**Final Project**

<Chess>  
June 8, 2016

**Jiwon Yoo**

CIS 17C – Data Structure  
Professor. Mark Lehr

<https://www.github.com/jiwonyoo1102/16SPR_CIS_17C_42102>

Table of Contents

[Chess 4](#_Toc453175000)

[Introduction 4](#_Toc453175001)

[Summary 5](#_Toc453175002)

[Class Diagram 6](#_Toc453175003)

[Pawns 7](#_Toc453175004)

[Movement 7](#_Toc453175005)

[Capturing 7](#_Toc453175006)

[Code 7](#_Toc453175007)

[Rooks 9](#_Toc453175008)

[Movement 9](#_Toc453175009)

[Capturing 9](#_Toc453175010)

[Code 9](#_Toc453175011)

[Knights 11](#_Toc453175012)

[Movement 11](#_Toc453175013)

[Capturing 11](#_Toc453175014)

[Code 11](#_Toc453175015)

[Bishops 12](#_Toc453175016)

[Movement 12](#_Toc453175017)

[Capturing 12](#_Toc453175018)

[Code 12](#_Toc453175019)

[Queens 13](#_Toc453175020)

[Movement: 13](#_Toc453175021)

[Capturing: 13](#_Toc453175022)

[Code 13](#_Toc453175023)

[Kings 15](#_Toc453175024)

[Movement: 15](#_Toc453175025)

[Capturing: 15](#_Toc453175026)

[Code 15](#_Toc453175027)

[References 17](#_Toc453175028)

[PreciseClock.h 17](#_Toc453175029)

[Colormod.h 19](#_Toc453175030)

[Full code 20](#_Toc453175031)

[Game 20](#_Toc453175032)

[Chess.h 20](#_Toc453175033)

[Chess.cpp 21](#_Toc453175034)

[PieceMove.h 30](#_Toc453175035)

[PieceMove.cpp 30](#_Toc453175036)

[Game Player 39](#_Toc453175037)

[Computer.h 39](#_Toc453175038)

[Player.h 39](#_Toc453175039)

[User.h 40](#_Toc453175040)

[Stack 41](#_Toc453175041)

[Node.h 41](#_Toc453175042)

[Stack.h 41](#_Toc453175043)

[Stack.cpp 42](#_Toc453175044)

*Code highlighting: https://tohtml.com*

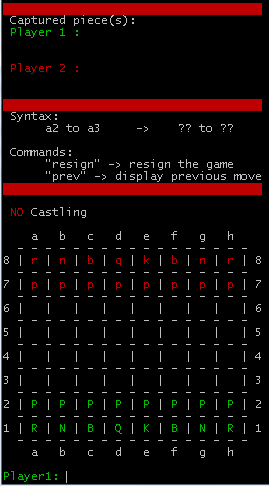
# Chess

Chess is a two-player strategy board game played on a chessboard, a checkered game board with 64 squares arranged in an eight-by-eight grid. Each player begins the game with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of the six piece types moves differently. The most powerful piece is the queen and the least powerful piece is the pawn.

The objective is to capture the opponent's king. To this end, a player's pieces are used to attack and capture the opponent's pieces, while supporting their own. The game can be won by voluntary resignation by the opponent, which typically occurs when too much material is lost, or if checkmate appears unavoidable.

*(Source: https://en.wikipedia.org/wiki/Chess)*

# Introduction

This chess game is a terminal based Player vs. Player game. Basic graphical user interface have been implemented using Linux console colors (“\033[##m“).

Player2 is colored ***red***. Player1, however, is different based on the running environment. If the environment of the machine is Windows based, Player1 will be ***green***. In Linux/Unix machine, Player1 will be ***blue*.**

The game rule is same as regular chess game and it runs exactly same except that the program does not include castling, checkmate, and stalemate. When game is done, the name of the winner will be saved in a file with the time which represents the total time spent making a move.

The first block (*“Captured piece(s)”)* shows all the pieces captured by each player in alphabetical order.



The second block shows the syntax of moving chess pieces. [*source*] to [*destination*]

The last or third block shows the current chess board with Player1’s pieces on the bottom and Player2’s pieces on top.

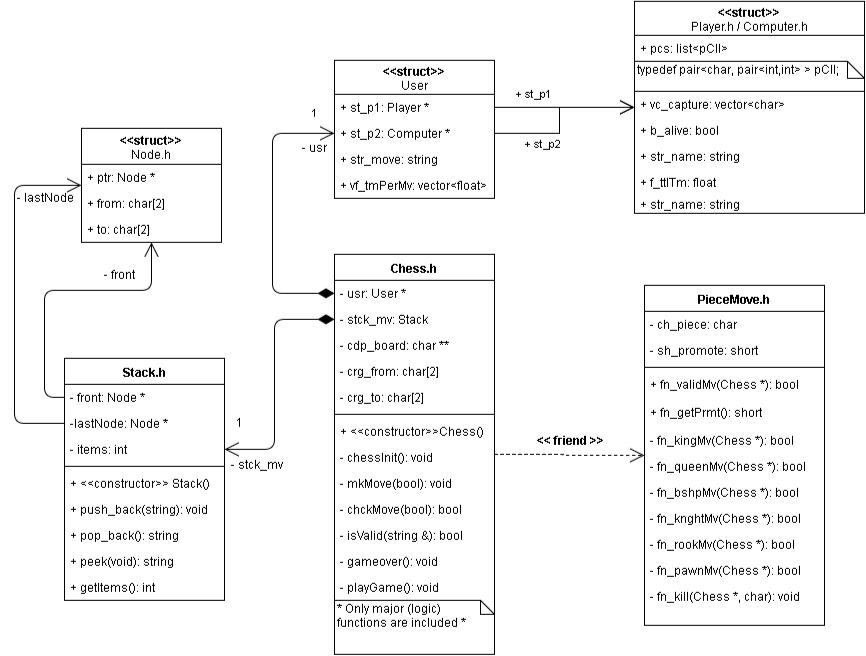
# Summary

The program is 1100+ lines long;

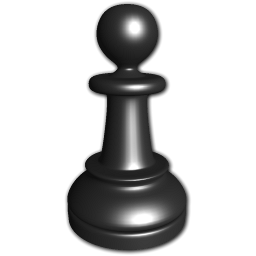
|  |  |
| --- | --- |
| What is not included: | |
| Hashing | |
| Trees | |
|  | |
| Major variables used: | |
| User \*usr | Player and computer’s structure |
| Char \*\*cdp\_board | Chess board |
| Char crg\_from[2] | Initial location of a piece |
| Char crg\_to[2] | Destination of a moving piece |
| Char ch\_piece | Determine which piece is moving |
| List<pair<char,pari<int,int> > > pcs | Store pieces’ position |
| Map< string, float > mymap; | Store winner’s name and spent time |
| Map< string, float>::iterator it; |  |
| Bool b\_alive | Determines when to stop the game |
|  | |
| Libraries used: | |
| <iostream> |  |
| <fstream> | File I/O |
| <ostream> | Colomod.h |
| <iomanip> |  |
| <cstdlib> |  |
| <string> / <string.h> |  |
| <list> | Player.h + Computer.h |
| <map> | Chess::scrBoard( .. ) |
| <algorithm> | Sort() |
| <vector> |  |
| <cmath> |  |
| <sys/time.h> | PreciseClock.h |
| <Windows.h> | PreciseClock.h |
| <ctime> | PreciseClock.h |
| “Stack.h” |  |

*\* This program runs best at unix/linux operating machine’s terminal (compile: g++ \*.cpp or make). If using Windows, unix bash is recommended such as git bash rather than the cmd prompt. This program can be run in NetBeans; other IDEs are not guaranteed to be run normal.*

# Class Diagram



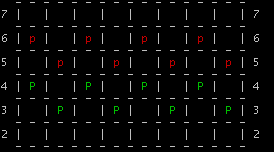
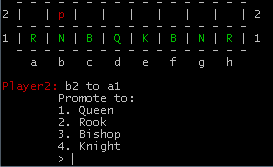
# Pawns

Movement:

* A pawn can move to the square directly in front of itself, if that square is clear;
* A pawn on its starting position has the option of moving two squares;

Capturing:

* A pawn captures diagonally; one square distance;  
    
  Promotion:
* A pawn that advances all the way to the opposite side of the board is promoted to   
   another piece of that player’s choice.

Code  
**bool** PieceMove::fn\_pawnMv(Chess \*cp) {

**switch**( ch\_piece ) {

**case** 'p': // player 2 || computer

// starter-> may move 2 places

**if**(\_FROM[0]==4 && \_FROM[1]==\_TO[1] &&

(\_FROM[0]+2==\_TO[0] || \_FROM[0]+4==\_TO[0]) &&

(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ') )

**return** **true**;

// return false if pawn moved more than 1 place

**else** **if**(\_FROM[0]+2!=\_TO[0]) {

cout << "\tPAWN: ILLEGAL MOVE\n";

**return** **false**;

}

// cannot move diagonal: same team

**else** **if**(\_LOWS && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'P': // player 1 || you

// starter-> may move 2 places

**if**(\_FROM[0]==14 && \_FROM[1]==\_TO[1] &&

(\_FROM[0]-2==\_TO[0] || \_FROM[0]-4==\_TO[0]) &&

(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ') )

**return** **true**;

// return false if pawn moved more than 1 place

**else** **if**(\_FROM[0]-2!=\_TO[0]) {

cout << "\tPAWN: ILLEGAL MOVE\n";

**return** **false**;

}

// cannot move diagonal: same team

**else** **if**(\_CAPS && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: WRONG TARGET\n";

**return** **false**;

}

}

// cannot move front: there is something in front

**if**(\_FROM[1]==\_TO[1] && cp->cdp\_board[\_TO[0]][\_TO[1]]!=' ') {

cout << "\tPAWN: CANNOT CAPTURE IN THE SAME DIRECTION ?\n";

**return** **false**;

}

// cannot move diagonal: there is no piece to capture

**else** **if**(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ' && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: TRYING TO CAPTURE SOMETHING?\n";

**return** **false**;

}

// Promotion -> pawn reached the otherside

**if**(\_TO[0]==2 || \_TO[0]==16) {

cout << "\tPromote to:\n";

cout << "\t1. Queen\n";

cout << "\t2. Rook\n";

cout << "\t3. Bishop\n";

cout << "\t4. Knight\n\t> ";

cin >> sh\_promote;

cin.ignore();

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

# Rooks

Movement:

* A rook can move to any square vertically or horizontally, if that square is clear and if no pieces are found in between source to destination;
* Cannot jump over any pieces;

Capturing:

* A rook captures directly; horizontally or vertically;

### Code

**bool** PieceMove::fn\_rookMv(Chess \*cp) {

// check if rook is moving diagonally

**if**(\_FROM[0]!=\_TO[0] && \_FROM[1]!=\_TO[1]) {

cout << "\tROOK: ILLEGAL MOVE\n";

**return** **false**;

}

**switch**( ch\_piece ) {

**case** 'r': // player 2

// check if player is trying to capture their own piece

**if**(\_LOWS) {

cout << "\tROOK: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'R': // player 1

// check if player is trying to capture their own piece

**if**(\_CAPS) {

cout << "\tROOK: WRONG TARGET\n";

**return** **false**;

}

}

**bool** b\_swap, b\_swap2;

b\_swap = b\_swap2 = **false**;

// Vertical checking

**if**(\_FROM[1]==\_TO[1]) {

// 2 to 8 (going downward), 8 to 2(going upward)

// swap values when (going up)

**if**(\_FROM[0] > \_TO[0]) {

swap(\_FROM[0],\_TO[0]);

b\_swap = **true**;

}

**for**(**int** i=\_FROM[0]+2; i<=\_TO[0]-2; i+=2)

**if**(cp->cdp\_board[i][\_FROM[1]]!=' ') {

cout << "\tROOK: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Horizontal checking

**else** {

// 2 to 8 (going right), 8 to 2(going left)

// swap values when (going left)

**if**(\_FROM[1] > \_TO[1]) {

swap(\_FROM[1],\_TO[1]);

b\_swap2 = **true**;

}

**for**(**int** i=\_FROM[1]+2; i<=\_TO[1]-2; i+=2)

**if**(cp->cdp\_board[\_FROM[0]][i]!=' ') {

cout << "\tROOK: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// if swapped, swap it back to original state

**if**(b\_swap)

swap(\_FROM[0],\_TO[0]);

**else** **if**(b\_swap2)

swap(\_FROM[1],\_TO[1]);

// Captured somthing?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

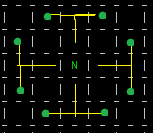
# Knights

Movement:

* A knight can move two squares horizontally and one square vertically or two squares  
   vertically and one square horizontally; Looks like an ‘L’ shape;
* Can jump over pieces;

Capturing:

* A knight captures an enemy piece by replacing it on its square, meaning if any piece is  
   located at the place where knight is moving, a knight can capture that piece;

L-shape;   
Yellow: path; Green: terminal point;

### Code

**bool** PieceMove::fn\_knghtMv(Chess \*cp) {

**short** sh\_x = abs(\_TO[0] - \_FROM[0]);

**short** sh\_y = abs(\_TO[1] - \_FROM[1]);

// regardless of the direction,

// night's abs() distance from A to B is (4,2) or (2,4)

**if**( !((sh\_x==4 && sh\_y==2)||(sh\_x==2 && sh\_y==4)) ) {

cout << "\tKNIGHT: ILLEGAL MOVE\n";

**return** **false**;

}

**switch**(ch\_piece) {

**case** 'n':

// check if player is tring to capture their own piece

**if**(\_LOWS) {

cout << "\tKNIGHT: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'N':

// check if player is tring to capture their own piece

**if**(\_CAPS) {

cout << "\tKNIGHT: WRONG TARGET\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

# Bishops

Movement:

* A bishop can move any squares diagonally;
* Cannot jump over any pieces;

Capturing:

* A bishop captures an enemy piece by replacing it on its square.

### Code

**bool** PieceMove::fn\_bshpMv(Chess \*cp) {

**short** sh\_x = \_TO[0] - \_FROM[0];

**short** sh\_y = \_TO[1] - \_FROM[1];

**if**( abs(sh\_x) != abs(sh\_y) ) {

cout << "\tBISHOP: INVALID MOVE\n";

**return** **false**;

}

**int** temp = abs(sh\_x)/sh\_x;

**int** temp2 = abs(sh\_y)/sh\_y;

**switch**(ch\_piece) {

**case** 'b':

// check if player is trying to capture their own piece

**if**(\_LOWS) {

cout << "\tBISHOP: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'B':

// check if player is tring to capture their own piece

**if**(\_CAPS) {

cout << "\tBISHOP: WRONG TARGET\n";

**return** **false**;

}

}

// check if player's piece is jumping over the pieces

**for**(**int** i=2; i<=abs(sh\_x)-2; i+=2) {

**if**( cp->cdp\_board[\_FROM[0]+(i\*temp)][\_FROM[1]+(i\*temp2)]!=' ') {

cout << "\tBISHOP: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

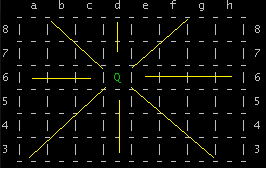
# Queens

### Movement:

* A queen is the most powerful piece in a chess game that it can move any   
   number of square in any direction: horizontal, vertical, or diagonal;

### Capturing:

* A queen captures an enemy piece by replacing it on its square.

can move any direction;

### Code

**bool** PieceMove::fn\_queenMv(Chess \*cp) {

**short** sh\_x = \_TO[0] - \_FROM[0];

**short** sh\_y = \_TO[1] - \_FROM[1];

**bool** b\_swap, b\_swap2;

b\_swap = b\_swap2 = **false**;

// make x always greater than y, x > y

**if**( abs(sh\_x) < abs(sh\_y) ) {

swap( sh\_x, sh\_y );

b\_swap = **true**;

}

// (vertical and horizontal move) || (diagonal moves)

**if**( !((sh\_y==0 && sh\_x%2==0) || (abs(sh\_x)==abs(sh\_y))) ) {

cout << "\tQUEEN: INVALID MOVE\n";

**return** **false**;

}

// if swapped, change it back

**if**( b\_swap ) {

swap( sh\_x, sh\_y );

b\_swap = **false**;

}

// Invalid move: capturing the same team

**switch**( ch\_piece ) {

**case** 'q':

**if**(\_LOWS) {

cout << "\tQUEEN: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'Q':

**if**(\_CAPS) {

cout << "\tQUEEN: WRONG TARGET\n";

**return** **false**;

}

}

// Check vertial moving

**if**(\_FROM[1]==\_TO[1]) {

// swap values when (going up)

**if**(\_FROM[0] > \_TO[0]) {

swap(\_FROM[0],\_TO[0]);

b\_swap = **true**;

}

// check if player's piece is jumping over any pieces

**for**(**int** i=\_FROM[0]+2; i<=\_TO[0]-2; i+=2)

**if**(cp->cdp\_board[i][\_FROM[1]]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Check horizontal moving

**else** **if**(\_FROM[0]==\_TO[0]){

// swap values when (going left)

**if**(\_FROM[1] > \_TO[1]) {

swap(\_FROM[1],\_TO[1]);

b\_swap2 = **true**;

}

// check if player's piece is jumping over any pieces

**for**(**int** i=\_FROM[1]+2; i<=\_TO[1]-2; i+=2)

**if**(cp->cdp\_board[\_FROM[0]][i]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// check diagonals

**else** {

// negative or positive

**int** temp = abs(sh\_x)/sh\_x;

**int** temp2 = abs(sh\_y)/sh\_y;

**for**(**int** i=2; i<=abs(sh\_x)-2; i+=2) {

**if**(cp->cdp\_board[\_FROM[0]+(i\*temp)]  
 [\_FROM[1]+(i\*temp2)]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

}

// if swapped, swap it back to original state

**if**(b\_swap)

swap(\_FROM[0],\_TO[0]);

**else** **if**(b\_swap2)

swap(\_FROM[1],\_TO[1]);

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

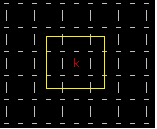
# Kings

## Movement:

* A king can move one square in any direction: horizontal, vertical, or diagonal.

## Capturing:

* Just like any other pieces, a king captures an enemy piece by replacing it on its  
   square.

one square, any direction;

### Code

**bool** PieceMove::fn\_kingMv(Chess \*cp) {

**short** sh\_x = abs(\_TO[0] - \_FROM[0]);

**short** sh\_y = abs(\_TO[1] - \_FROM[1]);

// absolute valued (x,y) pairs of distance from A to B.

// There are 4 pairs (2,2) (2,0) (2,4) (0,2)

**if**( !((sh\_x==2&&sh\_y==2) || (sh\_x==2&&sh\_y==0) ||

(sh\_x==2&&sh\_y==4) || (sh\_x==0&&sh\_y==2)) ) {

cout << "\tKING: INVALID MOVE\n";

**return** **false**;

}

**switch**(ch\_piece) {

**case** 'k':

**if**(\_LOWS) {

cout << "\tKING: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'K':

**if**(\_CAPS) {

cout << "\tKING: WRONG TARGET\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

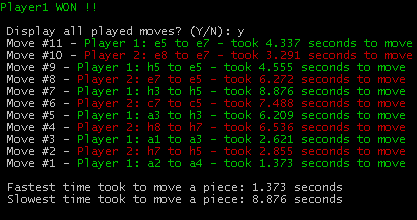
// Valid move

**return** **true**;

}

# References

## PreciseClock.h

PreciseClock.h file was used to clock each player’s movement time.   
When game is finished, each player’s moves, single move and total moves time will be listed and show which move was the fasted and slowest.  
  


/\*

Copyright (c) 2010-2013 Tommaso Urli

Tommaso Urli tommaso.urli@uniud.it University of Udine

Permission is hereby granted, free of charge, to any person obtaining

a copy of this software and associated documentation files (the

"Software"), to deal in the Software without restriction, including

without limitation the rights to use, copy, modify, merge, publish,

distribute, sublicense, and/or sell copies of the Software, and to

permit persons to whom the Software is furnished to do so, subject to

the following conditions:

The above copyright notice and this permission notice shall be

included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,

EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF

MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND

NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE

LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION

WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

\*/

#ifndef PRECISECLOCK\_H

#define PRECISECLOCK\_H

#ifdef \_WIN32

#include <Windows.h>

#else

#include <sys/time.h>

#include <ctime>

#endif

/\* Remove if already defined \*/

**typedef** **long** **long** int64;

**typedef** **unsigned** **long** **long** uint64;

/\* Returns the amount of milliseconds elapsed since the UNIX epoch. Works on both

 \* windows and linux. \*/

uint64 GetTimeMs64() {

#ifdef \_WIN32

/\* Windows \*/

FILETIME ft;

LARGE\_INTEGER li;

/\* Get the amount of 100 nano seconds intervals elapsed since January 1, 1601 (UTC) and copy it

\* to a LARGE\_INTEGER structure. \*/

GetSystemTimeAsFileTime(&ft);

li.LowPart = ft.dwLowDateTime;

li.HighPart = ft.dwHighDateTime;

uint64 ret = li.QuadPart;

ret -= 116444736000000000LL; /\*Convert from file time to UNIX epoch time.\*/

ret /= 10000; /\* From 100 nano seconds (10^-7) to 1 ms(10^-3) intervals \*/

**return** ret;

#else

/\* Linux \*/

**struct** timeval tv;

gettimeofday(&tv, NULL);

uint64 ret = tv.tv\_usec;

/\* Convert from micro seconds (10^-6) to ms (10^-3) \*/

ret /= 1000;

/\* Adds the seconds (10^0) after converting them to ms (10^-3) \*/

ret += (tv.tv\_sec \* 1000);

**return** ret;

#endif

}

#endif /\* PRECISECLOCK\_H \*/

## Colormod.h

Colormod.h file is used to change console colors in unix/linux terminal.

#ifndef COLORMOD\_H

#define COLORMOD\_H

#include <ostream>

**namespace** Color {

**enum** Code {

FG\_RED = 31,

FG\_GREEN = 32,

FG\_BLUE = 34,

FG\_DEFAULT = 39,

BG\_RED = 41,

BG\_GREEN = 42,

BG\_BLUE = 44,

BG\_DEFAULT = 49

};

**class** Modifier {

Code code;

**public**:

Modifier(Code pCode):code(pCode){}

**friend** std::ostream&

**operator**<<(std::ostream& os, **const** Modifier& mod) {

**return** os << "\033["<<mod.code<<"m";

}

}; // end class

} // end namespace

#endif

# Full code

## Game

### Chess.h

#ifndef CHESS\_H

#define CHESS\_H

// User Libraries

#include "../stack/Stack.h"

#include "../game\_player/User.h"

#include "../etc/Colormod.h"

**using** **namespace** Color;

**class** Chess {

// Let PieceMove class access Chess' private variables

**friend** **class** PieceMove;

**private**:

User \*usr;

Stack stck\_mv;

**char** \*\*cdp\_board; // chess board

**char** crg\_from[2]; // move piece location from->to

**char** crg\_to[2];

// Helper functions;

**void** chessInit( **void** ); // game - body/logic

**void** mkMove( **bool** );

**bool** chckMove( **bool** );

**bool** isValid( string & );

**void** gameover( **void** );

**void** playGame( **void** );

**void** display( **void** ); // game - graphic(?) part

**void** drawPcs( **void** );

**void** cls( **void** );

**void** status( **void** );

**void** scrBoard( **bool** ); // file IO

**public**:

Chess() { chessInit(); } // Constructor

~Chess( **void** ); // Destructor

**void** menu( **void** );

};

#endif

### Chess.cpp

// System Libraries

#include <algorithm> // sort

#include <iostream>

#include <fstream> // file IO

#include <cstdlib> // NULL

#include <iomanip> // setw

#include <string.h>

#include <map>

**using** **namespace** std;

// User Libraries

#include "../header/game/Chess.h"

#include "../header/game/PieceMove.h"

#include "../header/etc/PreciseClock.h"

// Symbolic Constants

#define ROW 19

#define COL 19

#define HEADER BGRED << setw(37) << ' ' << BGDEF

// Global Constants

#ifdef \_WIN32 // use GREEN in windows else BLUE

**const** Modifier GBLU(FG\_GREEN);

#else

**const** Modifier GBLU(FG\_BLUE);

#endif

**const** Modifier RED(FG\_RED); // change font color

**const** Modifier DEF(FG\_DEFAULT);

**const** Modifier BGRED(BG\_RED); // change background color

**const** Modifier BGDEF(BG\_DEFAULT);

// User defined libraries

**typedef** list <pCII >::iterator t\_it;

//////////////////////////////////////////////////////////////////////

// Destructor

// Destory char\*\* chessboard and sturctures

//////////////////////////////////////////////////////////////////////

Chess::~Chess( **void** ) {

**for**(**int** i=0;i<10;++i)

**delete** cdp\_board[i];

**delete** [] cdp\_board;

cdp\_board = NULL;

**delete** usr->st\_p1;

**delete** usr->st\_p2;

**delete** usr;

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : Initialize/set all required game data

//////////////////////////////////////////////////////////////////////

**void** Chess::chessInit( **void** ) {

// allocate memory to structures

usr = **new** User();

usr->st\_p1 = **new** Player();

usr->st\_p2 = **new** Computer();

// Allocate memmory to 19x19 charcter array

cdp\_board = **new** **char**\*[ROW];

**for**(**int** i=0;i<ROW;++i)

cdp\_board[i] = **new** **char**[COL];

// Initialize the chess board with blank spaces

**for**(**int** i=0; i<ROW; ++i)

**for**(**int** j=0; j<COL; ++j)

cdp\_board[i][j] = ' ';

// Label ranks(1-8) and files(a-h)

**for**(**int** i=0; i<ROW; ++i) {

**for**(**int** j=0; j<COL; ++j) {

// if odd # row, draw the horizontal grid

**if**( (i&1)==1 ) {

**if**(j!=0 && j!=COL-1)

cdp\_board[i][j] = '-';

}

// if even # row

**else** **if**( (i&0)==0 ) {

// in first and lost column, label ranks(1-8)

**if**( i>1 && i<ROW-1 && (j==0 || j==COL-1) )

cdp\_board[i][j] = '9'-(i/2);

// if odd # col, draw the vertical grid

**if**( (j&1)==1 )

**if**(i!=0 && i!=ROW-1)

cdp\_board[i][j] = '|';

// in first and last row, label flies(a-h)

**if**( (j&1)!=1 && (i==0 || i==ROW-1) && j>1 && j<COL-1)

cdp\_board[i][j] = 'a'+(j/2)-1;

}

} // end inner for

} // end outer for

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PARAMETER : bool player1

// PURPOSE : validate the player's move and move a piece

//////////////////////////////////////////////////////////////////////

**void** Chess::mkMove(**bool** p1) {

cls();

display();

// Display player's name

cout << endl << ((p1)?GBLU:RED)

<< ((p1)?(usr->st\_p1->str\_name):(usr->st\_p2->str\_name))

<< ": " << DEF;

// time how long it takes for a user to make a move

uint64 begin = GetTimeMs64();

getline(cin, usr->str\_move);

uint64 end = GetTimeMs64();

// while user input is invalid

**while**( !isValid(usr->str\_move) ){

cout << endl << ((p1)?GBLU:RED)

<< ((p1)?(usr->st\_p1->str\_name):(usr->st\_p2->str\_name))

<< ": " << DEF;

begin += GetTimeMs64();

getline(cin, usr->str\_move);

end += GetTimeMs64();

}

// player resigns

**if**( (usr->str\_move).compare("resign")==0 ) {

((p1)?(usr->st\_p1->b\_alive):(usr->st\_p2->b\_alive)) = **false**;

**return**;

}

// record moves

stck\_mv.push\_back(usr->str\_move);

// record times

usr->vf\_tmPerMv.push\_back((end-begin)/1000.0f);

// move selected piece

t\_it it = ((p1)?(usr->st\_p1->pcs.begin()):(usr->st\_p2->pcs.begin()));

**while**( it!=((p1)?(usr->st\_p1->pcs.end()):(usr->st\_p2->pcs.end())) ) {

**if**( it->second.first==crg\_from[0] && it->second.second==crg\_from[1] ) {

it->second.first = crg\_to[0];

it->second.second = crg\_to[1];

cdp\_board[crg\_from[0]][crg\_from[1]] = ' ';

it = ((p1)?(usr->st\_p1->pcs.end()):(usr->st\_p2->pcs.end()));

**return**;

}

++it;

} // end while

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PARAMETER : bool p1

// PURPOSE : validate the piece's movement && perform promotion

//////////////////////////////////////////////////////////////////////

**bool** Chess::chckMove( **bool** p1 ) {

PieceMove chkMv;

// check if p2 selected p1's piece or vice versa

**char** ch = cdp\_board[crg\_from[0]][crg\_from[1]];

**if**( ((p1)?(ch>='a'):(ch<'a')) ) {

cout << "\tERROR: NOT YOUR PIECE.\n";

**return** **false**;

}

// check whether piece's move is valid

**else** **if**( !chkMv.fn\_validMv(**this**) ) {

**return** **false**;

}

// check for pawn's promotion

**else** **if**( chkMv.fn\_getPrmt()!=0 ) {

// Pawn Promotion

**char** promo = (p1)?'P':'p';

**switch**(chkMv.fn\_getPrmt()) {

**case** 1: promo = ((p1)?'Q':'q'); **break**;

**case** 2: promo = ((p1)?'R':'r'); **break**;

**case** 3: promo = ((p1)?'B':'b'); **break**;

**case** 4: promo = ((p1)?'N':'n'); **break**;

} // end switch

// Promote the pawn to a desired piece

t\_it it = (p1)?(usr->st\_p1->pcs.begin()):(usr->st\_p2->pcs.begin());

**while**( it!=((p1)?(usr->st\_p1->pcs.end()):(usr->st\_p2->pcs.end())) ) {

**if**( it->first==((p1)?'P':'p')&&(it->second.first==crg\_from[0]&&

it->second.second==crg\_from[1]) ) {

it->first = promo;

it = ((p1)?(usr->st\_p1->pcs.end()):(usr->st\_p2->pcs.end()));

}

++it;

} // end while

}

**return** **true**;

}

//////////////////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : string

// PURPOSE : deteremine whether player's move is valid or not;

// return true if valid else false

//////////////////////////////////////////////////////////////////////

**bool** Chess::isValid( string &str ) {

// remove all spaces from the user input

**for**(**int** i=0; i<str.size(); ++i) {

**if**(str[i]==' ') str.erase(i--,1);

str[i] = tolower(str[i]);

} // end for

// player resigns the game;

**if**(str.compare("resign")==0) **return** **true**;

// player views the previous movement

**if**(str.compare("prev")==0) {

cout << " Previous move: " << stck\_mv.peek() << endl;

**return** **false**;

}

// a2toa3 -> size should be 6

**if**(str.size()!=6) {

cout << "\tSYNTAX ERROR: 6 CHARS PLEASE...\n";

**return** **false**;

}

// a2toa3 -> check for valid files[a-h]

**else** **if**( !(str[0]>=97&&str[0]<=104) || !(str[4]>=97&&str[4]<=104) ) {

cout << "\tSYNTAX ERROR: FILES [a-h]\n";

**return** **false**;

}

// a2toa3 -> check for valid ranks[1-8]

**else** **if**( !(str[1]>='1'&&str[1]<='8') || !(str[5]>='1'&&str[5]<='8') ) {

cout << "\tSYNTAX ERROR: 6 RANKS [1-8]\n";

**return** **false**;

}

// a2toa3 -> check for "to"

**else** **if**( str[2]!='t' || str[3]!='o' ) {

cout << "\tSYNTAX ERROR: 'to/TO'? \n";

**return** **false**;

}

// a2toa2 -> check for stupidity

**else** **if**( str[1]==str[5] && str[0]==str[4]) {

cout << "\tERROR: SUICIDE?\n";

**return** **false**;

}

**else** {

// extract files and ranks from the string

crg\_from[1] = (tolower(str[0])-'a'+1)\*2;

crg\_from[0] = 18-((str[1]-'0')\*2);

crg\_to[1] = (tolower(str[4])-'a'+1)\*2;

crg\_to[0] = 18-((str[5]-'0')\*2);

// check if blank piece was selected

**if**(cdp\_board[crg\_from[0]][crg\_from[1]] == ' ') {

cout << "\tERROR: YOU GOTTA MOVE SOMETHING?\n";

**return** **false**;

}

//PieceMove chkMv;

**switch**( (stck\_mv.getItems()%2!=0)?0:1 ) {

**case** 1: **return** chckMove(**true**); // player 1

**case** 0: **return** chckMove(**false**); // player 2

} // end switch

} // end else

**return** **true**;

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : display the final result

//////////////////////////////////////////////////////////////////////

**void** Chess::gameover() {

cls();

display();

cout << endl;

// Display the winner

**if**( usr->st\_p1->b\_alive ) cout << GBLU << usr->st\_p1->str\_name

<< " WON !!" << DEF << endl;

**else** cout << RED << usr->st\_p2->str\_name << " WON !!" << DEF << endl;

// Display all the moves if user wants to

cout << endl;

cout << " Display all played moves? (Y/N): ";

**char** temp; cin >> temp;

**float** timeP1=0.0f,timeP2=0.0f;

**if**( tolower(temp)=='y' ) {

**for**(**int** i=stck\_mv.getItems(),j=usr->vf\_tmPerMv.size()-1;i>0;--i,--j) {

cout << " Move #" << i << " - " << ((i%2!=0)?GBLU:RED);

cout << "Player " << ((i%2!=0)?1:2)

<< ": " << stck\_mv.pop\_back() << " - "

<< "took " << usr->vf\_tmPerMv[j] << " seconds to move "

<< DEF << endl;

} // end for

cout << endl;

} **else** **if** ( tolower(temp)!='n' )

cout << "error..." << endl;

// sum up player's total spent time moving pieces

**for**(**int** i=usr->vf\_tmPerMv.size()-1;i>0;--i) {

**if**(i%2==0) usr->st\_p1->f\_ttlTm+=usr->vf\_tmPerMv[i];

**else** usr->st\_p2->f\_ttlTm+=usr->vf\_tmPerMv[i];

}

// sort then display fastest and slowest time

**if**( usr->vf\_tmPerMv.size() ) {

sort( usr->vf\_tmPerMv.begin(), usr->vf\_tmPerMv.end() );

cout << " Fastest time took to move a piece: "

<< usr->vf\_tmPerMv[0] << " seconds\n";

cout << " Slowest time took to move a piece: "

<< usr->vf\_tmPerMv[usr->vf\_tmPerMv.size()-1] << " seconds\n\n";

}

// false-> only save data to a file; do not display

scrBoard( **false** );

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : start the game

//////////////////////////////////////////////////////////////////////

**void** Chess::playGame( **void** ) {

cls();

// Read in player names

cout << "Name of player1: ";

cin >> usr->st\_p1->str\_name;

cout << "Name of player2: ";

cin >> usr->st\_p2->str\_name;

cin.ignore();

// player turn

**bool** pTurn = **true**;

**while**(usr->st\_p1->b\_alive&&usr->st\_p2->b\_alive) {

**if**(pTurn) { mkMove(**true**); pTurn = **false**; }

**else** { mkMove(**false**); pTurn = **true**; }

} // end while

// display the final result

gameover();

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : dispaly the chessboard

//////////////////////////////////////////////////////////////////////

**void** Chess::display( **void** ) {

drawPcs(); // draw chess pieces onto the board

status();

// display the board

**for**(**int** i=0;i<ROW;++i) {

**for**(**int** j=0;j<COL;++j) {

**char** ch = cdp\_board[i][j];

// p1->blue, p2->red

**if**( ch>='a'&&ch<='z'&&(i!=0&&i!=ROW-1) ) cout << RED;

**else** **if**( ch>='A'&&ch<='Z' ) cout << GBLU;

cout << cdp\_board[i][j] << DEF << ' ';

} // end inner for

cout << endl;

} // end outer for

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : locate each chess pieces on the grid

//////////////////////////////////////////////////////////////////////

**void** Chess::drawPcs( **void** ) {

t\_it it = usr->st\_p1->pcs.begin();

**while**( it!=usr->st\_p1->pcs.end() )

cdp\_board[it->second.first][it->second.second] = (it++)->first;

it = usr->st\_p2->pcs.begin();

**while**( it!=usr->st\_p2->pcs.end() )

cdp\_board[it->second.first][it->second.second] = (it++)->first;

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : clear the screen

//////////////////////////////////////////////////////////////////////

**void** Chess::cls( **void** ) {

**for**(**int** i=0;i<50;++i) cout << endl;

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : Display player's status of captured pieces;

//////////////////////////////////////////////////////////////////////

**void** Chess::status( **void** ) {

// sort captured pieces

sort( usr->st\_p1->vc\_capture.begin(), usr->st\_p1->vc\_capture.end() );

sort( usr->st\_p2->vc\_capture.begin(), usr->st\_p2->vc\_capture.end() );

// Display Captured pieces

cout << endl << HEADER << endl;

cout << " Captured piece(s): " << endl;

cout << GBLU << " Player 1 :" << DEF << "\n\t";

**for**(**int** i=0;i<usr->st\_p1->vc\_capture.size();++i) {

cout << RED << usr->st\_p1->vc\_capture[i] << DEF << ' ';

**if**( (i+1)%8==0 ) cout << "\n\t";

}

cout << endl << endl;

cout << RED << " Player 2 :" << DEF << "\n\t";

**for**(**int** i=0;i<usr->st\_p2->vc\_capture.size();++i) {

cout << GBLU << usr->st\_p2->vc\_capture[i] << DEF << ' ';

**if**( (i+1)%8==0 ) cout << "\n\t";

}

cout << endl;

// Display Syntax Information

cout << endl << HEADER << endl;

cout << " Syntax: " << endl;

cout << " a2 to a3 -> ?? to ??\n\n";

cout << " Commands: " << endl;

cout << " \"resign\" -> resign the game\n";

cout << " \"prev\" -> display previous move";

cout << endl << HEADER << endl << endl;

cout << RED << " NO" << DEF << " Castling\n\n";

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PARAMETER : bool print

// PURPOSE : if print==true, only display data w/o writ'n to a file

// else only save data to the file w/o displaying

//////////////////////////////////////////////////////////////////////

**void** Chess::scrBoard( **bool** print ) {

map<**float**, string> mymap; // <ttlTime,name>

map<**float**, string>::iterator it;

fstream file("score\_board.dat",ios::in);

// read data from a file

**if**( !file.fail()) {

string str; **float** f;

**while**( file >> str >> f )

mymap[f] = str;

}

file.close();

// display the score board if print is true

**if**( print ) {

it = mymap.begin();

cout << setfill('=') << setw(40) << ' ' << setfill(' ') << endl;

cout << " Winner " << setw(10) << ' ' << "Total Time Spent\n";

cout << setfill('=') << setw(40) << ' ' << setfill(' ') << endl;

**while**( it!=mymap.end() )

cout << left << ' ' << setw(10) << (it++)->second << setw(10)

<< ' ' << it->first << " seconds" << endl;

cout << right << "\n Press Enter .............\n";

cin.ignore(255,'\n');

getchar();

**return**;

}

// add new winner and winner's time to the map

**if**( usr->st\_p1->b\_alive )

mymap[usr->st\_p1->f\_ttlTm] = usr->st\_p1->str\_name;

**else**

mymap[usr->st\_p2->f\_ttlTm] = usr->st\_p2->str\_name;

// write data to a file

file.open("score\_board.dat",ios::out);

it = mymap.begin();

**while**( it!=mymap.end() )

file << (it++)->second << setw(5) << ' ' << it->first << endl;

file.close();

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PURPOSE : Display the menu

//////////////////////////////////////////////////////////////////////

**void** Chess::menu( **void** ) {

**int** n;

**do**{

cls();

cout << " 1. Play Game\n";

cout << " 2. View Scoreboard\n";

cout << " 0. exit\n > ";

cin >> n;

**while**( cin.fail() ) {

cin.clear();

cin.ignore(255, '\n');

cout << " > ";

cin >> n;

}

**switch**(n) {

**case** 0: **return**;

**case** 1: playGame(); **break**;

**case** 2: scrBoard( **true** ); **break**; // true->only display data

}

}**while**( n!=1 );

}

### PieceMove.h

#ifndef PIECEMOVE\_H

#define PIECEMOVE\_H

#include "../game/Chess.h"

**class** PieceMove{

**private**:

**char** ch\_piece;

**short** sh\_promote;

// helper functions

**bool** fn\_kingMv(Chess \*);

**bool** fn\_queenMv(Chess \*);

**bool** fn\_bshpMv(Chess \*);

**bool** fn\_knghtMv(Chess \*);

**bool** fn\_rookMv(Chess \*);

**bool** fn\_pawnMv(Chess \*);

**void** fn\_kill(Chess \*, **char**);

**public**:

// Accessor

**short** fn\_getPrmt()

{ **return** sh\_promote; }

// Member functions

**bool** fn\_validMv(Chess \*); // check for valid move

};

#endif

### PieceMove.cpp

// System Libraries

#include <iostream>

#include <cmath>

**using** **namespace** std;

// User Libraries

#include "../header/game/PieceMove.h"

#include "../header/game/Chess.h"

// Symbolic Constants

#define \_FROM cp->crg\_from

#define \_TO cp->crg\_to

#define \_LOWS cp->cdp\_board[\_TO[0]][\_TO[1]]>=97 &&\

cp->cdp\_board[\_TO[0]][\_TO[1]]<=122

#define \_CAPS cp->cdp\_board[\_TO[0]][\_TO[1]]>=65 &&\

cp->cdp\_board[\_TO[0]][\_TO[1]]<=90

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine which piece is moving and

// return true if the move is valid else false

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_validMv(Chess \*chess) {

ch\_piece = chess->cdp\_board[chess->crg\_from[0]][chess->crg\_from[1]];

sh\_promote = 0;

**switch**( ch\_piece ) {

**case** 'k': **case** 'K':

**return** fn\_kingMv(chess);

**case** 'q': **case** 'Q':

**return** fn\_queenMv(chess);

**case** 'b': **case** 'B':

**return** fn\_bshpMv(chess);

**case** 'n': **case** 'N':

**return** fn\_knghtMv(chess);

**case** 'r': **case** 'R':

**return** fn\_rookMv(chess);

**case** 'p': **case** 'P':

**return** fn\_pawnMv(chess);

**default**:

**return** **false**;

} // end switch

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether pawn's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_pawnMv(Chess \*cp) {

**switch**( ch\_piece ) {

**case** 'p': // player 2 || computer

// starter-> may move 2 places

**if**(\_FROM[0]==4 && \_FROM[1]==\_TO[1] &&

(\_FROM[0]+2==\_TO[0] || \_FROM[0]+4==\_TO[0]) &&

(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ') )

**return** **true**;

// return false if pawn moved more than 1 place

**else** **if**(\_FROM[0]+2!=\_TO[0]) {

cout << "\tPAWN: ILLEGAL MOVE\n";

**return** **false**;

}

// cannot move diagonal: same team

**else** **if**(\_LOWS && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'P': // player 1 || you

// starter-> may move 2 places

**if**(\_FROM[0]==14 && \_FROM[1]==\_TO[1] &&

(\_FROM[0]-2==\_TO[0] || \_FROM[0]-4==\_TO[0]) &&

(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ') )

**return** **true**;

// return false if pawn moved more than 1 place

**else** **if**(\_FROM[0]-2!=\_TO[0]) {

cout << "\tPAWN: ILLEGAL MOVE\n";

**return** **false**;

}

// cannot move diagonal: same team

**else** **if**(\_CAPS && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: WRONG TARGET\n";

**return** **false**;

}

}

// cannot move front: there is something in front

**if**(\_FROM[1]==\_TO[1] && cp->cdp\_board[\_TO[0]][\_TO[1]]!=' ') {

cout << "\tPAWN: CANNOT CAPTURE IN THE SAME DIRECTION ?\n";

**return** **false**;

}

// cannot move diagonal: there is no piece to capture

**else** **if**(cp->cdp\_board[\_TO[0]][\_TO[1]]==' ' && (\_FROM[1]!=\_TO[1])) {

cout << "\tPAWN: TRYING TO CAPTURE SOMETHING?\n";

**return** **false**;

}

// Promotion -> pawn reached the otherside

**if**(\_TO[0]==2 || \_TO[0]==16) {

cout << "\tPromote to:\n";

cout << "\t1. Queen\n";

cout << "\t2. Rook\n";

cout << "\t3. Bishop\n";

cout << "\t4. Knight\n\t> ";

cin >> sh\_promote;

cin.ignore();

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether king's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_kingMv(Chess \*cp) {

**short** sh\_x = abs(\_TO[0] - \_FROM[0]);

**short** sh\_y = abs(\_TO[1] - \_FROM[1]);

// absolute valued (x,y) pairs of distance from A to B.

// There are 4 pairs (2,2) (2,0) (2,4) (0,2)

**if**( !((sh\_x==2&&sh\_y==2) || (sh\_x==2&&sh\_y==0) ||

(sh\_x==2&&sh\_y==4) || (sh\_x==0&&sh\_y==2)) ) {

cout << "\tKING: INVALID MOVE\n";

**return** **false**;

}

**switch**(ch\_piece) {

**case** 'k':

**if**(\_LOWS) {

cout << "\tKING: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'K':

**if**(\_CAPS) {

cout << "\tKING: WRONG TARGET\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether queen's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_queenMv(Chess \*cp) {

**short** sh\_x = \_TO[0] - \_FROM[0];

**short** sh\_y = \_TO[1] - \_FROM[1];

**bool** b\_swap, b\_swap2;

b\_swap = b\_swap2 = **false**;

// make x always greater than y, x > y

**if**( abs(sh\_x) < abs(sh\_y) ) {

swap( sh\_x, sh\_y );

b\_swap = **true**;

}

// (vertical and horizontal move) || (diagonal moves)

**if**( !((sh\_y==0 && sh\_x%2==0) || (abs(sh\_x)==abs(sh\_y))) ) {

cout << "\tQUEEN: INVALID MOVE\n";

**return** **false**;

}

// if swapped, change it back

**if**( b\_swap ) {

swap( sh\_x, sh\_y );

b\_swap = **false**;

}

// Invalid move: capturing the same team

**switch**( ch\_piece ) {

**case** 'q':

**if**(\_LOWS) {

cout << "\tQUEEN: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'Q':

**if**(\_CAPS) {

cout << "\tQUEEN: WRONG TARGET\n";

**return** **false**;

}

}

// Check vertial moving

**if**(\_FROM[1]==\_TO[1]) {

// swap values when (going up)

**if**(\_FROM[0] > \_TO[0]) {

swap(\_FROM[0],\_TO[0]);

b\_swap = **true**;

}

// check if player's piece is jumping over any pieces

**for**(**int** i=\_FROM[0]+2; i<=\_TO[0]-2; i+=2)

**if**(cp->cdp\_board[i][\_FROM[1]]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Check horizontal moving

**else** **if**(\_FROM[0]==\_TO[0]){

// swap values when (going left)

**if**(\_FROM[1] > \_TO[1]) {

swap(\_FROM[1],\_TO[1]);

b\_swap2 = **true**;

}

// check if player's piece is jumping over any pieces

**for**(**int** i=\_FROM[1]+2; i<=\_TO[1]-2; i+=2)

**if**(cp->cdp\_board[\_FROM[0]][i]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// check diagonals

**else** {

// negative or positive

**int** temp = abs(sh\_x)/sh\_x;

**int** temp2 = abs(sh\_y)/sh\_y;

**for**(**int** i=2; i<=abs(sh\_x)-2; i+=2) {

**if**( cp->cdp\_board[\_FROM[0]+(i\*temp)][\_FROM[1]+(i\*temp2)]!=' ') {

cout << "\tQUEEN: CANNOT GO THROUGH\n";

**return** **false**;

}

}

}

// if swapped, swap it back to original state

**if**(b\_swap)

swap(\_FROM[0],\_TO[0]);

**else** **if**(b\_swap2)

swap(\_FROM[1],\_TO[1]);

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether bishop's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_bshpMv(Chess \*cp) {

**short** sh\_x = \_TO[0] - \_FROM[0];

**short** sh\_y = \_TO[1] - \_FROM[1];

**if**( abs(sh\_x) != abs(sh\_y) ) {

cout << "\tBISHOP: INVALID MOVE\n";

**return** **false**;

}

**int** temp = abs(sh\_x)/sh\_x;

**int** temp2 = abs(sh\_y)/sh\_y;

**switch**(ch\_piece) {

**case** 'b':

// check if player is trying to capture their own piece

**if**(\_LOWS) {

cout << "\tBISHOP: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'B':

// check if player is tring to capture their own piece

**if**(\_CAPS) {

cout << "\tBISHOP: WRONG TARGET\n";

**return** **false**;

}

}

// check if player's piece is jumping over the pieces

**for**(**int** i=2; i<=abs(sh\_x)-2; i+=2) {

**if**( cp->cdp\_board[\_FROM[0]+(i\*temp)][\_FROM[1]+(i\*temp2)]!=' ') {

cout << "\tBISHOP: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether knight's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_knghtMv(Chess \*cp) {

**short** sh\_x = abs(\_TO[0] - \_FROM[0]);

**short** sh\_y = abs(\_TO[1] - \_FROM[1]);

// regardless of the direction,

// night's abs() distance from A to B is (4,2) or (2,4)

**if**( !((sh\_x==4 && sh\_y==2)||(sh\_x==2 && sh\_y==4)) ) {

cout << "\tKNIGHT: ILLEGAL MOVE\n";

**return** **false**;

}

**switch**(ch\_piece) {

**case** 'n':

// check if player is tring to capture their own piece

**if**(\_LOWS) {

cout << "\tKNIGHT: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'N':

// check if player is tring to capture their own piece

**if**(\_CAPS) {

cout << "\tKNIGHT: WRONG TARGET\n";

**return** **false**;

}

}

// Captured something?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : bool

// PARAMETER : Chess

// PURPOSE : Determine whether rook's move is valid

///////////////////////////////////////////////////////////

**bool** PieceMove::fn\_rookMv(Chess \*cp) {

// check if rook is moving diagonally

**if**(\_FROM[0]!=\_TO[0] && \_FROM[1]!=\_TO[1]) {

cout << "\tROOK: ILLEGAL MOVE\n";

**return** **false**;

}

**switch**( ch\_piece ) {

**case** 'r': // player 2

// check if player is trying to capture their own piece

**if**(\_LOWS) {

cout << "\tROOK: WRONG TARGET\n";

**return** **false**;

}

**break**;

**case** 'R': // player 1

// check if player is trying to capture their own piece

**if**(\_CAPS) {

cout << "\tROOK: WRONG TARGET\n";

**return** **false**;

}

}

**bool** b\_swap, b\_swap2;

b\_swap = b\_swap2 = **false**;

// Vertical checking

**if**(\_FROM[1]==\_TO[1]) {

// 2 to 8 (going downward), 8 to 2(going upward)

// swap values when (going up)

**if**(\_FROM[0] > \_TO[0]) {

swap(\_FROM[0],\_TO[0]);

b\_swap = **true**;

}

**for**(**int** i=\_FROM[0]+2; i<=\_TO[0]-2; i+=2)

**if**(cp->cdp\_board[i][\_FROM[1]]!=' ') {

cout << "\tROOK: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// Horizontal checking

**else** {

// 2 to 8 (going right), 8 to 2(going left)

// swap values when (going left)

**if**(\_FROM[1] > \_TO[1]) {

swap(\_FROM[1],\_TO[1]);

b\_swap2 = **true**;

}

**for**(**int** i=\_FROM[1]+2; i<=\_TO[1]-2; i+=2)

**if**(cp->cdp\_board[\_FROM[0]][i]!=' ') {

cout << "\tROOK: CANNOT GO THROUGH\n";

**return** **false**;

}

}

// if swapped, swap it back to original state

**if**(b\_swap)

swap(\_FROM[0],\_TO[0]);

**else** **if**(b\_swap2)

swap(\_FROM[1],\_TO[1]);

// Captured somthing?

**if**( cp->cdp\_board[\_TO[0]][\_TO[1]] != ' ')

fn\_kill(cp, cp->cdp\_board[\_TO[0]][\_TO[1]]);

// Valid move

**return** **true**;

}

///////////////////////////////////////////////////////////

// RETURN : void

// PARAMETER : Chess, char

// PURPOSE : Determine which piece was captured and

// decrease the remaining number of that piece

///////////////////////////////////////////////////////////

**void** PieceMove::fn\_kill(Chess \*cp, **char** ch) {

list<pCII >::iterator it;

**switch**( (cp->stck\_mv.getItems()%2!=0)?2:1 ) {

**case** 1: { // player 1

it = cp->usr->st\_p2->pcs.begin();

**while**( it!=cp->usr->st\_p2->pcs.end() ) {

**if**( it->first==ch&&(it->second.first==cp->crg\_to[0]&&

it->second.second==cp->crg\_to[1]) ) {

**if**( ch=='k' ) cp->usr->st\_p2->b\_alive = **false**;

cp->usr->st\_p2->pcs.erase(it);

it=cp->usr->st\_p2->pcs.end();

cp->usr->st\_p1->vc\_capture.push\_back(ch);

**continue**;

}

++it;

}

**break**;

}

**case** 2: { // player 2

it = cp->usr->st\_p1->pcs.begin();

**while**( it!=cp->usr->st\_p1->pcs.end() ) {

**if**( it->first==ch&&(it->second.first==cp->crg\_to[0]&&

it->second.second==cp->crg\_to[1]) ) {

**if**( ch=='K' ) cp->usr->st\_p1->b\_alive = **false**;

cp->usr->st\_p1->pcs.erase(it);

it=cp->usr->st\_p1->pcs.end();

cp->usr->st\_p2->vc\_capture.push\_back(ch);

**continue**;

}

++it;

}

}

}

}

## Game Player

### Computer.h

#ifndef COMPUTER\_H

#define COMPUTER\_H

// System Libraries

#include <list>

#include <vector>

// User defined types

**typedef** pair<**int**,**int**> pII;

**typedef** pair<**char**, pair<**int**,**int**> > pCII;

**struct** Computer {

list<pCII > pcs;

vector<**char**> vc\_capture;

**bool** b\_alive;

std::string str\_name;

**float** f\_ttlTm;

// initialize computer side pcs starting location

Computer() {

**for**(**int** i=0;i<8;++i)

pcs.push\_back( pCII('p', pII(4,(2\*i)+2)) );

pcs.push\_back( pCII('r', pII(2,2)) );

pcs.push\_back( pCII('r', pII(2,16) ));

pcs.push\_back( pCII('n', pII(2,4)) );

pcs.push\_back( pCII('n', pII(2,14) ));

pcs.push\_back( pCII('b', pII(2,6)) );

pcs.push\_back( pCII('b', pII(2,12) ));

pcs.push\_back( pCII('q', pII(2,8)) );

pcs.push\_back( pCII('k', pII(2,10) ));

b\_alive = **true**;

f\_ttlTm = 0.0f;

}

};

#endif

### Player.h

#ifndef PLAYER\_H

#define PLAYER\_H

// System Libraries

#include <list>

#include <vector>

// User defined types

**typedef** pair<**int**,**int**> pII;

**typedef** pair<**char**, pair<**int**,**int**> > pCII;

**struct** Player {

list<pCII > pcs;

vector<**char**> vc\_capture;

**bool** b\_alive;

std::string str\_name;

**float** f\_ttlTm;

// initialize player side pcs starting location

Player() {

**for**(**int** i=0;i<8;++i)

pcs.push\_back( pCII('P', pII(14,(2\*i)+2)) );

pcs.push\_back( pCII('R', pII(16,2)) );

pcs.push\_back( pCII('R', pII(16,16)) );

pcs.push\_back( pCII('N', pII(16,4)) );

pcs.push\_back( pCII('N', pII(16,14)) );

pcs.push\_back( pCII('B', pII(16,6)) );

pcs.push\_back( pCII('B', pII(16,12)) );

pcs.push\_back( pCII('Q', pII(16,8))) ;

pcs.push\_back( pCII('K', pII(16,10)) );

b\_alive = **true**;

f\_ttlTm = 0.0f;

}

};

#endif

### User.h

#ifndef USER\_H

#define USER\_H

// System Libraries

#include <list>

#include <vector>

// User Libraries

#include "Player.h"

#include "Computer.h"

**struct** User {

Player \*st\_p1;

Computer \*st\_p2;

vector<**float**> vf\_tmPerMv;

string str\_move;

};

#endif

## Stack

### Node.h

#ifndef NODE\_H

#define NODE\_H

**struct** Node {

Node \*ptr;

**char** from[2]; // chess moves [from->to]

**char** to[2];

};

#endif

### Stack.h

#ifndef STACK\_H

#define STACK\_H

// System Libraries

#include <cstdlib>

#include <iostream>

#include <string>

**using** **namespace** std;

// User Libraries

#include "Node.h"

**class** Stack {

**private**:

Node \*front;

Node \*lastNode;

**int** items;

**public**:

Stack( **void** ); // Constructor

~Stack( **void** ); // Destructor

// Member Functions

**void** push\_back( string );

string pop\_back( **void** );

string peek( **void** );

**int** getItems( **void** );

};

#endif

### Stack.cpp

#include <iostream>

#include "../header/stack/Stack.h"

//////////////////////////////////////////////////////////////////////

// Constructor

//////////////////////////////////////////////////////////////////////

Stack::Stack( **void** ) {

front = **new** Node;

front->from[0]='0';

front->from[1]='0';

front->to[0]='0';

front->to[1]='0';

lastNode = front;

items=0;

}

//////////////////////////////////////////////////////////////////////

// Destructor

//////////////////////////////////////////////////////////////////////

Stack::~Stack( **void** ) {

// delete each node

lastNode = NULL;

**while**( front->ptr!=NULL ) {

Node \*temp = front;

front = front->ptr;

**delete** temp;

}

// delet the last node

**delete** front;

**delete** lastNode;

}

//////////////////////////////////////////////////////////////////////

// RETURN : void

// PRE-CONDITION : -

// POST-CONDITION: add data at the end of the list

// PURPOSE : push Node at the end of the list

//////////////////////////////////////////////////////////////////////

**void** Stack::push\_back( string move ) {

// Create a new node

Node \*temp = **new** Node;

temp->from[0] = move[0];

temp->from[1] = move[1];

temp->to[0] = move[4];

temp->to[1] = move[5];

temp->ptr = NULL;

// Since I remembered the lastNode let lastNode points to the temp

// and now temp becomes the lastNode

lastNode->ptr=temp;

lastNode=temp;

++items;

}

//////////////////////////////////////////////////////////////////////

// RETURN : string

// PRE-CONDITION : list size > 0

// POST-CONDITION: last data is removed

// PURPOSE : pop data from the end of the list

//////////////////////////////////////////////////////////////////////

string Stack::pop\_back( **void** ) {

--items;

string mv;

mv+= lastNode->from[0];

mv+=lastNode->from[1];

mv+=" to ";

mv+=lastNode->to[0];

mv+=lastNode->to[1];

**if**(front->ptr==NULL) **return** mv;

// locate the one previous to the last Node

Node \*end = **new** Node;

end = front;

**while**(end->ptr->ptr!=NULL)

end=end->ptr;

// let one previous to the last one be the lastNode

lastNode = end;

// delete the last node

end = end->ptr;

**delete** end;

// lastNode points to empty Node

lastNode->ptr = NULL;

**return** mv;

}

//////////////////////////////////////////////////////////////////////

// RETURN : string

// PURPOSE : access the last data w/o modifying the list

//////////////////////////////////////////////////////////////////////

string Stack::peek( **void** ) {

string mv = "";

mv+=lastNode->from[0];

mv+=lastNode->from[1];

mv+=" to ";

mv+=lastNode->to[0];

mv+=lastNode->to[1];

**return** mv;

}

//////////////////////////////////////////////////////////////////////

// RETURN : int

// PURPOSE : return the number of items in a stack

//////////////////////////////////////////////////////////////////////

**int** Stack::getItems( **void** ) {

**return** items;

}