**Chapter 6**

We will develop code to display the moves of the game in this chapter. The game score will be saved to a file to allow the user the option of repeating the moves or setting up a position to determine why the chess program played as it did. At the suggestion of Visual Studio, I have used wide characters, WCHAR, to represent most of the text in Chess Code. However, when the game score is saved to a file, it is automatically converted to the familiar ASCII character set.

The structure of the game score requires an explanation. After attempting to record the moves, one by one, first white’s, then black’s, followed by a new line <lf> , ‘\n’, or endl, I realized that the black moves were not lined-up on the display, as white’s were. I considered changing fonts, but decided to use an idea seen in WChess by Borland. The white moves are kept in one string, and the black moves are kept in another. When the display fills with moves and it is time to scroll the display, both strings can be scrolled at the same time, just before displaying white’s next move.

The structure of the file which contains the game score is simpler. The ChessCode program will output information about the game in a game header. The header will contain the date, the computer’s color, and a place for the result. Then the moves are output as they occur, first white’s, then black’s followed by a new line. The display of the file and the alignment of black’s moves depend on the software and font used to view the file. I suggest using a simple text editor such as Notepad to obtain good results. The game will be displayed as if neatly recorded on a tournament style score sheet.

To start, update CCVariables.cpp and CCExterns.h with all the new variables we’ll be using to implement the score keeping. You can just add the lines below to the end of each file, removing the initial values and declaring them ‘extern’ in CCExterns.h. You’ll use the last two variables later when you add code to avoid draws by repetition.

// variables for display of score sheet

WCHAR gbuf[280]; // wide character global string buffer

WCHAR szWbuf[606], szBbuf[486]; // scoresheets, wht & blk

char Charbuf[21]; // need to retain " 1. . . . " until EndSetup()

int Wlen\_szbuf = 0, Blen\_szbuf = 0; // length of text in buffer

int Wlinecnt = 0, Blinecnt = 0; // line counts for wht scoresheet and blk scoresheet

short SCORELN; // score length: 14 lines for small graphics, 30 for large

int mate = 0; // move out of GUI, mate in n plies, n = 0-1f

// variables for streaming games to a file

bool reIdG = true;

long gamstpos; // file position at the start of each game in scoresheet

long position; // position in output file

long asterpos; // place holder for game result in scoresheet

long colorpos; // position in game label of color of player

long UndoPos[512]; // index by mvnum, 0 - 511 (up to 256 moves)

BOOL gamekeep = FALSE;

BOOL Log = TRUE;

int gamenumb = 1;

char result; // D = drawn, W = white won, B= black won

// variables for repeats

int Reps; // number of repetitions by machine

int OpReps; // number of repetitions by opponent

You will want to show when a checkmate or stalemate is discovered. This typically occurs after all legal moves have been generated and no legal moves are found. At that point, mate is announced and the previous move in the score sheet is modified to show the result. Find the line below in the RunChessCode() function in ChessCode.cpp.

if (n\_list\_mvs == 0)

{

Add:

ModifyPrevMv(com^1); // this is to fix the scoresheet when a mate is discovered.

then find:

if (CheckStatus[dph] == 0xffffffffffffffff)

and replace it with:

if (result == 'D') // ModifyPrevMv() found the 'result'

To accommodate this change, you will need to add the line below near the top of the file ChessCode.cpp with the other external function declarations.

extern void ModifyPrevMv(char som);

You will continue to modify ChessCode.cpp. You will be adding some code for saving the game score to a file. Begin by adding the following lines near the top of the file with the other #include statements.

#include <fstream>

using namespace std;

You’ll also need to add the 3 lines below to the list of declared external functions, also near the top of ChessCode.cpp. ShoMov() is responsible for formatting and displaying the moves on the display.

extern void ShoMov(int, UCHAR, Mov\_t, char);

extern void IdGame(void);

extern void ModifyPrevMv(char);

In the beginning of RunChessCode(), just after the conditional “if (gameover == true) return;” add the following:

if ((Log) && (reIdG))

{

IdGame();

}

This adds the header, which records the date, players and location, to each game recorded. The file, ChessCode.pgn, receives the score sheet as a game is played. You will need to add the following two lines to EndChessCode():

extern std::wfstream Scoresheet;

Scoresheet.close();

CCGUI.cpp will get many of the changes we need for score keeping. Start by adding the two lines below to the top of the file.

#include <fstream>

using namespace std;

Follow by adding the following 3 forward declarations to the list of forward declarations:

void IdGame(void);

void DispColor(WCHAR\*, int, COLOR);

void ColorTag(UCHAR cohp);

Then add the 3 lines below to the end of the static declarations. ASCIIpce[] will be used in recording moves and the constant number and letter declarations will show up on the edge of the chess board and help you identify squares when you are setting up chess problems or when the chessboard is flipped.

static WCHAR ASCIIpce[8] = L" PNBRQK";

const wchar\_t NumArray[8] = { '1','2','3','4','5','6','7','8' };

const wchar\_t CharArray[8] = { 'a','b','c','d','e','f','g','h' };

Just prior to the declarations for the small chess piece graphics add the 3 lines below.

WCHAR WClrStr[8] = L" White ";

WCHAR BClrStr[8] = L" Black ";

std::wfstream Scoresheet;

In LocatePt() add the conditional below before assigning values to x or y.

if (flip\_brd) sqr = 63 - sqr;

In DispBrd(), after the 1st for loop, add:

ShoNumLet();

In InitGraphicsCode() add the following two lines first at the top of the function:

Scoresheet.open(L"ChessCode.pgn", fstream::out | fstream::in | fstream::trunc);

hWaitCursor = LoadCursor(0, IDC\_WAIT);

Then a few lines later, immediately after the lines where you set the masks for the small graphics, add:

SCORELN = 14; // the number of lines in the score sheet

And immediately after the lines where you set the masks for the large graphics, add:

SCORELN = 30; // the number of lines in the score sheet

Between DispBrd(); and return true; add the next three lines:

DispInfoBoxes();

DispColor(WClrStr, 8, wht);

DispColor(BClrStr, 8, blk);

Six lines will be needed at the end of ClearBoard(). Just after ‘mvnum = 1;’ add

reIdG = true; // start of new game after this

// clear the score sheet not already logged.

Wlinecnt = 0;

for (int i = 0; i < 606; i++) szWbuf[i] = 0;

Blinecnt = 0;

for (int i = 0; i < 486; i++) szBbuf[i] = 0;

Several lines will be needed in NewGame(). Immediately after DispBrd(), add

IdGame();

Then after gameover = false, add the following lines:

// clear display of any moves played

Wlinecnt = 0;

for (int i=0; i<606; i++) szWbuf[i] = 0;

Blinecnt = 0;

for (int i=0; i<486; i++) szBbuf[i] = 0;

Wlen\_szbuf =0; Blen\_szbuf = 0;

mate = 0; // if 1st move is a check and previous game just ended in mate, ‘#’ was displayed

DispInfoBoxes();

DispColor(WClrStr, 8, wht);

DispColor(BClrStr, 8, blk);

A few lines are needed in YouAsBlack(). Start by declaring:

int i;

Then after mvnum = 2; add the following lines:

// clear the szWbuf and szBbuf

for (i = 0; i < 606; i++) szWbuf[i] = 0;

for (i = 0; i < 486; i++) szBbuf[i] = 0;

lstrcpy(szWbuf, L" 1. ... ");

lstrcpy(szBbuf, L"");

szWbuf[0x13] = '\n';

szWbuf[0x14] = 0;

Wlinecnt = 1;

Blinecnt = 0;

Wlen\_szbuf = 20;

swprintf(gbuf, Wlen\_szbuf, szWbuf);

if (Log)

{

// copy gbuf to Charbuf, dropping the leading 0 of the wide char

for (int i = 0; i < 20; i++)

Charbuf[i] = gbuf[i]; // save Charbuf contents for EndSetup()

}

Blen\_szbuf = 0;

mate = 0; // if 1st move is a check and previous game ended in mate, '#' was displayed

A few lines are needed in YouAsWhite(). Start by declaring:

int i;

Then after flip\_brd = false; add the following lines:

// clear the szWbuf and szBbuf

for (i = 0; i < 606; i++) szWbuf[i] = 0;

for (i = 0; i < 486; i++) szBbuf[i] = 0;

Wlen\_szbuf = 0;

Blen\_szbuf = 0;

mate = 0; // if 1st move is a check and previous game ended in mate, '#' was displayed

More lines are needed in ComputerAsBlack(). Start by declaring:

int i;

Then after mvnum = 2; add the following lines:

// clear the szWbuf and szBbuf

for (i = 0; i < 606; i++) szWbuf[i] = 0;

for (i = 0; i < 486; i++) szBbuf[i] = 0;

lstrcpy(szWbuf, L" 1. ... ");

lstrcpy(szBbuf, L"");

szWbuf[0x13] = '\n';

szWbuf[0x14] = 0;

Wlinecnt = 1;

Blinecnt = 0;

Wlen\_szbuf = 20;

swprintf(gbuf, Wlen\_szbuf, szWbuf);

if (Log)

{

// copy gbuf to Charbuf, dropping the leading 0 of the wide char

for (int i = 0; i < 20; i++)

Charbuf[i] = gbuf[i]; // save Charbuf contents for EndSetup()

}

Blen\_szbuf = 0;

mate = 0; // if 1st move is a check and previous game ended in mate, '#' was displayed

A few lines are needed in ComputerAsWhite(). Start by declaring:

int i;

Then after flip\_brd = true; add the following lines:

// clear the szWbuf and szBbuf

for (i = 0; i < 606; i++) szWbuf[i] = 0;

for (i = 0; i < 486; i++) szBbuf[i] = 0;

Wlen\_szbuf = 0;

Blen\_szbuf = 0;

mate = 0; // if 1st move is a check and previous game ended in mate, '#' was displayed

In the function EndSetup(), after the second ShowWindow(), add the line below:

EnableMenuItem(hMenu, ID\_EDIT\_UNDOMOVE, MF\_BYCOMMAND | MF\_DISABLED | MF\_GRAYED);

Also in the function EndSetup(), you’ll want to add the first three lines below after DispBrd(); and the next block of code below immediately after the third ShowWindow(). You’ll want to remove any lines which are duplicated.

DispInfoBoxes();

DispColor(WClrStr, 8, wht);

DispColor(BClrStr, 8, blk);

mmidx = 0; // don’t count any previously made moves toward draws after an edit

moveslisted = false; // allow the moves to be regenerated

if (reIdG) IdGame();

if (com) // if black

{

mvnum = 2;

if (CoHP == 0)

ComputerAsBlack();

else

YouAsBlack(); // this resets the move number and formats the scoresheet

}

else

{

mvnum = 1; // reset move number so moves made before edit are not checked for repeats

if (CoHP == 0)

YouAsWhite();

else

ComputerAsWhite();

}

After all these changes, you need to add the following large block of code in its entirety. It can be placed prior to PromoTW().

void DispChar(WCHAR\* CharStr, int xPos, int yPos, int len)

{

RECT Rect;

HDC hDC;

int x, y;

x = xPos;

y = yPos;

Rect = { x,y,x + 30, y + 40 };

hDC = GetDC(hWnd); // get device context for main window

DrawText(hDC, CharStr, len, &Rect, DT\_LEFT);

ReleaseDC(hWnd, hDC);

}

static void ShoNumLet(void)

{

int i;

int xPos, yPos;

HDC hDC;

WCHAR CharStr[2] = L"0";

xPos = HBOARDOFFSET + (8 \* BmpWidth);

yPos = (BmpWidth >> 1) - 8;

for (i = 7; i >= 0; i--)

{

if (flip\_brd)

{

CharStr[0] = NumArray[7 - i];

DispChar(CharStr, xPos, yPos, 1);

}

else

{

CharStr[0] = NumArray[i];

DispChar(CharStr, xPos, yPos, 1);

}

yPos += BmpWidth;

}

xPos = HBOARDOFFSET + (BmpWidth >> 1) - 2;

yPos = (8 \* BmpWidth);

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

{

if (flip\_brd)

{

CharStr[0] = CharArray[7 - i];

DispChar(CharStr, xPos, yPos, 1);

}

else

{

CharStr[0] = CharArray[i];

DispChar(CharStr, xPos, yPos, 1);

}

xPos += BmpWidth;

}

}

//

// draw text frame

//

void DrawFrame(HDC hDC, RECT& Rect)

{

int x1, y1, x2, y2;

POINT pArray[3];

HPEN hPen, hOldPen;

HBRUSH hOldBrush;

x1 = Rect.left;

x2 = Rect.right;

y1 = Rect.top;

y2 = Rect.bottom;

hOldBrush = (HBRUSH)SelectObject(hDC, GetStockObject(WHITE\_BRUSH));

hOldPen = (HPEN)SelectObject(hDC, GetStockObject(WHITE\_PEN));

SelectObject(hDC, hPen = CreatePen(PS\_SOLID, 1, RGB(0, 0, 0)));

Rectangle(hDC, x1 + 1, y1 + 1, x2 - 1, y2 - 1); // THIS LINE DRAWS THE RECTANGLE

DeleteObject(SelectObject(hDC, GetStockObject(WHITE\_PEN)));

SelectObject(hDC, hOldBrush);

DeleteObject(SelectObject(hDC, hOldPen));

}

//

// Display Info Boxes

//

void DispInfoBoxes(void)

{

HDC hDC;

RECT Rect{};

hDC = GetDC(hWnd); // find device context for the main window

Rect = { 2, 0, HBOARDOFFSET - 1, BmpWidth \* 6 + 1 };

DrawFrame(hDC, Rect);

if (ScreenHeight >= MINHEIGHT)

Rect = { 2, BmpWidth \* 6, HBOARDOFFSET - 1, (BmpWidth \* 8) };

else Rect = { 2, BmpWidth \* 6, HBOARDOFFSET - 1, (BmpWidth \* 8+5) };

DrawFrame(hDC, Rect);

ReleaseDC(hWnd, hDC);

}

//

// display COLOR on display

//

void DispColor(WCHAR\* ClrStr, int len, COLOR clr)

{

RECT ClrRect;

HDC hDC;

int x, y;

// display color

if (clr == blk)

x = 128 + 4;

else x = 4;

y = 4;

ClrRect = { x,y,x + 90,y + 40 };

hDC = GetDC(hWnd); // get device context for main window

DrawText(hDC, ClrStr, len, &ClrRect, DT\_LEFT);

ReleaseDC(hWnd, hDC);

}

//

// display moves on White's scoresheet

//

void DispScorW(WCHAR\* WMstr, int len)

{

RECT ScorWRect;

HDC hDC;

int x, y;

// display move

x = 4;

y = CharHeight - 2;

ScorWRect = { x,y,x + 240,y + 40 \* SCORELN };

hDC = GetDC(hWnd); // get device context for main window

DrawText(hDC, WMstr, len, &ScorWRect, DT\_LEFT);

ReleaseDC(hWnd, hDC);

}

//

// display moves on Black's scoresheet

//

void DispScorB(WCHAR\* BMstr, int len)

{

RECT ScorBRect;

HDC hDC;

int x, y;

// display move

x = 120 + 4;

y = CharHeight - 2;

ScorBRect = { x,y,x + 90,y + 40 \* SCORELN };

hDC = GetDC(hWnd); // get device context for main window

DrawText(hDC, BMstr, len, &ScorBRect, DT\_LEFT);

ReleaseDC(hWnd, hDC);

}

// determine which moves are ambiguous

// upon return

// if ambig & 0x0f, files are the same

// if ambig & 0xf0, ranks are the same

unsigned char Ambiguous(char From, char To, char Pce, int movetype)

{

uint64\_t controllers, pinmask;

uint8\_t from1 = From;

uint8\_t from2 = from1;

uint8\_t to = To;

uint8\_t pce = Pce;

unsigned char ambig = 0;

bool ambiguity = false;

COLOR com;

if (BitCnt(Pbb[pce]) == 1) return 0; // no ambiguity if only one of a kind

else if (((pce & 0x0e) == wp) || ((pce & 0x0e) == wb)) return 0;

else

{

com = COLOR(pce & 1);

// the following code is OK for ambiguous moves

pinmask = (RookPin[dph] | BishopPin[dph]); // pin masks just combined

if (movetype == 1) // if simple king move

pinmask = FindPinsDiscoveries(dph,com);

else if (setbit[To] & pinmask) // is the pin blocked?

pinmask = 0;

// find pieces controlling the 'to' square

controllers = FindControllers(0xffffffffffffffff, com, to, pinmask);

// find controllers, one at a time

for (; controllers; controllers &= clrbit[from2])

{

from2 = (uint8\_t)LSBit(controllers);

// when used for P.Var., pieces may not be on the same square as they were;

// one piece can appear to be on two different squares

// when showing move just made, piece moving is not on from1 any longer

if ((from2 != from1) && (Brd[from2].pce == pce) &&

((Brd[from1].pce == pce) || (Brd[from1].pce == nil)))

// same piece type if any

{

ambiguity = true;

if ((from2 & 7) == (from1 & 7)) // on the same file?

{

if ((pce & 0x0e) != wn)

{

// but one may be behind the other

if (((from1 < from2) && (from2 < to)) || ((to < from1)

&& (from1 < from2)) || ((from2 < from1) && (from1 < to)) ||

((to < from2) && (from2 < from1)))

{

ambiguity = false;

break;

}

}

ambig |= 0x01; // on same file, use rank

}

else if ((from2 & 0x38) == (from1 & 0x38)) // on the same rank?

{

if ((pce & 0x0e) != wn)

{

// but one may be behind the other

if (((from1 < from2) && (from2 < to)) || ((to < from1) &&

(from1 < from2)) || ((from2 < from1) && (from1 < to)) ||

((to < from2) && (from2 < from1)))

{

ambiguity = false;

break;

}

}

ambig |= 0x10; // on same rank, use file

}

}

}

// may not be on the same file or the same rank, but still ambiguous

if ((ambig == 0) && (ambiguity)) ambig = 0x10; //just choose one

return ambig;

}

}

// move text pre-move

void MovTxtPre(Mov\_t\* move, char\* pce, char\* to, char\* from, char\* promo, char\* ctm, char\* cop, char\* movetype, int deep)

{

\*movetype = \_pext\_u32((\*move).m, 0x0000c0c0);

\*from = \_pext\_u32((\*move).m, 0x00003f00);

\*to = \_pext\_u32((\*move).m, 0x0000003f);

if (\*movetype > 10) // promotions need some parsing here

{

\*ctm = (!((\*to & 8) >> 3)); // color to move

\*promo = (((\*from & 0x38) >> 2) | \*ctm); // or'ing in the color of the piece

// now fix 'from' by adding rank

\*from = (\*from & 0x07) + PromoRank[\*ctm];

}

if (deep == 1) \*pce = Brd[\*to].pce; // get piece moving after move is made

else if ((deep == -1) || (abs((\*move).move.Value) > 0x7fd8)) \*pce = Brd[\*from].pce; // get piece moving before move is made

else \*pce = (\*move).move.Value >> 8; // piece moving has replaced Value, it may have been captured in variation, not on board

\*ctm = \*pce & 1;

\*cop = \*ctm ^ 1;

}

// move text post-move

void MovTxtPost(Mov\_t\* move, char\* pce, char\* to, char\* from, char\* promo, char\* ctm, char\* cop, char\* movetype)

{

\*movetype = \_pext\_u32((\*move).m, 0x0000c0c0);

\*from = \_pext\_u32((\*move).m, 0x00003f00);

\*to = \_pext\_u32((\*move).m, 0x0000003f);

if (\*movetype > 10) // promotions need some parsing here

{

\*ctm = (!((\*to & 8) >> 3)); // color to move

\*promo = (((\*from & 0x38) >> 2) | \*ctm); // or'ing in the color of the piece

// now fix 'from' by adding rank

\*from = (\*from & 0x07) + PromoRank[\*ctm];

}

\*pce = Brd[\*to].pce; // get piece moving

\*ctm = \*pce & 1;

\*cop = \*ctm ^ 1;

}

/\*

\* convert a move to a string

\*/

WCHAR\* MovTxt(Mov\_t move, int deep, int cbit)

{

static WCHAR str[12];

char pce, to, from, promo, cap, amb = 0;

char ctm, cop, movetype;

// MovTxtPost is used after a move has been made on the chess board

// MovTxtPre is used before a move has been made, (see ShoPV())

if (deep == 0)

MovTxtPost(&move, &pce, &to, &from, &promo, &ctm, &cop, &movetype);

else

MovTxtPre(&move, &pce, &to, &from, &promo, &ctm, &cop, &movetype, deep);

switch (movetype)

{

case 0:

case 1:

case 2:

cap = 0;

break;

case 3:

cap = wp | cop;

break;

case 4:

cap = wn | cop;

break;

case 5:

cap = wb | cop;

break;

case 6:

cap = wr | cop;

break;

case 7:

cap = wq | cop;

break;

case 8:

case 9:

cap = 0;

break;

case 10:

cap = wp | cop;

break;

case 11:

cap = 0;

break;

case 12:

cap = wn | cop;

break;

case 13:

cap = wb | cop;

break;

case 14:

cap = wr | cop;

break;

case 15:

cap = wq | cop;

break;

}

wcscpy\_s(str, L" "); // 11 chars and null

if (pce != nil)

{

amb = Ambiguous(from, to, pce, movetype);

if (((pce & 0x0e) == wk) && (((to - from) & 3) == 2)) /\* castling \*/

{

if (to < from)

{

wcscpy\_s(str, L"O-O ");

if (cbit) // is this a check?

{

str[6] = '+';

if (mate == 1) // Checkmate in 'mate' ply has been found

{

str[6] = '#';

}

}

}

else

{

wcscpy\_s(str, L"O-O-O ");

if (cbit) // see case above

{

str[6] = '+';

if (mate == 1) // Checkmate in 'mate' ply

{

str[6] = '#';

}

}

}

}

else

{

str[0] = ASCIIpce[pce >> 1];

str[1] = 'h' - (from & 7);

str[2] = '1' + (from >> 3);

if (cap & 0x0e)

str[3] = 'x';

else

str[3] = '-';

str[4] = 'h' - (to & 7);

str[5] = '1' + (to >> 3);

}

if (cbit) // is this a check? see case above

{

str[6] = '+';

if (mate == 1) // Checkmate in 'mate' ply

{

str[6] = '#';

}

}

else

str[6] = ' ';

//

// Now look at promotions

if (movetype > 10)

{

if (str[3] != 'x') // non-capture promo

{

// change the notation from Pd7-d8+ to d8=Q+

str[3] = ASCIIpce[promo >> 1];

str[0] = str[4];

str[1] = str[5];

str[2] = '=';

str[4] = str[6];

str[5] = str[6] = ' ';

}

else

{

// change the notation from Pd7xc8+ to dxc8=N+

str[0] = str[1]; // file

str[1] = 'x';

str[2] = str[4];

str[3] = str[5];

str[4] = '=';

str[5] = ASCIIpce[promo >> 1];

// str[6] is already set

}

}

//

// for the remaining moves

// now discard unwanted characters

//

else if (((pce & 0x0e) == wp) && (str[3] != 'x'))

{ // discard Pd2- from Pd2-d4+ to get d4+

str[0] = str[4];

str[1] = str[5];

str[2] = str[6];

str[3] = str[4] = str[5] = str[6] = ' ';

}

else if (((pce & 0x0e) == wp) && (str[3] == 'x'))

{ // discard P & 3 from Pd3xc4+ to get dxc4+

str[0] = str[1];

str[1] = str[3];

str[2] = str[4];

str[3] = str[5];

str[4] = str[6];

str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && ((pce & 0x0e) == wk) && (str[3] != 'x'))

{ // discard g2- from Kg2-h2+ to get Kh2+

str[1] = str[4];

str[2] = str[5];

str[3] = str[6];

str[4] = str[5] = str[6] = ' ';

}

else if (((pce & 0x0e) == wk) && (str[3] == 'x'))

{ // discard d2 from Kd2xc3+ to get Kxc3+

str[1] = str[3];

str[2] = str[4];

str[3] = str[5];

str[4] = str[6];

str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] != 'x') && (amb == 0))

{ // discard d2- from Bd2-e3+ to get Be3+

str[1] = str[4];

str[2] = str[5];

str[3] = str[6];

str[4] = str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] == 'x') && (amb == 0))

{ // discard d2 from Nd2xc4+ to get Nxc4+

str[1] = str[3];

str[2] = str[4];

str[3] = str[5];

str[4] = str[6];

str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] != 'x') && ((amb & 0x11) == 0x01))

// on same file (a-h)

{ // discard e & - from Ne2-d4+ to get N2d4+

str[1] = str[2];

str[2] = str[4];

str[3] = str[5];

str[4] = str[6];

str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] == 'x') && ((amb & 0x11) == 0x01))

// on same file (a-h)

{ // discard e from Ne2xd4+ to get N2xd4+

str[1] = str[2];

str[2] = str[3];

str[3] = str[4];

str[4] = str[5];

str[5] = str[6];

str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] != 'x') && ((amb & 0x11) == 0x10))

{ // discard 2 & - from Nc2-d4+ to get Ncd4+

str[2] = str[4];

str[3] = str[5];

str[4] = str[6];

str[5] = str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] == 'x') && ((amb & 0x11) == 0x10))

{ // discard 2 from Nc2xd4+ to get Ncxd4+

str[2] = str[3];

str[3] = str[4];

str[4] = str[5];

str[5] = str[6];

str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] != 'x') && ((amb & 0x11) == 0x11))

{ // discard - from Nc2-d4+ to get Nc2d4+

str[3] = str[4];

str[4] = str[5];

str[5] = str[6];

str[6] = ' ';

}

else if ((str[0] != 'O') && (str[3] == 'x') && ((amb & 0x11) == 0x11))

{ // discard nothing from Nc2xd4+

}

}

if ((Reps == 2) || (Draw50 >= 100))

{

str[7] = '=';

result = 'D';

}

return (str);

}

void ModifyPrevMv(char som)

{

WCHAR charbuf[21];

short idx, limit, offset;

result = 'D';

// modify previous '+' by opponent to show mate

if (som) // if black to move

{

if (Blinecnt < SCORELN)

idx = (Blinecnt - 1) \* 16;

else

idx = (SCORELN - 1) \* 16;

limit = idx + 15;

// scan most recent move for result '#'

for (; idx < limit; idx++)

{

if ((szBbuf[idx] == '+') || (szBbuf[idx] == '#'))

{

szBbuf[idx] = '#';

result = 'B';

break;

}

}

if ((n\_list\_mvs == 0) && (idx == limit)) // no '#' was found

szBbuf[idx - 7] = '=';

if ((Reps >= 2) || (OpReps >= 2))

{

if (idx == limit) szBbuf[idx - 7] = '='; // repetition without '+'

else szBbuf[idx] = '='; // repetition with '+'

result = 'D';

mvnum++; // increment the move number so the code below will work

}

DispScorB(&szBbuf[0], Blen\_szbuf);

Scoresheet.seekp(asterpos);

Scoresheet << result;

limit = UndoPos[mvnum - 1] - UndoPos[mvnum - 2]; // get length of string

position = UndoPos[mvnum - 2]; // get previous recorded position

Scoresheet.seekp(position); // back up to there

}

else // white to move

{

if (Wlinecnt < SCORELN)

idx = (Wlinecnt - 1) \* 20;

else

idx = (SCORELN - 1) \* 20;

limit = idx + 20;

// scan most recent move for result '#'

for (; idx < limit; idx++)

{

if ((szWbuf[idx] == '+') || (szWbuf[idx] == '#'))

{

szWbuf[idx] = '#';

result = 'W';

break;

}

}

if ((n\_list\_mvs == 0) && (idx == limit)) // no '#' was found

szWbuf[idx - 7] = '=';

if ((Reps >= 2) || (OpReps >= 2))

{

if (idx == limit) szWbuf[idx - 7] = '='; // repetition without '+'

else szWbuf[idx] = '='; // repetition with '+'

result = 'D';

mvnum++; // increment the move number so the code below will work

// as it does for mates

}

DispScorW(&szWbuf[0], Wlen\_szbuf);

Scoresheet.seekp(asterpos);

Scoresheet << result;

limit = UndoPos[mvnum - 1] - UndoPos[mvnum - 2]; // get length of string

position = UndoPos[mvnum - 2]; // get previous recorded position

Scoresheet.seekp(position); // back up to there

Scoresheet >> charbuf; // read move NUMBER

position = Scoresheet.tellp(); // get this position

}

// now fix the string in Scoresheet

Scoresheet >> charbuf; // read mate move

Scoresheet.seekp(position); // notice new position

for (idx = 0; idx < limit; idx++)

if ((charbuf[idx] == '+') || (charbuf[idx] == '#'))

{

charbuf[idx] = '#';

if ((Reps >= 2) || (OpReps >= 2))

charbuf[idx] = '=';

break;

}

else if (charbuf[idx] == 0) // end of string?

{

charbuf[idx] = '='; // just put '=' at end of string

charbuf[++idx] = 0; // and move end of string over 1

break;

}

if (som) // black plays mate move

offset = 4;

else offset = 1;

// now insert 'offset' spaces

for (idx = 7; idx >= 0; idx--) charbuf[idx + offset] = charbuf[idx];

// now insert spaces

for (idx = 0; idx < offset; idx++) charbuf[idx] = ' ';

if ((!som) && ((Reps >= 2) || (OpReps >= 2)))

{

Scoresheet << charbuf;

Scoresheet << L" " << endl; // draw by repetition

}

else Scoresheet << charbuf << endl; // save move which announces mate

}

void ShoMov(int mvnum, UCHAR turn, Mov\_t move, char cbit)

{

int movenum, i, cnt;

WCHAR wszLf[2] = { '\n',0 };

size\_t sz;

movenum = (mvnum - 1) / 2 + 1;

if (movenum >= 3) gamekeep = TRUE; //save game if 2 moves

if (Log)

{

// If you are playing as black, Scoresheet has had 1. ... put

// into it and position in file (=0xb2) was saved. Then IdGame()

// was called from EndSetup(). This resets the position

// to 0x9f. So don't overwrite it below. Save it directly.

UndoPos[mvnum - 1] = Scoresheet.tellp(); // save previous position

}

if ((turn & 1) == wht) // is it white's turn?

{

sz = 20;

if (Wlinecnt >= SCORELN) // scroll score on display

{

// find the 1st <lf> for White

for (i = 0; (szWbuf[i] != '\n' && i < 20); i++)

cnt = i;

cnt = cnt + 2;

// now discard the beginning of the string

for (i = 0; i < (Wlen\_szbuf - cnt); i++)

szWbuf[i] = szWbuf[i + cnt];

Wlen\_szbuf -= cnt;

// now clear the end of the string

for (i = 0; i < cnt + 4; i++) szWbuf[Wlen\_szbuf + i] = ' '; // extra spaces

szWbuf[Wlen\_szbuf + cnt + 4] = 0;

DispScorW(&szWbuf[0], (Wlen\_szbuf + cnt + 5));

// now clear the end of the string for future concat

for (i = 0; i < cnt + 4; i++) szWbuf[Wlen\_szbuf + i] = 0;

// find the 1st <lf> for Black

for (i = 0; (szBbuf[i] != '\n' && i < 16); i++)

cnt = i;

cnt = cnt + 2;

// now discard the beginning of the string

for (i = 0; i < (Blen\_szbuf - cnt); i++)

szBbuf[i] = szBbuf[i + cnt];

Blen\_szbuf -= cnt;

// now clear the end of the string

for (i = 0; i < cnt + 4; i++) szBbuf[Blen\_szbuf + i] = ' '; // extra spaces

szBbuf[Blen\_szbuf + cnt + 4] = 0;

DispScorB(&szBbuf[0], (Blen\_szbuf + cnt + 5));

// now clear the end of the string for future concat

for (i = 0; i < cnt + 4; i++) szBbuf[Blen\_szbuf + i] = 0;

}

// offset of number depends on the number of digits in it

if (movenum < 10)

swprintf(gbuf, sz, L" %3.1d. %s", movenum, MovTxt(move,0,cbit));

else if (movenum < 100)

swprintf(gbuf, sz, L" %3.2d. %s", movenum, MovTxt(move,0,cbit));

else

swprintf(gbuf, sz, L" %3.3d. %s", movenum, MovTxt(move,0,cbit));

Wlen\_szbuf += sz; // size of move gbuf & <lf>

wcscat\_s(szWbuf, gbuf);

wcscat\_s(szWbuf, wszLf); // add <lf> to scoresheet

if (Log)

{

Scoresheet << gbuf;

position = Scoresheet.tellp();

UndoPos[mvnum] = position;

}

DispScorW(&szWbuf[0], Wlen\_szbuf);

Wlinecnt++;

}

else

{ // black's turn

sz = 16;

swprintf(gbuf, sz, L" %s", MovTxt(move,0,cbit));

Blen\_szbuf += sz; // size of move gbuf & <lf>

wcscat\_s(szBbuf, gbuf);

wcscat\_s(szBbuf, wszLf); // add <lf> to scoresheet

if (Log)

{

Scoresheet << gbuf << wszLf;

position = Scoresheet.tellp();

UndoPos[mvnum] = position;

}

DispScorB(&szBbuf[0], Blen\_szbuf);

Blinecnt++;

}

if ((Reps == 2) || (Draw50 >= 100))

{

position = Scoresheet.tellp();

Scoresheet.seekp(asterpos);

Scoresheet << result;

Scoresheet.seekp(position); // back to end of current move

}

}

//

// set color of player in Game Id

//

void ColorTag(UCHAR cohp)

{

long endcolpos;

long position = Scoresheet.tellp();

Scoresheet.seekp(colorpos);

if (!cohp) // if Color of Human Pieces is white (0)

{

Scoresheet << L"[White \"Player\"]\n";

Scoresheet << L"[Black \"ChessCode ver 1.0\"]\n";

}

else

{

Scoresheet << L"[White \"ChessCode ver 1.0\"]\n";

Scoresheet << L"[Black \"Player\"]\n";

}

endcolpos = Scoresheet.tellp();

if (position > endcolpos)

Scoresheet.seekp(position); // restore position in file when called

// else endcolpos is correct for 1st call

}

//

// all file streaming into char strings for now

//

void IdGame(void)

{

int i;

char date[9];

char sdate[11];

char buf[4] = { 0,0,0,0 };

// backup if false start, otherwise continue in file.

if (gamekeep == FALSE) Scoresheet.seekp(gamstpos);

else gamenumb++;

gamekeep = FALSE; // reset at beginning of every game

gamstpos = Scoresheet.tellp(); //save for next game

\_strdate\_s(date);

sdate[0] = '2';

sdate[1] = '0';

sdate[2] = date[6];

sdate[3] = date[7];

sdate[4] = ':';

sdate[5] = date[0];

sdate[6] = date[1];

sdate[7] = ':';

sdate[8] = date[3];

sdate[9] = date[4];

sdate[10] = 0;

//

sprintf\_s(buf, "%d", gamenumb);

//

// seven tags needed

//

Scoresheet << L"\n\n[Event \"Practice Games\"]\n";

Scoresheet << L"[Site \"Sometown, MO USA\"]\n";

Scoresheet << L"[Date \"";

// Scoresheet << 2008.02.14

for (i = 0; i < 10; i++) Scoresheet << sdate[i];

Scoresheet << L"\"]\n";

Scoresheet << L"[Round \"";

Scoresheet << buf;

Scoresheet << L"\"]\n";

colorpos = Scoresheet.tellp();

ColorTag(CoHP);

Scoresheet << L"[Result \"";

asterpos = Scoresheet.tellp();

Scoresheet << L"\*\"] \n\n";

//

// start of game or continuation from position setup

//

if (com == blk)

{

DispScorW(&szWbuf[0], Wlen\_szbuf);

if (Log)

{

Scoresheet << Charbuf;

position = Scoresheet.tellp();

}

}

reIdG = false;

}

Add the three external function declarations toward the top of CCGUI.cpp:

extern uint64\_t FindPinsDiscoveries(int dph, int colr);

extern uint64\_t FindControllers(uint64\_t CheckMask, int colr, int sq, uint64\_t pinmask);

extern void AnnounceEndofGame(WCHAR\* resultmsg, int len);

and add forward declarations for:

void DispInfoBoxes(void);

static void ShoNumLet(void);

Also, now that MakeMove() detects checks, you can simplify the code around //ShoMov() in EndSearch(). Locate:

//ShoMov(mvnum, Turn, move, checktoking[com^1]);

Change ‘checktoking[com^1]’ to ‘move.move.Value & 0x0001’. Make this same change in the line below it. Then remove the eight lines above it. Also remove the two slashes that make it a comment. Since checktoking[] (lower case only) is no longer needed, this variable can be removed from both CCVariables.cpp and CCExterns.h.

In the next section we’ll look at avoiding draw by repetition.

**Avoiding/Seeking Draws**

When winning, ChessCode.cpp sometimes needs to consider playing less than the best move to avoid a three time repetition of the position. When losing, ChessCode.cpp would do well to repeat the position. There are also cases when both sides can move without making any pawn moves or captures. If this continues for fifty moves, either side can claim a draw. So if one player is ahead, it may be best again to consider playing less than the best move to avoid running into this fifty move rule.

ChessCode will attempt to avoid a three time repetition when ahead in material. A three time repetition means the position is identical to two earlier positions with the same player moving. Actually, the second time the same position repeats is the third time the position occurs. To avoid this, ChessCode cuts the value of its move in half upon seeing the first repetition of the position. It will do the same if it sees a repetition of position by its opponent. By the second repetition, the value of its move is reduced to zero.

To avoid the 50 move rule draw, ChessCode will begin to reduce the value of its move by 1/16th per move starting on the 34th move without a capture or pawn move. The value of the move will be zero by the 50th move.

To add these changes, you’ll begin by adding a new variable to both CCVariables.cpp and CCExterns.h.

int Draw50; // number of moves since a capture or pawn move

As each move is made, it will be stored in the global array MovesMade[]. The index into the array will be the global ‘mmidx’ which will increment as each half-move is played. Although there are faster methods of looking for repeats, ChessCode uses a simple method of scanning this array backwards from the most recent move until it either sees a capture or pawn move, or it finds a position which is repeated. A temporary chessboard is used to unwind the moves of the game without affecting the current position. The scan ends when a capture or pawn move is seen in the MovesMade list, or the position on the temporary chessboard matches the current position for the 2nd time. The scan only occurs at the base level of any search, so the time used is not significant.

You’ll need to modify CCGenMoves.cpp by adding an external message declaration near the top of the file.

extern WCHAR RepeatPos[20];

Further down in the file, after ResolveTactics() and before GenListofMoves(), you’ll want to add the following function:

int ReptdPos(Mov\_t move)

{

int movetype, i, j, to, from, stm, reps = 0;

bool eqalbrds = false;

SQUARE tempbrd[64];

// first check that move is a decent candidate for a repetition

movetype = \_pext\_u32(move.m, 0x0000c0c0);

to = \_pext\_u32(move.m, 0x0000003f);

// only simple moves, but not pawn moves, can repeat

if ((movetype > 2) || ((Brd[to].pce & 0x0e) == wp) || (mvnum < 6)) return 0;

// copy the chess board to a temporary copy

for (i = 0; i < 64; i++) tempbrd[i] = Brd[i];

// current move has not been put into MovesMade list

from = \_pext\_u32(move.m, 0x00003f00);

// undo half a move

tempbrd[from].pce = tempbrd[to].pce;

stm = (tempbrd[from].pce & 1); // notice side which is moving

tempbrd[to].pce = nil;

// scan list of moves made

for (i = (mmidx - 2); i >= 1; i--)

{

movetype = \_pext\_u32(MovesMade[i].m, 0x0000c0c0);

to = \_pext\_u32(MovesMade[i].m, 0x0000003f);

// check that move is a decent candidate for a repetition

// only simple moves, but not pawn moves, can repeat

if ((movetype > 2) || ((tempbrd[to].pce & 0x0e) == wp) || (mvnum < 6)) return reps;

from = \_pext\_u32(MovesMade[i].m, 0x00003f00);

i--;

// undo 2nd half move

tempbrd[from].pce = tempbrd[to].pce;

tempbrd[to].pce = nil;

// compare boards

eqalbrds = true;

for (j = 0; j < 64; j++)

{

if (tempbrd[j].pce != Brd[j].pce)

{

eqalbrds = false;

break;

}

}

if (eqalbrds == true)

{

reps++; // duplicate position found

if (reps == 2) break;

}

// get first move of next 2 moves

movetype = \_pext\_u32(MovesMade[i].m, 0x0000c0c0);

to = \_pext\_u32(MovesMade[i].m, 0x0000003f);

// check that move is a decent candidate for a repetition

// only simple moves, but not pawn moves, can repeat

if ((movetype > 2) || ((tempbrd[to].pce & 0x0e) == wp) || (mvnum < 6)) return reps;

from = \_pext\_u32(MovesMade[i].m, 0x00003f00);

// undo half a move

tempbrd[from].pce = tempbrd[to].pce;

tempbrd[to].pce = nil;

}

return reps;

}

Also you’ll add to GenListofMoves(), after the assignment of values to PosVal and MatVal, replacing the line:

list\_mvs[i].move.Value = MatVal + PosVal; // this overwrites the check-bit

with the following:

// All ingrediants of Value are known, except if move is a repetition,

// if move is approaching 50 moves without a capture or pawn move,

// or if checkmate is seen. So it seems a good time to look for these

// conditions and save the results in list\_mvs[i].move.Value

Reps = ReptdPos(list\_mvs[i]); // count number of repeats with same side to move

if (Reps >= 2) list\_mvs[i].move.Value = 0; // drawn position is worth zero

else if (Reps == 1)

list\_mvs[i].move.Value = (MatVal + PosVal) >> 1; // 1/2 value to avoid

else if ((OpReps) && ((\_pext\_u32(list\_mvs[i].m, 0x0000c0c0) < 3) && // movetype < 3

(((Brd[\_pext\_u32(list\_mvs[i].m, 0x0000003f)].pce) & 0x0e) != wp))) // no pn move

list\_mvs[i].move.Value = (MatVal + PosVal) >> 1; // 1/2 value to avoid

else if ((Draw50 >= 83) && (Draw50 <= 99) &&

(((Brd[\_pext\_u32(list\_mvs[i].m, 0x0000003f)].pce) & 0x0e) != wp) &&

((\_pext\_u32(list\_mvs[i].m, 0x0000c0c0) < 3) ||

(\_pext\_u32(list\_mvs[i].m, 0x0000c0c0) != 8)))

{ // additional 1/16th of value lost on each move of this and next 15

int penalty = (MatVal + PosVal) >> 4; // 1/16th of value

list\_mvs[i].move.Value = (MatVal + PosVal) - ((Draw50 - 83) \* penalty);

}

else list\_mvs[i].move.Value = MatVal + PosVal; // the check-bit is overwritten

Next, you’ll modify ChessCode.cpp. by adding

#include <immintrin.h>

between using namespace std; and #include “framework.h”. Then add the following external function declaration near the top of the file.

extern int ReptdPos(Mov\_t);

At the end of the declarations of external messages add the next two lines:

extern WCHAR RepeatPos[20];

extern WCHAR Draw50msg[19];

Just above the function TogglePlayer(), add

void AnnounceEndofGame(WCHAR\* resultmsg, int len)

{

DispMsg(resultmsg, len); // could use pop-up to delay move

EnableMenuItem(hMenu, ID\_EDIT\_SETUPPOSITION, MF\_BYCOMMAND | MF\_DISABLED | MF\_GRAYED);

endsearch = true;

gameover = true;

}

Just after the declaration of RunChessCode() in that function add

int movetype;

Then replace the line with ‘DispMsg(Stalemate, 11);’ with

AnnounceEndofGame(Stalemate, 11);

And the line with ‘DispMsg(Checkmate, 11);’ with

AnnounceEndofGame(Checkmate, 11);

Remove three lines starting with:

endsearch = true;

In the else clause, after the line list\_mvs[0] = MakeMove(list\_mvs[0]), replace the line DispBrd() with the following lines:

Draw50++;

movetype = \_pext\_u32(list\_mvs[0].m, 0x0000c0c0);

if (((movetype >= 3) && (movetype != 8)) || ((Brd[ToSqr].pce & 0x0e) == wp))

Draw50 = 0; // reset if capture or pawn move

if ((list\_mvs[0].move.Value == 0))

{ // most likely this is a 3 time repetition

Reps = ReptdPos(list\_mvs[0]); // check to be sure

if (Reps >= 2)

{

AnnounceEndofGame(RepeatPos, 19); // could use pop-up to delay move

}

}

if (Draw50 >= 100)

{ // 50 moves without a capture or pawn move for both players

AnnounceEndofGame(Draw50msg, 18);

}

DispBrd();

// check bit is set in MakeMove()

ShoMov(mvnum, Turn, list\_mvs[0], list\_mvs[0].move.Value & 0x0001);

Finally, we’ll modify CCGUI.cpp for the support needed for declaring draws. Start by adding the line below to the external function declarations.

extern int ReptdPos(Mov\_t move);

Further down the list of declarations just after WCHAR Check[12] = L”Check + “; add the two lines below.

WCHAR RepeatPos[20] = L"Drawn by Repetition";

WCHAR Draw50msg[19] = L"Draw: 50 move rule";

At the end of EndSearch(), just after you see PrevTo = ToSqr; add the following lines:

Draw50++;

movetype = \_pext\_u32(move.m, 0x0000c0c0);

if (((movetype >= 3) && (movetype != 8)) || ((Brd[ToSqr].pce & 0x0e) == wp))

Draw50 = 0; // reset if capture or pawn move

else

OpReps = ReptdPos(move); // has opponent repeated position?

if (OpReps == 2)

{

ModifyPrevMv(com);

AnnounceEndofGame(RepeatPos, 19); // could display a pop-up here

}

Congratulations on getting to this point in the program’s development. Take time to remove any bugs you may find and test the program thoroughly. Once we get into the tree search, the fixing of problems gets increasingly difficult, although if you stick with it, you’ll develop the skill and confidence needed to debug any software!

In the next chapter we will start to develop the search tree, but before moving ahead, take time to save your files and back them up.