CIS-17C

Project 1 Write Up
- The Game of Chess

https://github.com/dabbingpenguin/17CProjectNetbeans

Ansh Srivastava

Introduction

I have coded a chess game for my project because I thought it would be an interesting challenge. Chess is a game that has many rules to it which makes it more complicated than other simpler games which would be good learning experience for me. I have been spent quite a few hours on it. It has 2 structures and 1 class. The whole project is around 1300 lines. The project is located on https://github.com/dabbingpenguin/17CProjectNetbeans .

Approach To Development

I started by coding as much as I could and then figuring things out as I made progress. At first I made it print the board and pieces and eventually I started to get user input and validate the moves the user made. Eventually, I decided to implement a check for checks and a multiset to store all the legal moves. When the set is empty, I would know that it is a checkmate or a stalemate. It was a great learning experience and I am glad I chose chess over other simpler games.

I used my personal Git repository in GitLab: https://gitlab.com/decisivedove/17cprojectchess and made commits everytime I felt like I made some significant progress. I also kept a backup of my recent working code locally in case something ever went wrong, but I would delete them when I felt like I had a better stable version. Once I was done with the project, I copied all of my code into Netbeans, created a GitHub account and pushed to it.

Oftentimes throughout the development process, I would come across bugs in my code. My approach to troubleshoot them was to print out as much information as possible to make debugging easier for example I would print out all the possible legal moves to troubleshoot things like checks not being detected or legal moves not being accepted.

My main game was in while a loop inside my main function. I would keep looping through until the game was over i.e., a checkmate or a stalemate. At first I had a lot of the logic in my main function (for example logic to check whose move it was) but as the game got more complex, I moved over a lot of the logic to my Board class for convenience while writing other functions like checking if a move was legal.

Once I was done with my code, I started changing the existing functionality in my code to use the containers and other requirements. I tried not to rewrite a lot of my code or add big features because that would lead to a higher probability of me breaking my code. The only feature I ended up adding was to check how many points each player based on the pieces they had captured. I refactored a lot of my code to have it fit in with the other requirements e.g., I changed from using for loops to iterators wherever I thought would be convenient, and I rewrote some functions using maps instead of switch cases.

I feel like I need more testing to make my code free of bugs. When I tried to test it, there were certain edge cases and things I overlooked while writing my code that causes issues only in certain circumstance. For example, I forgot that the distance the king moves is not always 1 mathematically. It is sqrt(2) when the king moves diagonally. I overlooked that while writing my code and then later caught it only while testing my game. I also fixed a bug with my knights not being able to check because I only included 4 possible paths for a knight to move instead of 8 in my functions that would

check for a checks. While the code should be mostly bug-free, I feel like if I did more testing, I could have identified even more bugs.

Game Rules

The rules of the game are the typical rules of chess:

The game starts off with the position given on the left. Each coordinate on the board is given by a letter and a number for example the square on the bottom left of the board is a1 and the square on the top right of the board is h8. These coordinates are important because that is how the user interacts with the game in my project.

The pieces in my game are shown by two letter combinations. The first letter is for the player the piece is for, and the second letter is the type of piece it is.

Piece	Name
BP	Black Pawn
WP	White Pawn
WN	White Knight
BN	Black Knight
WB	White Bishop
BB	Black Bishop
WR	White Rook
BR	Black Rook
WQ	White Queen
BQ	Black Queen
WK	White King
WQ	White Queen

The pieces and their names are given in the table to the left. Here are the rules for how each type of piece can move:

- Rook: A rook can move either horizontally or vertically. It cannot jump over other pieces.
- Bishop: A bishop can move diagonally. Just like a rook, it cannot jump over other pieces.
- Knight: A knight can move 2 squares horizontally or vertically and 1 square perpendicular to it. A knight can jump over other pieces.
- Queen: A queen can move vertically, horizontally, and diagonally. It cannot jump over pieces just like the bishop and rook.
- King: A king can move just like a queen but one square at a time i.e., either 1 square horizontally, vertically, or diagonally.
- Pawn: A pawn can only move one square forward at a time. It cannot capture vertically tough. If there is an opponent's piece in front of your pawn, the pawn cannot capture the piece or move forward. The only way a pawn can capture is one forward diagonally or using en passant which will be explained later in the documentation. A pawn can also move 2 squares in its very first move.

Here are some additional rules that are just as simple:

- You cannot capture your own pieces.
- A check is defined as when the king is threatened and seems like it can be captured by the opponent.
- You cannot make a move that puts you in check.
- When you are in check, you must make a move that gets you out of check either by blocking , capturing the piece that is causing the check, or moving your king.
- If there are no ways to get out of check, it is defined as a checkmate and the player who checkmates their opponent wins the game.

- If you are not under check but do not have any legal move to make, it is considered a stalemate and the game is a draw.

Here are some additional special moves you can make in the game of chess:

- Castling: In order to castle, your king must not be under check.
  - Kingside Castling: Your king moves two squares towards the rook closest to your king and your rook moves one square to the left of the king. In order to do it, the squares your king will go through can not be threatened and you can not have moved your king or the rook you are castling with. In order to castle kingside, type in **0-0** when my game asks you for a move.
  - Queenside Castling: It is identical to kingside castling but the difference is that the king moves two squares to the left and the rook comes to one square right of the king. The same rules of not having moved your king or rook, and the squares the king moves through must not be threatened apply. In order to castle queenside, type **0-0-0** when my game asks you for a move.
- En passant pawn capture: This is a special pawn move. If your opponent just moved a pawn for the first time and decided to move 2 squares and you have a pawn right next to it, you can capture that pawn and move your pawn one square forward from where your opponent's square was. This move has to be done right after your opponent makes the pawn move. En passant can only be done immediately after the pawn move was made by the opponent. In order to en passant, type in **ENP** when my game asks for your move.

Other rules not implemented includes the ability to resign, offer draw, and the fifty-move rule.

Description of Code

My structs and classes are in a namespace called chess. Here is a description of all the source files in my code:

- coordinate.hpp - This header file has the coordinate structure defined. The coordinate structure just has an int for rank and one for file. It also has an anonymous enum that defines all of the files in the game - A,B,C,...,H that way I can treat files as integers too.
- Move.hpp - This header file has the Move struct defined. The move struct uses aggregation with two chess::coordinate structs. Most moves in my game are just treated as an initial square and a final square. If some piece is in the final square, it will be captured. The only functions in this struct are operator overloading for the comparison operators so that it works with sets.
- Board.hpp - This is the header for my chess::Board class. It deals with the storing the pieces in the board, captured pieces, storing whose move it is, storing which moves are legal, printing out the board to the screen, and checking when the game ends by checkmate or stalemate.
- Board.cpp - The definitions for the functions declared in my Board.hpp file belong here. This is the largest file in the project with 1089 lines and contains most of the logic.
- main.cpp - This file contains the code that executes when my game runs. It creates a chess::Board object in the main function and takes user input in a while loop (while the game is not over) and passes it into the chess::Board object.

The logic of most of the source files are self-explanatory. I shall describe how the code works using a flow chart and UML in a later section.

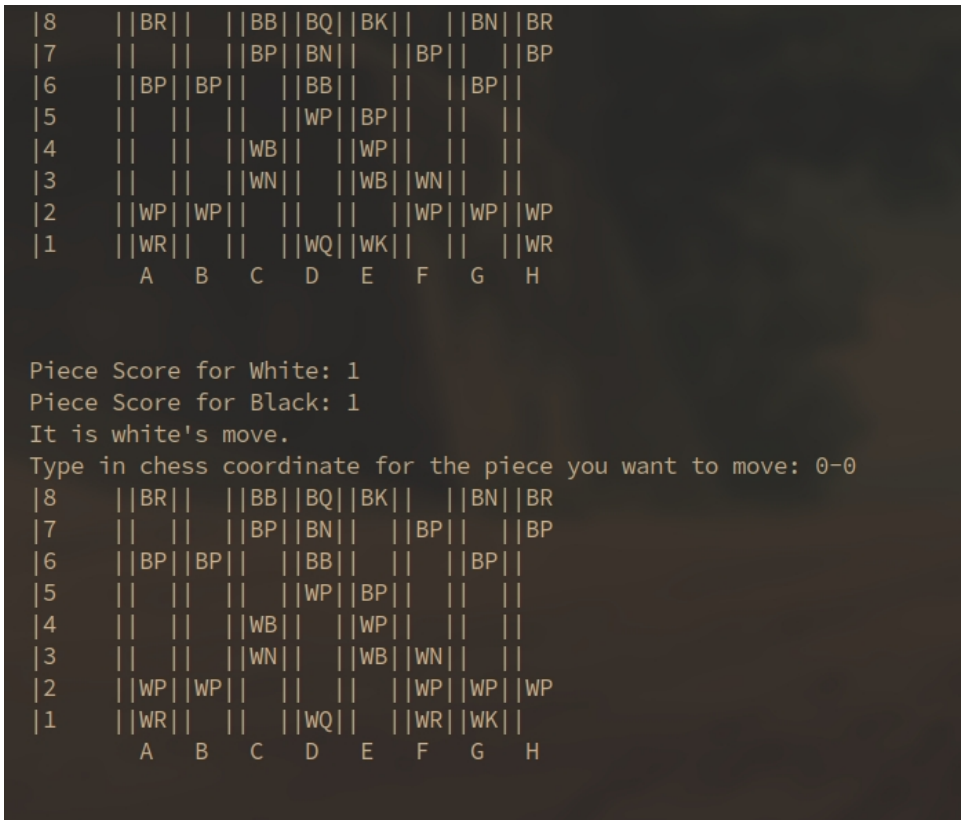
Sample Input/Output

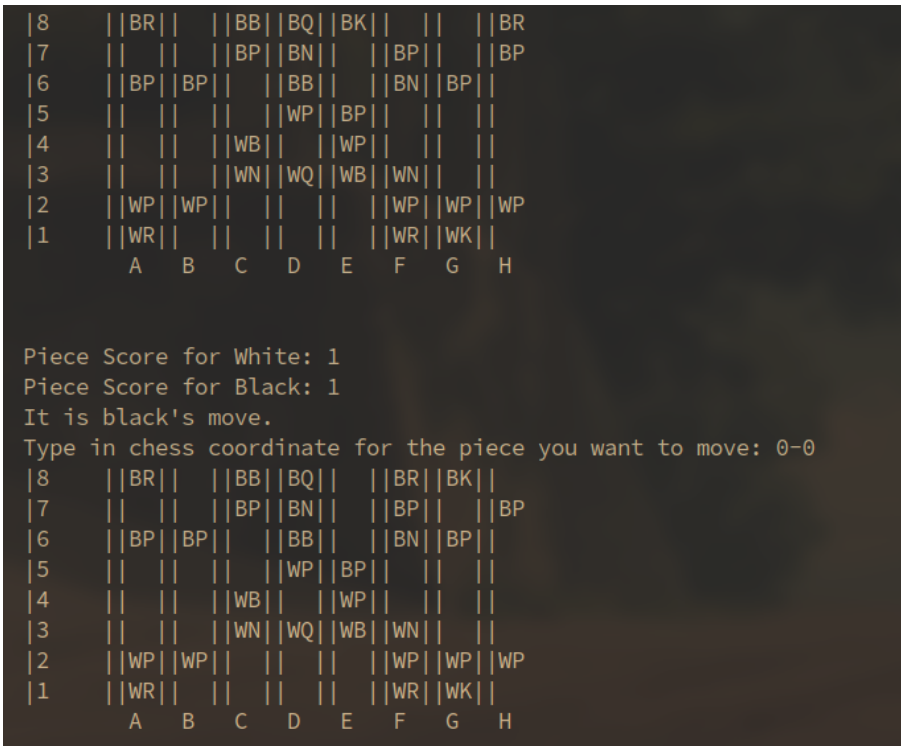
Here is a sample game of inputs and output:

```
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||  ||  ||
|4    ||  ||  ||  ||  ||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||WP||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
       A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: d2
Type in chess coordinate for the square you want to move your piece to: d4
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||  ||  ||
|4    ||  ||  ||  ||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
       A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is black's move.
Type in chess coordinate for the piece you want to move: d7
Type in chess coordinate for the square you want to move your piece to: d5
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||  ||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||BP||  ||  ||  ||
|4    ||  ||  ||  ||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
       A   B   C   D   E   F   G   H
```

```
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: c2
Type in chess coordinate for the square you want to move your piece to: c4
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||  ||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||BP||  ||  ||  ||
|4    ||  ||  ||WP||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||  ||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is black's move.
Type in chess coordinate for the piece you want to move: d5
Type in chess coordinate for the square you want to move your piece to: c4
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||  ||BP||BP||BP||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||  ||  ||
|4    ||  ||  ||BP||WP||  ||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||  ||  ||WP||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 1
It is white's move.
Type in chess coordinate for the piece you want to move: e2
Type in chess coordinate for the square you want to move your piece to: e4
```

Here is a demonstration of kingside castling:





Here is what happens when you try to play an illegal move:

```
|8    ||BR||  ||BB||BQ||  ||BR||BK||
|7    ||  ||  ||BP||BN||  ||BP||  ||BP
|6    ||BP||BP||  ||BB||  ||BN||BP||
|5    ||  ||  ||  ||WP||BP||  ||  ||
|4    ||  ||  ||WB||  ||WP||  ||  ||
|3    ||  ||  ||WN||WQ||WB||WN||  ||
|2    ||WP||WP||  ||  ||  ||WP||WP||WP
|1    ||WR||  ||  ||  ||  ||WR||WK||
        A   B   C   D   E   F   G   H


Piece Score for White: 1
Piece Score for Black: 1
It is white's move.
Type in chess coordinate for the piece you want to move: c4
Type in chess coordinate for the square you want to move your piece to: b6
You made an illegal move. Try making your move again.
It is white's move.
Type in chess coordinate for the piece you want to move:
```

This is what a checkmate looks like:

```
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||  ||  ||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||BP||BP||
|4    ||  ||  ||  ||WP||WP||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||  ||WP||WP||WP
|1    ||WR||WN||WB||WQ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0
It is white's move.
Type in chess coordinate for the piece you want to move: d1
Type in chess coordinate for the square you want to move your piece to: h5
|8    ||BR||BN||BB||BQ||BK||BB||BN||BR
|7    ||BP||BP||BP||BP||BP||  ||  ||BP
|6    ||  ||  ||  ||  ||  ||  ||  ||
|5    ||  ||  ||  ||  ||  ||BP||BP||WQ
|4    ||  ||  ||  ||WP||WP||  ||  ||
|3    ||  ||  ||  ||  ||  ||  ||  ||
|2    ||WP||WP||WP||  ||  ||WP||WP||WP
|1    ||WR||WN||WB||  ||WK||WB||WN||WR
        A   B   C   D   E   F   G   H


Piece Score for White: 0
Piece Score for Black: 0

Checkmate. White won!!!!!
◆ 17cprojectchess git:(master) ✗ >>>
```

Check Off Sheet

Sequences

A **std::list** is being used to store the captured pieces. The list is declared in the Board.hpp file:

```
std::list <int> capturedPieces;
```

Every time a piece is captured, it is added to the list in the Board.cpp file in the chess::Board::forceMove(Move mov) function:

```
std::swap(board[initialSquare.file][initialSquare.rank], board[finalSquare.file][finalSquare.rank]);
if (board[initialSquare.file][initialSquare.rank] != EMPTY)
  capturedPieces.push_back(board[initialSquare.file][initialSquare.rank]);
board[initialSquare.file][initialSquare.rank] = EMPTY;
```

Associative Containers

std::multiset

A **std::multiset<Move>** is declared in my Board.hpp file:

```
std::multiset<Move> legalMoves;
```

Whenever a move is legal, it is added to the legalMoves multiset in the chess::Board::fillset() function:

```
if (validateMove(mov))
{
  int a  = board[mov.firstSquare.file][mov.firstSquare.rank];
  int b = board[mov.lastSquare.file][mov.lastSquare.rank];
  if (currentMove.front() != pieceToPlayer(a))
    continue;
  board[mov.lastSquare.file][mov.lastSquare.rank] = a;
  board[mov.firstSquare.file][mov.firstSquare.rank] = chess::EMPTY;
  if ((currentMove.front() == Player::WHITE && !whiteIsUnderCheck()) || (currentMove.front() == Player::BLACK && !blackIsUnderCheck()))
    legalMoves.insert(mov);
  board[mov.firstSquare.file][mov.firstSquare.rank] = a;
  board[mov.lastSquare.file][mov.lastSquare.rank] = b;
}
```

The set is then checked to see if the move the user made is legal in the chess::Board::makeAMove(Move mov) function:

```
bool chess::Board::makeAMove(Move mov)
{
  fillSet();
  for (std::multiset<Move>::iterator it = legalMoves.begin();
       it != legalMoves.end();
       ++it)
  {
    Move x = *it;
    if (x.firstSquare.rank == mov.firstSquare.rank &&
        x.firstSquare.file == mov.firstSquare.file &&
        x.lastSquare.rank == mov.lastSquare.rank   &&
        x.lastSquare.file == mov.lastSquare.file     )
    {
    forceMove(mov);
    moves.push(mov);
    currentMove.push(currentMove.front());
    currentMove.pop();
    fillSet();
    // for (Move xy : legalMoves){ std::cout << xy.firstSquare.file << xy.firstSquare.rank << "->" << xy.lastSquare.file << xy.lastSquare.rank << "\t"; }
    // std::cout << '\n';
    return true;
    }
  }
  fillSet();
  return false;
}
```

std::map

Here is how I decided to use maps in my code:

```
std::pair<int, int> chess::Board::getPieceScore()
{
  std::map<int, int> pieceScores = {
      {WP, 1},
      {WR, 5},
      {WN, 3},
      {WB, 3},
      {WQ, 9},
      {BP, 1},
      {BR, 5},
      {BN, 3},
      {BB, 3},
      {BQ, 9}
    };
  int blackScore = 0;
  int whiteScore = 0;
  for (std::list<int>::iterator it = capturedPieces.begin();
      it != capturedPieces.end(); ++it)
  {
    if (pieceToPlayer(*it) == Player::WHITE)
      blackScore += pieceScores[*it];
    if (pieceToPlayer(*it) == Player::BLACK)
      whiteScore += pieceScores[*it];
  }
  std::pair <int, int> ret;
  ret.first = whiteScore;
  ret.second = blackScore;
  return ret;
}
```

```
int chess::getFileFromChar(char c)
{
  std::map<char, int> charToFile =
  {
    {'A', chess::A},
    {'B', chess::B},
    {'C', chess::C},
    {'D', chess::D},
    {'E', chess::E},
    {'F', chess::F},
    {'G', chess::G},
    {'H', chess::H},
    {'a', chess::A},
    {'b', chess::B},
    {'c', chess::C},
    {'d', chess::D},
    {'e', chess::E},
    {'f', chess::F},
    {'g', chess::G},
    {'h', chess::H}
  };
  return charToFile[c];
}

int chess::getRankFromChar(char c)
{
  std::map<char, int> charToRank =
  {
    {'1', 1},
    {'2', 2},
    {'3', 3},
    {'4', 4},
    {'5', 5},
    {'6', 6},
    {'7', 7},
    {'8', 8}
  };
  return charToRank[c];
}
```

```
chess::Player chess::pieceToPlayer(int piece)
{
  std::map<int, chess::Player> pToPlayer =
  {
    {chess::EMPTY, chess::Player::NOPLAYER},
    {chess::WP, chess::Player::WHITE},
    {chess::WB, chess::Player::WHITE},
    {chess::WR, chess::Player::WHITE},
    {chess::WN, chess::Player::WHITE},
    {chess::WK, chess::Player::WHITE},
    {chess::WQ, chess::Player::WHITE},
    {chess::BP, chess::Player::BLACK},
    {chess::BR, chess::Player::BLACK},
    {chess::BN, chess::Player::BLACK},
    {chess::BB, chess::Player::BLACK},
    {chess::BK, chess::Player::BLACK},
    {chess::BQ, chess::Player::BLACK}
  };
  return pToPlayer[piece];
}
```

Container Adapters:

std::stack

Board.hpp:

```
std::stack<Move> moves;
```

Board.cpp:

```
void chess::Board::forceMove(Move mov)
{
  // Check the rules
  // if (!validateMove(mov))
  //  return false;
  // Push to the stack
  moves.push(mov);
```

A std::stack was used to store the moves every player made. This is useful because it allows me to check if the previous move was a pawn move which is useful for en passant.

```
bool chess::Board::enpassant()
{
  if (!canEnpassant)
    return false;
  Move lastMove = moves.top();
```

and the lastMove variable is later used to perform all sorts of checks.

std::queue

Board.hpp:

```
std::queue<Player> currentMove;
```

and that is used in the constructor in Board.cpp:

```
currentMove.push(Player::BLACK);
currentMove.push(Player::WHITE);
```

whenever I need to check whose move it is, it uses *currentMove.front()*. In my makeAMove(Move mov) function, after every move, I change the move order like this:

```
currentMove.push(currentMove.front());
currentMove.pop();
```

Additional Containers:

std::pair

Pairs are used to store whether the players are allowed to castle in the chess::Board class:

```
std::pair <bool, bool> canCastleKingside;
std::pair <bool, bool> canCastleQueenside;
```

I found this more convenient than to have a different variables for each player.

2D std::vector

```
std::vector<std::vector<int>> board;
```

The 2D vector is declared in my chess::Board class. It is used to hold the piece in every square.

Iterators

Here are all the instances I used iterators in my Board.cpp file:

```
void chess::Board::printBoard()
{
    for (std::vector<std::vector<int>>::iterator it = board.end() - 1; it!= board.begin(); --it)
   {
      for (std::vector<int>::iterator jt = it->begin(); jt != it->end(); ++ jt)
      {
        int i = std::distance(board.begin(), it);
        int j = std::distance(it->begin(), jt);
        if (j==0)
        {
          if (i==8)
            std::cout << std::setw(2) << "|8" << std::setfill(' ');
          else
            std::cout << std::setw(2) << i << std::setfill(' ');
        }
        std::cout << std::setw(4) << getPieceName(board[j][i]) << std::setfill('|');
      }
      std::cout << '\n';
   }
  std::cout <<  "     " << std::setfill(' ')
```

```
bool chess::Board::blackIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
      it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //      jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), BK);
    if (*jt  == BK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForBlack(c);
}
```

```
bool chess::Board::whiteIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
      it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), WK);
    if (*jt  == WK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForWhite(c);
}
```

```
for (std::list<int>::iterator it = capturedPieces.begin();
    it != capturedPieces.end(); ++it)
{
  if (pieceToPlayer(*it) == Player::WHITE)
    blackScore += pieceScores[*it];
  if (pieceToPlayer(*it) == Player::BLACK)
    whiteScore += pieceScores[*it];
}
```

std::list, std::multiset, and std::map (eventough I did not need to use an iterator on map in my project) use bidirectional iterators. std::vector uses a random-access iterator. The difference between these iterators is that a bidirectional iterator can only increment or decrement by one element at a time but a random-access iterator can access any element in just a single step.
I did use std::find however in my code. It is supposed to return an input iterator. An input iterator is guaranteed to be able to increment but is not guaranteed to be able to decrement and may be read-only.

One thing to note is that a random access iterator is also a valid bidirectional iterator which is a valid forward iterator which is both a valid input and output iterator. All of these are valid trivial iterators because they may be de referenced. Here is a description of the types of iterators:

- Trivial Iterator- Is guaranteed to be able to be dereferenced.
- Input iterators - Is guaranteed to be able to incremented and read from.
- Output iterators - Is guaranteed to be able to incremented and written to.
- Forward iterator - Can be incremented, read, and written from.
- Bidirectional Iterators - Can be incremented, decremented, written, and read from.
- Random-Access Iterators - Can skip any number of elements to jump to another memory location in just one step.

std::list, std::multiset, std::map	**Bidirectional**, Forward, Output, Input, Trivial
std::vector	**Random-Access**, Bidirectional, Forward, Output, Input, Trivial

Algorithms

Non-mutating

std::distance

```
void chess::Board::printBoard()
{
    for (std::vector<std::vector<int>>::iterator it = board.end() - 1; it!= board.begin(); --it)
   {
      for (std::vector<int>::iterator jt = it->begin(); jt != it->end(); ++ jt)
      {
        int i = std::distance(board.begin(), it);
        int j = std::distance(it->begin(), jt);
        if (j==0)
```

^ std::distance was used because I needed the index of the element in my nested for loops.

std::find

```
bool chess::Board::blackIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
     it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), BK);
    if (*jt  == BK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForBlack(c);
}
```

```
bool chess::Board::whiteIsUnderCheck()
{
  coordinate c;
  for (std::vector<std::vector<int>>::iterator it = board.begin() + 1;
     it != board.end(); ++it)
  {
    // for (std::vector<int>::iterator jt = it->begin() + 1;
    //     jt != it->end(); ++jt)
    std::vector<int>::iterator jt = std::find(it->begin(), it->end(), WK);
    if (*jt  == WK && jt != it->end())
      {
        c.file = it - board.begin();
        c.rank = jt - it->begin();
      }
  }
  return isSquareThreatenedForWhite(c);
}
```

std::find is used to find the kings for purposes of checking for checks.

Mutating algorithms

std::swap

```
std::swap(board[initialSquare.file][initialSquare.rank], board[finalSquare.file][finalSquare.rank]);
if (board[initialSquare.file][initialSquare.rank] != EMPTY)
  capturedPieces.push_back(board[initialSquare.file][initialSquare.rank]);
board[initialSquare.file][initialSquare.rank] = EMPTY;
```

std::swap is used in my forceMove(Move) function to swap the values of two squared in the board.

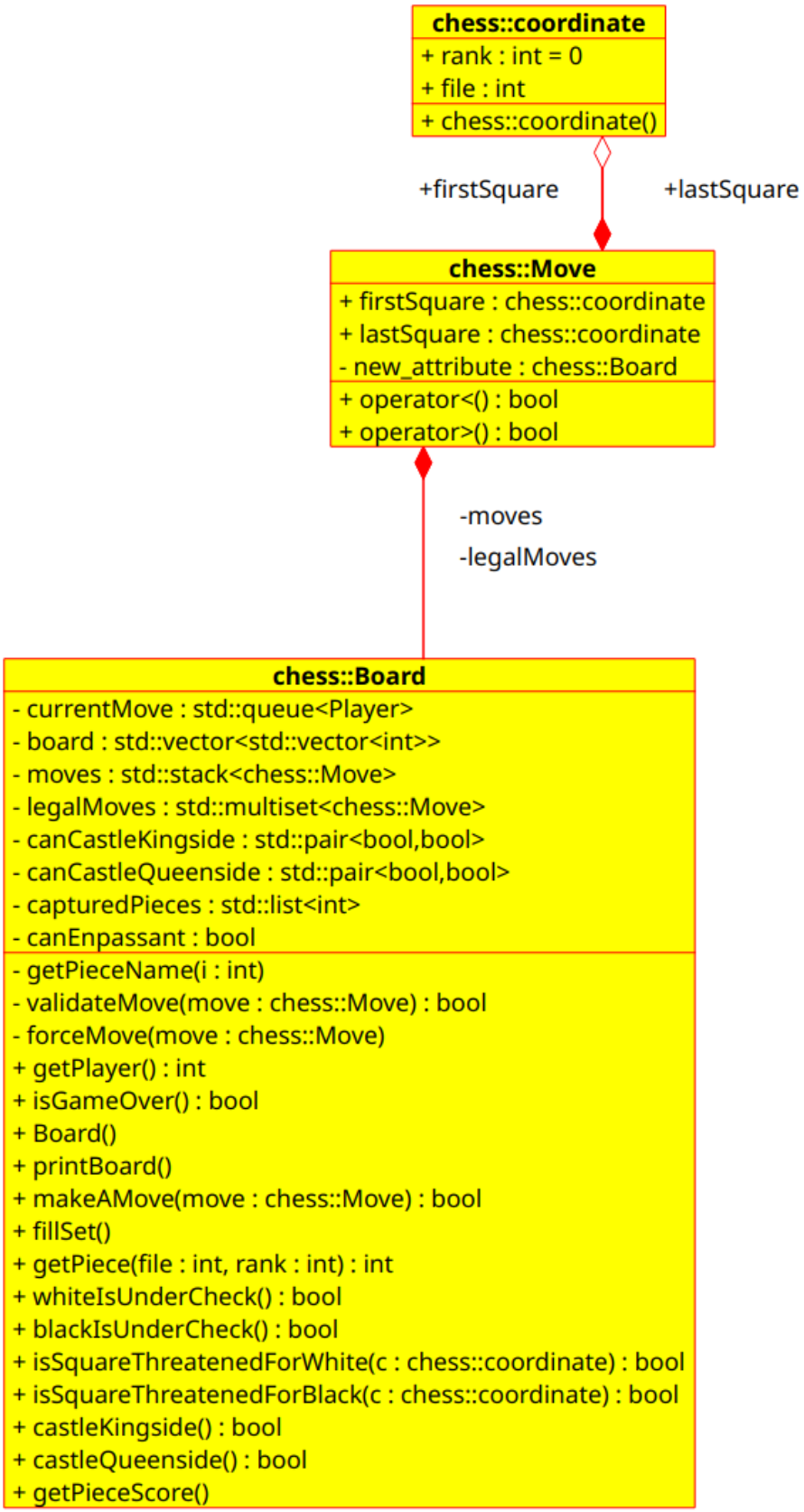
Organization
std::sort

```
currentMove.push(Player::BLACK);
currentMove.push(Player::WHITE);
auto sortQueue = [](std::queue<Player> q)
{
    std::vector<Player> pVector;
    while (!q.empty())
    {
      pVector.push_back(q.front());
      q.pop();
    }
    std::sort(pVector.begin(), pVector.end(), [] (Player a, Player b)
        {
          if (a == Player::WHITE && b == Player::BLACK)
            return true;
          return false;
        });
    std::queue<Player> ret (std::deque<Player> (pVector.begin(), pVector.end()));
    return ret;
};
currentMove = sortQueue(currentMove);
```

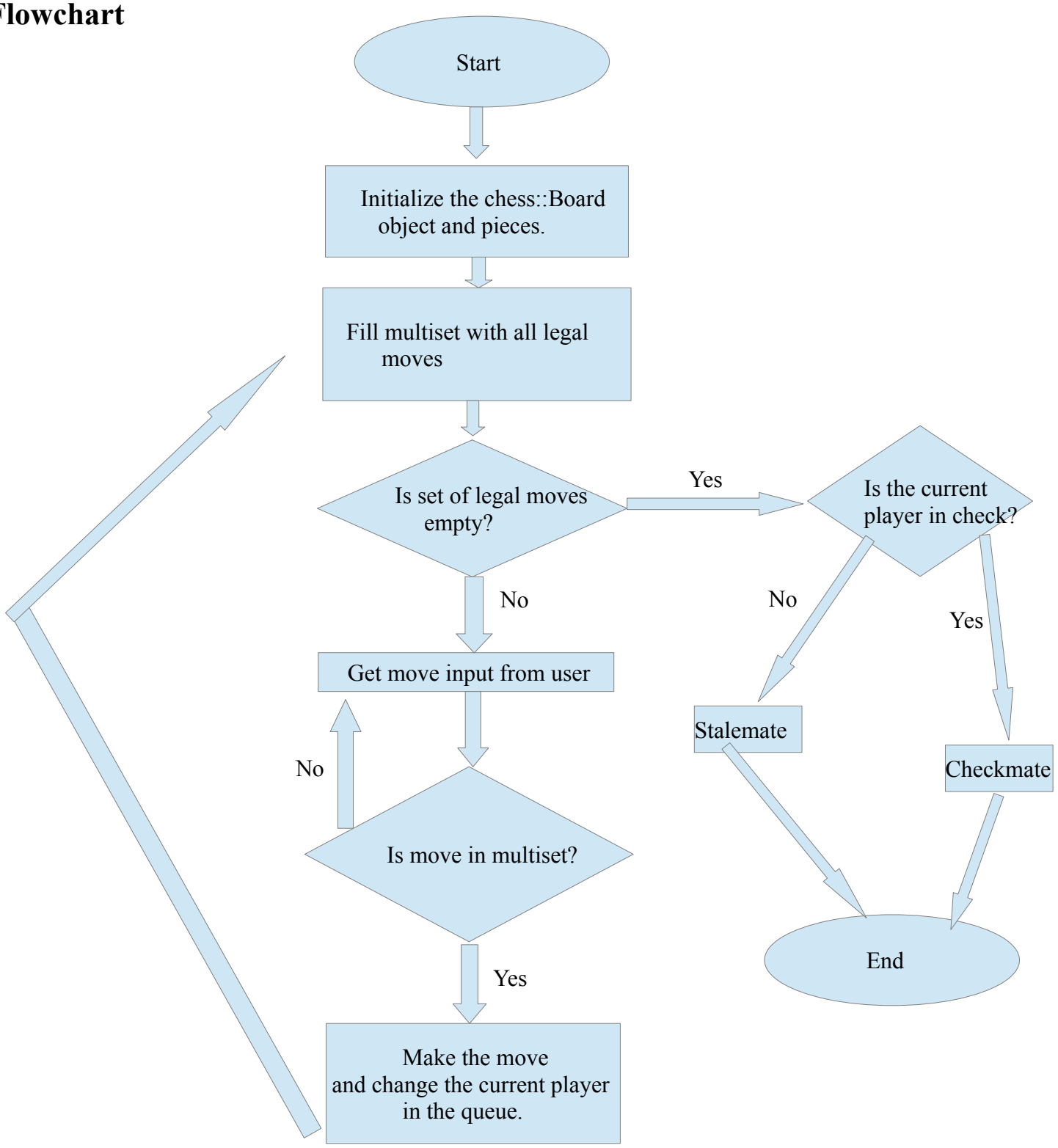
In the constructor for my chess::Board class, I push the items in the wrong order for the currentMove queue. I then write a custom lambda function to sort it after converting it to a vector and then changing it back to a queue.

Documentation

UML



Flowchart


Start
Initialize the chess::Board object and pieces.
Fill multiset with all legal moves
Is set of legal moves empty?
Yes
Is the current player in check?
No
Yes
No
Get move input from user
Stalemate
Checkmate
No
Is move in multiset?
Yes
End
Make the move and change the current player in the queue.

Pseudocode

I am going to give psuedocode of how I fill up the set with legal moves since that part is not clear from the flow chart.

LOOP through i = 1 to 8 for all integer values of i
LOOP through j = 1 to 8 for all integer values of j
LOOP through k = 1 to 8 for all integer values of k
LOOP through l = 1 to 8 for all integer values of l
    Check if the piece in the square [i][j] can move to [k][l] according to the rules of movement of the piece
    if no, continue to the next iteration of the loop
    if yes, perform the move

    check if current player is under check
    if yes, add the move for [i][j]->[k][l] to the multiset of legal moves

    Reverse the move

The above is how I add legal moves to the set. I perform the move if it follows the rules of movement and then I check if the player would be in check after the move and then I reverse the move. If not under check, I add the move to the multiset. Finally I reverse the move.

It is still not clear how I check for checks or check if something follows the rules of movement. Here is an example for checking if a move follows the rules of movement for a bishop:

Given square [i][j] -> [k][l] we want to know if it is a valid move for a bishop
CREATE a variable dx = k-i
CREATE a variable dy = l-j
Is absolute value of dx = the absolute value of dy
If not then it is not a legal bishop move. RETURN FALSE
If yes, we still need to check if there is a piece in between [i][j]→[k][l]
CREATE a variable length=absolute value of dx or dy (because both will be the same value)
NORMALIZE dx, and dy by setting dx = dx/length and dy = dy/length
LOOP from n=1 to n=length-1
    Check if the square [i + n*dx][j+n*dy] is EMPTY
    If NOT EMPTY, RETURN FALSE
RETURN TRUE if it made it this far

And I use a similar process to check if the king is under check. I use the same method of using a normalized change in the x and y positions and loop through them to check if there is a bishop or a queen in each diagonal. I use a similar process for all other pieces.