資訊傳播學系學士班B科技組

期末專題報告

象棋程式專題報告

Chess Program Report for Practical Project

指導老師：方文聘教授

作 者：鄭兆晉

蔡宜佳

中華民國一零八年十二月

**目錄**

封面…………………………………………………………………… 0

目錄…………………………………………………………………… 1

問題說明……………………………………………………………… 2

演算法………………………………………………………………… 3

1. 主要流程………………………………………………………3

1.1主程式………………………………………………………3

1.2判斷最佳解………………………………………………… 3

1.3 點對點判斷法……………………………………………… 4

程式碼………………………………………………………………… 5

1.基本判斷………………………………………………………5

1.1canMove………………………………………………………5

1.2判斷最佳解………………………………………………… 5

2.各棋子規則……………………………………………………6

2.1方格內的規則……………………………………………… 6

2.2需判讀移動間的棋子之則………………………………………7

2.3需判讀有無拐角問題之規則……………………………………8

2.4過河問題之規則………………………………………………9

3.其他函數………………………………………………………10

3.1.bool re\_canMove……………………………………………10

3.2.bool canDefense\_king……………………………………… 10

3.3.bool canDefense\_other………………………………………11

3.4.bool kill\_itself ………………………………………… 11

討論……………………………………………………………………12

附錄……………………………………………………………………13

**問題說明**

棋盤程式要如何運用已生成的table.txt來得知目前棋盤狀況，並且從比賽判斷程式獲得先後手資訊，藉此計算合理步法，以及最佳步法，將棋寫入play.txt，以便比賽判斷程式得以判讀勝負。

至於如何找尋到最佳步法，就必須運用到，這學期所學得各式邏輯與語法，好比各個棋子所對應的規則該如何呈現，又或者如何用函數去取代冗長的數學算式，這些都是我們該去深究的問題。

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片13.PNG |
| 圖1.1:檔案轉換流程 |

**演算法**

由比賽判斷程式可得先後手以及table.txt，藉此得知當前比賽狀況，將己方各個位置資訊傳給判斷最佳解(圖2.2)，即可取得最終步法，並將棋寫入play.txt。(圖2.1)

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片1.PNG |
| 圖2.1:主程式 |

主程式將各個位置資訊傳入判讀，主要判讀順序可分為，敵方將領可吃、我方將領即將被吃、敵方有實棋可吃、以及己方實棋即將被吃、隨機步法，其中每項判讀所運用到的規則都是由bool canMove(圖3.1.2)來判讀是否是合理步法的。(圖2.2)

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片2.PNG |
| 圖2.2:判斷最佳解 |

主要的運算方式都是將移動前與移動後的數值關係進行比較，假如兩者關係符合數學式，即可知道該步法為合理步法。

移動前與移動後分別代表piece A以及 piece B，運用雙重迴圈將棋盤從board[0][0]跑至board[10][9],因此點對點的每種可能性將可判斷，再將兩者資料引入至接下來要說明的函數內，藉此判斷出兩者關係是否成立，若成立則將移動後位置與其id存入pieces方法之中，以便之後可以計算出最佳解。

|  |
| --- |
| C:\Users\mogaco\Desktop\期末圖\789.JPG |
| 圖2.3:棋盤版面 |

|  |
| --- |
| C:\Users\mogaco\Desktop\489484948.PNG |
| 圖2.4:點對點判斷 程式碼 |

**程式碼**

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\1111\main\投影片3.PNG | C:\Users\mogaco\Desktop\1232.PNG |
| 圖3.1.1: canMove流程 | 圖3.1.2:canMove程式碼 |

判斷最佳解會將當前移動棋子資訊(piece A)以及移動位置資訊(piece B)傳入此函數，首先判斷移動是否為合理值(空格或敵方實棋)，再者判斷當前移動棋子id為何以便對應其規則，再者運用各id之規則函數來判讀是否正確。

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\1111\main\投影片4.PNG | C:\Users\mogaco\Desktop\5465.PNG |
| 圖3.2.1: num\_between流程 | 圖3.2.2: num\_between程式碼 |

int num\_between 是將兩者座標位置進行計算，先判斷是否為同一直排或橫排上，再將之間的實棋數用for迴圈進行計算以便得出sum值並回傳，此設計是以便將、車、炮上的判斷。

**方格內的規則**

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片5.PNG |
| 圖3.3: 將規則流程 |

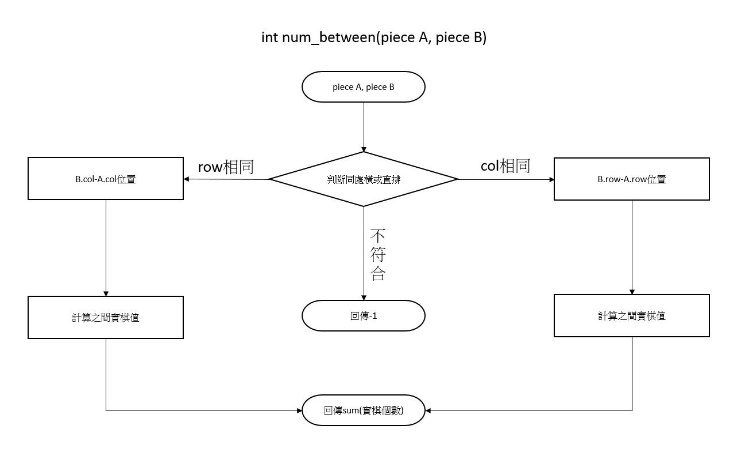
將的規則

先判讀移動位子是否在己方九宮格內，再者判斷前往的位置是否為敵方實棋或空格。

上下左右移動均為1，因此當橫排或直排取絕對值為零和一即可成立。

若王見王也可以直接飛去吃這就運用上所述的num\_between(圖3.2.2)由於王見王直接判輸贏因此不使用。

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片6.PNG |
| 圖3.4: 士規則流程 |



士的規則

先判讀移動位子是否在己方九宮格內，再者判斷前往的位置是否為敵方實棋或空格。

斜向移動均為1，因此當橫排與直排取絕對值均為一即可成立。

**需判讀其移動間的棋子數之規則**

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片9.PNG |
| 圖3.5: 車規則流程 |

車的規則

將當前位置與移動位置引入函數int num\_between(圖3.2.2)，計算其值。

若移動位置與原先位置之間無實棋(num\_between回傳0)，則步法成立。

|  |
| --- |
| C:\Users\mogaco\Desktop\1111\main\投影片10.PNGPU |
| 圖3.6: 炮規則流程 |

炮的規則

將當前位置與移動位置引入function num\_between(圖3.2.2)，計算其值。

若移動位置與原先位置之間無實棋(num\_between回傳0)，則移動位置須為空格。

若移動位置與原先位置之間無實棋為一(num\_between回傳1)，則移動位置須為敵方實棋。

**需判讀有無拐角問題之規則**

馬的規則

將當前位置之上下左右判斷是否有實棋存在，若無允許其對應方向的步法。

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\1111\main\投影片7.PNG | C:\Users\mogaco\Desktop\期末圖\MU1.JPG |
| 圖3.7.1: 馬規則流程 | 圖3.7.2:拐馬腳 |

象的規則

判斷移動為先手還是後手方，並且無法使象過河，將當前位置之斜方判斷是否有實棋存在，若無允許其對應方向的步法。

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\1111\main\投影片8.PNG | C:\Users\mogaco\Desktop\期末圖\SU.JPG |
| 圖3.8.1: 象規則流程 | 圖3.8.2:拐象腳 |

**過河問題之規則**

兵的規則

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\1111\main\投影片11.PNG | C:\Users\mogaco\Desktop\期末圖\BI1.JPG |
| 圖3.9.1: 兵規則流程 | 圖3.9.2:過核問題 |

有了上述的規則即可計算出所有當回合可移動的合理步法，再者就是計算哪一步才是最佳解，因此我們將會運用在bool re\_canMove、bool canDefense\_king、bool canDefense\_other以及bool kill\_itself這四種函數，藉此計算判斷最佳解(圖2.2)。

|  |  |
| --- | --- |
| C:\Users\mogaco\Desktop\期末圖\re_m.JPG | C:\Users\mogaco\Desktop\489485.PNG |
| 圖4.1.1: re\_canMove流程 | 圖4.1.2:re\_canMove程式碼 |

re\_canMove(圖4.1.2)與先前的canMove(圖3.1.2)觀念一樣，只是少去了判斷敵我關係，即使是己方的也可以吃，這將計算到互換關係，以便裡用於之後的函數判斷

|  |
| --- |
| C:\Users\mogaco\Desktop\main.png |
| 圖4.2.1:canDefense\_king流程圖 |

**canDefense\_king**

此函數是為了保護將而產生的，先將每步方法是一遍，若可以使敵方棋子進攻失敗則成立，否則回傳失敗。

|  |
| --- |
| C:\Users\mogaco\Desktop\489749848.PNG |
| 圖4.2.2:canDefense\_king程式碼 |

i,j為守護棋的移動方法

A=killer即將殺掉將領的敵棋

B=將領資訊

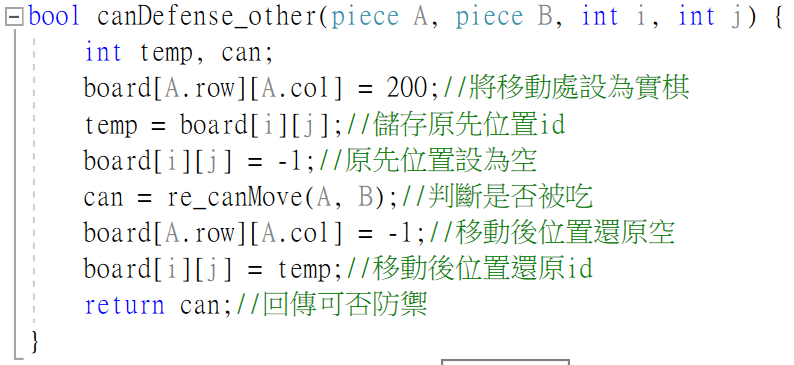
因此將守護棋移動後位置設為實棋，即可判斷是否守護成立(re\_canMove回傳T/F)。

|  |
| --- |
| C:\Users\mogaco\Desktop\main.png |
| 圖4.3.1:canDefense\_other流程圖 |

**canDefense\_other**

此函數是為了製造一換一而產生的，先將每步方法是一遍，若可以使敵方棋子進攻後被反吃則成立，否則回傳失敗。

|  |
| --- |
|  |
| 圖4.3.2:canDefense\_other程式碼 |



i,j =即將被吃掉之棋子

A=為守護棋的移動後的資訊

B=即將被吃掉之棋子

將守護棋移動後，再度判斷是否可以前往被吃棋子位置移動，若可即成立。

|  |
| --- |
| C:\Users\mogaco\Desktop\7879798781.PNGC:\Users\mogaco\Desktop\1948964.PNGC:\Users\mogaco\Desktop\58946.PNG |
| 圖4.4:kill\_itself預防自殺函數 |

**預防自殺函數**

顧名思義，防止自己把將領獻給敵人，每決定一步棋時，都需判斷是否會自殺，因此將每步棋設定好，並用敵人視角計算一遍，藉此確定這步棋不會自掘墳墓。

|  |
| --- |
| **討論**  **分析與證明程式之功能**  失敗一:己方棋子不斷去送死。  調整一:運用re\_canMove(圖4.1.2)計算出敵方步法(可吃自己棋)，以便推  算出移動後是否會送死或反殺。  失敗二:己方棋子都不會進攻。  調整二:原先運用觀察下步棋來推算是否可二步達成勝利，但並非最佳解。  失敗三:己方棋子只會反覆移動。  調整三:在調整二成立的情況下，有時會處於無限迴圈，因而將調整二移  除，改用隨機方式去做進攻。  失敗四:己方棋子都不會進攻，將領會自己去送死。  調整四:在調整三的運作下，時常移動的位置會把將領獻給敵隊，因而有些  缺陷在，衍生出了kill\_itself預防自殺函數(圖4.4)，再而每下  一步棋前，先確認是否自殺，使遊戲結束。  **缺陷還有與現有方法比較**  缺陷一:在於只會一味的防守，有時勝利只差兩步，卻依然保持防守狀態。  缺陷二:隨機情況下無法求得最佳進攻解。  調整二與調整三之間的比較論進攻性調整二為佳，但穩定性不高有時會產生無限循環，因此最後還是採用隨機方式去修改。  **特色**  將最差的結果排除並以隨機方式進攻，雖無法主動進攻，但絕對不會自掘墳墓，因此只要對方有失誤即可勝利。 |

**附錄**

列表1.1:完整程式碼

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <string>  #include <time.h>  using namespace std;  //棋子資料表格  struct piece {  int id;  int row;  int col;  };  void input(); //讀取棋盤  void output(int player, piece final\_move); //寫入最佳解  piece fill\_pieces(int row, int col); //填入棋子資料  piece fill\_other\_way(int id, int row, int col); //填入其餘資料  int board[10][9]; //棋盤  int player; //先後手  int best\_choose = -1; //判斷最佳旗子可吃  int choose = 200; //最佳判斷  int class\_system[16] = { 7,1,1,2,2,4,4,6,6,5,5,3,3,3,3,3 }; //旗子的階級分配  int num\_between(piece A, piece B); //計算兩期之間有多少實棋  bool be\_killed; //是否被敵方所吃  bool canDefense\_king(piece A, piece B, int i, int j); //自我犧牲保護  bool canDefense\_other(piece A, piece B, int i, int j); //封鎖對方  bool canMove(piece A, piece B); //判斷規則  bool re\_canMove(piece A, piece B); //判斷敵方規則(我方傷害)  bool kill\_itself(int id, int row, int col); //防止自殺  bool teammate(int moveid, int killid); //是否同陣營  bool JU(piece A, piece B); //將的規則  bool SI(piece A, piece B); //士的規則  bool SHU(piece A, piece B); //象的規則  bool MU(piece A, piece B); //馬的規則  bool GI(piece A, piece B); //車的規則  bool PU(piece A, piece B); //炮的規則  bool BI(piece A, piece B); //兵的規則  piece final\_move; //最終解  piece move\_piece; //原先資料存取  piece kill\_piece; //移動資料存取  struct piece pieces[16][17] = { 0 }; //我方方法  struct piece re\_pieces[16][17] = { 0 }; //敵方方法  struct piece random\_way[120] = { 0 }; //隨機存取方法  struct piece way[120] = { 0 }; //移動後所存取之方法  int main(int n, char\*\* argv) {  //--------引入先後手-------------------------  if (n >= 2) {  player = atoi(argv[1]);  }  //--------------讀取棋盤-----------------------  input();  //---------------判斷-----------------------  int killer = -1;  int m = 0;  int f = 0;  //---------------我方方法--------------------  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  move\_piece = fill\_pieces(i, j);  if (!(board[i][j] == -1) && board[i][j] / 100 == player) {  f = 0;  for (int r = 0; r < 10; r++) {  for (int c = 0; c < 9; c++) {  kill\_piece = fill\_pieces(r, c);  if (canMove(move\_piece, kill\_piece)) {  pieces[m][f] = fill\_other\_way(board[i][j], r, c);  f++;  }  }  }  m++;  }  }  }  m = 0;  //-----------------敵方方法------------  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  move\_piece = fill\_pieces(i, j);  if (!(board[i][j] == -1) && board[i][j] / 100 == abs(player - 1)) {  f = 0;  for (int r = 0; r < 10; r++) {  for (int c = 0; c < 9; c++) {  kill\_piece = fill\_pieces(r, c);  if (re\_canMove(move\_piece, kill\_piece)) {  re\_pieces[m][f] = fill\_other\_way(board[i][j], r, c);  f++;  }  }  }  m++;  }  }  }  //----------有將可以吃就吃------  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 17; j++) {  if (board[pieces[i][j].row][pieces[i][j].col] % 100 == 0) {  best\_choose = 100;  final\_move = pieces[i][j];  }  }  }  //--------將的防守----------------  if (best\_choose == -1) {  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  if ((board[i][j] % 100 == 0) && board[i][j] / 100 == player) {  be\_killed = false;  for (int k = 0; k < 16; k++) {  for (int l = 0; l < 17; l++) {  if (!(re\_pieces[k][l].id == 0 && re\_pieces[k][l].row == 0 && re\_pieces[k][l].col == 0)) {  if ((re\_pieces[k][l].row == i) && (re\_pieces[k][l].col == j)) {  be\_killed = true;  killer = re\_pieces[k][l].id;  }  }  //判斷是否可以直接吃掉對方  if (be\_killed == true) {  for (int m = 0; m < 16; m++) {  for (int o = 0; o < 17; o++) {  if (!(pieces[m][o].id == 0 && pieces[m][o].row == 0 && pieces[m][o].col == 0)) {  if (board[pieces[m][o].row][pieces[m][o].col] == killer) {  if (kill\_itself(pieces[m][o].id, pieces[m][o].row, pieces[m][o].col) == false) {  best\_choose = 100;  final\_move = fill\_other\_way(pieces[m][o].id, pieces[m][o].row, pieces[m][o].col);  }  }  }  }  }  //尋找他棋來守護將  if (best\_choose == -1) {  for (int p = 0; p < 16; p++) {  for (int q = 0; q < 17; q++) {  if (!(pieces[p][q].id == 0 && pieces[p][q].row == 0 && pieces[p][q].col == 0)) {  kill\_piece = fill\_pieces(i, j);  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  if (board[i][j] == killer)  move\_piece = fill\_pieces(i, j);  }  }  if (canDefense\_king(move\_piece, kill\_piece, pieces[p][q].row, pieces[p][q].col) == false && choose > class\_system[pieces[p][q].id % 100]&& kill\_itself(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col) == false) {  best\_choose = 100;  choose = class\_system[pieces[p][q].id % 100];  final\_move = fill\_other\_way(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col);  }  }  }  }  }  //將領自己逃跑  if (best\_choose == -1) {  for (int p = 0; p < 16; p++) {  if(pieces[p][0].id%100==0){  for(int q=0;q<17;q++){  if (!(pieces[p][q].row == 0 && pieces[p][q].col == 0)&& kill\_itself(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col) == false) {  best\_choose = 100;  final\_move = fill\_other\_way(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col);  }  }  }  }  }  }  }  }  }  }  }  }  //-------------------有敵棋分數最高並且被吃掉的-------------------------  if (best\_choose == -1) {  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 17; j++) {  if (!(pieces[i][j].id == 0 && pieces[i][j].row == 0 && pieces[i][j].col == 0)) {  if ((class\_system[board[pieces[i][j].row][pieces[i][j].col]] >= best\_choose) && !(board[pieces[i][j].row][pieces[i][j].col] == -1)) {  be\_killed = false;  for (int k = 0; k < 16; k++) {  for (int l = 0; l < 17; l++) {  if (!(re\_pieces[k][l].id == 0 && re\_pieces[k][l].row == 0 && re\_pieces[k][l].col == 0)) {  if ((re\_pieces[k][l].row == pieces[i][j].row) && (re\_pieces[k][l].col == pieces[i][j].col)) {  be\_killed = true;  break;  }  }  }  }  if (be\_killed == false && kill\_itself(pieces[i][j].id, pieces[i][j].row, pieces[i][j].col) == false) {  best\_choose = class\_system[board[pieces[i][j].row][pieces[i][j].col]];  final\_move = fill\_other\_way(pieces[i][j].id, pieces[i][j].row, pieces[i][j].col);  }  }  }  }  }  }  //判斷自己是否被吃  if (best\_choose == -1) {  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  if (board[i][j] / 100 == player) {\  be\_killed = false;  for (int k = 0; k < 16; k++) {  for (int l = 0; l < 17; l++) {  if (!(re\_pieces[k][l].id == 0 && re\_pieces[k][l].row == 0 && re\_pieces[k][l].col == 0)) {  if ((re\_pieces[k][l].row == i) && (re\_pieces[k][l].col == j)) {\  be\_killed = true;  killer = re\_pieces[k][l].id;  }  }  //尋找有無棋子可以封鎖敵方進攻若有不動  if (be\_killed == true) {  for (int p = 0; p < 16; p++) {  for (int q = 0; q < 17; q++) {  kill\_piece = fill\_pieces(i, j);  if (re\_canMove(pieces[p][q], kill\_piece)) {  be\_killed = false;  }  }  }  }  //是否可以直接吃掉敵人  if (be\_killed == true) {  for (int m = 0; m < 16; m++) {  for (int o = 0; o < 17; o++) {  if (!(pieces[m][o].id == 0 && pieces[m][o].row == 0 && pieces[m][o].col == 0)) {  if (board[pieces[m][o].row][pieces[m][o].col] == killer && kill\_itself(pieces[m][o].id, pieces[m][o].row, pieces[m][o].col) == false) {  best\_choose = 100;  final\_move = fill\_other\_way(pieces[m][o].id, pieces[m][o].row, pieces[m][o].col);  }  }  }  }  //尋找可以封鎖敵人的棋步  if (best\_choose == -1) {  for (int p = 0; p < 16; p++) {  for (int q = 0; q < 17; q++) {  if (!(pieces[p][q].id == 0 && pieces[p][q].row == 0 && pieces[p][q].col == 0)) {  kill\_piece = fill\_pieces(i, j);  move\_piece = fill\_other\_way(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col);  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  if (board[i][j] == pieces[p][q].id) {  if (canDefense\_other(move\_piece, kill\_piece, i, j) == true && kill\_itself(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col) == false) {  best\_choose = 100;  final\_move = fill\_other\_way(pieces[p][q].id, pieces[p][q].row, pieces[p][q].col);  }  }  }  }  }  }  }  }  }  }  }  }  }  }  }  //--------------隨機步法--------------------------  if (best\_choose == -1) {  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 17; j++) {  for (int x = 0; x < 16; x++) {  for (int y = 0; y < 17; y++) {  if (pieces[i][j].row == re\_pieces[x][y].row&&pieces[i][j].col == re\_pieces[x][y].col) {  pieces[i][j] = fill\_other\_way(0, 0, 0);  }  }  }  }  }  m = 0;  //確定最終解不會自殺  for (int i = 0; i < 16; i++) {  for (int j = 0; j < 17; j++) {  if (!(pieces[i][j].id == 0 && pieces[i][j].row == 0 && pieces[i][j].col == 0) && kill\_itself(pieces[i][j].id, pieces[i][j].row, pieces[i][j].col) == false) {  random\_way[m] = pieces[i][j];  m++;  }  }  }  //儲存最終解  do {  srand(time(NULL));  final\_move = random\_way[rand() % m];  } while (final\_move.id == 0 && final\_move.row == 0);  }  //-----------------輸出檔案----------------------------  output(player, final\_move);  return 1;  }  void input() {  ifstream infile("table.txt");//先讀取棋盤檔案  string value;  for (int i = 0; i < 10; i++) {//讀取每行  for (int j = 0; j < 8; j++) {//讀取每列  getline(infile, value, ' ');//每遇到空格停止  board[i][j] = atoi(value.c\_str());//將掃到的值存入board內  }  getline(infile, value, '\n');//每遇到換行停止  board[i][8] = atoi(value.c\_str());//將掃到的值存入board內  }  }  void output(int player, piece final\_move) {  ofstream outFile;  outFile.open("play.txt");//開啟檔案play.txt  outFile << player << " " << final\_move.id << " " << final\_move.col << " " << final\_move.row << endl;  //將(先後手)(棋子id)(棋子移動座標)寫入play.txt  outFile.close();//關閉檔案play.txt  }  int num\_between(piece A, piece B) {  int sum = 0;//總計值  if ((B.row - A.row) == 0) {//當兩座標處同行時  if (B.col - A.col > 0) {//當移動處比原先座標大時  for (int j = A.col + 1; j < B.col; j++) {//掃過之間每個棋子  if (!(board[A.row][j] == -1)) {//判斷中間有幾顆實棋  sum++;//當不是空的 總計加一  }  }  }  else if (B.col - A.col < 0) {//當移動處比原先座標小時  for (int j = A.col - 1; j > B.col; j--) {//掃過之間每個棋子  if (!(board[A.row][j] == -1)) {//判斷中間有幾顆實棋  sum++;//當不是空的 總計加一  }  }  }  return sum;//回傳總計值  }  if ((B.col - A.col) == 0) {//當兩座標處同列時  if (B.row - A.row > 0) {//當移動處比原先座標大時  for (int i = A.row + 1; i < B.row; i++) {//掃過之間每個棋子  if (!(board[i][A.col] == -1)) {//判斷中間有幾顆實棋  sum++;//當不是空的 總計加一  }  }  }  else if (B.row - A.row < 0) {//當移動處比原先座標小時  for (int i = A.row - 1; i > B.row; i--) {//掃過之間每個棋子  if (!(board[i][A.col] == -1)) {//判斷中間有幾顆實棋  sum++;//當不是空的 總計加一  }  }  }  return sum;//回傳總計值  }  return -1;//若不處於同行同列回傳-1  }  bool canDefense\_king(piece A, piece B, int i, int j) {  int temp, can;  temp = board[i][j];//存取移動位置id  board[i][j] = 200;//將移動處設為實棋  can = re\_canMove(A, B);//判斷敵方可否移動  board[i][j] = temp;//將移動處設為原值  return can;//回傳是否防禦  }  bool canDefense\_other(piece A, piece B, int i, int j) {  int temp, can;  board[A.row][A.col] = 200;//將移動處設為實棋  temp = board[i][j];//儲存原先位置id  board[i][j] = -1;//原先位置設為空  can = re\_canMove(A, B);//判斷是否被吃  board[A.row][A.col] = -1;//移動後位置還原空  board[i][j] = temp;//移動後位置還原id  return can;//回傳可否防禦  }  bool canMove(piece A, piece B) {  if (B.id == -1 || !teammate(A.id, B.id)) {//判斷是否為非我方棋子  switch (A.id % 100) {//將id化簡為0-15  case 0:  return JU(A, B);//將規則  break;  case 1:  case 2:  return SI(A, B);//士規則  break;  case 3:  case 4:  return SHU(A, B);//象規則  break;  case 5:  case 6:  return GI(A, B);//車規則  break;  case 7:  case 8:  return MU(A, B);//馬規則  break;  case 9:  case 10:  return PU(A, B);//砲規則  break;  case 11:  case 12:  case 13:  case 14:  case 15:  return BI(A, B);//兵規則  break;  }  }  return false;//若非回傳否  }  bool re\_canMove(piece A, piece B) {  switch (A.id % 100) {//將id化簡為0-15  case 0:  return JU(A, B);//將規則  break;  case 1:  case 2:  return SI(A, B);//士規則  break;  case 3:  case 4:  return SHU(A, B);//象規則  break;  case 5:  case 6:  return GI(A, B);//車規則  break;  case 7:  case 8:  return MU(A, B);//馬規則  break;  case 9:  case 10:  return PU(A, B);//砲規則  break;  case 11:  case 12:  case 13:  case 14:  case 15:  return BI(A, B);//兵規則  break;  }  return false;//若非回傳否  }  bool teammate(int moveid, int killid) {  if (moveid / 100 == killid / 100) {//判斷兩棋是否為同國  return true;//是回傳是  }  return false;//非回傳非  }  bool JU(piece A, piece B) {  //王見王  if (B.id % 100 == 0 && A.col == B.col) {  if (num\_between(A, B) == 0)  return true;  }  //先手方  if (B.row >= 0 && B.row <= 2 && B.col >= 3 && B.col <= 5) {  if ((abs(B.row - A.row) == 1 && abs(B.col - A.col) == 0) || (abs(B.row - A.row) == 0 && abs(B.col - A.col) == 1)) {  return true;  }  }  //後手方  else if (B.row >= 7 && B.row <= 9 && B.col >= 3 && B.col <= 5) {  if ((abs(B.row - A.row) == 1 && abs(B.col - A.col) == 0) || (abs(B.row - A.row) == 0 && abs(B.col - A.col) == 1)) {  return true;  }  }  return false;  }  bool SI(piece A, piece B) {  if (B.row >= 0 && B.row <= 3 && B.col >= 3 && B.col <= 5) {  if ((abs(B.row - A.row) == 1 && abs(B.col - A.col) == 1) || (abs(B.row - A.row) == 1 && abs(B.col - A.col) == 1)) {  return true;  }  }  else if (B.row >= 7 && B.row <= 9 && B.col >= 3 && B.col <= 5) {  if ((abs(B.row - A.row) == 1 && abs(B.col - A.col) == 1) || (abs(B.row - A.row) == 1 && abs(B.col - A.col) == 1)) {  return true;  }  }  return false;  }  bool SHU(piece A, piece B) {  //先手方  if (A.row >= 0 && A.row <= 4 && A.col >= 0 && A.col <= 8) {  //移動區不超越先制範圍  if (B.row >= 0 && B.row <= 4 && B.col >= 0 && B.col <= 8) {  //若右下一無實棋開放右下方移動位置  if (board[A.row + 1][A.col + 1] == -1) {  if (B.row - A.row == 2 && B.col - A.col == 2) {  return true;  }  }  //若左下一無實棋開放左下方移動位置  if (board[A.row + 1][A.col - 1] == -1) {  if (B.row - A.row == 2 && B.col - A.col == -2) {  return true;  }  }  //若右上一無實棋開放右上方移動位置  if (board[A.row - 1][A.col + 1] == -1) {  if (B.row - A.row == -2 && B.col - A.col == 2) {  return true;  }  }  //若左上一無實棋開放左上方移動位置  if (board[A.row - 1][A.col - 1] == -1) {  if (B.row - A.row == -2 && B.col - A.col == -2) {  return true;  }  }  }  }  //後手方  else if (A.row >= 5 && A.row <= 9 && A.col >= 0 && A.col <= 8) {  //移動區不超越先制範圍  if ((B.row >= 5 && B.row <= 9 && B.col >= 0 && B.col <= 8)) {  //若右下一無實棋開放右下方移動位置  if (board[A.row + 1][A.col + 1] == -1) {  if (B.row - A.row == 2 && B.col - A.col == 2) {  return true;  }  }  //若左下一無實棋開放左下方移動位置  if (board[A.row + 1][A.col - 1] == -1) {  if (B.row - A.row == 2 && B.col - A.col == -2) {  return true;  }  }  //若右上一無實棋開放右上方移動位置  if (board[A.row - 1][A.col + 1] == -1) {  if (B.row - A.row == -2 && B.col - A.col == 2) {  return true;  }  }  //若左上一無實棋開放左上方移動位置  if (board[A.row - 1][A.col - 1] == -1) {  if (B.row - A.row == -2 && B.col - A.col == -2) {  return true;  }  }  }  }  return false;  }  bool MU(piece A, piece B) {  //若下一無實棋開放下方移動位置  if (board[A.row + 1][A.col] == -1) {  if (B.row - A.row == 2 && abs(B.col - A.col) == 1) {  return true;  }  }  //若上一無實棋開放上方移動位置  if (board[A.row - 1][A.col] == -1) {  if (B.row - A.row == -2 && abs(B.col - A.col) == 1) {  return true;  }  }  //若右一無實棋開放右方移動位置  if (board[A.row][A.col + 1] == -1) {  if (abs(B.row - A.row) == 1 && B.col - A.col == 2) {  return true;  }  }  //若左一無實棋開放左方移動位置  if (board[A.row][A.col - 1] == -1) {  if (abs(B.row - A.row) == 1 && B.col - A.col == -2) {  return true;  }  }  return false;  }  bool GI(piece A, piece B) {  //計算中間是否有實棋  if (num\_between(A, B) == 0) {  return true;  }  return false;  }  bool PU(piece A, piece B) {  //計算是要飛還是要平移  if ((num\_between(A, B) == 0 && B.id == -1) || (num\_between(A, B) == 1) && !(B.id == -1)) {  return true;  }  return false;  }  bool BI(piece A, piece B) {  //先手方  if (A.id / 100 == 0) {  //過河前  if (A.row >= 0 && A.row <= 4 && A.col >= 0 && A.col <= 8) {  if ((B.row - A.row) == 1 && (B.col - A.col) == 0) {  return true;  }  }  //過河後  if (A.row >= 5 && A.row <= 9 && A.col >= 0 && A.col <= 8) {  if (((B.row - A.row) == 1) && (abs(B.col - A.col) == 0) || (abs(B.col - A.col) == 1) && ((B.row - A.row) == 0)) {  return true;  }  }  return false;  }  //後手方  if (A.id / 100 == 1) {  //過河後  if (A.row >= 0 && A.row <= 4 && A.col >= 0 && A.col <= 8) {  if (((B.row - A.row) == -1) && (abs(B.col - A.col) == 0) || (abs(B.col - A.col) == 1) && ((B.row - A.row) == 0)) {  return true;  }  }  //過河前  if (A.row >= 5 && A.row <= 9 && A.col >= 0 && A.col <= 8) {  if ((B.row - A.row) == -1 && (B.col - A.col) == 0) {  return true;  }  }  return false;  }  else return false;  }  bool kill\_itself(int id, int row, int col) {  int x, y,f,temp;  //尋找原先移動位置  for ( int i = 0; i < 10; i++) {  for ( int j = 0; j < 9; j++) {  if (id == board[i][j]) {  x = i;  y = j;  }  if (id == board[i][j]) {  x = i;  y = j;  }  }  }  //存取移動前後id,並將棋盤設為移動後  temp = board[row][col];  board[row][col] = id;  board[x][y] = -1;  f = 0;  //計算移動後棋盤敵人所可走位置  for (int i = 0; i < 10; i++) {  for (int j = 0; j < 9; j++) {  move\_piece = fill\_pieces(i, j);  if (!(board[i][j] == -1) && board[i][j] / 100 == abs(player - 1)) {  for (int r = 0; r < 10; r++) {  for (int c = 0; c < 9; c++) {  kill\_piece = fill\_pieces(r, c);  if (re\_canMove(move\_piece, kill\_piece)) {  way[f] = fill\_other\_way(board[i][j], r, c);  f++;  }  }  }  }  }  }  //尋找有無敵方步法可將我方將領吃掉  for (int i = 0; i < f; i++) {  if (player==0&&board[way[i].row][way[i].col] == 0) {  board[row][col] = temp;  board[x][y] = id;  return true;  }  if (player == 1 && board[way[i].row][way[i].col] == 100) {  board[row][col] = temp;  board[x][y] = id;  return true;  }  }  //回復原狀並回傳false  board[row][col] = temp;  board[x][y] = id;  return false;  }  piece fill\_pieces(int row, int col) {//指定當前座標填入資訊  piece pieces;  pieces.id = board[row][col];  pieces.row = row;  pieces.col = col;  return pieces;  }  piece fill\_other\_way(int id, int row, int col) {//填入特定資訊  piece way;  way.id = id;  way.row = row;  way.col = col;  return way;  } |