PROJECT 1

Chess Game

MAY 3, 2022

YAN CHENG

22SPR-CIS-17C-43484

Introduction

Title: Chess

Chess is played on an 8 X 8 board where the initial placement of pieces is as shown in the following figure. Fox the indexing scheme, the white king is on e1, and the black king is on e8.

The rules are as below:

a) A king moves only one square in any direction.

b) A queen combines the power of a rook and bishop and can move any number of squares along a rank, file, or diagonal, but a queen cannot leap over other pieces.

c) A pawn in the initial position may move one or two squares vertically forward to an empty square but cannot leap over any piece. Subsequently it can move only one square vertically forward to an empty square. A pawn may also capture (replace) an opponent's piece diagonally one square in front of it. Pawns can never move backwards. These are the only moves.

d) A rook can move any number of squares horizontally or vertically, forward or backward, as long as it does not have to leap over other pieces. At the end of the move, it can occupy a previously empty square or capture (replace) an opponent's piece but it cannot replace another piece of the same player.

e) A bishop can move any number of squares diagonally in any direction as long as it does not have to leap over other pieces. At the end of the move, it can occupy a previously empty square or capture (replace) an opponent's piece but it cannot replace another piece of the same player.

f) A knight's move forms an "L"-shape: two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The knight is the only piece that can leap over other pieces. it can move two squares vertically and one square horizontally, or two squares horizontally and one square vertically.

Description

The main point that I programmed this project is the utilization of STL library with iterators and algorithms.

Summary

Project size: about 1300 lines

The number of variables: 13

The number of classes: 8

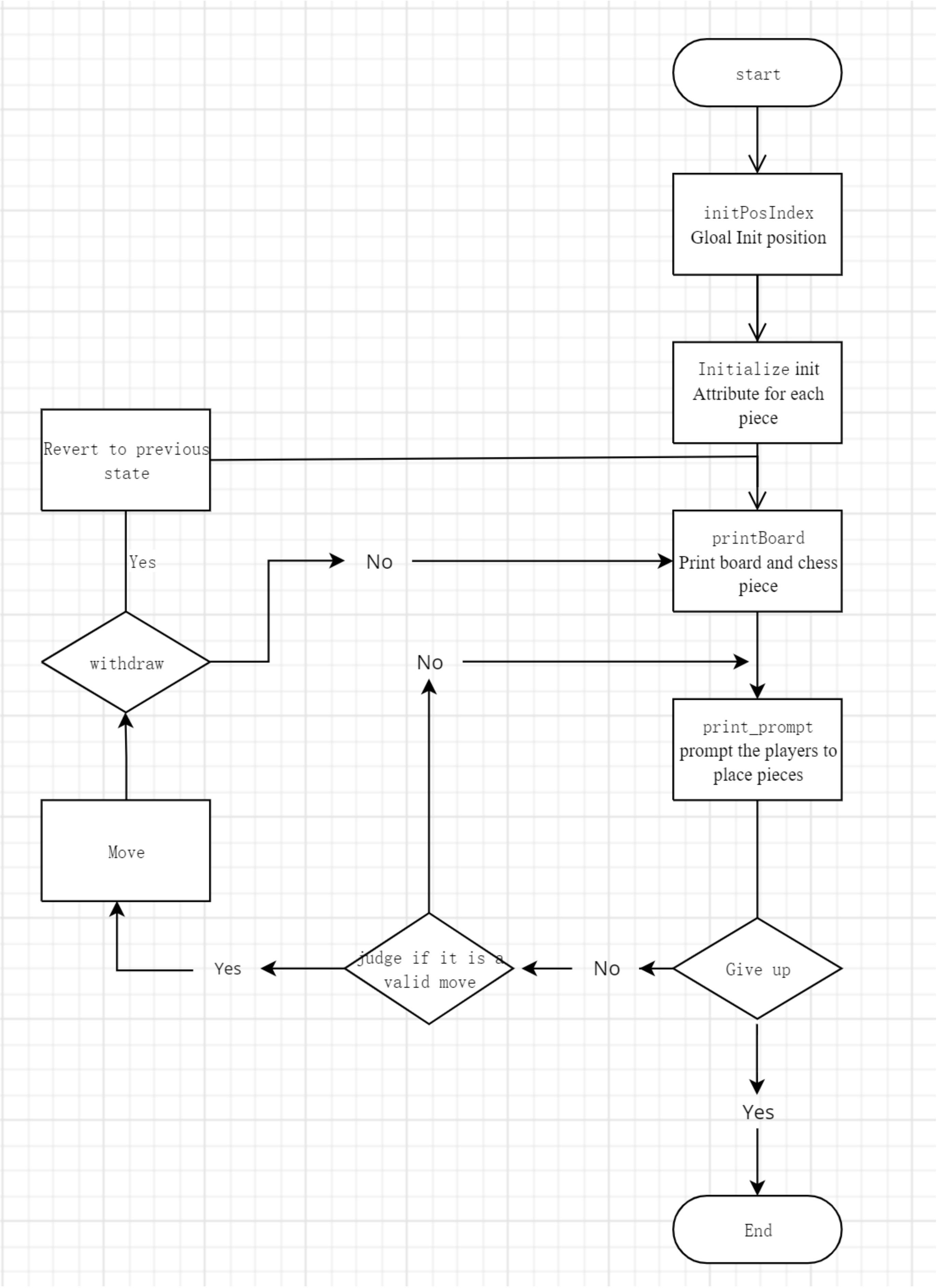
Chess game is the only board game that I am good at, and I don’t know any card games, so I decided to create this one. I created a toy version that set up the chess pieces on the board, move them and find possible moves given a certain position. The design is still not perfect because I didn’t consider some special rules, such as En passant, Castling, and so on. It can be improved in some areas, for example competing with the computer, designing a database to display records, and create a GUI to display the game.

It took about 25 days to finish, based on my knowledge on CIS-17A and 17C, and I utilized the container classes such as Associative containers map, sequence list, container adaptors queue and stack, iterators and algorithm find, count and swap. During the coding, I met a lot of problems, especially utilizing the map and stack, after referring to internet information and YouTube videos and asking help from the Virtual open lab, I solved most of them.

Github location:

https://github.com/YYCJW/CIS_17C_Project

Flow Chart


start
initPosIndex
Gloal Init position
Initialize init
Attribute for each
piece
Revert to previous
state
printBoard
Print board and chess
piece
Yes
No
withdraw
No
print_prompt
prompt the players to
place pieces
Move
Yes
judge if it is a
valid move
No
Give up
Yes
End

Pseudo Code

```
Initialize
Judge the state of the given position
    If there is no piece
        If the piece can move to the given position based on the game rules
            Move a piece to a given position
    If there is a piece
        If the piece at the given position has the same color
            Can't move
        Else if there is an opponent's piece at the given position
            Judge if the piece can move to the given position based on the game rules
                If it can move
                    Captures it and take the position
                Else can't move
```

Major Variables

Type	Variable Name	Description	Location

Int	BLOCK_SIZE	The size of each square	printBoard()
			initPosIndex()

	CHESS_SIZE	The size of the chess board	printBoard()
			legalMoves()

	row	The row of the chess board	initPosIndex()
			getPiece(string position)
			placePiece(ChessPiece* piece, string position)

			printBoard()

	column	The column of the chess board	initPosIndex()
			getPiece(string position)
			placePiece(ChessPiece* piece, string position)
			printBoard()

map	globalMapPosToVirtualIndex	Map the coordinate a1 to h8 of the chess pieces to (0, 0) to (8, 8) in the printing chess board	initPosIndex()
			getPiece(string position)
			placePiece(ChessPiece* piece, string position)
			printBoard()

	globalMapvirtualIndexToPos	Map the printing coordinate (0, 0) to (8, 8) of the chess pieces in the printing chess board to a1 to h8	initPosIndex()
			judgePieceStat()
			legalMoves()

	globalMapRealIndexToPos	The real printing coordinate of each chess piece in the printing chess board	initPosIndex()
			printBoard()

Stack<int>	stackStep	Record the move times	main()
queue<string> queueHistoryOp;	queueHistoryOp	Record the position of each move	main()
bool	m_bWhite	The color of player and the sequence of player moving	main()

enum	color	The color of chess pieces	initialize()
			placePiece(ChessPiece* piece, string position)

ChessBoard	*board	The chess board object	getPiece(string position)
			placePiece(ChessPiece* piece, string position)
			printBoard()
			initialize()

ChessPiece	board[][]	The chess pieces object	getPiece(string position)
			placePiece(ChessPiece* piece, string position)
			printBoard()
			initialize()

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <iostream>

#include <set>

#include <stack>

#include <queue>

#include <list>

#include <string>

#include <map>

#include <algorithm>

using namespace std;

#include "king.h"

#include "knight.h"
```

```
#include "bishop.h"

#include "rook.h"

#include "queen.h"

#include "pawn.h"

#include "chessBoard.h"


//Function prototypes

int print_prompt(bool);

int initPosIndex();

int destroy();


//Execution Begins Here

int main(int argc,char* argv[])

{

    //initialize all the variables

    char szfrom[8] = {0},szTo[8] = {0};

    int seque = 0;

    initPosIndex();

    ChessBoard* board = new ChessBoard;

    board->initialize();

    board->printBoard();

    while(1)
```

```
{
    print_prompt(board->m_bWhite);
    if(!board->stackStep.empty())
    {
        printf("do you lose? 1:yes 2:no\n");
    int succFlag = 0;
    fflush(stdin);
    scanf("%d",&succFlag);
    if(succFlag == 1)
    {
        if(board->m_bWhite)
        {
            printf("the white bide succ\n");
        }
        else
        {
            printf("the black bide succ\n");
        }
        break;
    }
    else
    {
            printf("you do not choose lose,please move continue\n");
        }
        }

        fflush(stdin);
    scanf("%s %s",&szfrom,&szTo);
    if( board->move(szfrom,szTo) == false)
    {
        continue;
    }

    board->printBoard();
    printf("is withdraw? 1:yes 2:no\n");
    fflush(stdin);
        scanf("%d",&seque);
        if(seque == 1)
        {
            if( !board->stackStep.empty())
            {
                int step = board->stackStep.top();
                board->stackStep.pop();
                ChessBoard* board1 = new ChessBoard;
    board1->initialize();
    string strHistroyMove;
    for(int i = 1; i < step;i++)
    {
        strHistroyMove = board->queueHistoryOp.front();
        board->queueHistoryOp.pop();
        size_t pos = strHistroyMove.find("_");
        string strfrom = strHistroyMove.substr(0,pos);
        string strTo = strHistroyMove.substr(pos+1);
        board1->move(strfrom,strTo);
        board1->m_iStep = i;
        board1->stackStep.push(i);
 board1->queueHistoryOp.push(strHistroyMove);
    }
    board1->printBoard();
    board1->m_bWhite = board->m_bWhite;
    delete board;
    board = board1;
    continue;
}
else
{
```

```
            ChessBoard* board1 = new ChessBoard;

            board1->initialize();

            board1->printBoard();

            board1->m_bWhite = board->m_bWhite;

            delete board;

            board = board1;

            continue;

        }

    }


board->m_iStep++;

    string strOperation = string(szfrom) + string("_") + szTo;

    board->stackStep.push(board->m_iStep);

    board->queueHistoryOp.push(strOperation);

   if(board->m_bWhite)

    {

        board->m_bWhite = false;

    }

    else

    {

        board->m_bWhite = true;

    }
    }
    delete board;
    destroy();
    return 0;
}

int print_prompt(bool flag)
{
    if(flag)
    {
        printf("please white side move\n");
    }
    else
    {
        printf("please black side move\n");
    }
    return 0;
}

int initPosIndex()
{
    globalMapPosToVirtualIndex.clear();
```

```
globalMapvirtualIndexToPos.clear();
globalMapRealIndexToPos.clear();
char szTemp[16] = {0};
char szTemp1[16] = {0};
SPieceIndex sPieceIndexTemp;
for(int i= CHESS_SIZE;i > 0; i--)
{
    for(int j = 'a'; j < 'i'; j++)
    {
        sprintf(szTemp,"%c%d",j,i);
        sPieceIndexTemp.row = CHESS_SIZE - i;
        sPieceIndexTemp.column = j -'a';

globalMapPosToVirtualIndex.insert(mapPosToVirtualIndex::value_type(szTemp,sPieceIndexTemp));
        sprintf(szTemp1,"%d_%d",CHESS_SIZE - i,j -'a');

globalMapvirtualIndexToPos.insert(mapVirtualIndexToPos::value_type(szTemp1,szTemp));
        int iRealRow = (CHESS_SIZE - i)*BLOCK_SIZE + BLOCK_SIZE/2;
        int iRealColumn = ( j - 'a')*BLOCK_SIZE + BLOCK_SIZE/2;
```

```
			sprintf(szTemp1,"%d_%d",iRealRow,iRealColumn);


	globalMapRealIndexToPos.insert(mapRealIndexToPos::value_type(szTemp1,szTemp));
		}
	}
}

int destroy()
{
	globalMapPosToVirtualIndex.clear();
	globalMapvirtualIndexToPos.clear();
	globalMapRealIndexToPos.clear();
}
```

```
#ifndef CHESSBOARD_H
#define CHESSBOARD_H
```

```
#include "chessPiece.h"


//Chess board
class ChessBoard{
public:
    //initialize the board to an 8X8 array, each element of the array is a square of the board
    ChessBoard();
    ~ChessBoard();
public:
    //initialize the board with an opening state, calling placePiece() method below to place the pieces in the right position
    void initialize();
    //returns the chess piece at a given position
    ChessPiece* getPiece(string position);
    //place the given piece at a give position, and returns true if successfully
    //returns false if there is already a piece of the same player in the given position
    //If an opponent's piece exists in the given position, that piece is captured, and returns true
      bool placePiece(ChessPiece* piece, string position);
```

```
        //checks if moving the piece from the fromPosition to toPosition is a legal move
        bool move(string fromPosition, string toPosition);

        //print the chess board
        void printBoard();

        //debug the program
        string toString();

        int destroy();

        int initPosIndex();
public:
      ChessPiece* board[8][8];
      stack<int> stackStep;
      queue<string> queueHistoryOp;
      int m_iStep;
      bool m_bWhite;
};

ChessBoard::ChessBoard()
```

```
{
    for(int i = 0;i < 8;i++)
    {
        for(int j =0;j < 8 ;j++)
        {
            board[i][j] = NULL;
        }
    }
    m_iStep = 0;
    m_bWhite = true;
}

ChessBoard::~ChessBoard()
{
    for(int i = 0;i < 8;i++)
    {
        for(int j =0;j < 8 ;j++)
        {
            if(board[i][j] != NULL)
            {
                delete board[i][j];
                board[i][j] = NULL;
            }
        }
    }

    while(!stackStep.empty())
    {
        stackStep.pop();
    }
    queue<string> empty;
  swap(empty,queueHistoryOp);
}

void ChessBoard::initialize()
{
    //Initialize the white pieces
    placePiece(new Rook(this,ChessPiece::WHITE,"\u2656"),"a1");
    placePiece(new Rook(this,ChessPiece::WHITE,"\u2656"),"h1");
    placePiece(new Knight(this,ChessPiece::WHITE,"\u2658"),"b1");
    placePiece(new Knight(this,ChessPiece::WHITE,"\u2658"),"g1");
    placePiece(new Bishop(this,ChessPiece::WHITE,"\u2657"),"c1");
    placePiece(new Bishop(this,ChessPiece::WHITE,"\u2657"),"f1");
    placePiece(new Queen(this,ChessPiece::WHITE,"\u2655"),"d1");
    placePiece(new King(this,ChessPiece::WHITE,"\u2654"),"e1");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"a2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"b2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"c2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"d2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"e2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"f2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"g2");
    placePiece(new Pawn(this,ChessPiece::WHITE,"\u2659"),"h2");

    //Initialize the black pieces
    placePiece(new Rook(this,ChessPiece::BLACK,"\u265C"),"a8");
    placePiece(new Rook(this,ChessPiece::BLACK,"\u265C"),"h8");
    placePiece(new Knight(this,ChessPiece::BLACK,"\u265E"),"b8");
    placePiece(new Knight(this,ChessPiece::BLACK,"\u265E"),"g8");
    placePiece(new Bishop(this,ChessPiece::BLACK,"\u265D"),"c8");
    placePiece(new Bishop(this,ChessPiece::BLACK,"\u265D"),"f8");
    placePiece(new Queen(this,ChessPiece::BLACK,"\u265B"),"d8");
    placePiece(new King(this,ChessPiece::BLACK,"\u265A"),"e8");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"a7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"b7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"c7");
```

```
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"d7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"e7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"f7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"g7");
    placePiece(new Pawn(this,ChessPiece::BLACK,"\u265F"),"h7");
}

//move a piece to a given position, and returns true if successfully
//if the given position has a piece of same color, returns false
//if the given position has the opponent's piece, then captures it and returns true
bool ChessBoard::placePiece(ChessPiece* piece, string position)
{
    mapPosToVirtualIndex::iterator itr = globalMapPosToVirtualIndex.find(position);
    if(itr == globalMapPosToVirtualIndex.end())
        return false;
    SPieceIndex sTmp = itr->second;
    if(board[sTmp.row][sTmp.column] == NULL)
    {
        board[sTmp.row][sTmp.column] = piece;
        piece->setRow(sTmp.row);
        piece->setColumn(sTmp.column);
        return true;
    }

    ChessPiece* ptr = board[sTmp.row][sTmp.column];
    if(ptr->getColor() == piece->getColor())
    {
        return false;
    }
    else
    {
        //capture the opponent's piece by the game rule and return true
        board[sTmp.row][sTmp.column] = piece;
        piece->setRow(sTmp.row);
        piece->setColumn(sTmp.column);
        return true;
    }

    return false;
}

void ChessBoard::printBoard()
{
    char szTemp[16] = {0};
    for(int i =0 ;i <= BLOCK_SIZE*CHESS_SIZE ; i++)
    {
        if(i == (i/BLOCK_SIZE)*BLOCK_SIZE + BLOCK_SIZE/2)
        {
            printf("%d",CHESS_SIZE- i/BLOCK_SIZE);
        }
        else
        {
            printf(" ");
        }
        for(int j = 0;j<= BLOCK_SIZE*CHESS_SIZE;j++)
        {
            sprintf(szTemp,"%d_%d",i,j);
            if(globalMapRealIndexToPos.find(szTemp) != globalMapRealIndexToPos.end())
            {
                string strPos = globalMapRealIndexToPos[szTemp];
                SPieceIndex sPieceIndex = globalMapPosToVirtualIndex[strPos];
                if(board[sPieceIndex.row][sPieceIndex.column] !=  NULL)
                {
                    printf("%s",(board[sPieceIndex.row][sPieceIndex.column])
->getStrCode().c_str());
                }
                else
                {
                    printf(" ");
                }
            }
            else if( i%BLOCK_SIZE == 0&& j%BLOCK_SIZE == 0)
        {
            printf("+");
        }
        else if(i%BLOCK_SIZE == 0 && j%BLOCK_SIZE != 0)
        {
            printf("-");
        }
        else if(j % BLOCK_SIZE == 0 && i % BLOCK_SIZE != 0)
        {
            printf("|");
        }
        else
        {
            printf(" ");
        }
        }
        /*
        if(i == (i/BLOCK_SIZE)*BLOCK_SIZE + BLOCK_SIZE/2)
        {
            printf(" %d",CHESS_SIZE- i/BLOCK_SIZE);
        }
        */
        printf("\n");
    }

    printf(" ");
    for(int j = 0;j<= BLOCK_SIZE*CHESS_SIZE;j++)
    {
        if(j == (j/BLOCK_SIZE)*BLOCK_SIZE + BLOCK_SIZE/2)
        {
            printf("%c", 'a' +    j/BLOCK_SIZE);
        }
        else
        {
            printf(" ");
        }
    }
    printf("\n");
}

bool ChessBoard::move(string fromPosition, string toPosition)
{
    //find the exact piece from its position
    mapPosToVirtualIndex::iterator itr =
globalMapPosToVirtualIndex.find(fromPosition);
    if(itr == globalMapPosToVirtualIndex.end())
    {
                //the range of chess piece's position is between a1 to a8
horizontally and h1 to h8 vertically,
                //mapping in the globalMapPosToVirtualIndex, where will be
illegal if the position is not included.
        printf("illegal position %s\n",fromPosition.c_str());
        return false;
    }

    SPieceIndex sPieceIndex = itr->second;
    ChessPiece* ptr = board[sPieceIndex.row][sPieceIndex.column];
    if( ptr == NULL)
    {
        // can't move if there are no chess pieces in the corresponding positions
        printf("position %s does not have a piece,can not move\n",fromPosition.c_str());
        return false;
    }
    else
    {
        list<string> lstResult;
        ptr->legalMoves(lstResult);
        if(lstResult.size() == 0)
        {
            printf("%s can not move,no appropriate position\n",fromPosition.c_str());
            return false;
        }
        else
        {
            int num = count(lstResult.begin(),lstResult.end(),toPosition);
            if(num == 0)
            {
                                    // can't move if there are no chess pieces in the
```

```
corresponding position.
                printf("%s can not move,no appropriate position\n",fromPosition.c_str());
                lstResult.clear();
                return false;
            }
            else
            {
                SPieceIndex sPieceIndexTmp = globalMapPosToVirtualIndex[toPosition];
                //use the chess piece from fromPosition to replace the piece from toPosotion
                board[sPieceIndexTmp.row][sPieceIndexTmp.column] = ptr;
                //set the coordinate after replacing
                ptr->setRow(sPieceIndexTmp.row);
                ptr->setColumn(sPieceIndexTmp.column);
                //clear the pieces in the fromPosition
                board[sPieceIndex.row][sPieceIndex.column] = NULL;
                lstResult.clear();
                return true;
            }
        }
```

```
    }

    return false;
}


//judge the state of each chess piece
int ChessPiece::judgePieceStat(int row,int column)
{
    char szTemp[16] = {0};
    sprintf(szTemp,"%d_%d",row,column);
    if(globalMapvirtualIndexToPos.find(szTemp)                              ==
globalMapvirtualIndexToPos.end())
    {
            //out of the boarder, and the move is an illegal position
            return -1;
    }

    ChessBoard* curChessBoard = getChessBoard();
    //if the position is null, then no chess piece can move to this position
    if(curChessBoard->board[row][column] == NULL)
    {
```

```
                return 0;
        }


        ChessPiece* ptr = curChessBoard->board[row][column];
        //if the position has a chess piece which has the same color, then can't capture
this piece
        //else, can capture this piece in this position by the opponent's piece
        if(ptr->getColor() == getColor())
        {
                return 1;
        }
        else
        {
                return 2;
        }
            return -1;
}
#endif /* CHESSBOARD_H */

#ifndef CHESSPIECE_H
#define CHESSPIECE_H
```

```
//define the size of each square be 3X3 in the chess board
#define BLOCK_SIZE 3
//define the size of the chess board
#define CHESS_SIZE 8

typedef struct SPieceIndex{
    int row;
    int column;
}SPieceIndex;

typedef map<string,SPieceIndex> mapPosToVirtualIndex ;
typedef map<string,string> mapVirtualIndexToPos;
typedef map<string,string> mapRealIndexToPos;

mapPosToVirtualIndex globalMapPosToVirtualIndex;
mapVirtualIndexToPos globalMapvirtualIndexToPos;
mapRealIndexToPos globalMapRealIndexToPos;

class ChessBoard;

class ChessPiece{
public:
```

```
enum Color {WHITE, BLACK};

//set the attribute and color of the chess board
ChessPiece(ChessBoard* pBoard, Color _color, string _strCode) : board(pBoard),color(_color),strCode(_strCode)  {}

//returns the row
int getRow(){return row;}

//returns the column
int getColumn(){return column;}

//set the row
void setRow(int i) { row = i;}

//set the column
void setColumn(int i) {column = i;}

//get piece color no need for a setColor method because a piece cannot change color.
Color getColor() {return color;}
```

```
    //returns the position of the piece in the format single letter (a..h) followed by a single digit (1..8).
    string getPosition();


    //sets the position of the piece to the appropriate row and column based on the argument
    //which in the format single letter (a..h) followed by a single digit (1..8)
    void setPosition(string position);


     //will be implemented in the concrete subclasses corresponding to each chess piece
     //This method returns a string composed of a single character that corresponds to which piece it is
/*
character       piece
----------     -----------
  "\u2654"     white king
  "\u2655"     white queen
  "\u2656"     white rook
  "\u2657"     white bishop
  "\u2658"     white knight
  "\u2659"     white pawn
  "\u265A"     black king
  "\u265B"     black queen
  "\u265C"     black rook
  "\u265D"     black bishop
  "\u265E"     black knight
  "\u265F"     black pawn
*/
    string toString();

    //This method will be implemented in the concrete subclasses corresponding to each chess piece.
    //This method returns all the legal moves that piece can make based on the rules described above in the assignment
    virtual int legalMoves(list<string>& lstResult)
    {
       lstResult.clear();
       return 0;
    };

    void setStrCode(string str) {strCode = str;}

    string getStrCode() {return strCode;}
```

```
        ChessBoard* getChessBoard() {return board;}

        /*judge the state of the pieces at the given position, returns 0 if there is no piece,
                  returns 1 if there is a piece with the same color, returns 2 if there is an opponent's piece*/
        int judgePieceStat(int row,int column);

protected:
      ChessBoard* board; // the board it belongs to, default null
      int row; // the index of the horizontal rows 0..7
      int column; // the index of the vertical column 0..7
      Color color; // the color of the piece
      string strCode;
};

//debug the program
string ChessPiece::toString()
{
      char szTemp[16] = {0};
      sprintf(szTemp,"%d_%d",row,column);
```

```
    if(globalMapvirtualIndexToPos.find(szTemp)                              ==
globalMapvirtualIndexToPos.end())
    {
            return string("");
    }
    else
    {
            return globalMapvirtualIndexToPos[szTemp];
    }
    return string("");
}

#endif /* CHESSPIECE_H */


#ifndef KING_H
#define KING_H


#include "chessPiece.h"
//King: each side has 1 king
class King :public ChessPiece{
public:
    King(ChessBoard* pBoard,Color _color, string
_strCode):ChessPiece(pBoard,_color,_strCode){}
public:
    virtual int legalMoves(list<string>& lstResult);

};

//**********************************************************************
************
// The king moves only one square in any direction.
*
//**********************************************************************
************

int King::legalMoves(list<string>& lstResult)
{
    int row = getRow();
    int column = getColumn();
    char szTemp[16] = {0};
    lstResult.clear();

    //calculate King's row first
```

```
int iRet = judgePieceStat(row + 1,column);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row + 1,column);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

iRet = judgePieceStat(row - 1,column);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row - 1,column);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//calculate King's column
iRet = judgePieceStat(row,column + 1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row,column + 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}
```

```
iRet = judgePieceStat(row,column - 1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row,column - 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//calculate the King's diagonal direction
//judge the upper right direction, both row and column increase
iRet = judgePieceStat(row + 1,column + 1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row + 1,column + 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//judge the lower right direction, the row decrease and column increase
iRet = judgePieceStat(row - 1,column + 1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row - 1,column + 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
    }


    //judge the upper left direction, the row increase and column decrease
    iRet = judgePieceStat(row + 1,column - 1);
    if(iRet == 0 || iRet == 2)
    {
        sprintf(szTemp,"%d_%d",row + 1,column - 1);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
    }


    //judge the lower left direction, both row and column decrease
    iRet = judgePieceStat(row - 1,column - 1);
    if(iRet == 0 || iRet == 2)
    {
        sprintf(szTemp,"%d_%d",row - 1,column - 1);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
    }
    return 0;
}


#endif /* KING_H */
```

```
#ifndef QUEEN_H
#define QUEEN_H

#include "chessPiece.h"
//Queen: each side has 1 queen
class Queen :public ChessPiece{
public:
    Queen(ChessBoard* pBoard,Color _color,string _strCode):ChessPiece(pBoard,_color,_strCode){}
public:
    int legalMoves(list<string>& lstResult);
};

//***********************************************************************
// A queen combines the power of a rook and bishop and can move any number *
// of squares along a rank, file, or diagonal, but a queen cannot leap     *
// over other pieces.                                                       *
//***********************************************************************
```

```
int Queen::legalMoves(list<string>& lstResult)
{
    int iRet = -1;
    int row = getRow();
    int column = getColumn();
    char szTemp[16] = {0};
    lstResult.clear();

    //calculate queen's row first
    for(int i = row + 1 ; i < CHESS_SIZE; i++ )
    {
        iRet = judgePieceStat(i,column);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,column);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    for(int i = row - 1 ; i >= 0 ; i-- )
    {
        iRet = judgePieceStat(i,column);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,column);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
```

```
}


//calculate queen's column
for(int i = column + 1 ; i < CHESS_SIZE; i++ )
{
    iRet = judgePieceStat(row,i);
    if(iRet == 0 || iRet == 2)
    {
        sprintf(szTemp,"%d_%d",row,i);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        if(iRet == 2)
        {
            break;
        }
    }
    else
    {
        break;
    }
}


for(int i = column - 1 ; i >= 0 ; i-- )
```

```
    {
        iRet = judgePieceStat(row,i);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row,i);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    //calculate the queen's diagonal direction
    //judge the upper right direction, both row and column increase
    for(int i = row + 1 ,j= column + 1;i < CHESS_SIZE && j < CHESS_SIZE;
i++,j++)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,j);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    //judge the upper left direction, the row increase and column decrease
    for(int i = row + 1 ,j= column - 1;i < CHESS_SIZE && j >= 0; i++,j--)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
```

```
        sprintf(szTemp,"%d_%d",i,j);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        if(iRet == 2)
        {
            break;
        }
    }
    else
    {
        break;
    }
}

//judge the lower left direction, both row and column decrease
for(int i = row - 1 ,j= column - 1;i >= 0 && j >= 0; i--,j--)
{
    iRet = judgePieceStat(i,j);
    if(iRet == 0 || iRet == 2)
    {
        sprintf(szTemp,"%d_%d",i,j);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        if(iRet == 2)
```

```
                {
                        break;
                }
        }
        else
        {
                break;
        }
}

//judge the lower right direction, the row decrease and column increase
for(int i = row - 1 ,j= column + 1;i >= 0 && j < CHESS_SIZE; i--,j++)
{
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
                sprintf(szTemp,"%d_%d",i,j);
                lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
                if(iRet == 2)
                {
                        break;
                }
        }
        else
        {
            break;
        }
    }
    return 0;
}

#endif /* QUEEN_H */



#ifndef KNIGHT_H
#define KNIGHT_H

#include "chessPiece.h"
//Knight: each side has 2 knights
class Knight :public ChessPiece{
public:
    Knight(ChessBoard*                pBoard,Color                _color,string
_strCode):ChessPiece(pBoard,_color,_strCode){}
public:
    int legalMoves(list<string>& lstResult);
};

//**********************************************************************
************
// A knight's move forms an "L"-shape: two squares vertically and one          *
// square horizontally, or two squares horizontally and one square             *
//                                                                   vertically.
*
// The knight is the only piece that can leap over other pieces.              *
// it can move two squares vertically and one square horizontally, or two    *
//     squares     horizontally     and     one     square     vertically
*
//**********************************************************************
************

int Knight::legalMoves(list<string>& lstResult)
{
    int row = getRow();
    int column = getColumn();
    char szTemp[16] = {0};
    lstResult.clear();
```

```
//there are 8 ways that a knight can move
 //the first way: move 1 column right and 2 rows upper
 int iRet = judgePieceStat(row + 2,column + 1);
 if(iRet == 0 || iRet == 2)
 {
      sprintf(szTemp,"%d_%d",row + 2,column + 1);
      lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
 }

 //the second way: move 2 columns right and 1 row upper
 iRet = judgePieceStat(row + 1,column + 2);
 if(iRet == 0 || iRet == 2)
 {
      sprintf(szTemp,"%d_%d",row + 1,column + 2);
      lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
 }

 //the third way: move 1 column right and 2 rows downward
 iRet = judgePieceStat(row -2,column + 1);
 if(iRet == 0 || iRet == 2)
 {
    sprintf(szTemp,"%d_%d",row -2,column + 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//the fourth way: move one column right and 1 row downwards
iRet = judgePieceStat(row -1,column +2);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row -1,column +2);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//the fifth way: move 1 column left and 2 rows upper
iRet = judgePieceStat(row + 2,column -1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row + 2,column -1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//the sixth way: move 2 columns left and 1 row upper
iRet = judgePieceStat(row + 1,column -2);
```

```
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row + 1,column -2);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//the seventh way: move 1 column left and 2 rows 7 downwards
iRet = judgePieceStat(row -2,column - 1);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row -2,column - 1);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}

//the eighth way: move 2 columns left and 1 row 8 downwards
iRet = judgePieceStat(row - 1,column - 2);
if(iRet == 0 || iRet == 2)
{
    sprintf(szTemp,"%d_%d",row - 1,column - 2);
    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
}
return 0;
```

```
}


#endif /* KNIGHT_H */




#ifndef BISHOP_H
#define BISHOP_H

#include "chessPiece.h"
//Bishop: each side has 2 bishops
class Bishop :public ChessPiece{
public:
    Bishop(ChessBoard*                    pBoard,Color                    _color,string
_strCode):ChessPiece(pBoard,_color,_strCode){}
public:
    int legalMoves(list<string>& lstResult);
};

//*******************************************************
// A bishop can move any number of squares diagonally in     *
// any direction as long as it does not have to leap over    *
// other pieces                                                    *
//*********************************************************


int Bishop::legalMoves(list<string>& lstResult)
{

    int iRet = -1;
    int row = getRow();
    int column = getColumn();
    char szTemp[16] = {0};
    lstResult.clear();

    //judge the bishop's upper right direction, both the row and column increase
    for(int i = row + 1 ,j= column + 1;i < CHESS_SIZE && j < CHESS_SIZE;
i++,j++)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,j);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    //judge the bishop's upper left direction, the row increase and the column decrease
    for(int i = row + 1 ,j= column - 1;i < CHESS_SIZE && j >= 0; i++,j--)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,j);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    //judge the bishop's lower left direction, both the row and column decrease
    for(int i = row - 1 ,j= column - 1;i >= 0 && j >= 0; i--,j--)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,j);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }

    //judge the bishop's lower right direction, the row decrease and the column increase
    for(int i = row - 1 ,j= column + 1;i >= 0 && j < CHESS_SIZE; i--,j++)
    {
        iRet = judgePieceStat(i,j);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,j);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
```

```
          }
     }
     return 0;
}

#endif /* BISHOP_H */

#ifndef ROOK_H
#define ROOK_H

#include "chessPiece.h"
//Rook: each side has 2 rooks
class Rook :public ChessPiece{
public:
     Rook(ChessBoard* pBoard,Color _color,string
_strCode):ChessPiece(pBoard,_color,_strCode){}
public:
     int legalMoves(list<string>& lstResult);
};

//**************************************************************
// A rook can move any number of squares horizontally and    *
// vertically, forward or backward, but rooks cannot leap    *
// over other pieces                                          *
//*****************************************************


int Rook::legalMoves(list<string>& lstResult)
{
     int iRet = -1;
     int row = getRow();
     int column = getColumn();
     char szTemp[16] = {0};
     lstResult.clear();


     //calculate the row first
   //judge the state and position of the previous rows for rook
     for(int i = row + 1 ; i < CHESS_SIZE; i++ )
     {
          iRet = judgePieceStat(i,column);
          if(iRet == 0 || iRet == 2)
          {
                sprintf(szTemp,"%d_%d",i,column);
                lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
                //the rook can move to the given position and capture the opponent's
```

```
piece, but can't move further
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
    //judge the state and position of the rear rows for rook
    for(int i = row - 1 ; i >= 0 ; i-- )
    {
        iRet = judgePieceStat(i,column);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",i,column);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
```

```
          }
      }
      else
      {
          break;
      }
  }

  //calculate the column for rook
//judge the state and position of the right columns for rook
  for(int i = column + 1 ; i < CHESS_SIZE; i++ )
  {
      iRet = judgePieceStat(row,i);
      if(iRet == 0 || iRet == 2)
      {
          sprintf(szTemp,"%d_%d",row,i);
          lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
          if(iRet == 2)
          {
              break;
          }
      }
```

```
        else
        {
            break;
        }
    }

    //judge the state and position of the left columns for rook
    for(int i = column - 1 ; i >= 0 ; i-- )
    {
        iRet = judgePieceStat(row,i);
        if(iRet == 0 || iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row,i);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 2)
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
    return 0;
}

#endif /* ROOK_H */

#ifndef PAWN_H
#define PAWN_H

#include "chessPiece.h"
//Pawn: each side has 8 pawns
class Pawn :public ChessPiece{
public:
    Pawn(ChessBoard* pBoard,Color _color,string
_strCode):ChessPiece(pBoard,_color,_strCode){}
public:
    int legalMoves(list<string>& lstResult);
};

//***************************************************************
***********
```

```
// A pawn in the initial position may move one or two squares vertically      *
// forward to an empty square, but cannot leap over any piece, and it can    *
// never move backwards. Subsequently it can move only one square
*
// vertically forward to an empty square.
*
// A pawn may also capture (replace) an opponent's piece diagonally one      *
// square in front of it.
*
//**********************************************************
***********

int Pawn::legalMoves(list<string>& lstResult)
{
    int iRet = -1;
    int row = getRow();
    int column = getColumn();
    SPieceIndex sPieceIndex;
    char szTemp[16] = {0};
    lstResult.clear();

    //if it is a white pawn, it moves upwards, which is from the row 2 to 8
```

```
corresponding to the opposite direction(6 to 0) in the printing board.
    if(getColor() == WHITE)
    {
        //judge if the white pawn can move one square
        iRet = judgePieceStat(row - 1,column);
        if(iRet == 0)
        {
            sprintf(szTemp,"%d_%d",row - 1,column);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            if(iRet == 0 && row == 6 /*6 is the initial row in the printing board*/)
            {
                //When the white pawn is in the initial position, if there is no other
piece in front of it, it can move two squares
                iRet = judgePieceStat(row - 2,column);
                if(iRet == 0)
                {
                    sprintf(szTemp,"%d_%d",row - 2,column);
                    lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
                }
            }
        }
        //judge if the pawn can replace the opponent's piece which is located in
the upper right position
        iRet = judgePieceStat(row - 1,column + 1);
        if(iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row - 1,column + 1);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        }

        //judge if the pawn can replace the opponent's piece which is located in the upper left position
        iRet = judgePieceStat(row - 1,column - 1);
        if(iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row - 1,column - 1);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        }
    }

    //else it is a black pawn, it moves downwards, which moves from the row 7 to 1
  //corresponding to the opposite direction(1 to 7) in the printing board.
    else
```

```
{
    iRet = judgePieceStat(row + 1,column);
    if(iRet == 0)
    {
        sprintf(szTemp,"%d_%d",row + 1,column);
        lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        //When the black pawn is in the initial position, if there is no other piece in front of it, it can move two squares
        if(iRet == 0 && row == 1)
        {
            iRet = judgePieceStat(row + 2,column);
            if(iRet == 0)
            {
                sprintf(szTemp,"%d_%d",row + 2,column);
                lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
            }
        }
    }

    //judge if the pawn can replace the opponent's piece which is located in the upper right position
    iRet = judgePieceStat(row + 1,column - 1);
```

```
        if(iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row + 1,column - 1);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        }

        //judge if the pawn can replace the opponent's piece which is located in the upper left position
        iRet = judgePieceStat(row + 1,column + 1);
        if(iRet == 2)
        {
            sprintf(szTemp,"%d_%d",row + 1,column + 1);
            lstResult.push_back(globalMapvirtualIndexToPos[szTemp]);
        }
    }

    return 0;
}

#endif /* PAWN_H */
```