Abstract:
We wished to implement a chess game which would run on a Windows machine, provide us with a great deal of practice with data structures and algorithms implemented in C++, and provide a pleasant distraction for CSE 331 students during finals week. The main problem with creating a chess game was the problems which were associated with the amount and complexity of rules in the normal chess game. Such rules usually center around where a piece can move, however, there are rules adding to the complexity of this problem such as checking endgame conditions like stalemate and checkmate. The other problem that faced us in implementing a chess game was developing an AI which was capable of making somewhat intelligent decisions. Developing a human-human version of the game has a great deal of complexity in making sure rules aren’t violated, but the implementation of an AI presents many more difficult questions and challenges in this project. Naturally, the way we overcame these challenges was to develop many different data structures and algorithms to deal with these situations. Not all data structures developed were complex, but the number and variety implemented required thorough knowledge of which structures to use and how to implement them. The separation of work was relatively strict in this project as Sean created a working and complete human-human console version of the game, and Tom developed the AI’s decision-making capabilities. We both worked on parts of the game interface in chess.cpp.

Keywords:
Stalemate – an endgame situation when it is a player’s turn and that player cannot move
Checkmate – an endgame situation when it is a player’s turn and that player cannot perform a move which uncheck’s their king
Check – a state in which a king can be taken by an opponent on the opponent’s next move. Check is a prerequisite for checkmate.
MoveTwo – a pawn special move only available to pawns which have not moved yet
Promotion – a pawn special state which occurs when a pawn reaches the end of the board and has no more remaining moves. This special state results in the promotion of the pawn to a queen piece type. (In standard chess, the player can decide which piece to promote a pawn to. This is eliminated in our implementation as 90% of the time, a player will promote the pawn to a queen, the most powerful piece in chess.)
Castling – a special move between the king and a rook in which both pieces move at the same time, swapping the relative locations of the king and rook and putting the king in a presumably defensive position.
MakeDecision – AI algorithm that takes care of determining which move to make next based upon the score given to all possible moves.
MakeScore – determines a score for the board given different constant values for each piece and different multipliers depending upon the board situation (i.e. piece can be taken or will be able to take another piece)
We wished to implement a chess game which would run on a Windows machine, provide us with a great deal of practice with data structures and algorithms implemented in C++, and provide a pleasant distraction for CSE 331 students during finals week. The main problem with creating a chess game was the problems which were associated with the amount and complexity of rules in the normal chess game. Such rules usual center around where a piece can move, however, there are rules adding to the complexity of this problem such as checking endgame conditions like stalemate and checkmate. The other problem that faced us in implementing a chess game was developing an AI which was capable of making somewhat intelligent decisions. Developing a human-human version of the game has a great deal of complexity in making sure rules aren’t violated, but the implementation of an AI presents many more difficult questions and challenges in this project. Naturally, the way we overcame these challenges was to develop many different data structures and algorithms to deal with these situations. Not all data structures developed were complex, but the number and variety implemented required thorough knowledge of which structures to use and how to implement them. The separation of work was relatively strict in this project as Sean created a working and complete human-human console version of the game, and Tom developed the AI’s decision-making capabilities.

We began this project by implementing a console, human-human version of the chess game which would provide us with an interactive medium to observe any AI and make changes as needed. It may sound easy to develop a simple game. However, many classes and algorithms had to be implemented to merely have a program which displayed valid output. It was decided that there needed to be ten classes written to help implement the game. These classes included Board, Piece, None, Pawn, Rook, Knight, Bishop, Queen, King, and Player, each with its own important properties. The Board was designed to hold pointers of Piece classes in a matrix of vector-vectors. By holding pointers in the Board and not the actual objects, information held in the board itself could be reduced and modifying the board could be hastened. Additionally, there is no real penalty for using vectors as opposed to other data structures as the size of the board remains the same and does not grow. A list could have been used, but it does not have as convenient of accessor functions such as the bracket operator. None and other chess piece classes all inherited from the Piece class and therefore had similar properties. It was decided early in the project that the Piece classes were to hold the important information of a move list which would contain specific destination coordinates on the board, limited by the simplest moving rule: a piece cannot move outside of the board.

Since Piece classes held general moving information, a decision had to be made as to how other more complicated moving rules could be enforced on the pieces. Many of the other moving rules in chess were observed to be dependant on the moves of other pieces and where other pieces were on the board. Therefore, two design possibilities came about:

1. Have Piece classes manipulate themselves, laying ontop of the Board class, and making decisions individually on what moves are allowed.

2. Have the Board class manipulate the move lists of the Piece classes dictating for a higher-level view which moves are allowed.
It became the preferred solution to have the board manipulate individual piece move lists as it had the advantage of having easily accessible information of which pieces existed in which slots. The simple procedure for building a correct move list for a piece with no special moves was to update the piece’s move list by the Piece::updateMove() function which built a move list assuming no other elements existed on the board, then using the Board::updateMove() function to scan the entire board and eliminate moves which are invalid because of check situations and piece’s being in the way. The Piece::updateMove() function manages certain special moves using private data member flags like CanMoveTwo in Pawn and CanCastle in King. The result should be a move list which doesn’t allow the player to select almost every invalid move.

This solution eliminates most invalid moves, but due to other algorithms, leaves some illegal moves on the list. In order to ensure a function called Board::hasSupport() works, the move list does not eliminate moves which would allow a capture of the same player’s piece. This is the case because Board::hasSupport() looks into the move lists of all same color pieces on the board and determines if a move in a same color piece’s list could “take” the target piece. If this is true, then that target piece has support. As a result, the Board::movePiece() function must prevent the user from capturing another piece owned by the same user. With this being the case, we’ve eliminated almost all illegal moves. The only remaining move situations not evaluated involve the pawn and king. As you may know, a pawn cannot take an opponent piece directly infront, but cannot move diagonally unless there is an enemy piece there. Special conditions take care of this case. The king also has to go through special conditions, as it cannot move into a space that is in check by an opponent piece. This case is eliminated by using the Board::inCheck() function on a None piece adjacent to the king and eliminating it if it returns true.

With all the move lists updated properly, the Board::movePiece() function is the next major algorithm implemented. The general goal of this function is to move a target piece to a destination without doing something illegal. The function basically checks to see if a destination is in the move list of the target piece and ensures that it is a valid move, otherwise moving the piece back to its original space. As stated earlier, it is in this function that moves where a piece is capturing a friendly piece are eliminated. It is also in this function where logic concerning castling is taken care of. Castling, as stated in the definitions, is a special move between a rook and a king. This special move occurs when neither the king nor the chosen rook has moved yet and a path is clear. Determining whether the pieces have moved yet is taken care of in this function by accessing a special private data member flag in the King or Rook classes. If the pieces have moved, their flags are set to false. The castling move is somewhat more difficult because two pieces move at once, and if the move does not work out then two pieces must be reset. The simplest way to deal with this is to add a case before the main move (a king to its destination) where the rook is moved. If the move is not legal due to postcondition status, then both pieces are reset. The biggest concern with moving a piece is ensuring that the postcondition status is not illegal, or in other words, the player’s king is not in check after the move. This is achieved by checking the king for check after the move and return the piece if Board::KinginCheck() is true. The final operation for the Board::movePiece() function is to update the entire board using Board::updateBoard()
which calls each piece’s update function and the the board’s update function on each piece.

At this point, the chess game has a board and pieces which can move around on the board without doing something illegally. There are two endgame situations which exist in chess: checkmate and stalemate. As stated in the definitions, checkmate is when the king is put in check and there is no move available which can remove the king from check. Stalemate is when the player cannot perform a move at all on his turn. Stalemate is the easier of these two situations as Board::isStalemate() merely checks the player’s pieces and returns true if any piece on the board can move (also eliminating same color captures temporarily). Checkmate requires many things to be true in order to return true. The function Board::isCheckmate() slowly eliminates situations which indicate a non-checkmate situation. The function final condition scans all the pieces on the board and finds the opponent piece which is putting the king in check. It then checks to see if the player can capture the checking piece and end the checkmate situation. This series of conditions eliminates almost all the conditions necessary to create a checkmate situation. The final condition is actually left up to the Board::movePiece() function which will prevent the user from moving into a situation which leaves them in check. This is a shortcut solution to the algorithm since Board::isCheckmate() does not return a definitive true. The situation where this final condition exists is when a player can move a piece into the path of the checking opponent piece. The condition could be checked inside Board::isCheckmate(), but would require the function to additionally move every piece for the player temporarily and seeing if it results in a false from Board::KinginCheck(). As you can see, this adds a significant amount of computations for every time the board is checked for a checkmate situation. Please feel free to ask Sean for a complete explanation if this is not clear.

The final class written for the human-human game was the Player class which primarily ran which functions in the Board class were called at a certain time. The class holds a pointer to the board, preventing an entire board from being copied and allowing two players to manipulate the same board, a name for the player, and the color they will be using. It is inside this class that some of the user interface dialogues were run. The most interesting function inside the Player class is Player::movePiece() which calls the Board::movePiece() function and returns a bool which indicates whether a piece was moved or not. Other functions within the Player class allow the player to touch a piece, allowing a player to query or move a piece, or query a piece displaying vital information and the moves available for the piece at a given turn. The entire program ran from chess.cpp and basically contained a large while loop which ran the user interface and which color’s turn it was. This file mostly manipulated the Player class but also contained the actual Board object for manipulation and checking endgame conditions. With all these pieces (and many more as seen in the file) in place, the human-human version of the game was operational.

The AI created for the game was implemented as the Opponent class which inherits from the Player class and redefines the movePiece() function. The functions makeDecision() and makeScore() were the two main functions that were added for AI support. stringifyBoard() was also added as a way to facilitate the creation of temporary boards for the AI to test moves with. The basic concept of the AI is that it virtually makes every possible move for the turn before making a move. The virtual move is then
scored by makeScore() which returns a double value for the worth of the board. The worth is determined by the computer’s subscore – the user’s subscore. The subscores are determined by which pieces are still on the board and which pieces are in danger of being captured/could potentially capture another piece. The algorithm in place now only checks the situation for the next turn, but our implementation would allow for looking ahead several turns. The difficulty in that would be determining how to effectively predict user moves.

Each possible move is stored as a MOVE struct which contains the coordinates of the piece, the destination of the piece and the score of the move. Each struct is then inserted into a decision tree (class DTree). Each node of a DTree (class DTnode) contains a MOVE, a pointer to its parent, and a vector of pointers to children. In the case of our chess game, a child would be a turn that immediately follows a parent. Because every child depends on its parent happening, deleting a node renders all its children irrelevant so they too can be pruned from the tree. In the future it may be worth wile to change the vector to a doubly-linked linked, but for an initial attempt we decided that a vector would be easier to manipulate. DTree has a Bestmove() function that is called by makeDecision() which right now is a max finder but could be changed to for instance determine the average score of all children of a node and find the node with the highest child average. We did not take advantage of the DTree too much because we only considered the next immediate move, but we went about creating our AI in such a way that it would be easily extendable.

Future Work:
- Smarter Artificial Intelligence
- Graphical User Interface
- Cross-platform
- Networked

References:

Appendix:

```cpp
bool Board::isStalemate(int color){ //Returns true if the board is in stalemate
    bool result=true;
    for(int r=0; (result && (r < 8)); r++){
        for(int c=0; (result && (c < 8)); c++){
            if((my_board[r][c]->getColor() == color) && (my_board[r][c]->getListSize() > 0))
                result = false; //if any piece on the board of a color can move then that color is not possibly in stalemate
        }
    }
    return result;
}

bool Board::isCheckmate(int color){ //Returns true if board is checkmate; check in higher level which piece_color is inCheck(piece_color);
    bool result;
```
Piece *temp;
temp = &getKing(color);
int listSize = temp->getListSize(), kr = temp->getRow(), kc = temp->getCol(); //gets king's information

for (int i = 0; i < temp->getListSize(); i++) { //reduces list size (formerly includes captures of same color
    if (my_board[temp->getMoveR(i)][temp->getMoveC(i)]->getColor() == color)
        listSize--;
}

if (!KingInCheck(color))
    result = false;
else if (listSize > 0)
    result = false;
else {
    for (int r = 0; r < 8; r++) //scans entire board
        for (int c = 0; c < 8; c++){
            if (my_board[r][c]->getColor() != NONE && my_board[r][c]->getColor() != color){ //if the piece is an enemy of king of color
                for (int j = 0; j < my_board[r][c]->getListSize(); j++) { //scan move list
                    if ((my_board[r][c]->getMoveR(j) == kr) && (my_board[r][c]->getMoveC(j) == kc))
                        if (!inCheck(*my_board[r][c])){ //if no piece can stop the enemy piece (player must forfeit if no other piece can block path legally)
                            result = true;
                            break;
                        }
                }
            }
        }
    }

return result;
}

//Input piece and destination coordinates, cross-checks with piece moveList if move is valid, then updates //piece coordinates, moves it within board private vectors. Performs all necessary erasing of pieces //Returns true if move is valid and complete. Otherwise false.
bool Board::movePiece(Piece& p, int destrow, int destcol){
    bool flag = false, result = false, Castling = false;
    Piece *temp, *Castemp;
    int trow = p.getRow(), tcol = p.getCol(), castRow, castCol;
    char promo;

    if (valid_row(destrow) && valid_column(destcol)){ //Makes sure that move is in board
        for (int i = 0; (!flag && (i < my_board[trow][tcol]->getListSize())); i++){
            if ((p.getMoveR(i) == destrow) && (p.getMoveC(i) == destcol))
                flag = true;
        }
    }

    //---________________________________________---Castling logic
    if (p.getType() == KING && flag){
        if (my_board[destrow][destcol]->getType() == NON) {
            if (p.getCastle()){
                if (p.getColor() == WHITE){ //sees what type of piece it is
                    if (my_board[0][0]->getType() == ROOK) && (destrow == 0) &&
                        (destcol == 1)) { //sees what moves piece could possibly castle in
                        Castling = true; //temporarily castles rook
                        my_board[0][2] = my_board[0][0];
                        my_board[0][0] = new None(0, 0);
                        Castemp = my_board[0][2];
                        castRow = 0;
                    }
                }
            }
        }
    }
}
castCol = 0;
}
else
flag = false;
}
else if((my_board[0][7]->getType() == ROOK) && (destrow == 0) && (destcol == 6)){//same as above
if(my_board[0][7]->getCastle()){
  Castling = true;
  my_board[0][5] = my_board[0][7];
  my_board[0][5]->setPos(0, 5);
  my_board[7][5]->setCastle(false);
  my_board[0][7] = new None(0, 7);
  Castemp = my_board[0][5];
  castRow = 0;
  castCol = 7;
} else
flag = false;
}
else{
if((my_board[7][0]->getType() == ROOK) && (destrow == 7) && (destcol == 1)){
if(my_board[7][0]->getCastle() && (my_board[7][0]->getType() != NON)) {
  Castling = true;
  my_board[7][2] = my_board[7][0];
  my_board[7][2]->setPos(7, 2);
  my_board[7][5]->setCastle(false);
  my_board[7][0] = new None(7, 0);
  Castemp = my_board[7][2];
  castRow = 7;
  castCol = 0;
} else
flag = false;
}
else if((my_board[7][7]->getType() == ROOK) && (destrow == 7) && (destcol == 6)){
if(my_board[7][7]->getCastle()){
  Castling = true;
  my_board[7][5] = my_board[7][7];
  my_board[7][5]->setPos(7, 5);
  my_board[7][5]->setCastle(false);
  my_board[7][7] = new None(7, 7);
  Castemp = my_board[7][5];
  castRow = 7;
  castCol = 7;
} else
flag = false;
}
}
} // END CASTLING LOGIC

if(flag){ //if we can move so far
  if(my_board[destrow][destcol]->getColor() != p.getColor()){//makes sure you
don't capture your own piece
  temp = my_board[destrow][destcol];//moves piece and stores any captured
  piece in temp (performs king's move in castle)
  my_board[destrow][destcol] = &p;
  p.setPos(destrow, destcol);
  my_board[trow][tcol] = new None(trow, tcol);
  if(p.getType() == KING)
    if(p.getCastle())
      p.setCastle(false);
updateBoard(); // updates the board to evaluate check situation

if(KinginCheck(p.getColor())){ //if you put yourself in check, reset board
    my_board[trow][tcol] = &p;
    my_board[destrow][destcol] = temp;
    p.setPos(trow, tcol);
    result = false;

    if(Castling){ //RESET ROOK IN CASTLE IF KING IS IN CHECK IN CASTLE
        my_board[castRow][castCol] = Castemp;
        my_board[Castemp->getRow()][Castemp->getCol()] = new None(Castemp->getRow(), Castemp->getCol());
        Castemp->setPos(castRow, castCol);
        Castemp->setCastle(true);
        p.setCastle(true);
    }
    updateBoard();
}
else{// you made a valid move
    result = true;
    if(p.getType() == PAWN){//toggles pawn's move two ability
        if(p.getMoveTwo())
            p.toggleMoveTwo();
        else if(destrow == 7){//promotes pawn to queen if able
            cout << "Pawn promoted to queen!!!\n";
            my_board[destrow][destcol] = new Queen(p.getColor(), destrow, destcol);
            updateBoard();
        }
    }
    else if(p.getType() == ROOK) //sets rook castle ability to false
        if(p.getCastle())
            p.setCastle(false);
}
else{
    cout << "You tried taking one of your own pieces.\n";
    result = false;
}
else
    cout << "Invalid destination coordinates possibly (castling when not allowed).\n";
else
    cout << "Invalid destination coordinates.\n";
return result;
}

void Board::updateMove(int r, int c){ //Examines piece move_list in space and calls removeMove on invalid moves outside
    int endr, endc, tempr, tempc;
    bool flag=false;
    if(my_board[r][c]->getType() == KING){ //SPECIAL CONDITIONS FOR KING TO ELIMINATE CASTLE POSSIBILITIES
        if(my_board[r][c]->getCastle()){
            updateCastle(r, c);
        }
    }
    for(int i=0; i < my_board[r][c]->getListSize(); i++)// checks every move in list
        temp = my_board[r][c]->getMoveR(i);
        tempc = my_board[r][c]->getMoveC(i);
if (my_board[tempr][tempc]->getColor() == my_board[r][c]->getColor()) { // if there is a piece of same color in path, prune it
    endr = temprr;
    endc = tempcc;
    flag = true;
}
else if (my_board[tempr][tempc]->getColor() != NONE) { // case for there being an enemy piece in path
    if (my_board[r][c]->getType() == PAWN && (tempc == c)) { // pawn cannot take a piece straight ahead
        my_board[r][c]->removeMove(i);
        i = -1;
    }
    if (my_board[r][c]->getType() == KING && hasSupport(*my_board[tempr][tempc])) { // King cannot capture an enemy piece if it puts the king in check
        my_board[r][c]->removeMove(i);
        i = -1;
    }
    endr = temprr;
    endc = tempcc;
    flag = true;
}
else if (my_board[r][c]->getType() == PAWN && (tempc != c)) { // pawn can only move diagonally when capturing
    Piece* temp;
    if (my_board[r][c]->getColor() == BLACK)
        temp = &getKing(WHITE);
    else
        temp = &getKing(BLACK);
    for (int k = 0; k < temp->getListSize(); k++)
        if (temp->getMoveR(k) == tempr && temp->getMoveC(k) == tempc)
            temp->removeMove(k);
    i = -1;
    flag = false;
}
else if (my_board[r][c]->getType() == KING) { // if a spot if in check then the king cannot move there
    if (my_board[r][c]->getColor() == WHITE)
        my_board[tempr][tempc]->switchColor(WHITE);
    else
        my_board[tempr][tempc]->switchColor(BLACK);
    if (inCheck(*my_board[tempr][tempc])){
        my_board[r][c]->removeMove(i);
        i = -1;
        flag = false;
    }
    my_board[tempr][tempc]->switchColor(NONE);
} else{
    flag = false;
}
if (flag) { // ALL SPECIAL CASE TO ELIMINATE MOVES THAT OCCUR AFTER endr and endc
    if (my_board[r][c]->getType() == ROOK) {
        for (int j = 0; j < my_board[r][c]->getListSize(); j++){
            temprr = my_board[r][c]->getMoveR(j);
            if (temprr > endr) {
                my_board[r][c]->removeMove(j);
                j--;
            }
        }
    }
}
i= -1;
}
else if((endr < r) && (tempr < endr)){
    my_board[r][c]->removeMove(j);
    j--;  
    i= -1;
}
else if((endc < c) && (tempc < endc)){
    my_board[r][c]->removeMove(j);
    j--;  
    i= -1;
}
else if((endc > c) && (tempc > endc)){
    my_board[r][c]->removeMove(j);
    j--;  
    i= -1;
}
else if(my_board[r][c]->getType() == BISHOP){
    for(int j=0; j < my_board[r][c]->getListSize(); j++){
        tempc = my_board[r][c]->getMoveC(j);
        if((endr > r) && (tempr > endr) && (endc > c) && (tempc > endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr < r) && (tempr < endr) && (endc > c) && (tempc > endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr > r) && (tempr > endr) && (endc < c) && (tempc < endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr < r) && (tempr < endr) && (endc < c) && (tempc < endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
    }
}
else if(my_board[r][c]->getType() == QUEEN){
    for(int j=0; j < my_board[r][c]->getListSize(); j++){
        tempr = my_board[r][c]->getMoveR(j);
        if((endr > r) && (tempr > endr) && (endc > c) && (tempc > endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr < r) && (tempr < endr) && (endc > c) && (tempc > endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr > r) && (tempr > endr) && (endc < c) && (tempc < endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
        else if((endr < r) && (tempr < endr) && (endc < c) && (tempc < endc)){
            my_board[r][c]->removeMove(j);
            j--;  
            i= -1;
        }
    }
}
else if((endr > r) && (tempr > endr) && (tempc==c) && (endc==c)){

my_board[r][c]->removeMove(j);
    j--;
i = -1;
} else if((endr < r) && (tempr < endr) && (tempc==c) && (endc==c)) {
    my_board[r][c]->removeMove(j);
    j--;
i = -1;
} else if((endc < c) && (tempc < endc) && (tempr==r) && (endr==r)) {
    my_board[r][c]->removeMove(j);
    j--;
i = -1;
} else if((endc > c) && (tempc > endc) && (tempr==r) && (endr==r)) {
    my_board[r][c]->removeMove(j);
    j--;
i = -1;
}
    //     queryPiece(r, c);
}
}
else if(my_board[r][c]->getType() == PAWN) {
    for(int j=0; j < my_board[r][c]->getListSize(); j++) {
        tempr = my_board[r][c]->getMoveR(j);
tempc = my_board[r][c]->getMoveC(j);
        if((endr > r) && (tempr > endr)) {
            my_board[r][c]->removeMove(j);
            j--;
i = -1;
        } else if ((endr < r) && (tempr < endr)) {
            my_board[r][c]->removeMove(j);
            j--;
i = -1;
        }
    }
}
}