# 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

*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)*

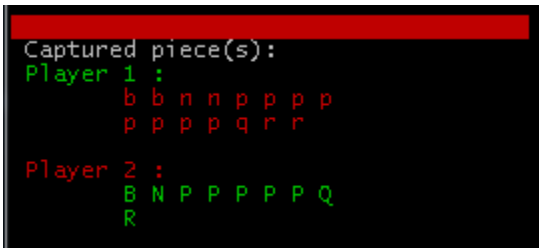
# Introduction

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*****.**

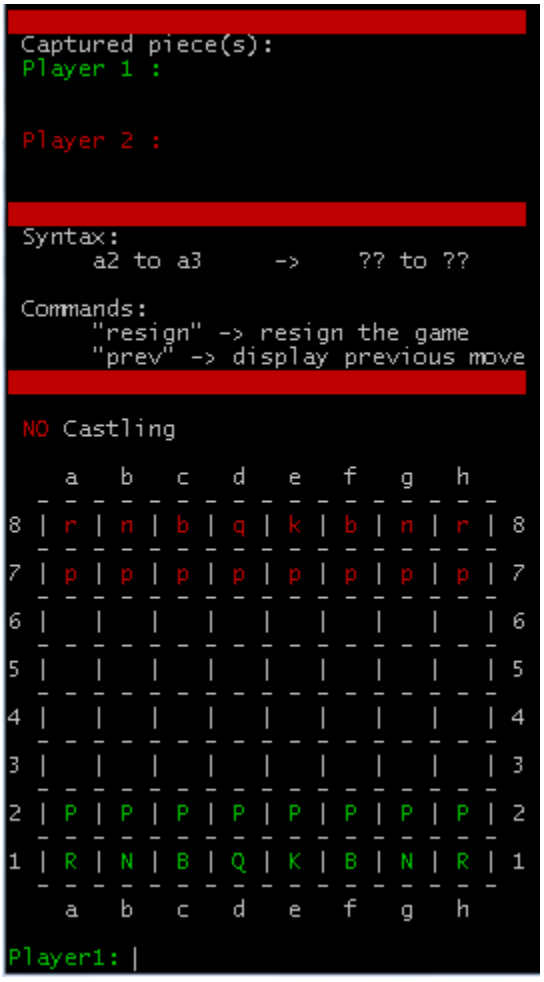
The game rule is same as regular chess game and it runs exactly same except that the program does not include castling, checkmate, and stalemate. When game is done, the name of the winner will be saved in a file with the time which represents the total time spent making a move.

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:

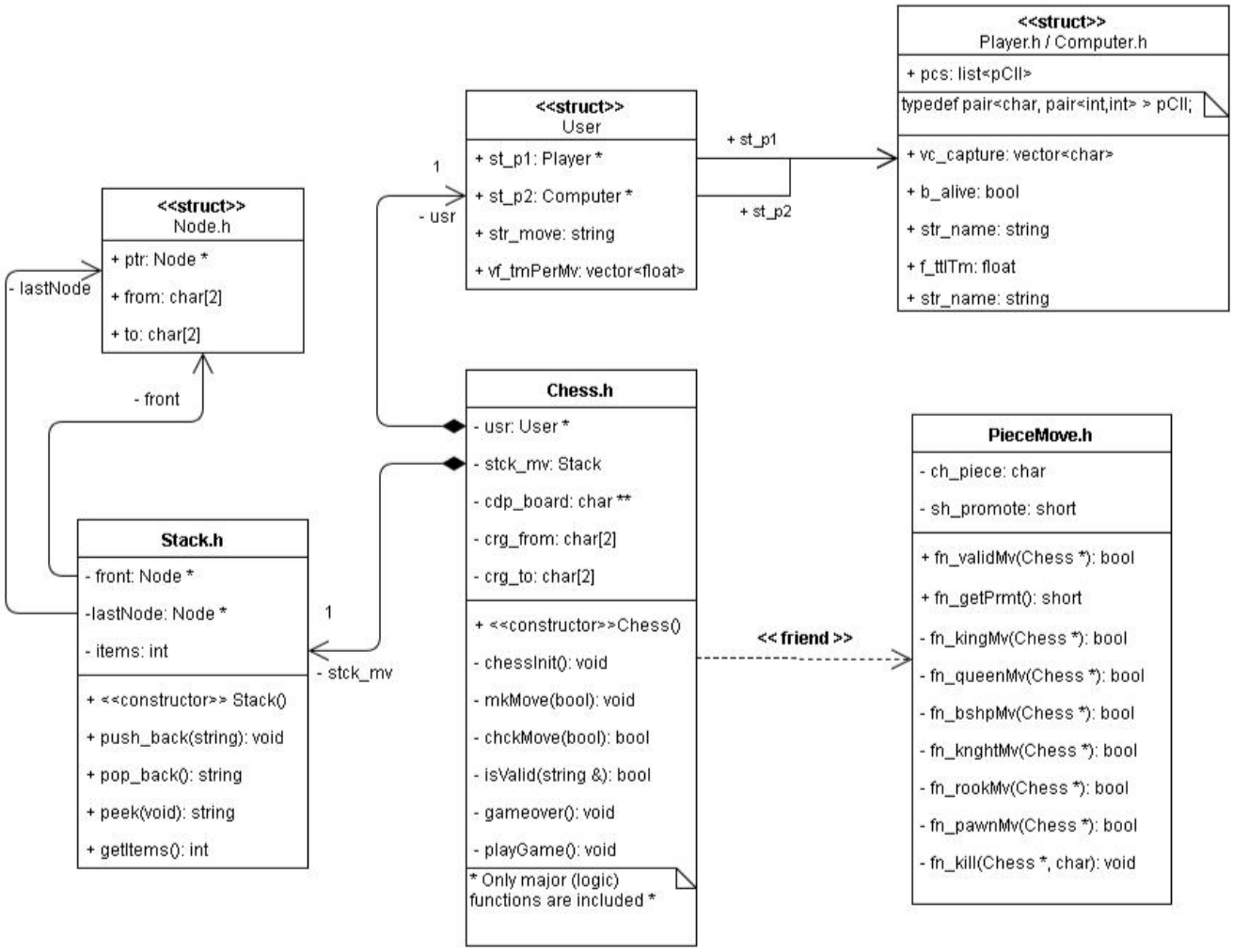
| | |
|---|---|
| User *usr | Player and computer's structure |
| Char **cdp_board | Chess board |
| Char crg_from[2] | Initial location of a piece |
| Char crg_to[2] | Destination of a moving piece |
| Char ch_piece | Determine which piece is moving |
| List<pair<char,pari<int,int> > > pcs | Store pieces' 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:

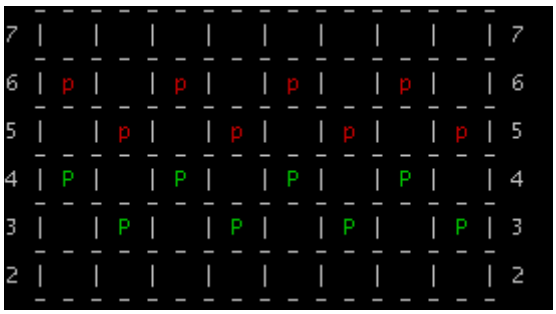
- A pawn can move to the square directly in front of itself, if that square is clear;
- A pawn on its starting position has the option of moving two squares;

## 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;
}
```

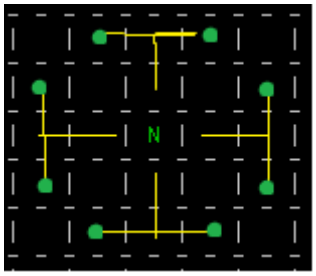
# Knights

## 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;
}
```

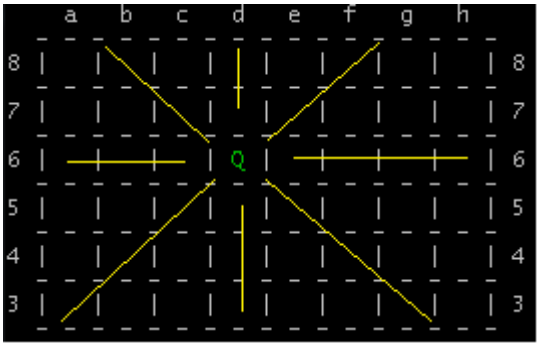
# Queens

## 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;
}
```

---

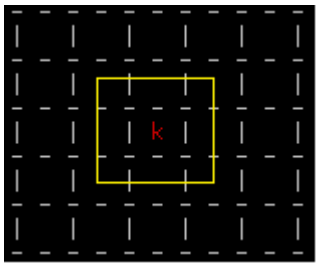
# Kings

### 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

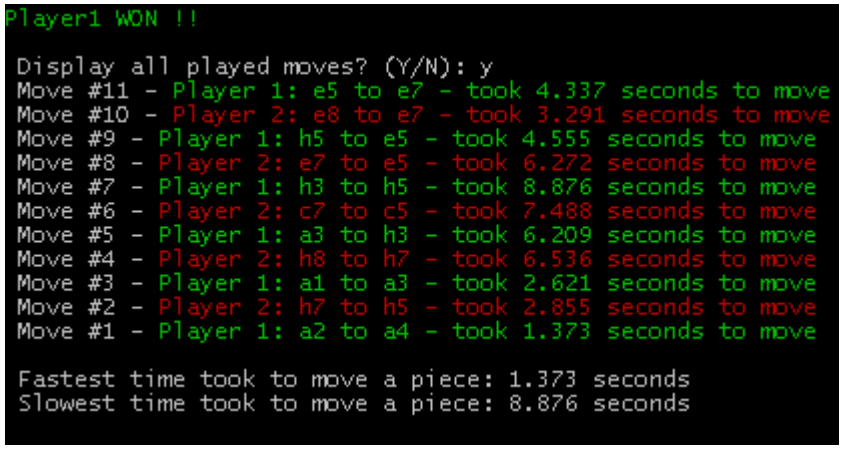
```
bool PieceMove::fn_kingMv(Chess *cp) {
      short sh_x = abs(_TO[0] - _FROM[0]);
      short sh_y = abs(_TO[1] - _FROM[1]);
      //    absolute valued (x,y) pairs of distance from A to B.
      //    There are 4 pairs (2,2)  (2,0)  (2,4)  (0,2)
```

```
	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
```

```c
/* 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;
}
```