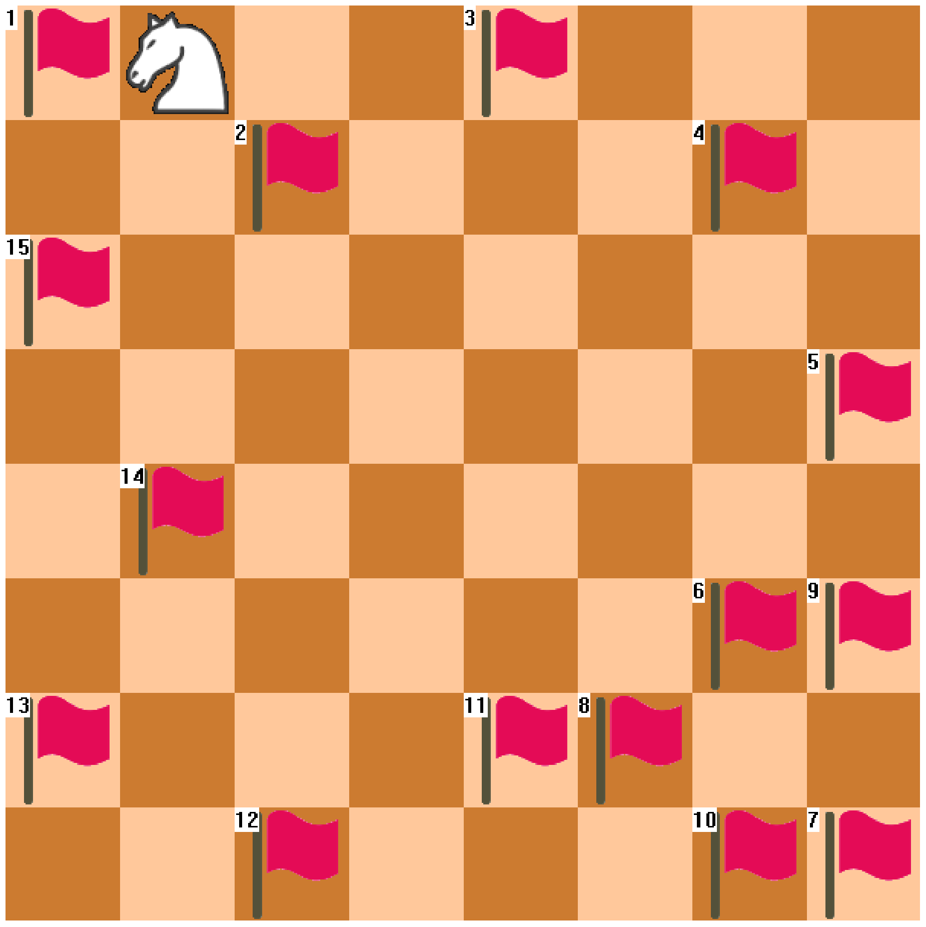
Knight's Tour



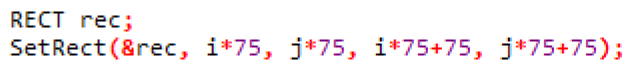
Christian Stiehl 500669869

Requirements

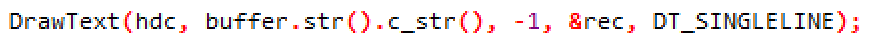
1 Programming language has to be C++.

I programmed the knight's tour in C++ using the win32 API. The IDE I used was Dev-C++.

2 The program makes use of pointers and references.

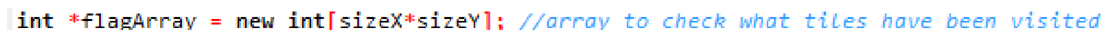


```
RECT rec;
SetRect(&rec, i*75, j*75, i*75+75, j*75+75);
```



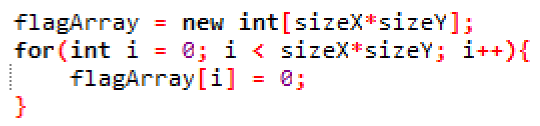
```
DrawText(hdc, buffer.str().c_str(), -1, &rec, DT_SINGLELINE);
```

Because I had not used C++ a lot before, knowing when using references is better is not 100% clear to me yet. However the functions SetRect and DrawText require LPRECTs which you can get by declaring a regular RECT and then giving the function a reference to your RECT variable.



```
int *flagArray = new int[sizeX*sizeY]; //array to check what tiles have been visited
```

I used this array of pointers so that I could change the array size whenever I needed to by just making it point to a new array and setting all the ints in that array to 0.

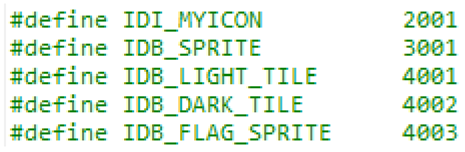


```
flagArray = new int[sizeX*sizeY];
for(int i = 0; i < sizeX*sizeY; i++){
    flagArray[i] = 0;
}
```

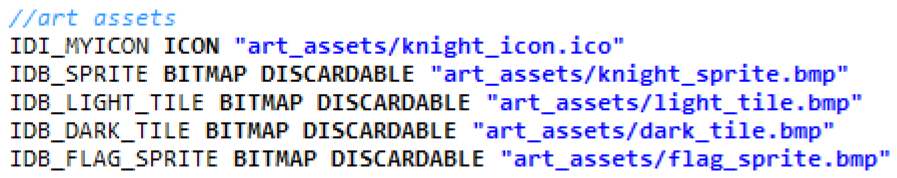
I know that int temp = 20; would store an integer of value 20 to a data point in the computer, for example 1700. Then int reference = &temp; would be 1700. And int pointer = *reference; would be 20. However why this implementation is useful is not clear to me yet. But I am confident I can learn this going on with the project.

3 There is a visual representation.

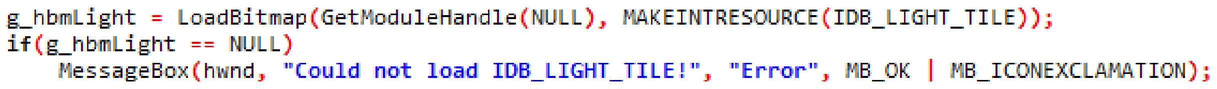
I used Win32’s window features to visualize navigation and used bitmaps to draw sprites on screen. See cover page for screenshot of the program.



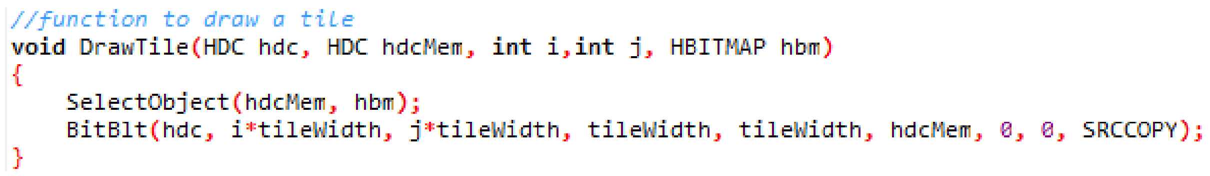
```
#define IDI_MYICON          2001
#define IDB_SPRITE          3001
#define IDB_LIGHT_TILE      4001
#define IDB_DARK_TILE       4002
#define IDB_FLAG_SPRITE     4003
```



```
//art assets
IDI_MYICON ICON "art_assets/knight_icon.ico"
IDB_SPRITE BITMAP DISCARDABLE "art_assets/knight_sprite.bmp"
IDB_LIGHT_TILE BITMAP DISCARDABLE "art_assets/light_tile.bmp"
IDB_DARK_TILE BITMAP DISCARDABLE "art_assets/dark_tile.bmp"
IDB_FLAG_SPRITE BITMAP DISCARDABLE "art_assets/flag_sprite.bmp"
```



```
g_hbmLight = LoadBitmap(GetModuleHandle(NULL), MAKEINTRESOURCE(IDB_LIGHT_TILE));
if(g_hbmLight == NULL)
    MessageBox(hwnd, "Could not load IDB_LIGHT_TILE!", "Error", MB_OK | MB_ICONEXCLAMATION);
```



```
//function to draw a tile
void DrawTile(HDC hdc, HDC hdcMem, int i,int j, HBITMAP hbm)
{
    SelectObject(hdcMem, hbm);
    BitBlt(hdc, i*tileWidth, j*tileWidth, tileWidth, tileWidth, hdcMem, 0, 0, SRCCOPY);
}
```

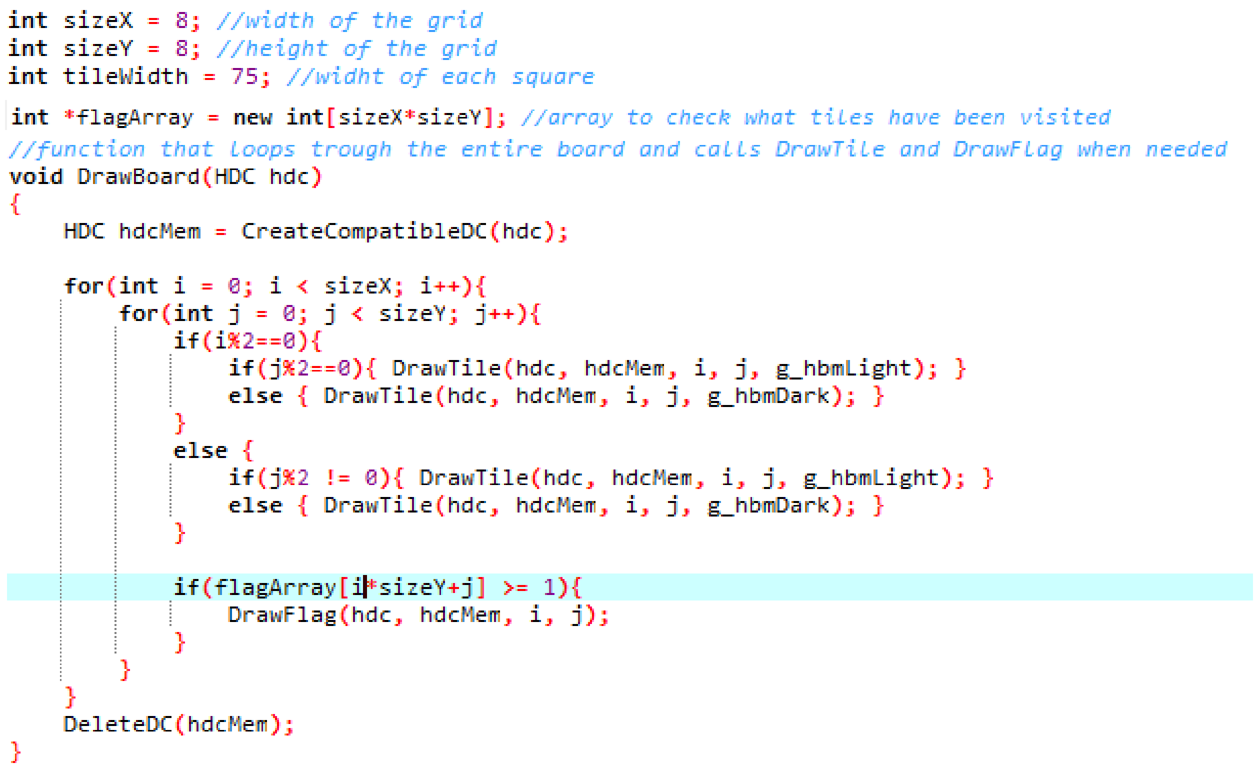
First I define IDs for all the sprites in resource.h, then I assign files to the IDs in resource.rc. In the code I first declare a bunch of variables (g_hbmLight for example) and assign these to NULL. Then in the WM_CREATE switch statement I use the function LoadBitmap to load the bitmap that is saved in IDB_LIGHT_TILE into the variable. If after this function the variable is still NULL, loading the bitmap was not successful, so the program presents an error warning.

Then to draw the tiles, knight and flags I have different functions, for example DrawTile. DrawTile takes 2 HDC objects these are what the bitmaps need to be drawn on. Then it takes 2 ints for the position and the HBITMAP for either a dark tile or a light tile.

4 The board is at least 5x5 in size.

In my version of the Knight's Tour the board can be any size. I accomplished this by using a double for-loop that draws the board.

To keep track of what tiles had already been visited I use an array of size sizeX*sizeY (sizeX and sizeY are the width and height of the board). To represent the board being two dimensional in an one dimensional array I find objects using X*sixeY+Y (X and Y are the positions of the tile e.g. (0,0) or (6,5) and sizeY is the height of the board).



```
int sizeX = 8; //width of the grid
int sizeY = 8; //height of the grid
int tileWidth = 75; //widht of each square

int *flagArray = new int[sizeX*sizeY]; //array to check what tiles have been visited
//function that loops trough the entire board and calls DrawTile and DrawFlag when needed
void DrawBoard(HDC hdc)
{
    HDC hdcMem = CreateCompatibleDC(hdc);

    for(int i = 0; i < sizeX; i++){
        for(int j = 0; j < sizeY; j++){
            if(i%2==0){
                if(j%2==0){ DrawTile(hdc, hdcMem, i, j, g_hbmLight); }
                else { DrawTile(hdc, hdcMem, i, j, g_hbmDark); }
            }
            else {
                if(j%2 != 0){ DrawTile(hdc, hdcMem, i, j, g_hbmLight); }
                else { DrawTile(hdc, hdcMem, i, j, g_hbmDark); }
            }

            if(flagArray[i*sizeY+j] >= 1){
                DrawFlag(hdc, hdcMem, i, j);
            }
        }
    }
    DeleteDC(hdcMem);
}
```

Here we see the declaration of the width and height of the board (default 8) and the width of a tile (this is used for the bitmaps, every bitmap is 75 pixels high and wide).

As mentioned earlier the flagArray is the one dimensional array that keeps track of tiles that have already been visited. We can also see in the highlighted line of code how a position is called. In this case var i is the x axis and var j is the y axis. The loop checks if a tile has been visited (the index of the array is greater or equal to 1) and if it has been visited draws a flag on that tile.

To draw the board I used a double for loop so for each amount of sizeX the program loops once trough the amount of sizeY. Then it checks if the current X is even or odd (i%2==0) to determine if the even or odd tiles should be dark in that column.

5 The knight has to move according to its movement rules.

To accomplish this I first define a struct chess_moves. Then I fill an array with all 8 possible moves a knight could make from any position.

```
//defines a structure for a chess move
typedef struct chess_moves {
   //x and y position on the board
   int x,y;
}chess_moves;

//create moveArray and fill it with all 8 possible moves a knight could make from any square
//these are ordered in the standard warnsdorf priority order
chess_moves moveArray[8] = { {2,1}, {1,2},{-1,2},{-2,1}, {-2,-1},{-1,-2},{1,-2},{2,-1} };
```

These are ordered in the standard Warnsdorff priority order. This means the order the program tries them in.

```
//these for loops go to 8 because there are 8 moves possible in moveArray, does not get affected by board size
for(int i = 0; i < 8; i++){
    //reset score to 0 and apply the next possible move to the current position
    tempScore = 0;
    tempMove.x = g_knightInfo.x + moveArray[i].x;
    tempMove.y = g_knightInfo.y + moveArray[i].y;
```

To try moves I simple run a for loop and store the knights new position in tempMove. tempMove is the current x and y position of the knight + the x and y of the move I want to try. For example from (0,0) to (2, 1) or to (1, 2).

6 The knight can only visit each square once.

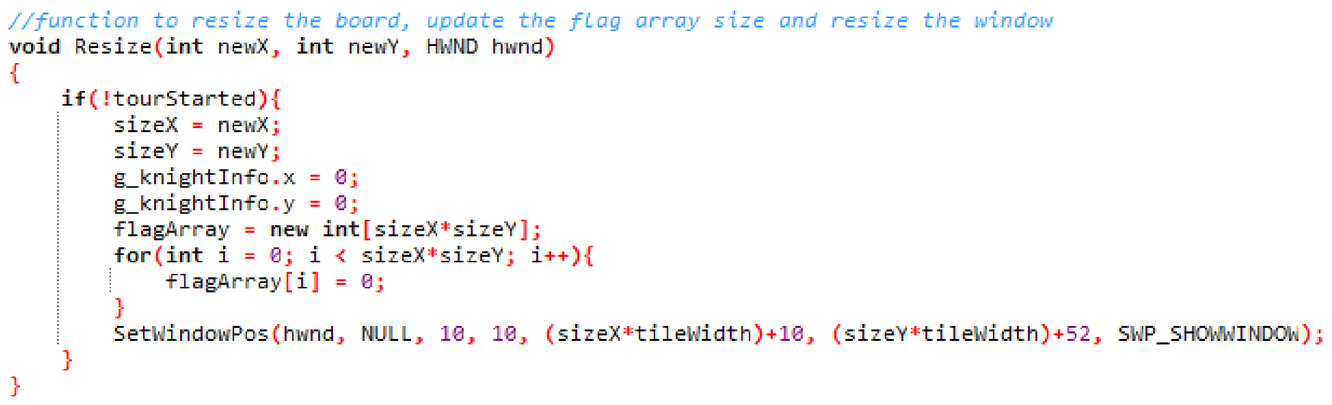
To make sure that the knight does not move out of bounds or to a tile that has already been visited I have a function called isMovePossible. This function takes one chess_move (as defined earlier) and sees if that move is possible.

```
//function to check if a move would place the knight out of bounds or on a square that has already been visited
bool isMovePossible(chess_moves next_move) {
    int i = next_move.x;
    int j = next_move.y;
    if ((i >= 0 && i < sizeX) && (j >= 0 && j < sizeY) && (flagArray[i*sizeY+j] == 0) ){
        return true;
    }
    return false;
}
```

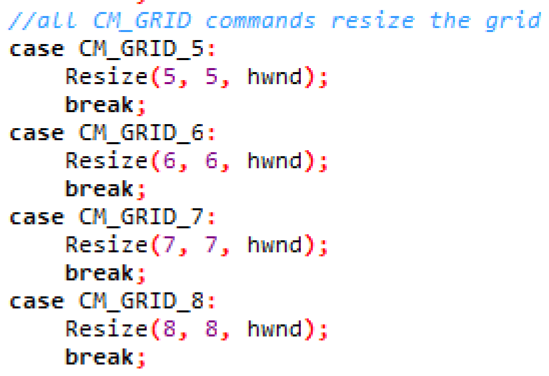
As seen in the function above i and j represent the x and y of the move. The function then checks if i is larger or equal than zero but lower than the width of the board. And if j is larger or equal than zero but lower than the height of the board. It then checks if the tile has not been visited before (flagArray[i*sizeY+j) == 0, if all of these factors are true, the function returns true because the move is possible. If any of the requirements are not met the if statement is skipped and return false is called.

Exemplar

1 Board size can be set by the user.



```
//function to resize the board, update the flag array size and resize the window
void Resize(int newX, int newY, HWND hwnd)
{
    if(!tourStarted){
        sizeX = newX;
        sizeY = newY;
        g_knightInfo.x = 0;
        g_knightInfo.y = 0;
        flagArray = new int[sizeX*sizeY];
        for(int i = 0; i < sizeX*sizeY; i++){
            flagArray[i] = 0;
        }
        SetWindowPos(hwnd, NULL, 10, 10, (sizeX*tileWidth)+10, (sizeY*tileWidth)+52, SWP_SHOWWINDOW);
    }
}
```

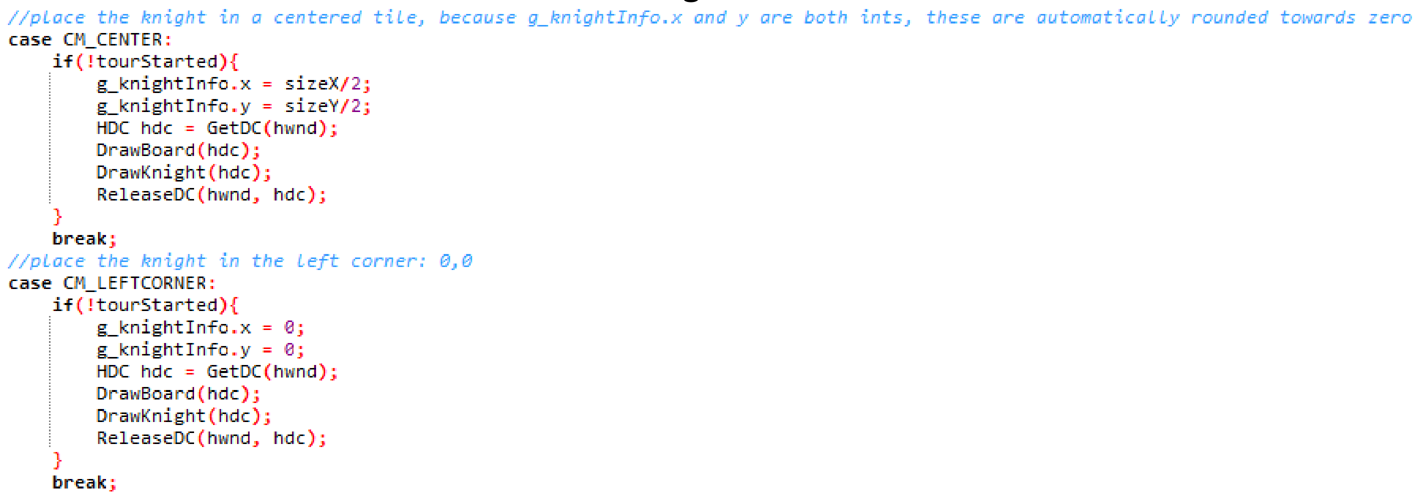


```
//all CM_GRID commands resize the grid
case CM_GRID_5:
    Resize(5, 5, hwnd);
    break;
case CM_GRID_6:
    Resize(6, 6, hwnd);
    break;
case CM_GRID_7:
    Resize(7, 7, hwnd);
    break;
case CM_GRID_8:
    Resize(8, 8, hwnd);
    break;
```

The user can select the board size from a drop down menu. This menu supports 5x5, 6x6, 7x7, 8x8, 9x9, 10x10, 15x15 and 16x8 board sizes. More options are possible but I did not find a good way to implement user input for text/numbers, so a dropdown menu was my best solution.

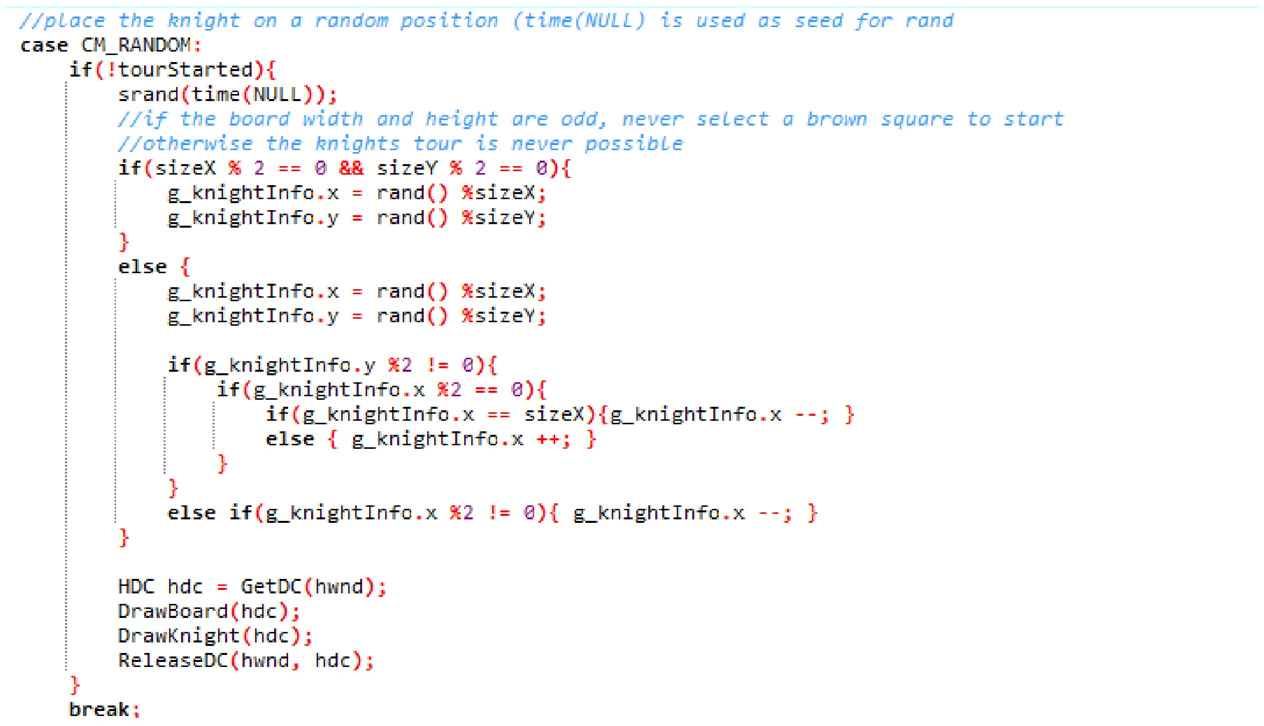
When the user selects for example 5x5, CM_GRID_5 is called in the events switch statement. This will call the Resize function and pass along 5 (width), 5 (height) and hwnd (the current window). The resize function checks if the tour is not in progress (!tourStarted) and sets the height and width of the board to the new variables. It also resets the position of the knight to 0, 0 (because when the knight is at (6,7) on an 8x8 board, he would not be in the bounds of the board on a 5x5 board for example). The flag array also has to be set to the new size and filled with 0s again. Finally it calls SetWindowPos to resize the window (this also automatically calls the WM_PAINT event which draws the board and the knight in their new sizes and positions.

2 The user can decide where the knight starts.



```
//place the knight in a centered tile, because g_knightInfo.x and y are both ints, these are automatically rounded towards zero
case CM_CENTER:
    if(!tourStarted){
        g_knightInfo.x = sizeX/2;
        g_knightInfo.y = sizeY/2;
        HDC hdc = GetDC(hwnd);
        DrawBoard(hdc);
        DrawKnight(hdc);
        ReleaseDC(hwnd, hdc);
    }
    break;
//place the knight in the left corner: 0,0
case CM_LEFTCORNER:
    if(!tourStarted){
        g_knightInfo.x = 0;
        g_knightInfo.y = 0;
        HDC hdc = GetDC(hwnd);
        DrawBoard(hdc);
        DrawKnight(hdc);
        ReleaseDC(hwnd, hdc);
    }
    break;
```

As mentioned above I did not find a good way to implement user input for text/numbers. Because of this a dropdown menu had to do for repositioning the knight. I decided to give the user 3 choices for the position of the knight. Top left (0,0), center (sizeX/2, sizeY/2) and random.



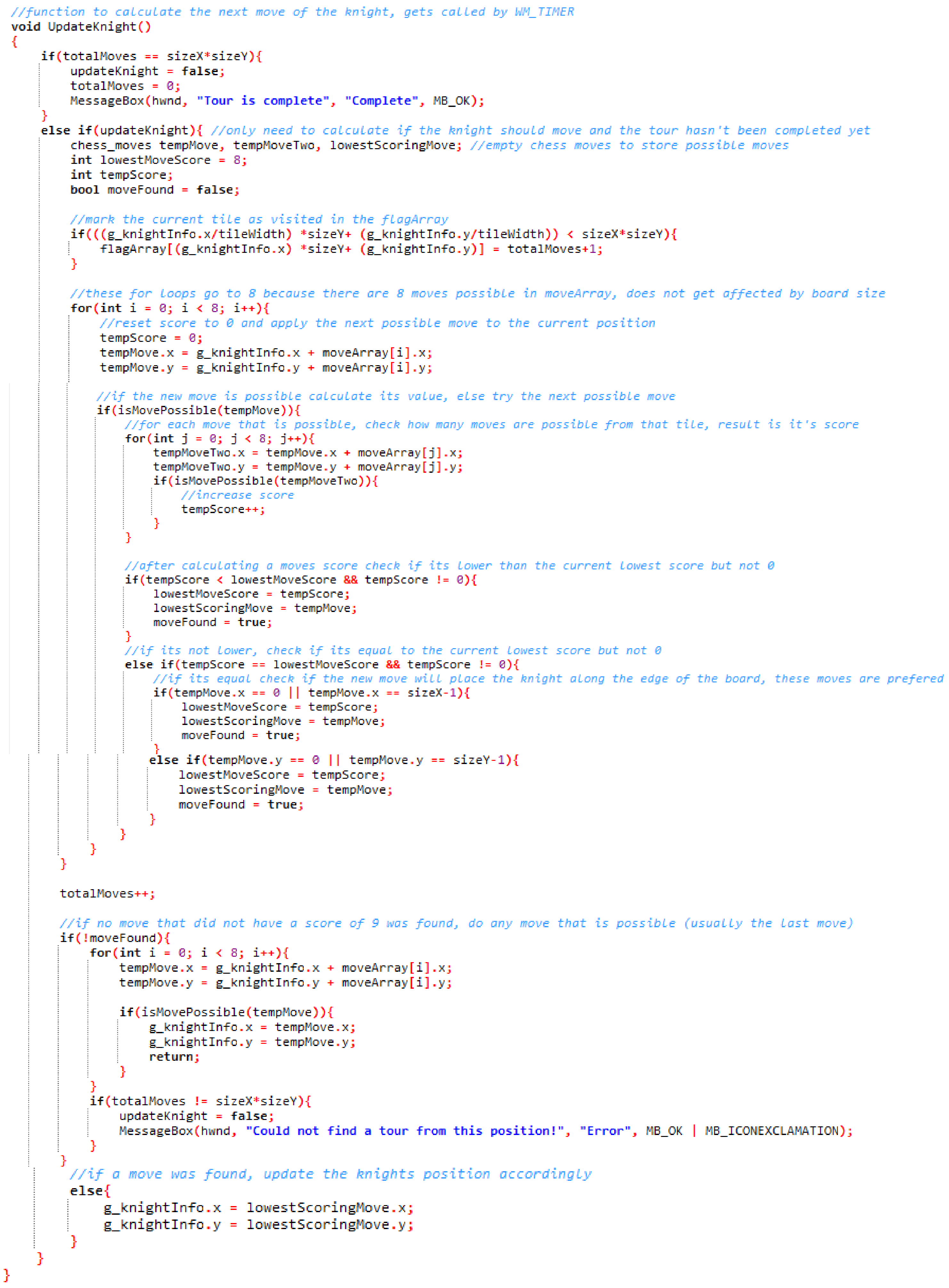
```
//place the knight on a random position (time(NULL) is used as seed for rand
case CM_RANDOM:
    if(!tourStarted){
        srand(time(NULL));
        //if the board width and height are odd, never select a brown square to start
        //otherwise the knights tour is never possible
        if(sizeX % 2 == 0 && sizeY % 2 == 0){
            g_knightInfo.x = rand() %sizeX;
            g_knightInfo.y = rand() %sizeY;
        }
        else {
            g_knightInfo.x = rand() %sizeX;
            g_knightInfo.y = rand() %sizeY;

            if(g_knightInfo.y %2 != 0){
                if(g_knightInfo.x %2 == 0){
                    if(g_knightInfo.x == sizeX){g_knightInfo.x --; }
                    else { g_knightInfo.x ++; }
                }
            }
            else if(g_knightInfo.x %2 != 0){ g_knightInfo.x --; }
        }

        HDC hdc = GetDC(hwnd);
        DrawBoard(hdc);
        DrawKnight(hdc);
        ReleaseDC(hwnd, hdc);
    }
    break;
```

This event randomizes the position of the knight using time(NULL) as a seed for the random numbers. It first checks if the tour has not started yet, then it checks if the board is even or odd. If the board is even the position can be any tile, if the board is odd the tour can only start from a white tile. This is because it is mathematically impossible to complete a knight’s tour on a NxN board where N is odd from a black tile.

3 Uses a more efficient solution than brute force.

I used Warnsdorff’s rule to solve the knight’s tour. Along with an amendment to the rule which says the knight should prefer tiles further away from the middle in the case of a tie. Warnsdorff’s rule is that the knight should go to the square that has the least possible moves available. See the full code on the next page.

What the code boils down to is a for loop that goes up to 8 and checks every possible move a knight can make from the tile the knight is currently on. For each move it checks if that move is a valid move. If the move is valid it loops trough another for loop of 8 for that move and checks how many moves are possible from that tile, each move that is possible increases the score of that tile by 1. After the second for loop, or if the move is not possible at all it checks if the moveScore is not 0 and lower than the current best. If it isn’t lower, it checks if it’s a tie. If it is a tie my algorithm just checks if the new move is on any of the furthest edges. This implementation works and I was not in the mood to make a check that actually checks for distance from the middle for both moves.



```
//function to calculate the next move of the knight, gets called by WM_TIMER
void UpdateKnight()
{
    if(totalMoves == sizeX*sizeY){
        updateKnight = false;
        totalMoves = 0;
        MessageBox(hwnd, "Tour is complete", "Complete", MB_OK);
    }
    else if(updateKnight){ //only need to calculate if the knight should move and the tour hasn't been completed yet
        chess_moves tempMove, tempMoveTwo, lowestScoringMove; //empty chess moves to store possible moves
        int lowestMoveScore = 8;
        int tempScore;
        bool moveFound = false;

        //mark the current tile as visited in the flagArray
        if(((g_knightInfo.x/tileWidth) *sizeY+ (g_knightInfo.y/tileWidth)) < sizeX*sizeY){
            flagArray[(g_knightInfo.x) *sizeY+ (g_knightInfo.y)] = totalMoves+1;
        }

        //these for loops go to 8 because there are 8 moves possible in moveArray, does not get affected by board size
        for(int i = 0; i < 8; i++){
            //reset score to 0 and apply the next possible move to the current position
            tempScore = 0;
            tempMove.x = g_knightInfo.x + moveArray[i].x;
            tempMove.y = g_knightInfo.y + moveArray[i].y;

            //if the new move is possible calculate its value, else try the next possible move
            if(isMovePossible(tempMove)){
                //for each move that is possible, check how many moves are possible from that tile, result is it's score
                for(int j = 0; j < 8; j++){
                    tempMoveTwo.x = tempMove.x + moveArray[j].x;
                    tempMoveTwo.y = tempMove.y + moveArray[j].y;
                    if(isMovePossible(tempMoveTwo)){
                        //increase score
                        tempScore++;
                    }
                }

                //after calculating a moves score check if its lower than the current lowest score but not 0
                if(tempScore < lowestMoveScore && tempScore != 0){
                    lowestMoveScore = tempScore;
                    lowestScoringMove = tempMove;
                    moveFound = true;
                }
                //if its not lower, check if its equal to the current lowest score but not 0
                else if(tempScore == lowestMoveScore && tempScore != 0){
                    //if its equal check if the new move will place the knight along the edge of the board, these moves are prefered
                    if(tempMove.x == 0 || tempMove.x == sizeX-1){
                        lowestMoveScore = tempScore;
                        lowestScoringMove = tempMove;
                        moveFound = true;
                    }
                    else if(tempMove.y == 0 || tempMove.y == sizeY-1){
                        lowestMoveScore = tempScore;
                        lowestScoringMove = tempMove;
                        moveFound = true;
                    }
                }
            }
        }

        totalMoves++;

        //if no move that did not have a score of 9 was found, do any move that is possible (usually the last move)
        if(!moveFound){
            for(int i = 0; i < 8; i++){
                tempMove.x = g_knightInfo.x + moveArray[i].x;
                tempMove.y = g_knightInfo.y + moveArray[i].y;

                if(isMovePossible(tempMove)){
                    g_knightInfo.x = tempMove.x;
                    g_knightInfo.y = tempMove.y;
                    return;
                }
            }
            if(totalMoves != sizeX*sizeY){
                updateKnight = false;
                MessageBox(hwnd, "Could not find a tour from this position!", "Error", MB_OK | MB_ICONEXCLAMATION);
            }
        }
        //if a move was found, update the knights position accordingly
        else{
            g_knightInfo.x = lowestScoringMove.x;
            g_knightInfo.y = lowestScoringMove.y;
        }
    }
}
```