move"
- Computer 2 demonstrates preferring to capture the highest pair when black knight moved from f6 to e4 to capture the white bishop.
- Computer 2 demonstrates preferring checks when black knight moved from d2 to f3 to put the white king in check
- Computer 1 is performing random legal moves, as desired
- Plays a simple game, about 10 moves each, demonstrating basic capturing/attacking

For the desired output, please see: **c1vsc2.out**

Command: ./chess < computer_king_self_check.in
Demonstrates the following:

- The Computer 2 avoids putting its own king into a self check

For the desired output, please see: **computer_king_self_check.out**

Command: ./chess < l1_king_bait.in
Demonstrates the following:

- The Computer 1 avoids putting its own king into a self check

For the desired output, please see: **l1_king_bait.out**

Command: ./chess < king_bait.in
Demonstrates the following:

- The Computer 2 knows in advance that eating the pawn will cause the King to be in check by the black bishop, and so since there's no capture or check possible, it performs a random legal move.

For the desired output, please see: **king_bait.out**

Setup Functionality

Command: ./chess < invalid_king_setup.in
Demonstrates the following:

- Shows that it's an invalid setup because there isn't exactly one white and black king

For the desired output, please see: **invalid_king_setup.out**

Command: ./chess < invalid_pawn_setup.in
Demonstrates the following:

- Shows that it's an invalid setup because there exist a pawn on the first and last row

For the desired output, please see: **invalid_pawn_setup.out**

Command: ./chess < invalid_pawn_king_setup.in
Demonstrates the following:

- Shows that it's an invalid setup because there isn't exactly one white and black king and there exists a pawn on the first row

For the desired output, please see: **invalid_pawn_king_setup.out**

Command: ./chess < incorrect_spelling.in
Demonstrates the following:

- Program does not break down if a command is misspelled (i.e., "move" being spelled incorrectly still functions, as desired)

For the desired output, please see: **incorrect_spelling.out**

Scoring Functionality

Command: ./chess < scoring.in
Demonstrates the following:

- Multiple games can be played using the resign command as an example
- Final score displayed is correct

For the desired output, please see: **scoring.out**

Edge Cases

Command: ./chess < weird_checkmate.in
Demonstrates the following:

- We had a case where this scenario would place the White king into "checkmate" even though this should simply be a "check".
- In this test, we simply just want to verify that it says "White is in check." instead of "Checkmate! Black wins!"

For the desired output, please see: **weird_checkmate.out**

Command: ./chess < undoCheck1.in
Demonstrates the following:

- We had an edge case where in this situation, the king would still move and put itself in check, which is not valid.
- In this test, we want to simply verify that the king will classify this as invalid and display it to the user.

For the desired output, please see: **undoCheck1.out**

Extra Features:

- We have implemented RAII and the use of smart pointers

Notes:

- We did not implement en passant
- We did not implement computer level 3