Guide to Programming a Chess Engine

This document is a product of a rather rash decision in mid 2008 to learn to program my own Chess Game, hence began my journey into the art of computer chess. Over the last 2 years I have made some significant progress. I have learned quite a bit about the art of Computer Chess and amazingly I managed to build a reasonable strong chess engine. This Document is dedicated to recording the development of my Chess Game as well as the field of **Computer Chess** in general

Table of Contents

Project Goals

Goal 1

Create a chess game that was capable of consistently winning games against me.
Status: Achieved as of August 18, 2008.

Goal 2

Defeat Chess Titans at level 10, the free Chess Game that comes with Windows Vista.

Status: Achieved as of November 7th, 2009 although not consistently

Goal 3

Interface with WinBoard, the open source chess board. This is necessary to submit my Chess Engine to most Computer Chess tournaments.
Status: Achieved as of December 18th 2009:

Goal 4

Enter a computer chess tournament and beat someone's chess engine.
Status: Acheived as of April 5th 2010, Entered WBEC Ridderkerk's 18th Edition Tournament and beat more than one chess engine.

Goal 5

Defeat Little Chess Partner , the free Java Chess Game from the nice people at Lokasoft.
Status: October 28th 2008, tied based on three move repetition

Goal 6

Create Mobile Chess Game (iOS & Android)
Status: Acheived as of October 20th 2016,

Choice of Programming Language

For my implementation of the Chess Game I selected C# mainly because it is a computer language that:

- I know best and work with every day.
- I would like to improve my skills in.
- Should be around for a while giving the project a longer life.

There are however negative aspects to my choice. C# cannot compete with C or assembly language in terms of performance. This will means that a chess engine created in C# will probably be slightly slower in move searches compared to the same chess engine created in a programming language that compiles into machine code.

I have read sites that claim that the .NET framework is just as efficient and quick as unmanaged C++. However I am skeptical at best. I have seen a managed C# program choke on things that C++ would breeze through. Maybe some of this has to do with implementation or optimization. However I am a firm believer that optimized C++ unmanaged executable will usually be faster then optimized C# managed executable.

As proof I would like to submit the fact that it has been years since the introduction of C# and yet the Windows OS is still entirely written in unmanaged code… even notepad.

However I stated before my goal is not to make the best Chess Engine that ever existed, just a fairly good one. I believe the C# programming language will serve that purpose.

Chess Piece Representation

The first 2 tasks in creating a chess engine are the description of the Chess Board and the Chess Pieces. This page I will discuss my C# representation of the Chess Piece.
Throughout the chess engine many decisions will have to be made based on 2 concepts. The chess piece type and it's color. For this purpose I have declared two enumerated types, chess piece color and chess piece type:

```
public enum ChessPieceColor
{
    White,
    Black
}

public enum ChessPieceType
{
    King,
    Queen,
    Rook,
    Bishop,
    Knight,
    Pawn,
    None
}
```

Originally these two enumerated types were located inside the Chess Piece class, however I later realized that these constructs must be available to assemblies outside of the chess engine, and therefore must be made public. Since I found that internal classes perform faster than public classes, the only way to make my Chess Piece class internal is to remove the enumerated types out of the Chess Piece class.
The next concept that will be required is the idea of a Chess Board position. This will be needed in order to list valid moves for a chess piece inside the Chess Piece class. In the current version of my chess engine, board position coordinates are stored as a single byte, 0 for A8 and 63 for A1.
Originally I stored chess board positions as a set of two bytes. The first byte was used to represent the column and second byte for the row. This made my chess engine easier to understand but cost me approximately 30% more in performance.
The class representing my Chess Pieces will be declared as internal sealed:

```
internal sealed class Piece
```

During my performance testing I found that the sealed and internal keywords improved speed significantly. I also found that private methods and objects perform much faster so I strongly suggest using the private keyword as much as possible.
To make descriptions of my chess pieces easier and more strongly typed, I will be using the above defined enumerated types as representations of the chess piece color and type.

```
internal ChessPieceColor PieceColor;
internal ChessPieceType PieceType;
```

The next line of code will describe the piece value used in the evaluation of positions. Obviously each piece type will have a different value; a Queen is worth more than a pawn etc.

```
internal short PieceValue;
```

In order to keep track the value of pieces that are currently attacking and defending this chess piece we will declare the following 2 variables:

```
internal short AttackedValue;
internal short DefendedValue;
```

Each piece will have what I call the Piece Action Value, this is the value added or subtracted from the score when the chess piece is either attacking or defending another chess piece. Different chess piece types will have different action values. This follows the logic that it is better to risk a pawn than it is to risk a queen.

```
internal short PieceActionValue;
```

The next line describes if the chess piece is currently selected on the board by the user. This will help later when coding a graphical user interface.

```
internal bool Selected;
```

The following variable desribes if our chess piece has been moved. This will be very usefull in the evaluation function where we can give penalties or bonueses if the chess piece has not yet made a move.

```
internal bool Moved;
```

Each piece will also contain a list of valid moves, or board positions that the piece can move too.

```
internal Stack<byte> ValidMoves;
```

Constructors

There are 2 constructors in the Chess Piece class

Copy Constructor, which is needed during move generation.

```
internal Piece(Piece piece)
{
    PieceColor = piece.PieceColor;
    PieceType = piece.PieceType;
    Moved = piece.Moved;
    PieceValue = piece.PieceValue;

    if (piece.ValidMoves != null)
        LastValidMoveCount = piece.ValidMoves.Count;
}
```

Constructor used to initiate the chess board with chess pieces.

```
internal Piece(ChessPieceType chessPiece, ChessPieceColor chessPieceColor)
{
    PieceType = chessPiece;
    PieceColor = chessPieceColor;

    ValidMoves = new Stack<byte>();
    PieceValue = CalculatePieceValue(PieceType);
```

```
}
```

Methods

Calculate Piece Value and is used during object construction to calculate and record the chess piece's value.

```
private static short CalculatePieceValue(ChessPieceType pieceType)
{
    switch (pieceType)
    {
        case ChessPieceType.Pawn:
            {
                return 100;

            }
        case ChessPieceType.Knight:
            {
                return 320;
            }
        case ChessPieceType.Bishop:
            {
                return 325;
            }
        case ChessPieceType.Rook:
            {
                return 500;
            }

        case ChessPieceType.Queen:
            {
                return 975;
            }
        case ChessPieceType.King:
            {
                return 32767;
            }
        default:
            {
                return 0;
            }
    }
}
```

The Piece Action value, the added or subtracted from the score when the chess piece is either attacking or defending another chess piece is also calculated on construction, using the following method:

```
private static short CalculatePieceActionValue(ChessPieceType pieceType)
{
    switch (pieceType)
    {
```

```
            case ChessPieceType.Pawn:
                {
                    return 6;
                }
            case ChessPieceType.Knight:
                {
                    return 3;
                }
            case ChessPieceType.Bishop:
                {
                    return 3;
                }
            case ChessPieceType.Rook:
                {
                    return 2;
                }
            case ChessPieceType.Queen:
                {
                    return 1;
                }
            case ChessPieceType.King:
                {
                    return 1;
                }
            default:
                {
                    return 0;
                }
        }
}
```

Note that the above method is designed to encourage the computer to protect and attack using the lower valued pieces first.

Chess Board Square

In this post I will discuss the chess board square representation. Before we can discuss the chess board we need to model how each of the chess board squares will be represented on our board.
The chess board square will be declared as an internal struct. I have found that small structs seem to perform better then small classes. Also making objects internal or even better private, tends to increase performance.

```
internal struct Square
```

Each chess board square can contain a chess piece.

```
internal Piece Piece;
```

Each Board Square will also have the following copy constructor that will copy the chess piece from the copied chess board square or set the chess piece to null, signifying that the current square is empty.

```
internal Square(Piece piece)
{
    Piece = new Piece(piece);
}
```

Chess Board Representation

Prior to Reading this post I suggest reviewing the pages explaining the Board Square and Chess Piece classes.
The chess board class is again declared as internal sealed to improve performance.

```
internal sealed class Board
```

Our chess board will contain 64 board squares represented by an array of [64] items.

Originally I used a multidimensional array [][]. This way I can reference board position by columns and rows. Although this made my code easier to understand, it also made move searching approximately 30%

```
internal Square[] Squares;
```

At this point I would like to explain some simple concepts related to how we represent a chess board using the above 64 item array of board squares. Array item 0 will represent the top left most square on the board (A8). Array item 63 will represent the bottom right most square on the board, H1.

```
0  1  2  3  4  5  6  7
8  9  10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
```

```
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
```

When dealing with a single index to reference chess board positions there are certain things that one must know to make life easier. For example how do you know that two positions are both on the same row or column? There is an easy trick to figure that out.

Row

To figure out the row of a position you divide the position by 8 and take the integer portion of the result. For example position 63 divided by 8 is 7.875 which equals row 7. Position 3 divided by 8 is 0.375 so 0. In C# by casting to an integer you will always get just the integer portion of the number, hence:

```
Row = (int)(position / 8)
```

Column

To figure out the column of a position you use the modulus operator by performing position modulus 8. For example position 24 modulus 8 is column 0. Position 15 modulus 8 is 7, hence

```
Column = position % 8
```

Armed with these two concepts we can convert any position on our 64 square board to a column and row.

Properties

The next property is the Board Score. This is implemented as an internal integer. The score works by increasing better positions for White and decreasing for better positions for Black. Hence in our search methods Black is always trying to find boards with the lowest score and White with the highest.

```
internal int Score;
```

The next set of properties that contain information related to king checks and mates. True if white king is in check, false if not etc.

```
internal bool BlackCheck;
internal bool BlackMate;
internal bool WhiteCheck;
internal bool WhiteMate;
internal bool StaleMate;
```

The next two variables are counters that allow us to keep track of the two tie scenarios related to the 50 move rule and the 3 move repetitions rule. If the fifty move count reaches 50 or repeat move count reaches 3 we know that a tie has occurred.

```
internal byte FiftyMove;
internal byte RepeatedMove;
```

The two following flags are used to track if any of the two sides have castled. This information is needed for the evaluation function to give bonus scores for castling and the move generator to allow for castling to occur if the circumstance is correct.

```
internal bool BlackCastled;
internal bool WhiteCastled;
```

The next flag tracks if the board is in the middle game or end game state. This is determined later on by the amount of pieces remaining on the board in the Evaluation Function. If the chess board is in an end game state certain behaviors will be modified to increase king safety and mate opportunities.

```
internal bool EndGamePhase;
```

The board will also keep track of the last move that occurred. This is implemented as a Move Content class which we will discuss later.

```
internal MoveContent LastMove;
```

The next flags relate to the EnPassant rule, which was actually a bit of a pain to implement. For now all we need to know is that our board will contain 2 pieces of information related to EnPassant.

1. Which side has last made a move that can cause an EnPassant (Which side moved the pawn 2 spots).

```
internal ChessPieceColor EnPassantColor;
```

2. The Board Square of the EnPassant position, which is the position directly behind the pawn that moved 2 spots.

```
internal byte EnPassantPosition;
```

The Board will keep track of whose move it is

```
internal ChessPieceColor WhosMove;
```

As well as how many moves have occurred.

```
internal int MoveCount;
```

Constructors

The Board class with have 4 constructors as follows:
Default Constructor:

```
internal Board()
{
    Squares = new Square[64];
```

```
    for (byte i = 0; i < 64; i++)
    {
        Squares[i] = new Square();
    }
    LastMove = new MoveContent();
}
```

Copy Constructor:

```
internal Board(Board board)
{
    Squares = new Square[64];

    for (byte x = 0; x < 64; x++)
    {
        if (board.Squares[x].Piece != null)
        {
            Squares[x] = new Square(board.Squares[x].Piece);
        }
    }
    EndGamePhase = board.EndGamePhase;
    FiftyMove = board.FiftyMove;
    RepeatedMove = board.RepeatedMove;
    WhiteCastled = board.WhiteCastled;
    BlackCastled = board.BlackCastled;
    BlackCheck = board.BlackCheck;
    WhiteCheck = board.WhiteCheck;
    StaleMate = board.StaleMate;
    WhiteMate = board.WhiteMate;
    BlackMate = board.BlackMate;
    WhosMove = board.WhosMove;
    EnPassantPosition = board.EnPassantPosition;
    EnPassantColor = board.EnPassantColor;
    Score = board.Score;
    LastMove = new MoveContent(board.LastMove);
    MoveCount = board.MoveCount;
}
```

Constructor that allows to pass in the default Score. This is useful during move searching where we can initially construct the best Board we found so far to something ridiculous like int.MinValue

```
internal Board(int score) : this()
{
    Score = score;
}
```

Constructor that will accept an array of Board Squares

```
private Board(Square[] squares)
```

```
{
    Squares = new Square[64];

    for (byte x = 0; x < 64; x++)
    {
        if (squares[x].Piece != null)
        {
            Squares[x].Piece = new Piece(squares[x].Piece);
        }
    }
}
```

As you may have noticed above the copy constructor is actually quite meaty. There are too many fields to copy and this has a performance impact during move generation. For this reason I created another method called Fast Copy. The idea here is that during move generation some fields will get overwritten anyways, so I don't really care what the previous values of these fields were. The Fast Copy method will copy only the values that must persist from one board to another during move generation.

```
internal Board FastCopy()
{
    Board clonedBoard = new Board(Squares);

    clonedBoard.EndGamePhase = EndGamePhase;
    clonedBoard.WhoseMove = WhoseMove;
    clonedBoard.MoveCount = MoveCount;
    clonedBoard.FiftyMove = FiftyMove;
    clonedBoard.BlackCastled = BlackCastled;
    clonedBoard.WhiteCastled = WhiteCastled;
    return clonedBoard;
}
```

Board Movement

The following listings are a set of methods that will help us with chess piece movement on our board. Before we can actually write the main movement method, we need to handle all of the special scenarios such as pawn promotion, en passant and castling. These helper methods basically have a set of hard coded positions and some logic that states, if I am in this position and this piece type, do something different. Else the move will be handled by the main move method.

Pawn Promotion

The Promote Pawns method will check for the destination position of the pawn and promote it to a Queen Piece. Most Chess programs allow the user to choose the piece they promote the pawn too; however in most cases I don't see why you would not choose a queen anyways. Furthermore choosing the queen always simplifies the implementation for now.

```
private static bool PromotePawns(Board board, Piece piece, byte dstPosition,
                    ChessPieceType promoteToPiece)
{
    if (piece.PieceType == ChessPieceType.Pawn)
    {
        if (dstPosition < 8)
```

```
        {
            board.Squares[dstPosition].Piece.PieceType = promoteToPiece;
            return true;
        }
        if (dstPosition > 55)
        {
            board.Squares[dstPosition].Piece.PieceType = promoteToPiece;
            return true;
        }
    }

    return false;
}
```

En Passant

The Record En Passant method sets the En Passant flag if the piece currently moving is a pawn that moves 2 squares.

```
private static void RecordEnPassant(ChessPieceColor pcColor, ChessPieceType pcType,
        Board board, byte srcPosition, byte dstPosition)
{
    //Record En Passant if Pawn Moving
    if (pcType == ChessPieceType.Pawn)
    {
        //Reset FiftyMoveCount if pawn moved
        board.FiftyMove = 0;

        int difference = srcPosition - dstPosition;
        if (difference == 16 || difference == -16)
        {
            board.EnPassantPosition = (byte)(dstPosition + (difference / 2));
            board.EnPassantColor = pcColor;
        }
    }
}
```

Set En Passant Move Method will move the En Passant piece and kill the advanced pawn based on the En Passant flags of the board and the destination move requested.

```
private static bool SetEnpassantMove(Board board, byte dstPosition,
                    ChessPieceColor pcColor)
{
    //En Passant
    if (board.EnPassantPosition == dstPosition)
    {
        //We have an En Passant Possible
        if (pcColor != board.EnPassantColor)
        {
            int pieceLocationOffset = 8;

            if (board.EnPassantColor == ChessPieceColor.White)
            {
                pieceLocationOffset = -8;
            }
```

```
            dstPosition = (byte)(dstPosition + pieceLocationOffset);
            Square sqr = board.Squares[dstPosition];
            board.LastMove.TakenPiece =
                new PieceTaken(sqr.Piece.PieceColor, sqr.Piece.PieceType,
                        sqr.Piece.Moved, dstPosition);
            board.Squares[dstPosition].Piece = null;

            //Reset FiftyMoveCount if capture
            board.FiftyMove = 0;
            return true;
        }
    }
    return false;
}
```

Castling

The next Method will move the Rook to its correct position if castling is requested.

```
private static void KingCastle(Board board, Piece piece,
         byte srcPosition, byte dstPosition)
{
    if (piece.PieceType != ChessPieceType.King)
    {
        return;
    }

    //Lets see if this is a casteling move.
    if (piece.PieceColor == ChessPieceColor.White &&
           srcPosition == 60)
    {
        //Castle Right          if (dstPosition == 62)
        {
            //Ok we are casteling we need to move the Rook
            if (board.Squares[63].Piece != null)
            {
                board.Squares[61].Piece = board.Squares[63].Piece;
                board.Squares[63].Piece = null;
                board.WhiteCastled = true;
                board.LastMove.MovingPieceSecondary =
                 new PieceMoving(board.Squares[61].Piece.PieceColor,
                     board.Squares[61].Piece.PieceType,
                          board.Squares[61].Piece.Moved, 63, 61);
                board.Squares[61].Piece.Moved = true;
                return;
            }
        }
        //Castle Left
        else if (dstPosition == 58)
        {
            //Ok we are casteling we need to move the Rook
            if (board.Squares[56].Piece != null)
            {
                board.Squares[59].Piece = board.Squares[56].Piece;
                board.Squares[56].Piece = null;
                board.WhiteCastled = true;
```

```
                board.LastMove.MovingPieceSecondary =
                     new PieceMoving(board.Squares[59].Piece.PieceColor,
                            board.Squares[59].Piece.PieceType,
                                 board.Squares[59].Piece.Moved, 56, 59);
                board.Squares[59].Piece.Moved = true;
                return;
            }
        }
    }
    else if (piece.PieceColor == ChessPieceColor.Black &&
           srcPosition == 4)
    {
        if (dstPosition == 6)
        {
            //Ok we are casteling we need to move the Rook
            if (board.Squares[7].Piece != null)
            {
                board.Squares[5].Piece = board.Squares[7].Piece;
                board.Squares[7].Piece = null;
                board.BlackCastled = true;
                board.LastMove.MovingPieceSecondary =
                    new PieceMoving(board.Squares[5].Piece.PieceColor,
                             board.Squares[5].Piece.PieceType,
                                  board.Squares[5].Piece.Moved, 7, 5);
                board.Squares[5].Piece.Moved = true;
                return;
            }
        }
        //Castle Left
        else if (dstPosition == 2)
        {
            //Ok we are casteling we need to move the Rook
            if (board.Squares[0].Piece != null)
            {
                board.Squares[3].Piece = board.Squares[0].Piece;
                board.Squares[0].Piece = null;
                board.BlackCastled = true;
                board.LastMove.MovingPieceSecondary =
                   new PieceMoving(board.Squares[3].Piece.PieceColor,
                             board.Squares[3].Piece.PieceType,
                                  board.Squares[3].Piece.Moved, 0, 3);
                board.Squares[3].Piece.Moved = true;
                return;
            }
        }
    }
    return;
}
```

This is the actual Move Method, where each piece is moved, captured. The logic here basically boils down to, recording the move, and assigning the moving piece to the new square, while clearing the old one. This method also calls the helper movement methods we have just listed above to handle the more complex scenarios such as castling, pawn promotion and En Passant.

```
internal static MoveContent MovePiece(Board board, byte srcPosition,
                    byte dstPosition,
                    ChessPieceType promoteToPiece)
```

```
{
    Piece piece = board.Squares[srcPosition].Piece;

    //Record my last move
    board.LastMove = new MoveContent();
    //Add One to FiftyMoveCount to check for tie.
    board.FiftyMove++;

    if (piece.PieceColor == ChessPieceColor.Black)
    {
        board.MoveCount++;
    }
    //En Passant
    if (board.EnPassantPosition > 0)
    {
        board.LastMove.EnPassantOccured =
              SetEnpassantMove(board, dstPosition, piece.PieceColor);
    }
    if (!board.LastMove.EnPassantOccured)
    {
        Square sqr = board.Squares[dstPosition];
        if (sqr.Piece != null)
        {
            board.LastMove.TakenPiece =
                  new PieceTaken(sqr.Piece.PieceColor, sqr.Piece.PieceType,
                                       sqr.Piece.Moved, dstPosition);
            board.FiftyMove = 0;
        }
        else
        {
            board.LastMove.TakenPiece =
                  new PieceTaken(ChessPieceColor.White, ChessPieceType.None,
                          false, dstPosition);

        }
    }
    board.LastMove.MovingPiecePrimary =
              new PieceMoving(piece.PieceColor, piece.PieceType,
                      piece.Moved, srcPosition, dstPosition);
    //Delete the piece in its source position
    board.Squares[srcPosition].Piece = null;
    //Add the piece to its new position
    piece.Moved = true;
    piece.Selected = false;
    board.Squares[dstPosition].Piece = piece;
    //Reset EnPassantPosition     board.EnPassantPosition = 0;

    //Record En Passant if Pawn Moving
    if (piece.PieceType == ChessPieceType.Pawn)
    {
       board.FiftyMove = 0;
       RecordEnPassant(piece.PieceColor, piece.PieceType,
                board, srcPosition, dstPosition);
    }
    board.WhoseMove = board.WhoseMove == ChessPieceColor.White
              ? ChessPieceColor.Black
              : ChessPieceColor.White;
    KingCastle(board, piece, srcPosition, dstPosition);
```

```
    //Promote Pawns
    if (PromotePawns(board, piece, dstPosition, promoteToPiece))
    {
        board.LastMove.PawnPromoted = true;
    }
    else
    {
        board.LastMove.PawnPromoted = false;
    }
    if ( board.FiftyMove >= 50)
    {
        board.StaleMate = true;
    }
    return board.LastMove;
}
```

If you compile this listing along with the Chess Piece, Move Content and Board Square classes you should have all the necessary code for declaring and moving pieces around the board. Of course you still don't have a graphical chess board or the move generator.

Chess Piece Moves

This post at one point discussed the Chess Piece Motion Class. I have since then divided the code from this class into two separate classes.

Piece Moves
Piece Valid Moves

This post will discuss Piece Moves class. This class is responsible for providing all available chess piece moves regardless of the state of the chess board. The information stored in this class will not change throughout the game play so it is static and calculated only once before the game starts. Having a set of possible moves for any chess piece at any position allows us to later to generate only the valid moves for each chess piece based on the current state of the board.

The Chess Piece Moves listing will contain a Valid Move Set struct. This struct will be used to store a set of moves available from a single position.

```
internal struct PieceMoveSet
{
    internal readonly List<byte> Moves;

    internal PieceMoveSet(List<byte> moves)
    {
        Moves = moves;
    }
}
```

Furthermore we will need some additional array to store all the above move sets for every position on the board.

For example KnightMoves[0]. Moves will return a Knight Moves available from position 0 or A8. KnightMoves[63] will return all of the possible moves for position 63 or H1.
Some chess pieces can move in a single direction for an undefined number of squares until they reach the end of the board or another chess piece. For this purpose moves sets for some pieces are divided into several arrays, each describing a move in a certain direction. This makes it easier to manage these movements in the Chess Piece Valid Moves Class by having the ability to loop through each array until a chess piece or the end of the board is reached and no further.
One other explanation is required around the Total Moves arrays. Example there is an array called KnightTotalMoves. This array will hold the number of moves available for every position on the chess board. This is a performance related addition as it allows me to replace all my foreach loops with regular for loops. It's a small performance gain (1%-2%) but they all add up.

```
internal struct MoveArrays
{
    internal static PieceMoveSet[] BishopMoves1;
    internal static byte[] BishopTotalMoves1;

    internal static PieceMoveSet[] BishopMoves2;
    internal static byte[] BishopTotalMoves2;

    internal static PieceMoveSet[] BishopMoves3;
    internal static byte[] BishopTotalMoves3;
    internal static PieceMoveSet[] BishopMoves4;
    internal static byte[] BishopTotalMoves4;
    internal static PieceMoveSet[] BlackPawnMoves;
    internal static byte[] BlackPawnTotalMoves;
    internal static PieceMoveSet[] WhitePawnMoves;
    internal static byte[] WhitePawnTotalMoves;
    internal static PieceMoveSet[] KnightMoves;
    internal static byte[] KnightTotalMoves;
    internal static PieceMoveSet[] QueenMoves1;
    internal static byte[] QueenTotalMoves1;
    internal static PieceMoveSet[] QueenMoves2;
    internal static byte[] QueenTotalMoves2;
    internal static PieceMoveSet[] QueenMoves3;
    internal static byte[] QueenTotalMoves3;
    internal static PieceMoveSet[] QueenMoves4;
    internal static byte[] QueenTotalMoves4;
    internal static PieceMoveSet[] QueenMoves5;
    internal static byte[] QueenTotalMoves5;
    internal static PieceMoveSet[] QueenMoves6;
    internal static byte[] QueenTotalMoves6;
    internal static PieceMoveSet[] QueenMoves7;
    internal static byte[] QueenTotalMoves7;
    internal static PieceMoveSet[] QueenMoves8;
    internal static byte[] QueenTotalMoves8;
    internal static PieceMoveSet[] RookMoves1;
```

```
    internal static byte[] RookTotalMoves1;
    internal static PieceMoveSet[] RookMoves2;
    internal static byte[] RookTotalMoves2;
    internal static PieceMoveSet[] RookMoves3;
    internal static byte[] RookTotalMoves3;
    internal static PieceMoveSet[] RookMoves4;
    internal static byte[] RookTotalMoves4;
    internal static PieceMoveSet[] KingMoves;
    internal static byte[] KingTotalMoves;
}
```

To make use of the above structs we will declare a static class called Piece Moves:

internal static class PieceMoves
To make life a bit easier, we will add a helper method called Position. This method will accept a chess board column and row, and return a single byte representing the chess board position. Usually we would not want to use a method like this because it will slow things down. However this method is only used when the Chess Engine starts when super fast performance is not really all that necessary.

```
private static byte Position(byte col, byte row)
{
    return (byte)(col + (row * 8));
}
```

Initiate Chess Piece Motion Class is called only once in the Chess Engine Constructor. It will construct all of the arrays and call the methods responsible for populating the Move Set arrays will all of the moves for each position on the board for each chess piece.

```
internal static void InitiateChessPieceMotion()
{
    MoveArrays.WhitePawnMoves = new PieceMoveSet[64];
    MoveArrays.WhitePawnTotalMoves = new byte[64];

    MoveArrays.BlackPawnMoves = new PieceMoveSet[64];
    MoveArrays.BlackPawnTotalMoves = new byte[64];
    MoveArrays.KnightMoves = new PieceMoveSet[64];
    MoveArrays.KnightTotalMoves = new byte[64];
    MoveArrays.BishopMoves1 = new PieceMoveSet[64];
    MoveArrays.BishopTotalMoves1 = new byte[64];
    MoveArrays.BishopMoves2 = new PieceMoveSet[64];
    MoveArrays.BishopTotalMoves2 = new byte[64];
    MoveArrays.BishopMoves3 = new PieceMoveSet[64];
    MoveArrays.BishopTotalMoves3 = new byte[64];
    MoveArrays.BishopMoves4 = new PieceMoveSet[64];
    MoveArrays.BishopTotalMoves4 = new byte[64];
    MoveArrays.RookMoves1 = new PieceMoveSet[64];
    MoveArrays.RookTotalMoves1 = new byte[64];
    MoveArrays.RookMoves2 = new PieceMoveSet[64];
    MoveArrays.RookTotalMoves2 = new byte[64];
    MoveArrays.RookMoves3 = new PieceMoveSet[64];
```

```
    MoveArrays.RookTotalMoves3 = new byte[64];
    MoveArrays.RookMoves4 = new PieceMoveSet[64];
    MoveArrays.RookTotalMoves4 = new byte[64];
    MoveArrays.QueenMoves1 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves1 = new byte[64];
    MoveArrays.QueenMoves2 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves2 = new byte[64];
    MoveArrays.QueenMoves3 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves3 = new byte[64];
    MoveArrays.QueenMoves4 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves4 = new byte[64];
    MoveArrays.QueenMoves5 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves5 = new byte[64];
    MoveArrays.QueenMoves6 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves6 = new byte[64];
    MoveArrays.QueenMoves7 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves7 = new byte[64];
    MoveArrays.QueenMoves8 = new PieceMoveSet[64];
    MoveArrays.QueenTotalMoves8 = new byte[64];
    MoveArrays.KingMoves = new PieceMoveSet[64];
    MoveArrays.KingTotalMoves = new byte[64];

    SetMovesWhitePawn();
    SetMovesBlackPawn();
    SetMovesKnight();
    SetMovesBishop();
    SetMovesRook();
    SetMovesQueen();
    SetMovesKing();
}
```

Set Moves methods are responsible for populating the Move Arrays with the moves available for each chess piece from a given position. I am not going to explain this much further. These methods basically add some predetermined positions to the arrays defined above. Again performance is not really all that key here, since these methods run only once when the Chess Engine starts.

```
private static void SetMovesBlackPawn()
{
    for (byte index = 8; index <= 55; index++)
    {
        var moveset = new PieceMoveSet(new List<byte>());

        byte x = (byte)(index % 8);
        byte y = (byte)((index / 8));

        //Diagonal Kill
        if (y < 7 && x < 7)
        {
            moveset.Moves.Add((byte)(index + 8 + 1));
            MoveArrays.BlackPawnTotalMoves[index]++;
        }
```

```
        if (x > 0 && y < 7)
        {
            moveset.Moves.Add((byte)(index + 8 - 1));
            MoveArrays.BlackPawnTotalMoves[index]++;
        }

        //One Forward
        moveset.Moves.Add((byte)(index + 8));
        MoveArrays.BlackPawnTotalMoves[index]++;

        //Starting Position we can jump 2          if (y == 1)
        {
            moveset.Moves.Add((byte)(index + 16));
            MoveArrays.BlackPawnTotalMoves[index]++;
        }
        MoveArrays.BlackPawnMoves[index] = moveset;
    }
}
private static void SetMovesWhitePawn()
{
    for (byte index = 8; index <= 55; index++)
    {
        byte x = (byte)(index % 8);
        byte y = (byte)((index / 8));
        var moveset = new PieceMoveSet(new List<byte>());

        //Diagonal Kill
        if (x < 7 && y > 0)
        {
            moveset.Moves.Add((byte)(index - 8 + 1));
            MoveArrays.WhitePawnTotalMoves[index]++;
        }
        if (x > 0 && y > 0)
        {
            moveset.Moves.Add((byte)(index - 8 - 1));
            MoveArrays.WhitePawnTotalMoves[index]++;
        }
        //One Forward
        moveset.Moves.Add((byte)(index - 8));
        MoveArrays.WhitePawnTotalMoves[index]++;
        //Starting Position we can jump 2
        if (y == 6)
        {
            moveset.Moves.Add((byte)(index - 16));
            MoveArrays.WhitePawnTotalMoves[index]++;
        }
        MoveArrays.WhitePawnMoves[index] = moveset;
    }
}
private static void SetMovesKnight()
{
```

```
for (byte y = 0; y < 8; y++)
{
    for (byte x = 0; x < 8; x++)
    {
        byte index = (byte)(y + (x * 8));
        var moveset = new PieceMoveSet(new List<byte>());

        byte move;
        if (y < 6 && x > 0)
        {
            move = Position((byte)(y + 2), (byte)(x - 1));
            if (move < 64)
            {
                moveset.Moves.Add(move);
                MoveArrays.KnightTotalMoves[index]++;
            }
        }
        if (y > 1 && x < 7)
        {
            move = Position((byte)(y - 2), (byte)(x + 1));
            if (move < 64)
            {
                moveset.Moves.Add(move);
                MoveArrays.KnightTotalMoves[index]++;
            }
        }
        if (y > 1 && x > 0)
        {
            move = Position((byte)(y - 2), (byte)(x - 1));
            if (move < 64)
            {
                moveset.Moves.Add(move);
                MoveArrays.KnightTotalMoves[index]++;
            }
        }
        if (y < 6 && x < 7)
        {
            move = Position((byte)(y + 2), (byte)(x + 1));
            if (move < 64)
            {
                moveset.Moves.Add(move);
                MoveArrays.KnightTotalMoves[index]++;
            }
        }
        if (y > 0 && x < 6)
        {
            move = Position((byte)(y - 1), (byte)(x + 2));
            if (move < 64)
            {
                moveset.Moves.Add(move);
                MoveArrays.KnightTotalMoves[index]++;
```

```
                }
            }
            if (y < 7 && x > 1)
            {
                move = Position((byte)(y + 1), (byte)(x - 2));
                if (move < 64)
                {
                    moveset.Moves.Add(move);
                    MoveArrays.KnightTotalMoves[index]++;
                }
            }
            if (y > 0 && x > 1)
            {
                move = Position((byte)(y - 1), (byte)(x - 2));
                if (move < 64)
                {
                    moveset.Moves.Add(move);
                    MoveArrays.KnightTotalMoves[index]++;
                }
            }

            if (y < 7 && x < 6)
            {
                move = Position((byte)(y + 1), (byte)(x + 2));
                if (move < 64)
                {
                    moveset.Moves.Add(move);
                    MoveArrays.KnightTotalMoves[index]++;
                }
            }
            MoveArrays.KnightMoves[index] = moveset;
        }
    }
}
private static void SetMovesBishop()
{
    for (byte y = 0; y < 8; y++)
    {
        for (byte x = 0; x < 8; x++)
        {
            byte index = (byte)(y + (x * 8));
            var moveset = new PieceMoveSet(new List<byte>());
            byte move;
            byte row = x;
            byte col = y;
            while (row < 7 && col < 7)
            {
                row++;
                col++;
                move = Position(col, row);
                moveset.Moves.Add(move);
```

```
                MoveArrays.BishopTotalMoves1[index]++;
            }
            MoveArrays.BishopMoves1[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row < 7 && col > 0)
            {
                row++;
                col--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.BishopTotalMoves2[index]++;
            }
            MoveArrays.BishopMoves2[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row > 0 && col < 7)
            {
                row--;
                col++;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.BishopTotalMoves3[index]++;
            }
            MoveArrays.BishopMoves3[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row > 0 && col > 0)
            {
                row--;
                col--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.BishopTotalMoves4[index]++;
            }
            MoveArrays.BishopMoves4[index] = moveset;
        }
    }
}
private static void SetMovesRook()
{
    for (byte y = 0; y < 8; y++)
    {
        for (byte x = 0; x < 8; x++)
        {
            byte index = (byte)(y + (x * 8));
            var moveset = new PieceMoveSet(new List<byte>());
            byte move;
```

```
            byte row = x;
            byte col = y;
            while (row < 7)
            {
                row++;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.RookTotalMoves1[index]++;
            }
            MoveArrays.RookMoves1[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row > 0)
            {
                row--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.RookTotalMoves2[index]++;
            }
            MoveArrays.RookMoves2[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (col > 0)
            {
                col--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.RookTotalMoves3[index]++;
            }
            MoveArrays.RookMoves3[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (col < 7)
            {
                col++;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.RookTotalMoves4[index]++;
            }
            MoveArrays.RookMoves4[index] = moveset;
        }
    }
}
private static void SetMovesQueen()
{
    for (byte y = 0; y < 8; y++)
    {
        for (byte x = 0; x < 8; x++)
```

```
{
    byte index = (byte)(y + (x * 8));
    var moveset = new PieceMoveSet(new List<byte>());
    byte move;
    byte row = x;
    byte col = y;
    while (row < 7)
    {
        row++;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.QueenTotalMoves1[index]++;
    }
    MoveArrays.QueenMoves1[index] = moveset;
    moveset = new PieceMoveSet(new List<byte>());
    row = x;
    col = y;
    while (row > 0)
    {
        row--;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.QueenTotalMoves2[index]++;
    }
    MoveArrays.QueenMoves2[index] = moveset;
    moveset = new PieceMoveSet(new List<byte>());
    row = x;
    col = y;
    while (col > 0)
    {
        col--;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.QueenTotalMoves3[index]++;
    }
    MoveArrays.QueenMoves3[index] = moveset;
    moveset = new PieceMoveSet(new List<byte>());
    row = x;
    col = y;
    while (col < 7)
    {
        col++;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.QueenTotalMoves4[index]++;
    }
    MoveArrays.QueenMoves4[index] = moveset;
    moveset = new PieceMoveSet(new List<byte>());
    row = x;
    col = y;
    while (row < 7 && col < 7)
```

```
            {
                row++;
                col++;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.QueenTotalMoves5[index]++;
            }
            MoveArrays.QueenMoves5[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row < 7 && col > 0)
            {
                row++;
                col--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.QueenTotalMoves6[index]++;
            }
            MoveArrays.QueenMoves6[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row > 0 && col < 7)
            {
                row--;
                col++;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.QueenTotalMoves7[index]++;
            }
            MoveArrays.QueenMoves7[index] = moveset;
            moveset = new PieceMoveSet(new List<byte>());
            row = x;
            col = y;
            while (row > 0 && col > 0)
            {
                row--;
                col--;
                move = Position(col, row);
                moveset.Moves.Add(move);
                MoveArrays.QueenTotalMoves8[index]++;
            }
            MoveArrays.QueenMoves8[index] = moveset;
        }
    }
}
private static void SetMovesKing()
{
    for (byte y = 0; y < 8; y++)
    {
```

```
for (byte x = 0; x < 8; x++)
{
    byte index = (byte)(y + (x * 8));
    var moveset = new PieceMoveSet(new List<byte>());
    byte move;
    byte row = x;
    byte col = y;
    if (row < 7)
    {
        row++;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.KingTotalMoves[index]++;
    }
    row = x;
    col = y;
    if (row > 0)
    {
        row--;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.KingTotalMoves[index]++;
    }
    row = x;
    col = y;
    if (col > 0)
    {
        col--;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.KingTotalMoves[index]++;
    }
    row = x;
    col = y;
    if (col < 7)
    {
        col++;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.KingTotalMoves[index]++;
    }
    row = x;
    col = y;
    if (row < 7 && col < 7)
    {
        row++;
        col++;
        move = Position(col, row);
        moveset.Moves.Add(move);
        MoveArrays.KingTotalMoves[index]++;
    }
```

```
                row = x;
                col = y;
                if (row < 7 && col > 0)
                {
                    row++;
                    col--;
                    move = Position(col, row);
                    moveset.Moves.Add(move);
                    MoveArrays.KingTotalMoves[index]++;
                }
                row = x;
                col = y;
                if (row > 0 && col < 7)
                {
                    row--;
                    col++;
                    move = Position(col, row);
                    moveset.Moves.Add(move);
                    MoveArrays.KingTotalMoves[index]++;
                }

                row = x;
                col = y;
                if (row > 0 && col > 0)
                {
                    row--;
                    col--;
                    move = Position(col, row);
                    moveset.Moves.Add(move);
                    MoveArrays.KingTotalMoves[index]++;
                }
                MoveArrays.KingMoves[index] = moveset;
            }
        }
}
```

This concludes the Chess Piece Moves class.

Chess Piece Valid Moves

Originally the code in this post was part of the Chess Piece Motion class. However since I posted the original code I have divided that class into 2 separate classes. Chess Piece Moves and **Chess Piece Valid** moves which is discussed here.

This class will be responsible for dynamically figuring out only the valid moves for the current chess board and assigning only the valid moves to each chess piece. This class will also figure out what pieces are attacking

each other, is the king in check, has en passant occurred and assign the information to each piece for the purpose of score evaluation.

The Chess Piece Valid Moves class will be declared as follows:

```
internal static class PieceValidMoves
```

To help us understand what is board squares are being attacked we will defina 2 arrays. One for storing board squares attacked by White, and one for Black. These arrays are crucial in figuring out things like valid moves for a king, since kings cannot move onto a square that is currently attacked by an opponent.

```
internal static bool[] BlackAttackBoard;
internal static bool[] WhiteAttackBoard;
```

Furthermore we can't correctly check for the kings valid moves until we examine all other chess pieces. This is due to the fact that we won't know if the chess board square is attacked if we don't look at every single chess piece first. For this reason when we come across a king during our analysis, we don't analyze its possible moves but rather store its position for later analysis. The following 2 variables are used to store that information.

```
private static byte BlackKingPosition;
private static byte WhiteKingPosition;
```

The Analyze Move method will perform a deep analysis or examination of the move itself and its effect on the chess board. For example this method will record an En Passant scenario as well as recording Checks and Kills. The Analyze Move method will return true if the move is considered not blocked (not resulting in a kill or blocked by another chess piece of the same color). Similarly it will return false if the move is blocked and movement cannot continue past this position. This is very important since this return value will be used to end a loop of moves in a certain direction. This method is called for all chess pieces other than pawns and kings.

```
private static bool AnalyzeMove(Board board, byte dstPos, Piece pcMoving)
{
    //If I am not a pawn everywhere I move I can attack
    if (pcMoving.PieceColor == ChessPieceColor.White)
    {
        WhiteAttackBoard[dstPos] = true;
    }
    else
    {
        BlackAttackBoard[dstPos] = true;
    }

    //If there no piece there I can potentialy kill just add the move and exit
    if (board.Squares[dstPos].Piece == null)
    {
        pcMoving.ValidMoves.Push(dstPos);
        return true;
    }
```

```
    Piece pcAttacked = board.Squares[dstPos].Piece;
    //if that piece is a different color
    if (pcAttacked.PieceColor != pcMoving.PieceColor)
    {
        pcAttacked.AttackedValue += pcMoving.PieceActionValue;
        //If this is a king set it in check
        if (pcAttacked.PieceType == ChessPieceType.King)
        {
            if (pcAttacked.PieceColor == ChessPieceColor.Black)
            {
                board.BlackCheck = true;
            }
            else
            {
                board.WhiteCheck = true;
            }
        }
        else
        {
            //Add this as a valid move
            pcMoving.ValidMoves.Push(dstPos);
        }

        //We don't continue movement past this piece
        return false;
    }
    //Same Color I am defending
    pcAttacked.DefendedValue += pcMoving.PieceActionValue;
    //Since this piece is of my kind I can't move there
    return false;
}
```

Pawns behave slightly differently than regular pieces in that not all of their moves can result in a kill. A move straight ahead cannon result in a kill while a diagonal move can. For this reason, there are two separate methods to analyze pawn moves. One Parent that loops through all available pawn moves and one child that analyzes each move at a time.

Parent:

```
private static void CheckValidMovesPawn(List<byte> moves, Piece pcMoving,
          byte srcPosition,
          Board board, byte count)
{
 for (byte i = 0; i < count; i++)
 {
  byte dstPos = moves[i];

  if (dstPos%8 != srcPosition%8)
  {
   //If there is a piece there I can potentialy kill
   AnalyzeMovePawn(board, dstPos, pcMoving);
   if (pcMoving.PieceColor == ChessPieceColor.White)
   {
    WhiteAttackBoard[dstPos] = true;
   }
```

```
   else
   {
    BlackAttackBoard[dstPos] = true;
   }
  }
   // if there is something if front pawns can't move there
  else if (board.Squares[dstPos].Piece != null)
  {
   return;
  }
  //if there is nothing in front of me (blocked == false)
  else
  {
   pcMoving.ValidMoves.Push(dstPos);
  }
 }
}
```

Child:

```
private static void AnalyzeMovePawn(Board board, byte dstPos, Piece pcMoving)
{
    //Because Pawns only kill diagonaly we handle the En Passant scenario specialy
    if (board.EnPassantPosition > 0)
    {
        if (pcMoving.PieceColor != board.EnPassantColor)
        {
            if (board.EnPassantPosition == dstPos)
            {
                //We have an En Passant Possible
                pcMoving.ValidMoves.Push(dstPos);

                if (pcMoving.PieceColor == ChessPieceColor.White)
                {
                    WhiteAttackBoard[dstPos] = true;
                }
                else
                {
                    BlackAttackBoard[dstPos] = true;
                }
            }
        }
    }
    Piece pcAttacked = board.Squares[dstPos].Piece;
    //If there no piece there I can potentialy kill
    if (pcAttacked == null)
        return;
    //Regardless of what is there I am attacking this square
    if (pcMoving.PieceColor == ChessPieceColor.White)
    {
        WhiteAttackBoard[dstPos] = true;
        //if that piece is the same color
        if (pcAttacked.PieceColor == pcMoving.PieceColor)
        {
            pcAttacked.DefendedValue += pcMoving.PieceActionValue;
            return;
```

```
        }
        //else piece is different so we are attacking
        pcAttacked.AttackedValue += pcMoving.PieceActionValue;
        //If this is a king set it in check
        if (pcAttacked.PieceType == ChessPieceType.King)
        {
            board.BlackCheck = true;
        }
        else
        {
            //Add this as a valid move
            pcMoving.ValidMoves.Push(dstPos);
        }
    }
    else
    {
        BlackAttackBoard[dstPos] = true;
        //if that piece is the same color
        if (pcAttacked.PieceColor == pcMoving.PieceColor)
        {
            return;
        }
        //If this is a king set it in check
        if (pcAttacked.PieceType == ChessPieceType.King)
        {
            board.WhiteCheck = true;
        }
        else
        {
            //Add this as a valid move
            pcMoving.ValidMoves.Push(dstPos);
        }
    }
    return;
}
```

Check Valid Moves King Castle Method handles the complicated castling scenarios by examining the chess board squares between the king and the rook to make sure they are empty, not attacked and that the rook and king are both in their starting positions.

```
private static void GenerateValidMovesKingCastle(Board board, Piece king)
{
 if (king == null)
 {
  return;
 }

 if (king.Moved)
 {
  return;
 }
 if (king.PieceColor == ChessPieceColor.White &&
  board.WhiteCastled)
 {
  return;
 }
```

```
if (king.PieceColor == ChessPieceColor.Black &&
 board.BlackCastled)
{
 return;
}
if (king.PieceColor == ChessPieceColor.Black &&
 board.BlackCheck)
{
 return;
}
if (king.PieceColor == ChessPieceColor.White &&
 board.WhiteCheck)
{
 return;
}

//This code will add the castleling move to the pieces available moves
if (king.PieceColor == ChessPieceColor.White)
{
 if (board.WhiteCheck)
 {
  return;
 }
 if (board.Squares[63].Piece != null)
 {
  //Check if the Right Rook is still in the correct position
  if (board.Squares[63].Piece.PieceType == ChessPieceType.Rook)
  {
   if (board.Squares[63].Piece.PieceColor == king.PieceColor)
   {
    //Move one column to right see if its empty
    if (board.Squares[62].Piece == null)
    {
     if (board.Squares[61].Piece == null)
     {
      if (BlackAttackBoard[61] == false &&
       BlackAttackBoard[62] == false)
      {
       //Ok looks like move is valid lets add it
       king.ValidMoves.Push(62);
       WhiteAttackBoard[62] = true;
      }
     }
    }
   }
  }
 }
 if (board.Squares[56].Piece != null)
 {
  //Check if the Left Rook is still in the correct position
  if (board.Squares[56].Piece.PieceType == ChessPieceType.Rook)
  {
   if (board.Squares[56].Piece.PieceColor == king.PieceColor)
   {
    //Move one column to right see if its empty
```

```
    if (board.Squares[57].Piece == null)
    {
     if (board.Squares[58].Piece == null)
     {
      if (board.Squares[59].Piece == null)
      {
       if (BlackAttackBoard[58] == false &&
        BlackAttackBoard[59] == false)
       {
        //Ok looks like move is valid lets add it
        king.ValidMoves.Push(58);
        WhiteAttackBoard[58] = true;
       }
      }
     }
    }
   }
  }
 }
}
else if (king.PieceColor == ChessPieceColor.Black)
{
 if (board.BlackCheck)
 {
  return;
 }
 //There are two ways to castle, scenario 1:
 if (board.Squares[7].Piece != null)
 {
  //Check if the Right Rook is still in the correct position
  if (board.Squares[7].Piece.PieceType == ChessPieceType.Rook
   && !board.Squares[7].Piece.Moved)
  {
   if (board.Squares[7].Piece.PieceColor == king.PieceColor)
   {
    //Move one column to right see if its empty
    if (board.Squares[6].Piece == null)
    {
     if (board.Squares[5].Piece == null)
     {
      if (WhiteAttackBoard[5] == false && WhiteAttackBoard[6] == false)
      {
       //Ok looks like move is valid lets add it
       king.ValidMoves.Push(6);
       BlackAttackBoard[6] = true;
      }
     }
    }
   }
  }
 }
 //There are two ways to castle, scenario 2:
 if (board.Squares[0].Piece != null)
 {
  //Check if the Left Rook is still in the correct position
```

```
   if (board.Squares[0].Piece.PieceType == ChessPieceType.Rook &&
    !board.Squares[0].Piece.Moved)
   {
    if (board.Squares[0].Piece.PieceColor ==
     king.PieceColor)
    {
     //Move one column to right see if its empty
     if (board.Squares[1].Piece == null)
     {
      if (board.Squares[2].Piece == null)
      {
       if (board.Squares[3].Piece == null)
       {
        if (WhiteAttackBoard[2] == false &&
         WhiteAttackBoard[3] == false)
        {
         //Ok looks like move is valid lets add it
         king.ValidMoves.Push(2);
         BlackAttackBoard[2] = true;
        }
       }
      }
     }
    }
   }
  }
 }
}
```

The last method in this class is the only non-private method in this listing. The Generate Valid Moves method will be called directly by the chess engine to create valid moves for each chess board it examines. This method will loop through all chess pieces on the chess board and examine each possible move based on the chess piece's position. Please note that we have already calculated each pieces move in the Chess Piece Moves class for every single position on the chess board. We now just have to figure out which one of those moves is valid and what their results will be. While we are looping through all of the chess pieces we also collect some information such as how many pawns are in each file, how many pawns are isolated. At the end of this method we will sum up this information and store it on the Chess Board.

```
internal static void GenerateValidMoves(Board board)
{
 // Reset Board
 board.BlackCheck = false;
 board.WhiteCheck = false;

 WhiteAttackBoard = new bool[64];
 BlackAttackBoard = new bool[64];
 //Generate Moves
 for (byte x = 0; x < 64; x++)
 {
  Square sqr = board.Squares[x];
  if (sqr.Piece == null)
   continue;
  sqr.Piece.ValidMoves = new Stack<byte>(sqr.Piece.LastValidMoveCount);
  switch (sqr.Piece.PieceType)
```

```
{
 case ChessPieceType.Pawn:
  {
   if (sqr.Piece.PieceColor == ChessPieceColor.White)
   {
    CheckValidMovesPawn(MoveArrays.WhitePawnMoves[x].Moves, sqr.Piece, x,
         board,
         MoveArrays.WhitePawnTotalMoves[x]);
    break;
   }
   if (sqr.Piece.PieceColor == ChessPieceColor.Black)
   {
    CheckValidMovesPawn(MoveArrays.BlackPawnMoves[x].Moves, sqr.Piece, x,
         board,
         MoveArrays.BlackPawnTotalMoves[x]);
    break;
   }
   break;
  }
 case ChessPieceType.Knight:
  {
   for (byte i = 0; i < MoveArrays.KnightTotalMoves[x]; i++)
   {
    AnalyzeMove(board, MoveArrays.KnightMoves[x].Moves[i], sqr.Piece);
   }
   break;
  }
 case ChessPieceType.Bishop:
  {
   for (byte i = 0; i < MoveArrays.BishopTotalMoves1[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.BishopMoves1[x].Moves[i],
        sqr.Piece) ==
     false)
    {
     break;
    }
   }
   for (byte i = 0; i < MoveArrays.BishopTotalMoves2[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.BishopMoves2[x].Moves[i],
        sqr.Piece) ==
     false)
    {
     break;
    }
   }
   for (byte i = 0; i < MoveArrays.BishopTotalMoves3[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.BishopMoves3[x].Moves[i],
        sqr.Piece) ==
     false)
    {
```

```
      break;
     }
    }
    for (byte i = 0; i < MoveArrays.BishopTotalMoves4[x]; i++)
    {
     if (
      AnalyzeMove(board, MoveArrays.BishopMoves4[x].Moves[i],
         sqr.Piece) ==
      false)
     {
      break;
     }
    }
   break;
  }
 case ChessPieceType.Rook:
  {
   for (byte i = 0; i < MoveArrays.RookTotalMoves1[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.RookMoves1[x].Moves[i], sqr.Piece) ==
     false)
    {
     break;
    }
   }
   for (byte i = 0; i < MoveArrays.RookTotalMoves2[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.RookMoves2[x].Moves[i], sqr.Piece) ==
     false)
    {
     break;
    }
   }
   for (byte i = 0; i < MoveArrays.RookTotalMoves3[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.RookMoves3[x].Moves[i], sqr.Piece) ==
     false)
    {
     break;
    }
   }
   for (byte i = 0; i < MoveArrays.RookTotalMoves4[x]; i++)
   {
    if (
     AnalyzeMove(board, MoveArrays.RookMoves4[x].Moves[i], sqr.Piece) ==
     false)
    {
     break;
    }
   }
   break;
  }
 case ChessPieceType.Queen:
```

```
{
 for (byte i = 0; i < MoveArrays.QueenTotalMoves1[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves1[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves2[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves2[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves3[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves3[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves4[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves4[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves5[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves5[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves6[x]; i++)
 {
  if (
   AnalyzeMove(board, MoveArrays.QueenMoves6[x].Moves[i], sqr.Piece) ==
   false)
  {
   break;
  }
 }
 for (byte i = 0; i < MoveArrays.QueenTotalMoves7[x]; i++)
```

```
      {
       if (
        AnalyzeMove(board, MoveArrays.QueenMoves7[x].Moves[i], sqr.Piece) ==
        false)
       {
        break;
       }
      }
      for (byte i = 0; i < MoveArrays.QueenTotalMoves8[x]; i++)
      {
       if (
        AnalyzeMove(board, MoveArrays.QueenMoves8[x].Moves[i], sqr.Piece) ==
        false)
       {
        break;
       }
      }
      break;
     }
    case ChessPieceType.King:
     {
      if (sqr.Piece.PieceColor == ChessPieceColor.White)
      {
       WhiteKingPosition = x;
      }
      else
      {
       BlackKingPosition = x;
      }
      break;
     }
   }
  }

  if (board.WhoseMove == ChessPieceColor.White)
  {
   GenerateValidMovesKing(board.Squares[BlackKingPosition].Piece, board,
           BlackKingPosition);
   GenerateValidMovesKing(board.Squares[WhiteKingPosition].Piece, board,
           WhiteKingPosition);
  }
  else
  {
   GenerateValidMovesKing(board.Squares[WhiteKingPosition].Piece, board,
           WhiteKingPosition);
   GenerateValidMovesKing(board.Squares[BlackKingPosition].Piece, board,
           BlackKingPosition);
  }

  //Now that all the pieces were examined we know if the king is in check
  GenerateValidMovesKingCastle(board, board.Squares[WhiteKingPosition].Piece);
  GenerateValidMovesKingCastle(board, board.Squares[BlackKingPosition].Piece);
 }
```

This concludes the Chess Piece Valid Moves class.

Move Content

In our Chess Engine we will need to describe movement as it occurs. This will be useful to keep track move history, the best move at each search level or even the result of our Alpha Beta Search.
The Move Content class has 2 major components. The first component describes the moving chess piece(s). The second component describes the chess piece taken or captured during the move.
These two components described as 2 structs:

```
public struct PieceMoving
{
 public byte DstPosition;
 public bool Moved;
 public ChessPieceColor PieceColor;
 public ChessPieceType PieceType;
 public byte SrcPosition;

 public PieceMoving(ChessPieceColor pieceColor, ChessPieceType pieceType,
bool moved,
        byte srcPosition, byte dstPosition)
 {
  PieceColor = pieceColor;
  PieceType = pieceType;
  SrcPosition = srcPosition;
  DstPosition = dstPosition;
  Moved = moved;
 }
 public PieceMoving(PieceMoving pieceMoving)
 {
  PieceColor = pieceMoving.PieceColor;
  PieceType = pieceMoving.PieceType;
  SrcPosition = pieceMoving.SrcPosition;
  DstPosition = pieceMoving.DstPosition;
  Moved = pieceMoving.Moved;
 }
 public PieceMoving(ChessPieceType pieceType)
 {
  PieceType = pieceType;
  PieceColor = ChessPieceColor.White;
  SrcPosition = 0;
  DstPosition = 0;
  Moved = false;
 }
}



public struct PieceTaken
```

```
{
 public bool Moved;
 public ChessPieceColor PieceColor;
 public ChessPieceType PieceType;
 public byte Position;

 public PieceTaken(ChessPieceColor pieceColor, ChessPieceType pieceType,
bool moved,
       byte position)
 {
  PieceColor = pieceColor;
  PieceType = pieceType;
  Position = position;
  Moved = moved;
 }
 public PieceTaken(ChessPieceType pieceType)
 {
  PieceColor = ChessPieceColor.White;
  PieceType = pieceType;
  Position = 0;
  Moved = false;
 }
}
```

The Move Content class itself makes use of the above 2 structs by declaring them into 3 fields:

- PieceMoving MovingPiecePrimary – The primary piece that has moved.
- PieceMoving MovingPieceSecondary; - The secondary piece that has moved. This is usually null unless a king has castled. In this case the primary Piece would be the king and the secondary piece would be the rook.
- PieceTaken TakenPiece – The chess piece that was capture or taken during the move.

The other fields like **Score**, **Pawn Promoted** and **En Passant Occurred** are self explanatory. However the last method ToString requires a bit of an explanation.
The Move Content class will be used to generate Portable Game Notation (PGN) string of the game. For this reason I overwrote the ToString() method for the class so that it will return a portion of the PGN string for the move that has occurred.

```
public new string ToString()
{
    string value = "";

    var srcCol = (byte) (MovingPiecePrimary.SrcPosition%8);
    var srcRow = (byte)(8 - (MovingPiecePrimary.SrcPosition / 8));
    var dstCol = (byte) (MovingPiecePrimary.DstPosition%8);
    var dstRow = (byte) (8 - (MovingPiecePrimary.DstPosition/8));
    if (MovingPieceSecondary.PieceType == ChessPieceType.Rook)
    {
        if (MovingPieceSecondary.PieceColor == ChessPieceColor.Black)
        {
```

```
            if (MovingPieceSecondary.SrcPosition == 7)
            {
                value += "O-O";
            }
            else if (MovingPieceSecondary.SrcPosition == 0)
            {
                value += "O-O-O";
            }
        }
        else if (MovingPieceSecondary.PieceColor == ChessPieceColor.White)
        {
            if (MovingPieceSecondary.SrcPosition == 63)
            {
                value += "O-O";
            }
            else if (MovingPieceSecondary.SrcPosition == 56)
            {
                value += "O-O-O";
            }
        }
    }
    else
    {
        value += GetPgnMove(MovingPiecePrimary.PieceType);
        switch (MovingPiecePrimary.PieceType)
        {
            case ChessPieceType.Knight:
                value += GetColumnFromInt(srcCol + 1);
                value += srcRow;
                break;
            case ChessPieceType.Rook:
                value += GetColumnFromInt(srcCol + 1);
                value += srcRow;
                break;
            case ChessPieceType.Pawn:
                if (srcCol != dstCol)
                {
                    value += GetColumnFromInt(srcCol + 1);
                }
                break;
        }
        if (TakenPiece.PieceType != ChessPieceType.None)
        {
            value += "x";
        }
        value += GetColumnFromInt(dstCol + 1);
        value += dstRow;
        if (PawnPromoted)
        {
            value += "=Q";
        }
```

```
    }
    return value;
}
```

Notice: Since the Move Content class and all its components will be used to display information outside of the chess engine like the user interface all the Move Content components and the class itself were declared as public.
The above ToString() method requires a few helper methods that convert objects to strings:

```
private static string GetColumnFromInt(int column)
{
    switch (column)
    {
        case 1:
            return "a";
        case 2:
            return "b";
        case 3:
            return "c";
        case 4:
            return "d";
        case 5:
            return "e";
        case 6:
            return "f";
        case 7:
            return "g";
        case 8:
            return "h";
        default:
            return "Unknown";
    }
}

private static string GetPgnMove(ChessPieceType pieceType)
{
    switch (pieceType)
    {
        case ChessPieceType.Bishop:
            return "B";
        case ChessPieceType.King:
            return "K";
        case ChessPieceType.Knight:
            return "N";
        case ChessPieceType.Queen:
            return "Q";
        case ChessPieceType.Rook:
            return "R";
        default:
            return "";
    }
```

```
}
```

This concludes the Move Content class.

Starting the chess engine

Bringing it all together, starting the chess engine.
This post will bring all of the previous sections together in the discussion of the chess engine class. At this time I will assume that you have already read the previous sections related to Chess Board Square, Chess Board and Chess Piece Representation as well as the Chess Piece Moves and Chess Piece Valid Moves. Today I will not provide a complete chess engine listing because we have not yet discussed move searching and Chess Board Evaluation. However at the end of this section we will have a basic chess engine that can:

1. Track chess piece locations on the chess board
2. Provide a list of valid moves for each chess piece, including en passant and castling
3. Track whose move it is.
4. Track move history.
5. Setup a starting chess board.

This in theory once we create a graphical user interface this skeleton chess engine would allow you to play a two human player chess game.
Chess Engine class is declared as public sealed

```
public sealed class Engine
```

Chess Engine class will contain 3 members that will represent the current chess board, previous chess board whose move it currently is. Previous Chess Board will store the last chess board prior to the last move. Please notice that the Previous Chess Board member will potentially give us easy undo functionality.

```
internal Board ChessBoard;
internal Board PreviousChessBoard;

public ChessPieceColor WhoseMove
{
    get { return ChessBoard.WhoseMove; }
    set { ChessBoard.WhoseMove = value; }
}
```

The constructor is a bit complicated as it performs the following actions:

- Instantiate above members and set the initial move to White
- Initiate Chess Piece Motion (Pre-calculate all possible moves for all pieces on all chess board squares possible)
- Assign Chess pieces to the chess board in the starting position of a standard chess game.
- Generate valid moves for the chess pieces in their current positions.

```
public Engine()
{
    ChessBoard = new Board();
    MoveHistory = new Stack<MoveContent>();

    RegisterStartingBoard();
    ChessBoard.WhoseMove = ChessPieceColor.White;

    ChessPieceMoves.InitiateChessPieceMotion();
    PieceValidMoves.GenerateValidMoves(ChessBoard);
}
```

Notice the Constructor uses a method called Register Starting Board. This method constructs all the chess pieces necessary for the starting chess board and registers them with the chess board object.
In the above code a helper method was used called Register Piece. This method assigns the created chess piece to the desired location on the chess board.

```
private void RegisterPiece(byte boardColumn, byte boardRow, ChessPiece
Piece)
{
    byte position = (byte)(boardColumn + (boardRow * 8));

    ChessBoard.Squares[position].Piece = Piece;
    return;
}
```

The remaining method that I will indroduce today is the MovePiece method. This code will allow you to move chess pieces around the chess board. The method will return true if the move was successful and false if the move was not valid.

```
public bool MovePiece(byte sourceColumn, byte sourceRow,
        byte destinationColumn, byte destinationRow)
{
 byte srcPosition = (byte)(sourceColumn + (sourceRow * 8));
 byte dstPosition = (byte)(destinationColumn + (destinationRow * 8));

 Piece Piece = ChessBoard.Squares[srcPosition].Piece;
 PreviousChessBoard = new Board(ChessBoard);

 Board.MovePiece(ChessBoard, srcPosition, dstPosition, PromoteToPieceType);
 PieceValidMoves.GenerateValidMoves(ChessBoard);

 //If there is a check in place, check if this is still true;
 if (Piece.PieceColor == ChessPieceColor.White)
 {
  if (ChessBoard.WhiteCheck)
  {
   //Invalid Move
   ChessBoard = new Board(PreviousChessBoard);
```

```
    PieceValidMoves.GenerateValidMoves(ChessBoard);
    return false;
   }
  }
  else if (Piece.PieceColor == ChessPieceColor.Black)
  {
   if (ChessBoard.BlackCheck)
   {
    //Invalid Move
    ChessBoard = new Board(PreviousChessBoard);
    PieceValidMoves.GenerateValidMoves(ChessBoard);
    return false;
   }
  }
  MoveHistory.Push(ChessBoard.LastMove);
  return true;
 }
```

Generating a Starting Chess Position

At this point we it would be nice if we were able to add some chess pieces to our chess board. Originally I wrote a method that would declare 32 chess pieces and assign them to the correct chess board square. However eventually I wanted to implement FEN notation into my chess engine. FEN notation is an easy way to describe chess board positions. It is somewhat of a standard in computer chess circles. Hence once I implemented a method that can read a FEN string and setup the chess board based on the FEN string values, I had an easy way to create my starting chess position.

```
ChessBoard = new Board("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1");
```

The full source code for the FEN methods can be found on the FEN Section

To summarize, our chess engine class contains the current chess board (Board ChessBoard) as well as a copy of the chess board as it looked prior to the last move (Board PreviousChessBoard). The Chess Engine knows whose move it currently is (ChessPiece.ChessPieceColor WhoseMove). The constructor of the Chess Engine creates all the chess pieces required for a standard chess game and registers them with the chess board using the Register Piece method. The chess engine constructor will also Initiate Chess Piece Motion and Assign valid moves to each chess piece based on the pieces current position and the board layout. Moving chess pieces around the board is handled by the MovePiece method.

Piece Square Table

Today I want to discuss the **Piece Square Table**.
Originally the piece square tables were declared in a separate class that was used by the Evaluation function. However I found that it is more efficient to save the extra method calls and perform the piece square table lookups directly in the evaluation function.

Hence I have modified this post to simply describe the piece square table and the logic behind the numbers assigned to each position.

As I have already stated the piece square tables are used chess board Evaluation class to score points based on the current position of the chess piece. The main idea behind this code is that certain positions for chess pieces are better than others. Fox example it is better for knights to stay away from the edge of the board. Pawns should control the center of the board and advance forward.
I have decided not to create a piece square table for every single chess piece. I have omitted Queens and Rooks. I could not find good enough positional tactical advantages for Rooks and Queens to warrant the performance cost of a table lookup for each of their positions.

Here are the piece square tables used by my chess engine:

Pawns are encouraged to stay in the center and advance forward:

```
private static readonly short[] PawnTable = new short[]
{
      0,  0,  0,  0,  0,  0,  0,  0,
     50, 50, 50, 50, 50, 50, 50, 50,
     10, 10, 20, 30, 30, 20, 10, 10,
      5,  5, 10, 27, 27, 10,  5,  5,
      0,  0,  0, 25, 25,  0,  0,  0,
      5, -5,-10,  0,  0,-10, -5,  5,
      5, 10, 10,-25,-25, 10, 10,  5,
      0,  0,  0,  0,  0,  0,  0,  0
};
```

Knights are encouraged to control the center and stay away from edges to increase mobility:

```
private static readonly short[] KnightTable = new short[]
{
     -50,-40,-30,-30,-30,-30,-40,-50,
     -40,-20,  0,  0,  0,  0,-20,-40,
     -30,  0, 10, 15, 15, 10,  0,-30,
     -30,  5, 15, 20, 20, 15,  5,-30,
     -30,  0, 15, 20, 20, 15,  0,-30,
     -30,  5, 10, 15, 15, 10,  5,-30,
     -40,-20,  0,  5,  5,  0,-20,-40,
     -50,-40,-20,-30,-30,-20,-40,-50,
};
```

Bishops are also encouraged to control the center and stay away from edges and corners:

```
private static readonly short[] BishopTable = new short[]
{
     -20,-10,-10,-10,-10,-10,-10,-20,
     -10,  0,  0,  0,  0,  0,  0,-10,
     -10,  0,  5, 10, 10,  5,  0,-10,
     -10,  5,  5, 10, 10,  5,  5,-10,
     -10,  0, 10, 10, 10, 10,  0,-10,
     -10, 10, 10, 10, 10, 10, 10,-10,
     -10,  5,  0,  0,  0,  0,  5,-10,
```

```
    -20,-10,-40,-10,-10,-40,-10,-20,
};
```

Kings have 2 piece square tables, one for the end game and one for the middle game. During the middle game kings are encouraged to stay in the corners, while in the end game kings are encouraged to move towards the center.

```
private static readonly short[] KingTable = new short[]
{
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -20, -30, -30, -40, -40, -30, -30, -20,
  -10, -20, -20, -20, -20, -20, -20, -10,
   20,  20,   0,   0,   0,   0,  20,  20,
   20,  30,  10,   0,   0,  10,  30,  20
};

private static readonly short[] KingTableEndGame = new short[]
{
    -50,-40,-30,-20,-20,-30,-40,-50,
    -30,-20,-10,  0,  0,-10,-20,-30,
    -30,-10, 20, 30, 30, 20,-10,-30,
    -30,-10, 30, 40, 40, 30,-10,-30,
    -30,-10, 30, 40, 40, 30,-10,-30,
    -30,-10, 20, 30, 30, 20,-10,-30,
    -30,-30,  0,  0,  0,  0,-30,-30,
    -50,-30,-30,-30,-30,-30,-30,-50
};
```

The above tables are used during the evaluation method to lookup the positional values to help calculate the chess board score.
Here is an example of how the above tables would be used to lookup a value for a white pawn position:

```
score += PawnTable[position];
```

And here is the code to perform the same lookup for a black pawn:

```
byte index = (byte)(((byte)(position + 56)) - (byte)((byte)(position / 8)
* 16));

score += PawnTable[index];
```

Chess Board Evaluation

From what information I could gather on the internet, it seems that most experts agree that the evaluation function of a chess engine has the greatest capability to dictate the strength of your chess game. Rybka, currently considered to be the top computer chess engine in the world, is not famous for its search speed but rather for its advanced chess board evaluation written by an international chess master. Furthermore the evaluation function can make your move searching faster by allowing your chess engine to search better moves first.

Unfortunately for you Rybka's evaluation function is not available. To make things worse I am not considered by anyone to be a chess master.

Chess Piece Evaluation

As seen in the Chess Piece Class, the following values are assigned to chess pieces:

Pawn 100
Knight 320
Bishop 325
Rook 500
Queen 975
King 32767

About the numbers.

Writing your chess board evaluation method, think about the numbers as percentages of a pawn. In my chess engine a pawn is worth 100 points. If you award 10 points to a tactical position you are telling the chess engine that this position is worth 1/10th of a pawn. The problem can arise when a move combines several tactical advantages for the moving side and several tactical penalties for the opponent. This can result in unnecessary pawn sacrifice for set of minor tactical advantages. When writing your evaluation function always keep this in mind.

The Chess Board Score is represented by a signed integer. The higher the score the better it is for White. The lower the score the better it is for black. Using a single integer makes everything a bit faster since I can always just reference one variable.

For example if the value of a single pawn is 100 points and there was only one single black pawn on the chess board the score would be -100. If there were two pawns, one white and one black the score would be 0. If white had 2 pawns and black 1 pawn the score would be 100.

On to the code

Chess Bin Board evaluation is implemented as a static class with two static methods.

```
internal static class Evaluation
```

We first declare our Piece Square Tables discussed in the previous post:

```
private static readonly short[] PawnTable = new short[]
{
  0,  0,  0,  0,  0,  0,  0,  0,
 50, 50, 50, 50, 50, 50, 50, 50,
 10, 10, 20, 30, 30, 20, 10, 10,
  5,  5, 10, 27, 27, 10,  5,  5,
  0,  0,  0, 25, 25,  0,  0,  0,
  5, -5,-10,  0,  0,-10, -5,  5,
  5, 10, 10,-25,-25, 10, 10,  5,
  0,  0,  0,  0,  0,  0,  0,  0
};

private static readonly short[] KnightTable = new short[]
{
 -50,-40,-30,-30,-30,-30,-40,-50,
 -40,-20,  0,  0,  0,  0,-20,-40,
 -30,  0, 10, 15, 15, 10,  0,-30,
 -30,  5, 15, 20, 20, 15,  5,-30,
 -30,  0, 15, 20, 20, 15,  0,-30,
 -30,  5, 10, 15, 15, 10,  5,-30,
 -40,-20,  0,  5,  5,  0,-20,-40,
 -50,-40,-20,-30,-30,-20,-40,-50,
};
private static readonly short[] BishopTable = new short[]
{
 -20,-10,-10,-10,-10,-10,-10,-20,
 -10,  0,  0,  0,  0,  0,  0,-10,
 -10,  0,  5, 10, 10,  5,  0,-10,
 -10,  5,  5, 10, 10,  5,  5,-10,
 -10,  0, 10, 10, 10, 10,  0,-10,
 -10, 10, 10, 10, 10, 10, 10,-10,
 -10,  5,  0,  0,  0,  0,  5,-10,
 -20,-10,-40,-10,-10,-40,-10,-20,
};
private static readonly short[] KingTable = new short[]
{
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -30, -40, -40, -50, -50, -40, -40, -30,
  -20, -30, -30, -40, -40, -30, -30, -20,
  -10, -20, -20, -20, -20, -20, -20, -10,
   20,  20,   0,   0,   0,   0,  20,  20,
   20,  30,  10,   0,   0,  10,  30,  20
};
private static readonly short[] KingTableEndGame = new short[]
{
 -50,-40,-30,-20,-20,-30,-40,-50,
 -30,-20,-10,  0,  0,-10,-20,-30,
 -30,-10, 20, 30, 30, 20,-10,-30,
 -30,-10, 30, 40, 40, 30,-10,-30,
```

```
 -30,-10, 30, 40, 40, 30,-10,-30,
 -30,-10, 20, 30, 30, 20,-10,-30,
 -30,-30,  0,  0,  0,  0,-30,-30,
 -50,-30,-30,-30,-30,-30,-30,-50
};
```

The first method will evaluate the score for a single chess piece on the board.

```
private static int EvaluatePieceScore(Square square, byte position, bool castled,
                                      bool endGamePhase,ref byte bishopCount,
                                      ref bool insufficientMaterial)
```

We declare a single variable representing the score for the current chess piece. This score will always start at 0 and go up if the position is evaluated to be better, and go down if the position is evaluated to be worse. When the score is returned to the main Evaluation Function, it will be subtracted for black and added for white.

```
int score = 0;
```

We also declare one local variable that will hold a position we will use to look up the Pieces Piece Square Table value. As you saw above the Piece Square Table is an array of 64 elements, basically representing the chess board from white's perspective. In that case we can simply use white's position value to lookup it's Piece Square Table value. However if the chess piece is black, we need to invert its position to use the same tables for both white and black.

```
byte index = position;

if (square.Piece.PieceColor == ChessPieceColor.Black)
{
 index = (byte)(63-position);
}
```

Each piece type has a value. For example a pawn is worth 100 points, a Rook 500 etc. We add that value to the score.

```
score += square.Piece.PieceValue;
```

During move generation we record how many pieces are protecting each piece on the board. This is done by adding the protecting pieces PieceProtectionValue to the protected pieces ProtectedValue. In the evaluation method we add this Protected Value to the score. I know this sounds confusing. The trick here is that I don't want to simply count the number of pieces protecting or attacking another piece. Rather I want to give more points for being attacked or protected by a Pawn and fewer points for being attacked or protected by a minor or major piece.

```
score += square.Piece.DefendedValue;
```

Similarly during move generation we added each attacking piece's Piece Attack Value to the attacked pieces Attacked Value. We now subtract the attacked value from the score. The idea here is that we reward the computer for protecting its pieces and penalize it for having the pieces attacked.

```
score -= square.Piece.AttackedValue;
```

Furthermore if the chess piece is getting attacked and it is not protected then will consider that piece EnPrise, meaning we are about to lose it. There is an additional penalty applied by subtracting the Attack Value from the Score again.

```
//Double Penalty for Hanging Pieces
if (square.Piece.DefendedValue < square.Piece.AttackedValue)
{
 score -= ((square.Piece.AttackedValue - square.Piece.DefendedValue)* 10);
}
}
```

We will also add score for mobility. This discourages trapped pieces and blocked pawns.

```
if (square.Piece.ValidMoves != null)
{
 score += square.Piece.ValidMoves.Count;
}
```

The remainder of the code is chess piece specific, starting with Pawns.

The following code will perform 3 Evaluations:

- Remove some points for pawns on the edge of the board. The idea is that since a pawn of the edge can only attack one way it is worth 15% less.
- Give an extra bonus for pawns that are on the 6th and 7th rank as long as they are not attacked in any way.
- Add points based on the Pawn Piece Square Table Lookup.

We will also keep track in what column we find each pawn. This will not be a pawn count but a value of what that pawn means in that column if we later find him to be isolated passed or doubled. This information will be used later to score isolated passed and doubled pawns.

```
if (square.Piece.PieceType == ChessPieceType.Pawn)
{
 insufficientMaterial = false;

 if (position % 8 == 0 || position % 8 == 7)
 {
  //Rook Pawns are worth 15% less because they can only attack one way
  score -= 15;
 }
 //Calculate Position Values
 score += PawnTable[index];
 if (square.Piece.PieceColor == ChessPieceColor.White)
 {
  if (whitePawnCount[position % 8] > 0)
  {
```

```
  //Doubled Pawn
  score -= 16;
 }
 if (position >= 8 && position <= 15)
 {
  if (square.Piece.AttackedValue == 0)
  {
   whitePawnCount[position % 8] += 200;
   if (square.Piece.DefendedValue != 0)
    whitePawnCount[position % 8] += 50;
  }
 }
 else if (position >= 16 && position <= 23)
 {
  if (square.Piece.AttackedValue == 0)
  {
   whitePawnCount[position % 8] += 100;

   if (square.Piece.DefendedValue != 0)
    whitePawnCount[position % 8] += 25;
  }
 }
 whitePawnCount[position % 8]+=10;
}
else
{
 if (blackPawnCount[position % 8] > 0)
 {
  //Doubled Pawn
  score -= 16;
 }
 if (position >= 48 && position <= 55)
 {
  if (square.Piece.AttackedValue == 0)
  {
   blackPawnCount[position % 8] += 200;
   if (square.Piece.DefendedValue != 0)
    blackPawnCount[position % 8] += 50;
  }
 }
 //Pawns in 6th Row that are not attacked are worth more points.
 else if (position >= 40 && position <= 47)
 {
  if (square.Piece.AttackedValue == 0)
  {
   blackPawnCount[position % 8] += 100;
   if (square.Piece.DefendedValue != 0)
    blackPawnCount[position % 8] += 25;
  }
 }
 blackPawnCount[position % 8] += 10;
```

```
 }
}
```

Knights also get a value from the Piece Square Table. However knights are worth less in the end game since it is difficult to mate with a knight hence they lose 10 points during the end game.

```
else if (square.Piece.PieceType == ChessPieceType.Knight)
{
 score += KnightTable[index];

 //In the end game remove a few points for Knights since they are worth less
 if (endGamePhase)
 {
  score -= 10;
 }
}
```

Opposite to Knights, Bishops are worth more in the end game, also we add a small bonus for having 2 bishops since they complement each other by controlling different ranks.

```
else if (square.Piece.PieceType == ChessPieceType.Bishop)
{
 bishopCount++;

 if (bishopCount >= 2)
 {
  //2 Bishops receive a bonus
  score += 10;
 }
 //In the end game Bishops are worth more
 if (endGamePhase)
 {
  score += 10;
 }
 score += BishopTable[index];
}
```

We want to encourage Rooks not to move out of their corner positions before castling has occurred. Also if a rook is present on the board we will set a variable called Insufficient Material to false. This allows us to catch a tie scenario if insufficient material is present on the board.

```
else if (square.Piece.PieceType == ChessPieceType.Rook)
{
 insufficientMaterial = false;

 if (square.Piece.Moved && castled == false)
 {
  score -= 10;
 }
}
```

Furthermore we want to discourage Queen movement in the beginning of the game so we give a small penalty if the Queen has moved.

```
else if (square.Piece.PieceType == ChessPieceType.Queen)
{
 insufficientMaterial = false;

 if (square.Piece.Moved && !endGamePhase)
 {
  score -= 10;
 }
}
```

Finally in order to encourage the computer to castle we will remove points if castling is no longer possible. Furthermore we will also take away points if the king has less than 2 moves. The idea here is that if the king has only one move, we are possibly one move away from a mate.

```
else if (square.Piece.PieceType == ChessPieceType.King)
{
 if (square.Piece.ValidMoves.Count < 2)
 {
  score -= 5;
 }
 if (endGamePhase)
 {
  score += KingTableEndGame[index];
 }
 else
 {
  score += KingTable[index];

  if (square.Piece.Moved && castled == false)
  {
   score -= 30;
  }
 }
}
```

At the end our method simply returns the score:

```
return score;
```

The second method in the Board Evaluation Class is a static method that accepts the chess board and the currently moving side. This is the main Evaluation Function used in the chess engine. It will evaluate board specific events such as check and mate as well as loop through all of the chess pieces on the board and call the above described Evaluate Piece Score for each piece.

```
internal static void EvaluateBoardScore(Board board)
```

At the beginning of the evaluation the score will always be set to 0.

```
board.Score = 0;
```

We will also declare a variable called insufficient material that will be set to true only if enough chess pieces (material) are found to prevent the Insufficient Material Tie Rule.

```
bool insufficientMaterial = true;
```

The evaluation method will first examine board wide specific events. Is there a check, stale mate, has any side castled etc. Most of the work on figuring out these events has already been done in the Chess Piece Motion or Chess Board classes; all we are doing now is summing up the score.

```
if (board.StaleMate)
{
    return;
}
if (board.FiftyMoveCount >= 50)
{
    return;
}
if (board.RepeatedMoveCount >= 3)
{
    return;
}
```

Next we give bonuses and penalties for board level scenarios such as King Checks, Mates and Castles.

```
if (board.BlackMate)
{
 board.Score = 32767;
 return;
}
if (board.WhiteMate)
{
 board.Score = -32767;
 return;
}
if (board.BlackCheck)
{
 board.Score += 75;
 if (board.EndGamePhase)
  board.Score += 10;
}
else if (board.WhiteCheck)
{
 board.Score -= 75;
 if (board.EndGamePhase)
  board.Score -= 10;
}
if (board.BlackCastled)
```

```
{
 board.Score -= 40;
}
if (board.WhiteCastled)
{
 board.Score += 40;
}
//Add a small bonus for tempo (turn)
if (board.WhoseMove == ChessPieceColor.White)
{
 board.Score += 10;
}
else
{
 board.Score -= 10;
}
```

The following two integers will keep track of how many bishops and knights are remaining on the board. This will allow us to give an additional bonus if a player has both bishops. Also if there 2 are 2 knights we can't call the Insufficient Material Tie rule.

```
byte blackBishopCount = 0;
byte whiteBishopCount = 0;
byte knightCount = 0;
```

The following integer will calculate remaining material on the board. We will use this later on to make the decision if we are currently in the middle or end game. End game evaluation can differ from the middle game. For example in the middle game Knights are very useful since they can hop behind enemy lines and take out vulnerable pieces, however in an end game Knights have a hard time placing the king in a corner for a mate so their value is slightly diminished. Also castling in an end game has a diminished bonus.

```
int RemainingPieces = 0;
```

We also need to keep track how many pawns exist in each column so that later we can figure out if we have any isolated and doubled pawns.

```
blackPawnCount = new short[8];
whitePawnCount = new short[8];
```

The next step of the evaluation is a loop through all of the chess pieces on the chess board and call the EvaluatePieceScore method defined above.

```
for (byte x = 0; x < 64; x++)
{
 Square square = board.Squares[x];

 if (square.Piece == null)
  continue;
 //Calculate Remaining Material for end game determination
 remainingPieces++;
 if (square.Piece.PieceColor == ChessPieceColor.White)
```

```
 {
  board.Score += EvaluatePieceScore(square, x, board.WhiteCastled,
board.EndGamePhase,
   ref whiteBishopCount, ref insufficientMaterial);
 }
 else if (square.Piece.PieceColor == ChessPieceColor.Black)
 {
  board.Score -= EvaluatePieceScore(square, x, board.BlackCastled,
board.EndGamePhase,
   ref blackBishopCount, ref insufficientMaterial);
 }
 if (square.Piece.PieceType == ChessPieceType.Knight)
 {
  knightCount++;
  if (knightCount > 1)
  {
   insufficientMaterial = false;
  }
 }
 if ((blackBishopCount + whiteBishopCount) > 1)
 {
  insufficientMaterial = false;
 }
}
```

Next section does will handle the remaining board level events, such as calling a tie if there is insufficient material on the chess board.

After looping through all of the chess pieces we also know how many pieces are remaining on the chess board. If there are less than 10 pieces we will mark the chess board as being in an endgame. This way the next evaluation will change slightly based on the end game rules defined.

```
if (insufficientMaterial)
{
 board.Score = 0;
 board.StaleMate = true;
 board.InsufficientMaterial = true;
 return;
}

if (remainingPieces < 10)
{
 board.EndGamePhase = true;
 if (board.BlackCheck)
 {
  board.Score += 10;
 }
 else if (board.WhiteCheck)
 {
  board.Score -= 10;
 }
```

```
}
```

The last section of code uses the previously stored pawn position information and figures out if there are any doubled and isolated pawns. Each one of these pawns are given a strong penalty.

```
//Black Isolated Pawns
if (blackPawnCount[0] >= 1 && blackPawnCount[1] == 0)
{
 board.Score += 12;
}
if (blackPawnCount[1] >= 1 && blackPawnCount[0] == 0 &&
 blackPawnCount[2] == 0)
{
 board.Score += 14;
}
if (blackPawnCount[2] >= 1 && blackPawnCount[1] == 0 &&
 blackPawnCount[3] == 0)
{
 board.Score += 16;
}
if (blackPawnCount[3] >= 1 && blackPawnCount[2] == 0 &&
 blackPawnCount[4] == 0)
{
 board.Score += 20;
}
if (blackPawnCount[4] >= 1 && blackPawnCount[3] == 0 &&
 blackPawnCount[5] == 0)
{
 board.Score += 20;
}
if (blackPawnCount[5] >= 1 && blackPawnCount[4] == 0 &&
 blackPawnCount[6] == 0)
{
 board.Score += 16;
}
if (blackPawnCount[6] >= 1 && blackPawnCount[5] == 0 &&
 blackPawnCount[7] == 0)
{
 board.Score += 14;
}
if (blackPawnCount[7] >= 1 && blackPawnCount[6] == 0)
{
 board.Score += 12;
}

//White Isolated Pawns
if (whitePawnCount[0] >= 1 && whitePawnCount[1] == 0)
{
 board.Score -= 12;
}
if (whitePawnCount[1] >= 1 && whitePawnCount[0] == 0 &&
```

```
 whitePawnCount[2] == 0)
{
 board.Score -= 14;
}
if (whitePawnCount[2] >= 1 && whitePawnCount[1] == 0 &&
 whitePawnCount[3] == 0)
{
 board.Score -= 16;
}
if (whitePawnCount[3] >= 1 && whitePawnCount[2] == 0 &&
 whitePawnCount[4] == 0)
{
 board.Score -= 20;
}
if (whitePawnCount[4] >= 1 && whitePawnCount[3] == 0 &&
 whitePawnCount[5] == 0)
{
 board.Score -= 20;
}
if (whitePawnCount[5] >= 1 && whitePawnCount[4] == 0 &&
 whitePawnCount[6] == 0)
{
 board.Score -= 16;
}
if (whitePawnCount[6] >= 1 && whitePawnCount[5] == 0 &&
 whitePawnCount[7] == 0)
{
 board.Score -= 14;
}
if (whitePawnCount[7] >= 1 && whitePawnCount[6] == 0)
{
 board.Score -= 12;
}
//Black Passed Pawns
if (blackPawnCount[0] >= 1 && whitePawnCount[0] == 0)
{
 board.Score -= blackPawnCount[0];
}
if (blackPawnCount[1] >= 1 && whitePawnCount[1] == 0)
{
 board.Score -= blackPawnCount[1];
}
if (blackPawnCount[2] >= 1 && whitePawnCount[2] == 0)
{
 board.Score -= blackPawnCount[2];
}
if (blackPawnCount[3] >= 1 && whitePawnCount[3] == 0)
{
 board.Score -= blackPawnCount[3];
}
if (blackPawnCount[4] >= 1 && whitePawnCount[4] == 0)
```

```
{
 board.Score -= blackPawnCount[4];
}
if (blackPawnCount[5] >= 1 && whitePawnCount[5] == 0)
{
 board.Score -= blackPawnCount[5];
}
if (blackPawnCount[6] >= 1 && whitePawnCount[6] == 0)
{
 board.Score -= blackPawnCount[6];
}
if (blackPawnCount[7] >= 1 && whitePawnCount[7] == 0)
{
 board.Score -= blackPawnCount[7];
}
//White Passed Pawns
if (whitePawnCount[0] >= 1 && blackPawnCount[1] == 0)
{
 board.Score += whitePawnCount[0];
}
if (whitePawnCount[1] >= 1 && blackPawnCount[1] == 0)
{
 board.Score += whitePawnCount[1];
}
if (whitePawnCount[2] >= 1 && blackPawnCount[2] == 0)
{
 board.Score += whitePawnCount[2];
}
if (whitePawnCount[3] >= 1 && blackPawnCount[3] == 0)
{
 board.Score += whitePawnCount[3];
}
if (whitePawnCount[4] >= 1 && blackPawnCount[4] == 0)
{
 board.Score += whitePawnCount[4];
}
if (whitePawnCount[5] >= 1 && blackPawnCount[5] == 0)
{
 board.Score += whitePawnCount[5];
}
if (whitePawnCount[6] >= 1 && blackPawnCount[6] == 0)
{
 board.Score += whitePawnCount[6];
}
if (whitePawnCount[7] >= 1 && blackPawnCount[7] == 0)
{
 board.Score += whitePawnCount[7];
}
```

This concludes the Chess Piece Evaluation. The ChessBin evaluation function is by no means complete. I will be spending some additional time on this evaluation function soon and post any updates to this page.

In my experience modifying the evaluation function is extremely dangerous. Even small changes can significantly weaken your chess engine. The problem is that we are dealing here with a system of values that relate to each other in various and complicated ways. It is difficult to tell ahead of time how a small change in a bonus or penalty will affect how the engine will interpret this change in various situations. When modifying the chess engine evaluation function is best to always test against the previous version.
I always play the new version of my chess engine against the previously released one. This means that you should save different version of your chess game as you go along. Furthermore if you find your chess engine made a blunder, save the game, correct the mistake in your code then load the save game to see if that solved it. Over time you will have a few save games that will allow you to quickly test your new code. I found that often new code re-introduced some old bug I solved in an older save game and actually made the chess engine weaker.

Search for Mate

Before we can discus move searching and the Alpha Beta algorithm we need a way to check for check mates. This method will actually be located in our static Search class. This method will loop through all of the moves for all the chess pieces on the board and see if they actually have a valid move. If they do then we are not in a check mate situation. If there aren't any valid moves for this board position that we know that this is either a check mate (if the king is in check) or a stale mate.
This method is very expensive to execute so we only call it if there is a check on any king on the board or if there are 0 possible moves. I will explain in more detail in my Alpha Beta method.

The **Search for Mate** method returns true if a check mate or stale mate is found. The actual values of type of mate and side mated are stored in the three reference variables passed into the method.

```
internal static bool SearchForMate(ChessPieceColor movingSide, Board
examineBoard, ref bool blackMate, ref bool whiteMate, ref bool staleMate)
{
 bool foundNonCheckBlack = false;
 bool foundNonCheckWhite = false;

 for (byte x = 0; x < 64; x++)
 {
  Square sqr = examineBoard.Squares[x];
  //Make sure there is a piece on the square
  if (sqr.Piece == null)
   continue;
  //Make sure the color is the same color as the one we are moving.
  if (sqr.Piece.PieceColor != movingSide)
   continue;
  //For each valid move for this piece
  foreach (byte dst in sqr.Piece.ValidMoves)
  {
   //We make copies of the board and move so we don't change the original
   Board board = examineBoard.FastCopy();
   //Make move so we can examine it
```

```
   Board.MovePiece(board, x, dst, ChessPieceType.Queen);
   //We Generate Valid Moves for Board
   PieceValidMoves.GenerateValidMoves(board);
   if (board.BlackCheck == false)
   {
    foundNonCheckBlack = true;
   }
   else if (movingSide == ChessPieceColor.Black)
   {
    continue;
   }
   if (board.WhiteCheck == false )
   {
    foundNonCheckWhite = true;
   }
   else if (movingSide == ChessPieceColor.White)
   {
    continue;
   }
  }
 }
 if (foundNonCheckBlack == false)
 {
  if (examineBoard.BlackCheck)
  {
   blackMate = true;
   return true;
  }
  if (!examineBoard.WhiteMate && movingSide != ChessPieceColor.White)
  {
   staleMate = true;
   return true;
  }
 }
 if (foundNonCheckWhite == false)
 {
  if (examineBoard.WhiteCheck)
  {
   whiteMate = true;
   return true;
  }
  if (!examineBoard.BlackMate && movingSide != ChessPieceColor.Black)
  {
   staleMate = true;
   return true;
  }
 }
 return false;
}
```

The Search for Mate method is also used in your Chess Engine Make Move method which will check if the last move made by the human player caused a check mate. This will be done every time a player makes a move.

Move Searching and Alpha Beta

Out of all of the computer chess programming concepts I discussed on this website I found move searching to be the most complicated for me to understand, and the most time consuming to write. Until this point all of the concepts of writing a chess engine were easy to understand. Most developers will figure out some way of representing chess pieces, chess board and movement. However I found that move searching is not like that. Reinventing the wheel by writing your own move search algorithm is just not a good idea. There are algorithms that you just have to implement for your engine to have a shot at searching enough moves to simulate even average chess playing.

Min Max & Negamax

Probably like most people starting to program a chess engine I started by looking at implementing the Min Max algorithm. The idea is fairly simple, I look at all my moves I can make, evaluate them and make the best move. Then I do the same for the opponent from his point of view.
This led me to some really crappy code that did not really work very well. Actually what I found out later is that min max works by going deep into the furthest leaf of the search tree and working backwards. That's not the same thing as going from the top down because what ended up being a spectacular move 5 nodes down could have been a really crappy move at the beginning. A good example of this is a chess piece sacrifice to get a check mate. So I look to all the possibilities of every move I can make along with all of the moves my opponent can make work backwards and I choose the root move that will get me the best score at the end.

```
private static int MinMax(Board examineBoard, byte depth)
{
 if (depth == 0)
 {
   //Evaluate Score
  Evaluation.EvaluateBoardScore(examineBoard);
  //Invert Score to support Negamax
  return SideToMoveScore(examineBoard.Score, examineBoard.WhoseMove);
 }

 List<Position> positions = EvaluateMoves(examineBoard, depth);
 if (examineBoard.WhiteCheck || examineBoard.BlackCheck || positions.Count == 0)
 {
  if (SearchForMate(examineBoard.WhoseMove, examineBoard, ref
examineBoard.BlackMate, ref examineBoard.WhiteMate, ref examineBoard.StaleMate))
  {
   if (examineBoard.BlackMate)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return -32767-depth;
    return 32767 + depth;
   }
   if (examineBoard.WhiteMate)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return 32767 + depth;
    return -32767 - depth;
   }
   //If Not Mate then StaleMate
```

```
    return 0;
   }
  }
  int bestScore = -32767;

  foreach (Position move in positions)
  {
   //Make a copy
   Board board = examineBoard.FastCopy();
   //Move Piece
   Board.MovePiece(board, move.SrcPosition, move.DstPosition,
ChessPieceType.Queen);
   //We Generate Valid Moves for Board
   PieceValidMoves.GenerateValidMoves(board);
   if (board.BlackCheck)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
    {
     //Invalid Move
     continue;
    }
   }
   if (board.WhiteCheck)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.White)
    {
     //Invalid Move
     continue;
    }
   }
   int value = -MinMax(board, (byte)(depth-1));
   if (value > bestScore)
   {
    bestScore = (int)value;
   }
  }
  return bestScore;
}
```

The above **Mix-Max** implementation is not actually used anywhere in my chess engine, however I wanted to make sure it is we understand how it works before we move to the actual implementation of Alpha Beta.

The first point I would like to make about the above code is that the algorithm is recursive. It will call itself up to its furthest leaf and then return the score back to each parent branch so that the branch can evaluate which leaf was the best back as many levels as we decide are needed.
The second point on the above Min-Max implementation is related to the depth variable. It will set the limit of how far should our algorithm search. This is often referred to as ply. One ply equals one move by either side. So if we set our algorithm depth to 1 ply the Min Max algorithm would simple search one move deep and return the best move available to the current side. A depth 2 or ply 2 search would search each possible move and each possible response to every move.

Furthermore the above algorithm is actually a variation of Min-Max often called **Negamax** because as mentioned above it always looks for the maximizing score rather than having to branches looking for either the maximum or minimum value depending on the chess piece color moving.
There is however a trick to implementing **Negamax**. The issue is that the algorithm has to always look for the highest score, so we will need a helper method to help us out here by inverting the score for black.

```
private static int SideToMoveScore(int score, ChessPieceColor color)
{
 if (color == ChessPieceColor.Black)
  return -score;

 return score;
}
```

This above method is key, since without it your algorithm would not return the best move for black but rather the worst, highest scored.

If we you had tried to implemented this algorithm in your chess engine you would find that move searching was probably very slow. It was probably ok down to ply 3 or 4

Alpha Beta

The next evolution of my search algorithm was Alpha Beta. It took me of weeks reading articles on min max and alpha beta trying to fully grasp exactly was has to be coded and why it works.
The main idea behind Alpha Beta is the fact that we don't need to search every possible move. Some moves just do not make sense.
Let's imagine your opponent has 5 bags of items. You get to keep one of the items from one of the 5 bags. You get to choose the bag, however your opponent will get to choose which item you get. Your opponent does not want to give away his valuable items so he will choose the one that is least valuable to you. So you must choose the bag where the least valuable item is more valuable than the least valuable item in all of the other bags.
So let's say you open the first bag and you look inside. You see a gold ring, a diamond and a shovel. You know your opponent is not going to give you the gold ring or the diamond. If you choose the first bag you will end up with a shovel. The shovel is the least valuable item in that bag you remember that for later.
So you look into the second bag and you see a laptop computer. This is more valuable than a shovel, so you keep looking. However the second item is a clump of dirt. Dirt is less valuable than a shovel. So you don't need to keep looking through the other items in that bag, because you know that whatever else you find in the bag even if it is more valuable you will just end up with dirt. Instead you can move on to the next bag and keep looking for something better than a shovel.
This is how alpha beta works. You stop looking for responses to your move (bag) when you find one that has a worst result than the worst result from your previous move. The name Alpha Beta refers to the two variables that you will pass around the algorithm that will keep the best scores for you and your opponent (The Shovel)
The main advantage of Alpha Beta is that it is free. It does not affect the quality of the moves made by the computer. It simply discards moves that would not have been considered anyways.
One additional note I would like to make is that Alpha Beta works best when the moves are sorted in the order of best first. If we think of our example above it is in our best interest to find the bag with the shovel first before finding the bag with the clump of dirt. If you had found the clump of dirt first you would still have to look through all the other items in the second bag.

For this purpose it is in our best interest to sort the moves prior to trying Alpha Beta. For that we need to declare a few methods.

Evaluate Moves

Evaluate Moves is a pseudo move generator and an evaluation function combined. It organizes all the valid moves for a position into a list of positions and assigns them a score based on a very basic evaluation. We don't use this evaluation score to make any serious decisions we just use it to sort our moves. You may be tempted to use your regular chess board evaluation function here. This would improve sorting quite a bit, however a full evaluation is slow and there would be allot of wasted effort because you don't need the actual score of the chess board until you get to depth 0 (the last ply you are going to look at). In my tests I found that doing a simple sort based on a score resulting from Most Valuable Vitim Least Valuable Attacker, performs quite well. Basically the idea is that you want to try a pawn attacking a queen before you try a queen attacking a pawn. I achieve this by subtracting the values of the attacker and defender. This generates lots of node cut-offs. In addition to MVV/LVA I add some small easy evaluation points for castling moves and Piece Action Value.

Evaluate Moves stores its results in a List of Positions.

```
private struct Position
{
 internal byte SrcPosition;
 internal byte DstPosition;
 internal int Score;
}
```

Evaluate Moves also requires a helper sort method.

```
private static int Sort(Position s2, Position s1)
{
 return (s1.Score).CompareTo(s2.Score);
}
```

The actual Evaluate Moves method loops through all of the chess pieces on the board and records the source position and destination position of the move along its pseudo score.

```
private static List<Position> EvaluateMoves(Board board)
{

 //We are going to store our result boards here
 List<Position> positions = new List<Position>();
 for (byte x = 0; x < 64; x++)
 {
  Piece piece = board.Squares[x].Piece;
  //Make sure there is a piece on the square
  if (piece == null)
   continue;
  //Make sure the color is the same color as the one we are moving.
  if (piece.PieceColor != board.WhoseMove)
```

```
 continue;
//For each valid move for this piece
foreach (byte dst in piece.ValidMoves)
{
 Position move = new Position();
 move.SrcPosition = x;
 move.DstPosition = dst;
 Piece pieceAttacked = board.Squares[move.DstPosition].Piece;
 //If the move is a capture add it's value to the score
 if (pieceAttacked != null)
 {
  move.Score += pieceAttacked.PieceValue;
  if (piece.PieceValue < pieceAttacked.PieceValue)
  {
   move.Score += pieceAttacked.PieceValue - piece.PieceValue;
  }
 }
 if (!piece.Moved)
 {
  move.Score += 10;
 }
 move.Score += piece.PieceActionValue;
 //Add Score for Castling
 if (!board.WhiteCastled && board.WhoseMove == ChessPieceColor.White)
 {
  if (piece.PieceType == ChessPieceType.King)
  {
   if (move.DstPosition != 62 && move.DstPosition != 58)
   {
    move.Score -= 40;
   }
   else
   {
    move.Score += 40;
   }
  }
  if (piece.PieceType == ChessPieceType.Rook)
  {
   move.Score -= 40;
  }
 }
 if (!board.BlackCastled && board.WhoseMove == ChessPieceColor.Black)
 {
  if (piece.PieceType == ChessPieceType.King)
  {
   if (move.DstPosition != 6 && move.DstPosition != 2)
   {
    move.Score -= 40;
   }
   else
   {
```

```
        move.Score += 40;
       }
      }
      if (piece.PieceType == ChessPieceType.Rook)
      {
       move.Score -= 40;
      }
     }
     positions.Add(move);
    }
   }
   return positions;
}
```

Now for the actual implementation of **Alpha Beta**.

In our chess engine we need to introduce the concept of the variables alpha and beta. These will be used during our recursive search to affectively keep the leaf score during our search. This will allow us to make the decision of whether or not we need to continue or we can cut the search short and return.

- Alpha will be the current best score for this leaf.
- Beta will be the best score for the upper leaf thus far or the opponent's best score for positions already searched.

If Alpha > Beta, meaning my move is better than my opponents other best move thus far we have reached the scenario where searching other moves are not relevant because a Shovel is better than clump of dirt.
Here is the Alpha Beta code from my chess engine

```
private static int AlphaBeta(Board examineBoard, byte depth, int alpha, int beta)
{
 nodesSearched++;

 if (examineBoard.FiftyMove >= 50 || examineBoard.RepeatedMove >= 3)
  return 0;
 if (depth == 0)
 {
  //Evaluate Score
  Evaluation.EvaluateBoardScore(examineBoard);
  //Invert Score to support Negamax
  return SideToMoveScore(examineBoard.Score, examineBoard.WhoseMove);
 }
 List<Position> positions = EvaluateMoves(examineBoard);
 if (examineBoard.WhiteCheck || examineBoard.BlackCheck || positions.Count == 0)
 {
  if (SearchForMate(examineBoard.WhoseMove, examineBoard, ref
examineBoard.BlackMate, ref examineBoard.WhiteMate, ref examineBoard.StaleMate))
  {
   if (examineBoard.BlackMate)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return -32767-depth;
    return 32767 + depth;
   }
   if (examineBoard.WhiteMate)
   {
```

```
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return 32767 + depth;
    return -32767 - depth;
   }
   //If Not Mate then StaleMate
   return 0;
  }
 }
 positions.Sort(Sort);
 foreach (Position move in positions)
 {
  //Make a copy
  Board board = examineBoard.FastCopy();
  //Move Piece
  Board.MovePiece(board, move.SrcPosition, move.DstPosition,
ChessPieceType.Queen);
  //We Generate Valid Moves for Board
  PieceValidMoves.GenerateValidMoves(board);
  if (board.BlackCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.Black)
   {
    //Invalid Move     continue;
   }
  }
  if (board.WhiteCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.White)
   {
    //Invalid Move
    continue;
   }
  }
  int value = -AlphaBeta(board, (byte)(depth-1), -beta, -alpha);
  if (value >= beta)
  {
   // Beta cut-off    return beta;
  }
  if (value > alpha)
  {
   alpha = value;
  }
 }

 return alpha;
}
```

As you see this code is almost identical to the Min-Max code above with the exception of move sorting as well as the Alpha and Beta variables. Furthermore if we do find a cut-off (alpha > beta) then we simply return beta as the best score.

Initially the Alpha Beta method is called with Alpha being the smallest possible integer and Beta being the highest possible integer. This ensures that we search at least one move all the way down to its last ply before

performing any cut-offs. In the next post I will discuss how to make that initial Alpha Beta call from my chess engine and some of the other modifications of Alpha Beta that will improve its speed and accuracy.

Move Searching Alpha Beta Part 2

Last time I discussed Min Max and the Alpha Beta algorithms. However you might have noticed that the algorithm I showed last time does not really tell you which of the available moves is the best, but rather which was the best score out of all the available moves. To figure out which resulting chess board is the best I have implemented another method called Alpha Beta Root.
Alpha Beta Root is very similar to the regular Alpha Beta Method with the exception of keeping track of the best board found so far. Alpha Beta Root is also our entry method into searching; it calls the regular Alpha Beta method. You pass it a chess board and it returns Move Content containing the best move you can make. Alpha Beta Root does not also need to perform a Quiescence Search since it is already performed in the regular Alpha Beta method.
The code below can be divided into 3 sections.

1. Initial examination of what legal moves I can make and what their resulting score is. This is followed by a sort to give us the best chance of trying the best move first
2. Initial 1 ply call of Alpha Beta to see if there is an instant check mate so we can exit.
3. Regular Alpha Beta call.

Before we get started we will need a helper struct to keep a list of our starting positions.

```
internal struct ResultBoards
{
 internal List<Board> Positions;
}
```

Now onto the main Alpha Beta Root Method:

```
internal static MoveContent AlphaBetaRoot(Board examineBoard, byte depth)
{
 int alpha = -400000000;
 const int beta = 400000000;

 Board bestBoard = new Board(short.MinValue);
 //We are going to store our result boards here
 ResultBoards succ = new ResultBoards
 {
  Positions = new List<Board>(30)
 };
 for (byte x = 0; x < 64; x++)
 {
  Square sqr = examineBoard.Squares[x];
  //Make sure there is a piece on the square
  if (sqr.Piece == null)
   continue;
  //Make sure the color is the same color as the one we are moving.
  if (sqr.Piece.PieceColor != examineBoard.WhoseMove)
   continue;
```

```
  //For each valid move for this piece
  foreach (byte dst in sqr.Piece.ValidMoves)
  {
   //We make copies of the board and move so that we can move it without
effecting the parent board
   Board board = examineBoard.FastCopy();
   //Make move so we can examine it
   Board.MovePiece(board, x, dst, ChessPieceType.Queen);
   //We Generate Valid Moves for Board
   PieceValidMoves.GenerateValidMoves(board);
   //Invalid Move
   if (board.WhiteCheck && examineBoard.WhoseMove == ChessPieceColor.White)
   {
    continue;
   }
   //Invalid Move
   if (board.BlackCheck && examineBoard.WhoseMove == ChessPieceColor.Black)
   {
    continue;
   }
   //We calculate the board score
   Evaluation.EvaluateBoardScore(board);
   //Invert Score to support Negamax
   board.Score = SideToMoveScore(board.Score, board.WhoseMove);
   succ.Positions.Add(board);
  }
 }
 succ.Positions.Sort(Sort);
 //Can I make an instant mate?
 foreach (Board pos in succ.Positions)
 {
  int value = -AlphaBeta(pos, 1, -beta, -alpha);
  if (value >= 32767)
  {
   return pos.LastMove;
  }
 }
 depth--;
 byte plyDepthReached = ModifyDepth(depth, succ.Positions.Count);
 int currentBoard = 0;
 alpha = -400000000;
 succ.Positions.Sort(Sort);
 foreach (Board pos in succ.Positions)
 {
  currentBoard++;
  int value = -AlphaBeta(pos, plyDepthReached, -beta, -alpha);
  pos.Score = value;
  //If value is greater then alpha this is the best board
  if (value > alpha)
  {
   alpha = value;
   bestBoard = new Board(pos);
  }
 }
 return bestBoard.LastMove;
```

```
}
```

The obvious question might be why do we do this? Why not simply copy the best board in the regular Alpha Beta method and return it. The simple answer is performance. Because the regular Alpha Beta method is recursive we want it to be as fast as possible. It is much faster to copy integers rather than calling the copy constructor for the board object.
One last piece of code that I would like to add here is the Modify Ply method. One thing I noticed while testing my chess engine is that during the end game my engine made moves at a much faster rate than it did during the opening and middle game. This had a very simple explanation as during the end game there are far fewer chess pieces and there are less moves to calculate. For this reason I added a small method to that adds 2 plies to my search if there are less then 6 root moves on the board. This way I can search deeper during the end game, increasing my odds of finding a check mate.

```
private static byte ModifyDepth(byte depth, int possibleMoves)
{
 if (possibleMoves <= 15)
 {
  depth += 1;
 }

 return depth;
}
```

If you have any questions about this post feel free to post a comment below. Chances are someone else has the same question and I would love a chance for improvement.

Quiescence Search and Extensions

Now that we have discussed the basics of Alpha Beta we can add some small modifications to increase the strength of our move searching and solve the problems of the Horizon Affect

Horizon Affect

The Horizon Affect is a problem that affects every chess engine whose search depth is limited to a certain depth or ply. The Horizon of what your computer chess engine can see is set by the ply or depth limit on the Alpha Beta Search. If we asked the computer to examine 5 moves then the computer will not see that the sixth move made by the opponent will possibly be detrimental to its position. This is especially problematic when we start considering the long exchanges of chess pieces that often occur in a chess game.
There are two ways to battle this problem. The first solution is a simple hack. The trick is to make sure that we always ask the computer to search to an odd ply or depth. This means that the last move the computer will always consider will be the opponents move. This very easily prevents the computer from leaving hanging or unprotected pieces.

However this odd ply solution does not solve the issue related to long exchanges. During the middle game opponents will often build up an attack and defense of a weak piece located in a crucial position such as the center of the chess board. It is not unusual to have a center pawn both protected and attacked by 5-6 chess pieces, including knight's pawns bishops and even queens. If your chess engine is limited to 5 or even 7 ply, it

will never be able to search these exchanges to the end. This means your chess engine will never know if during the course of the piece exchange it will come out on top or lose an extra piece. To solve this issue most chess engines implement what is called a Quiescence Search.

Quiescence Search

The word Quiescence means calm or still. Quiescence Search in computer chess terms simply means searching for a calm position.
The idea behind a Quiescence Search is simple, once your reach your horizon, the last ply you will search, perform a deeper search considering only moves that capture other chess pieces. This deeper search occurs in the same Alpha Beta algorithm. Now you might think that this will significantly slow down your search. In practice however this is not the case. First not every single move is evaluated, only captures so there are much fewer moves to evaluate. Second with every single ply searched more pieces are captured and there are even less moves to evaluate. Very quickly you will often end up with 0 possible captures a few ply down. Third captures often make for very good beta cutoffs in our Alpha Beta. At most your quiescence search should not take more than 10-20% of your search. The difference in the accuracy of your search algorithm however is much higher than 20%. Implemented correctly you will see a significant improvement in the strength of your chess engine.

Extensions

Extensions are a fairly simple concept. An extension will allow your Alpha Beta to search deeper by 1 ply if the position is especially interesting. Interesting positions can be checks or really valuable captures.
Extensions are really needed if your Alpha Beta search is limited to a shallow search such as 5 ply. The usefulness of Extensions diminishes with the increase of the depth of the main Alpha Beta Search.
First I will list the modified Alpha Beta code that makes use of Extensions and Quiescence at depth 0.

```
private static int AlphaBeta(Board examineBoard, byte depth, int alpha, int beta,
bool extended)
{
 if (examineBoard.FiftyMove >= 50 || examineBoard.RepeatedMove >= 3)
  return 0;


 //End Main Search with Quiescence
 if (depth == 0)
 {
  if (!extended && examineBoard.BlackCheck || examineBoard.WhiteCheck)
  {
   depth++;
   extended = true;
  }
  else
  {
   //Perform a Quiessence Search
   return Quiescence(examineBoard, alpha, beta);
  }
 }
 List<Position> positions = EvaluateMoves(examineBoard);
 if (examineBoard.WhiteCheck || examineBoard.BlackCheck || positions.Count == 0)
 {
```

```
  if (SearchForMate(examineBoard.WhoseMove, examineBoard, ref examineBoard.BlackMate,
ref examineBoard.WhiteMate, ref examineBoard.StaleMate))
  {
   if (examineBoard.BlackMate)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return -32767-depth;
    return 32767 + depth;
   }
   if (examineBoard.WhiteMate)
   {
    if (examineBoard.WhoseMove == ChessPieceColor.Black)
     return 32767 + depth;
    return -32767 - depth;
   }
   //If Not Mate then StaleMate
   return 0;
  }
 }
 positions.Sort(Sort);
 foreach (Position move in positions)
 {
  //Make a copy
  Board board = examineBoard.FastCopy();
  //Move Piece
  Board.MovePiece(board, move.SrcPosition, move.DstPosition, ChessPieceType.Queen);
  //We Generate Valid Moves for Board
  PieceValidMoves.GenerateValidMoves(board);
  if (board.BlackCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.Black)
   {
    //Invalid Move
    continue;
   }
  }
  if (board.WhiteCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.White)
   {
    //Invalid Move
    continue;
   }
  }
  value = -AlphaBeta(board, (byte)(depth - 1), -beta, -alpha, extended);
  if (value >= beta)
  {
   return beta;
  }
  if (value > alpha)
  {
   alpha = (int)value;
  }
 }
 return alpha;
}
```

I would like you to notice that in the call to Alpha Beta now has a Boolean variable called: Extended. No matter what we only want to extend the depth of the search a maximum of one time per leaf of the search. Else we run the risk that the search will continue to perform extensions indefinitely. For this reason once we start one of the extension we will set the Boolean value to true, preventing it from occurring again. Here is the listing for the Quiescence Search method:

```
private static int Quiescence(Board examineBoard, int alpha, int beta)
{
 //Evaluate Score
 Evaluation.EvaluateBoardScore(examineBoard);
 //Invert Score to support Negamax
 examineBoard.Score = SideToMoveScore(examineBoard.Score, examineBoard.WhoseMove);

 if (examineBoard.Score >= beta)
  return beta;
 if (examineBoard.Score > alpha)
  alpha = examineBoard.Score;
 List<Position> positions = EvaluateMovesQ(examineBoard);
 if (positions.Count == 0)
 {
  return examineBoard.Score;
 }
 positions.Sort(Sort);
 foreach (Position move in positions)
 {
  //Make a copy
  Board board = examineBoard.FastCopy();
  //Move Piece
  Board.MovePiece(board, move.SrcPosition, move.DstPosition, ChessPieceType.Queen);
  //We Generate Valid Moves for Board
  PieceValidMoves.GenerateValidMoves(board);
  if (board.BlackCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.Black)
   {
    //Invalid Move
    continue;
   }
  }
  if (board.WhiteCheck)
  {
   if (examineBoard.WhoseMove == ChessPieceColor.White)
   {
    //Invalid Move
    continue;
   }
  }

  int value = -Quiescence(board, - beta, -alpha);
  if (value >= beta)
  {
   return beta;
  }
  if (value > alpha)
   alpha = value;
 }
 return alpha;
```

```
}
```

You may have noticed that the Quiescence Search uses a new method called EvaluateMovesQ. This method is virtually the same as the previously discussed Evaluate Moves except it only collects moves that capture.

```
private static List<Position> EvaluateMovesQ(Board examineBoard)
{
 //We are going to store our result boards here
 List<Position> positions = new List<Position>();

 for (byte x = 0; x < 64; x++)
 {
  Piece piece = examineBoard.Squares[x].Piece;
  //Make sure there is a piece on the square
  if (piece == null)
   continue;
  //Make sure the color is the same color as the one we are moving.
  if (piece.PieceColor != examineBoard.WhoseMove)
   continue;
  //For each valid move for this piece
  foreach (byte dst in piece.ValidMoves)
  {
   if (examineBoard.Squares[dst].Piece == null)
   {
    continue;
   }
   Position move = new Position();
   move.SrcPosition = x;
   move.DstPosition = dst;
   Piece pieceAttacked = examineBoard.Squares[move.DstPosition].Piece;
   move.Score += pieceAttacked.PieceValue;
   if (piece.PieceValue < pieceAttacked.PieceValue)
   {
    move.Score += pieceAttacked.PieceValue - piece.PieceValue;
   }
   move.Score += piece.PieceActionValue;

   positions.Add(move);
  }
 }
 return positions;
}
```

During the Quiescence Search we will often reach a position where no more captures are available. Hence there will be no positions to evaluate. For this reason we have to add the following line of code:

```
 if (positions.Count == 0)
 {
  return examineBoard.Score;
 }
```

This last piece of code that I want to explain is called the Stand Pad. It's just a complicated way of saying that during the quiescence search we will set a default value for alpha that is equal to the board being examined. It just prevents you from examining moves that make your situations worse than it already is.

```
if (examineBoard.Score >= beta)
 return beta;

if (examineBoard.Score > alpha)
 alpha = examineBoard.Score;
```

That completes the post on Quiescence Search and Extension.

Forsyth–Edwards Notation

In this post I am going to discuss Forsyth-Edwards Notation (FEN) and its implementation in a chess engine. FEN is a standard way of describing a chess position, containing enough information to restart the chess game from that position. It is based on a notation developed by a Scottish journalist, David Forsyth in the 19th century.

Why is FEN useful to us?

1. We can use FEN to store game history allowing us to search for move repetitions as well as display the history of the game to the user. Furthermore if we find a FEN position that has occurred in the past, we can skip searching for the best move and use the same response we used before.

2. We can use FEN strings to implement an Opening Book. With two FEN strings I can store position pairs representing a starting position and the prescribed response.

The implementation of Forsyth–Edwards Notation

FEN notation uses only ASCII characters stored in a single line. A FEN string or record contains 6 fields. These are separated by a space.
Piece placement from white's perspective. Each row is noted, starting from row 8 (blacks row0 and ending with row 1 (white's row). Each piece is described from column to column h. Each piece is identified by a single letter.

Pawn: **P**
Knight: **K**
Bishop: **B**
Rook: **R**
Queen: **Q**
King: **K**

White pieces are noted using capital letters and black using lower case. So P would be a white pawn and p would signify a black pawn.

Empty squares (spaces) are described using numbers, each number representing the number of empty squares before the next chess pieces. The number 8 would describe a completely empty row.
The character / describes a new row.

So for a starting position we may see:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR

The second column in the Forsyth–Edwards Notation represents whose turn it is. A single character is used w for white and b for black.

The third column represents if castling is still allowed. If neither side can castle then the character – is used. Otherwise the following letters are used. K means white can castle King Side, Q means White can castle Queen side. Lower case k and q mean the same for black.
The fourth column represents an En Passant target square. The square that the pawn hopped to get to its row, or the position behind the pawn. If there is no En Passant square then the character – is used. So if the last move was pawn to e4, we will record e3 in this column.
The fifth column contains the number of half moves since the last pawn move or capture. This is used to determine the 50 move draw scenario.

The last column contains the full move number. The number starts at 1 and is incremented after black's move.

Examples:

FEN for the starting position:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
FEN after the white pawn moved to E4:
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
FEN after the black pawn moved to C5
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2
And then after the white knight moves to F3:
rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2

Forsyth–Edwards Notation Code

In my chess engine FEN is implemented in two methods. The first method will produce a FEN string for any chess Board.

```
internal static string Fen(bool boardOnly, Board board)
{
 string output = String.Empty;
 byte blankSquares = 0;

 for (byte x = 0; x < 64; x++)
 {
  byte index = x;
  if (board.Squares[index].Piece != null)
  {
   if (blankSquares > 0)
```

```
    {
     output += blankSquares.ToString();
     blankSquares = 0;
    }
    if (board.Squares[index].Piece.PieceColor == ChessPieceColor.Black)
    {
     output +=
Piece.GetPieceTypeShort(board.Squares[index].Piece.PieceType).ToLower();
    }
    else
    {
     output += Piece.GetPieceTypeShort(board.Squares[index].Piece.PieceType);
    }
   }
   else
   {
    blankSquares++;
   }
   if (x % 8 == 7)
   {
    if (blankSquares > 0)
    {
     output += blankSquares.ToString();
     output += "/";
     blankSquares = 0;
    }
    else
    {
     if (x > 0 && x != 63)
     {
      output += "/";
     }
    }
   }
  }
  if (board.WhoseMove == ChessPieceColor.White)
  {
   output += " w ";
  }
  else
  {
   output += " b ";
  }
  string spacer = "";
  if (board.WhiteCastled == false)
  {
   if (board.Squares[60].Piece != null)
   {
    if (board.Squares[60].Piece.Moved == false)
    {
     if (board.Squares[63].Piece != null)
     {
      if (board.Squares[63].Piece.Moved == false)
      {
       output += "K";
       spacer = " ";
```

```
      }
     }
     if (board.Squares[56].Piece != null)
     {
      if (board.Squares[56].Piece.Moved == false)
      {
       output += "Q";
       spacer = " ";
      }
     }
    }
   }
  }
  if (board.BlackCastled == false)
  {
   if (board.Squares[4].Piece != null)
   {
    if (board.Squares[4].Piece.Moved == false)
    {
     if (board.Squares[7].Piece != null)
     {
      if (board.Squares[7].Piece.Moved == false)
      {
       output += "k";
       spacer = " ";
      }
     }
     if (board.Squares[0].Piece != null)
     {
      if (board.Squares[0].Piece.Moved == false)
      {
       output += "q";
       spacer = " ";
      }
     }
    }
   }

  }
  if (output.EndsWith("/"))
  {
   output.TrimEnd('/');
  }

  if (board.EnPassantPosition != 0)
  {
   output += spacer + GetColumnFromByte((byte)(board.EnPassantPosition % 8)) + ""
+ (byte)(8 - (byte)(board.EnPassantPosition / 8)) + " ";
  }
  else
  {
   output += spacer + "- ";
  }
  if (!boardOnly)
  {
   output += board.FiftyMove + " ";
```

```
  output += board.MoveCount + 1;
 }
 return output.Trim();
}
```

The second method is a Board constructor that will accept a FEN string and create a Chess Board based on the content of the string. Strictly speaking you will not need this code. I only use this to allow people to enter FEN strings in the user interface. Since FEN is a standard used in many chess programs allowing users to input FEN strings will allow them to visualize chess positions they might find on the internet.

```
internal Board(string fen) : this()
{
 byte index = 0;
 byte spc = 0;

 WhiteCastled = true;
 BlackCastled = true;
 byte spacers = 0;
 WhoseMove = ChessPieceColor.White;
 if (fen.Contains("a3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 40;
 }
 else if (fen.Contains("b3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 41;
 }
 else if (fen.Contains("c3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 42;
 }
 else if (fen.Contains("d3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 43;
 }
 else if (fen.Contains("e3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 44;
 }
 else if (fen.Contains("f3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 45;
 }
 else if (fen.Contains("g3"))
 {
  EnPassantColor = ChessPieceColor.White;
  EnPassantPosition = 46;
 }
 else if (fen.Contains("h3"))
```

```
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 47;
}

if (fen.Contains("a6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 16;
}
else if (fen.Contains("b6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 17;
}
else if (fen.Contains("c6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition =18;
}
else if (fen.Contains("d6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 19;
}
else if (fen.Contains("e6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 20;
}
else if (fen.Contains("f6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 21;
}
else if (fen.Contains("g6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 22;
}
else if (fen.Contains("h6"))
{
 EnPassantColor = ChessPieceColor.White;
 EnPassantPosition = 23;
}
foreach (char c in fen)
{
 if (index < 64 && spc == 0)
 {
  if (c == '1' && index < 63)
  {
   index++;
  }
  else if (c == '2' && index < 62)
  {
   index += 2;
```

```
    }
    else if (c == '3' && index < 61)
    {
     index += 3;
    }
    else if (c == '4' && index < 60)
    {
     index += 4;
    }
    else if (c == '5' && index < 59)
    {
     index += 5;
    }
    else if (c == '6' && index < 58)
    {
     index += 6;
    }
    else if (c == '7' && index < 57)
    {
     index += 7;
    }
    else if (c == '8' && index < 56)
    {
     index += 8;
    }
    else if (c == 'P')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Pawn, ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'N')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Knight,
ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'B')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Bishop,
ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'R')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Rook, ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'Q')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Queen,
ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
```

```
     index++;
    }
    else if (c == 'K')
    {
     Squares[index].Piece = new Piece(ChessPieceType.King, ChessPieceColor.White);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'p')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Pawn, ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'n')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Knight,
ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'b')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Bishop,
ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'r')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Rook, ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'q')
    {
     Squares[index].Piece = new Piece(ChessPieceType.Queen,
ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == 'k')
    {
     Squares[index].Piece = new Piece(ChessPieceType.King,
ChessPieceColor.Black);
     Squares[index].Piece.Moved = true;
     index++;
    }
    else if (c == '/')
    {
     continue;
    }
    else if (c == ' ')
    {
     spc++;
    }
```

```
}
else
{
 if (c == 'w')
 {
  WhoseMove = ChessPieceColor.White;
 }
 else if (c == 'b')
 {
  WhoseMove = ChessPieceColor.Black;
 }
 else if (c == 'K')
 {
  if (Squares[60].Piece != null)
  {
   if (Squares[60].Piece.PieceType == ChessPieceType.King)
   {
    Squares[60].Piece.Moved = false;
   }
  }
  if (Squares[63].Piece != null)
  {
   if (Squares[63].Piece.PieceType == ChessPieceType.Rook)
   {
    Squares[63].Piece.Moved = false;
   }
  }
  WhiteCastled = false;
 }
 else if (c == 'Q')
 {
  if (Squares[60].Piece != null)
  {
   if (Squares[60].Piece.PieceType == ChessPieceType.King)
   {
    Squares[60].Piece.Moved = false;
   }
  }
  if (Squares[56].Piece != null)
  {
   if (Squares[56].Piece.PieceType == ChessPieceType.Rook)
   {
    Squares[56].Piece.Moved = false;
   }
  }
  WhiteCastled = false;
 }
 else if (c == 'k')
 {
  if (Squares[4].Piece != null)
  {
   if (Squares[4].Piece.PieceType == ChessPieceType.King)
   {
    Squares[4].Piece.Moved = false;
   }
  }
```

```
 if (Squares[7].Piece != null)
 {
  if (Squares[7].Piece.PieceType == ChessPieceType.Rook)
  {
   Squares[7].Piece.Moved = false;
  }
 }
 BlackCastled = false;
}
else if (c == 'q')
{
 if (Squares[4].Piece != null)
 {
  if (Squares[4].Piece.PieceType == ChessPieceType.King)
  {
   Squares[4].Piece.Moved = false;
  }
 }
 if (Squares[0].Piece != null)
 {
  if (Squares[0].Piece.PieceType == ChessPieceType.Rook)
  {
   Squares[0].Piece.Moved = false;
  }
 }
 BlackCastled = false;
}
else if (c == ' ')
{
 spacers++;
}
else if (c == '1' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 1);
}
else if (c == '2' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 2);
}
else if (c == '3' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 3);
}
else if (c == '4' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 4);
}
else if (c == '5' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 5);
}
else if (c == '6' && spacers == 4)
{
 FiftyMove = (byte)((FiftyMove * 10) + 6);
}
else if (c == '7' && spacers == 4)
```

```
 {
  FiftyMove = (byte)((FiftyMove * 10) + 7);
 }
 else if (c == '8' && spacers == 4)
 {
  FiftyMove = (byte)((FiftyMove * 10) + 8);
 }
 else if (c == '9' && spacers == 4)
 {
  FiftyMove = (byte)((FiftyMove * 10) + 9);
 }
 else if (c == '0' && spacers == 4)
 {
  MoveCount = (byte)((MoveCount * 10) + 0);
 }
 else if (c == '1' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 1);
 }
 else if (c == '2' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 2);
 }
 else if (c == '3' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 3);
 }
 else if (c == '4' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 4);
 }
 else if (c == '5' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 5);
 }
 else if (c == '6' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 6);
 }
 else if (c == '7' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 7);
 }
 else if (c == '8' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 8);
 }
 else if (c == '9' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 9);
 }
 else if (c == '0' && spacers == 5)
 {
  MoveCount = (byte)((MoveCount * 10) + 0);
 }
}
```

```
  }

}
```

This concludes the post on Forsyth–Edwards Notation.

Some Performance Optimization Advice

Over the last year of development of my chess engine, much of the time has been spent optimizing my code to allow for better and faster move searching. Over that time I have learned a few tricks that I would like to share with you.

Measuring Performance

Essentially you can improve your performance in two ways:

1. Evaluate your nodes faster
2. Search fewer nodes to come up with the same answer

Your first problem in code optimization will be measurement. How do you know you have really made a difference? In order to help you with this problem you will need to make sure you can record some statistics during your move search. The ones I capture in my chess engine are:

1. Time it took for the search to complete.
2. Number of nodes searched

This will allow you to benchmark and test your changes. The best way to approach testing is to create several save games from the opening position, middle game and the end game. Record the time and number of nodes searched for black and white.

After making any changes I usually perform tests against the above mentioned save games to see if I have made improvements in the above two matrices: number of nodes searched or speed.

To complicate things further, after making a code change you might run your engine 3 times and get 3 different results each time. Let's say that your chess engine found the best move in 9, 10 and 11 seconds. That is a spread of about 20%. So did you improve your engine by 10%-20% or was it just varied load on your pc. How do you know? To fight this I have added methods that will allow my engine to play against itself, it will make moves for both white and black. This way you can test not just the time variance over one move, but a series of as many as 50 moves over the course of the game. If last time the game took 10 minutes and now it takes 9, you probably improved your engine by 10%. Running the test again should confirm this.

Finding Performance Gains

Now that we know how to measure performance gains lets discuss how to identify potential performance gains.

If you are in a .NET environment then the .NET profiler will be your friend. If you have a Visual Studio for Developers edition it comes built in for free, however there are other third party tools you can use. This tool has saved me hours of work as it will tell you where your engine is spending most of its time and allow you to concentrate on your trouble spots. If you do not have a profiler tool you may have to somehow log the time stamps as your engine goes through different steps. I do not suggest this. In this case a good profiler is worth its weight in gold. Red Gate ANTS Profiler is expensive but the best one I have ever tried. If you can't afford one, at least use it for their 14 day trial.

Your profiler will surly identify things for you, however here are some small lessons I have learned working with C#:

- Make everything private
- Whatever you can't make private, make it sealed
- Make as many methods static as possible.
- Don't make your methods chatty, one long method is better than 4 smaller ones.
- Representing your chess board as an array [8][8] is slower then representing it as an array [64]
- Replace int with byte where possible.
- Return from your methods as early as possible.
- Stacks are better than lists
- Arrays are better than stacks and lists.
- If you can define the size of the list before you populate it.
- Casting, boxing, un-boxing is evil.

Further Performance Gains:

I find move generation and ordering is extremely important. However here is the problem as I see it. If you evaluate the score of each move before you sort and run Alpha Beta, you will be able to optimize your move ordering such that you will get extremely quick Alpha Beta cutoffs. This is because you will be able to mostly try the best move first.

However the time you have spent evaluating each move will be wasted. For example you might have evaluated the score on 20 moves, sort your moves try the first 2 and received a cut-off on move number 2. In theory the time you have spent on the other 18 moves was wasted.

On the other hand if you do a lighter and much faster evaluation say just captures, your sort will not be that good and you will have to search more nodes (up to 60% more). On the other hand you would not do a heavy evaluation on every possible move. As a whole this approach is **usually faster**.

Finding this perfect balance between having enough information for a good sort and not doing extra work on moves you will not use, will allow you to find huge gains in your search algorithm. Furthermore if you choose the poorer sort approach you will want to first to a shallower search say to ply 3, sort your move before you go into the deeper search (this is often called Iterative Deepening). This will significantly improve your sort and allow you to search much fewer moves.

Performance Reconstruction Phase Two

It's this time again. The time where I realize that I made a poor decision somewhere in my design and a small portion of my code has to be re-written.

I already spoke about this briefly about this. Currently my chess engine will make all possible moves for a position ahead of time and evaluate each resulting chess board before entering Alpha Beta. Although this gives me an almost perfect sorting algorithm (just sort the fully evaluated positions), on a whole this design is not very efficient. The problem lies in the fact that often the algorithm will be evaluating moves that I will never explore due to an Alpha Beta cut-off. Imagine having 30 possible moves, making each move, and evaluating the score of the resulting position, just to later find out there is a cut off in the 5th position. This means that I have just evaluated 25 unnecessary positions.

The new design makes one move at a time and generates the next move only after the Alpha Beta call. This means that I need a separate algorithm for sorting, choosing which moves to make first to give me the best chance for a cut-off. This sorting is done by a tested algorithm called:

MVV/LVA *Most Valuable Victim, Least Valuable Attacker*. This works exactly as it reads, a move where a pawn attacks a queen would be sorted first, king attacking pawn would be sorted last.

I have already implemented this new Alpha Beta and noticed a significant performance improvement over the previous version. The current version with the new algorithm of Chess Bin Chess now searches to ply 6 (from 5) on Medium Setting and Ply 7 on Hard setting. Although previously the Hard setting was already searching to ply 7 it was painfully slow. On ply 7 I can now easily do one move per 30 seconds average.

I will be updating all of the posts with the newest version of the source code over the next few weeks.

Update January 19th 2010 - Well it took longer than I thought but all the posts are now updated to the new faster source code. It took so much longer to debug all the code then expected. I found so many mistakes, however I think now I have a fairly stable and bug free version.

Transposition Table and Zobrist Hashing

Due to the complexity of this topic I have divided this post into two entries. This article will discuss the Transposition Table and Zobrist Hashing techniques with no code samples. The second part in the series will discuss the code used in my chess engine.

An Alpha Beta search will often have to consider the same position several times. This occurs due to the fact that in chess there are many ways to reach the same piece arrangement. For this reason, most chess engines implement a Transposition Table that stores previously searched positions and evaluations.

The problems

The first problem is that although same chess positions do often re-occur during an alpha beta search, we can only count on having this happen somewhere between 15%-20%. We can't know which positions these will be ahead of time so for every 100 entries we save in our Transposition Table we might only use 20 entries. Hence whatever Transposition Table scheme we choose, it has to be very efficient because we will be storing and searching more useless entries than useful ones.
In a perfect world we would have the ability to simply save every single node we search in our Transposition Table. Unfortunately this is simply not practical. The memory requirements for this scheme would be higher than most computers can accommodate. Furthermore the time wasted on searching such a large table would outweigh any time saving benefits. So we have to accept that our Transposition Table is limited in size will not store all the nodes we search.

Implementation

First we need to figure out how uniquely identify each position we come across. This has to be extremely efficient since we will have to do this for every node in our search. Simply converting the chess board to a string of values like FEN is far too slow.

Zobrist Hashing

Fortunately for us a process for indexing game positions called Zobrist Hashing has already been invented by professor at the University of Wisconsin by the name of Albert Zobrist.

The Zobrist Hash uniquely representing our chess board will be a 64 bit variable. In C# we can implement this as an unsigned long (ulong). We calculate a chess boards Zobrist Hash using the following steps

1. Generate an **array of 12 * 64 Pseudo Random** numbers representing each chess piece type on each of the 64 positions on a chess board. You only do this once at when your chess engine initiates. This will give you a single hash value for every single chess piece for every single position on the board. You will have to elect which portions of the array represent which chess piece. Let's say you choose to have the first 64 entries in your array represent the white pawn. So the 9th value in your array will be White Pawns A2 square etc.
2. For each chess piece on the chess board XOR its positions random number against the current Zobrist hash value. So a white pawn on B2 might be the 10th value in the array.

This is easy enough, however we don't necessary want to calculate the Zobrist hash from scratch each time we make a move. That would be very slow and would make the whole exercise futile. For this reason each time a move is made on our chess board, whether during game play or alpha beta search we simply update the Zobrist Hash by:

1. XOR the chess pieces previous positions random number against the current Zobrist hash value, this erases the chess piece from our hash.
2. XOR the chess pieces new positions random number against the current Zobrist hash value adding the chess piece back to its new position.

A bit about the **XOR** operator:

If you don't understand what I mean by XOR in the above paragraphs you may want to read this:

XOR or Exclusive Or is one of the standard bit operators available to you in most programming languages. By bit operator I mean it allows you to manipulate the individual bits in a value. If you are not sure what that means, you might need to Google this first. The one nice side effect of the XOR operator is that if you XOR a value 2 times by the same value it will return to its original value.

For example:
1110 XOR 1001 = 0111
0111 XOR 1001 = **1110**

In order to add and remove chess pieces from our hash we will be using the XOR operator to add the chess piece to a position and then use the XOR again to erase it once it moves. For example if we XOR the Zobrist Hash representing our chess board against the 64 bit number representing a white pawn on A2 it would be like adding the pawn to the chess board on A2. If we do it again we remove or erase the pawn from A2. We can than XOR the hash against the 64 bit value representing a white pawn on A3. The last 2 operations would essentially move the white pawn from A2 to A3.

Collisions

Right about now you might have noticed that a 64bit value is not large enough to represent every single possible chess position. So using the above scheme it is possible to have 2 different chess positions evaluate to the same 64 bit hash value. This is called a collision and no matter how you implement this, you will always have a small chance of a collision. The key here is to minimise the chance of a collision down to a small enough value so that the speed gain you get from having a transposition table (and perhaps constantly deeper search) outweighs the negative effect of the possibility of a collision. In most chess circles a 64 bit value is considered to be large enough to make collisions not a practical problem.

Transposition Table Contents

So what goes in a transposition table entry? Here are the items included in my Transposition Table:
Hash: This is a Zobrist Hash representing the chess position
Depth: The depth remaining in the alpha beta search. So depth 5 would mean the score is recorded for a 5 ply search. This can also be referred to as the Depth of the Search Tree.
Score: The evaluation score for the position.
Ancient: Boolean flag, if false the node will not be replaced with a newer entry.
Node Type: There are 3 node types, **Exact**, **Alpha** and **Beta**. Exact means this is an exact score for the tree. However in the events that an alpha beta cut-off occurs we can't use the score as an exact score. An

Alpha Node Type means the value of the node was at **most** equal to Score. The Beta Node Type means the value is at **least** equal to score.

Replacement Schemes

Since your Transposition Table can't hold all the moves searched in a game you will have to start replacing your entries fairly soon. In the same time you don't simply want to replace all entries regardless of their usefulness. For this reason in the event that I find an entry that is useful (was used in a lookup), I set a Boolean flag Ancient to false, meaning doesn't replace. This way you always replace entries that are unused and keep the ones that were historically useful. To prevent your table from filling up with Ancient nodes from 10 turns ago, the Ancient flag gets set to true for every entry after every search.

Table Lookup

The last problem we have to find is how to we quickly search a Zobrist Table. We can't just do a for loop. This would be slow. The trick is actually in how we store the entries in the first place. Rather than simply adding an entry in the order we received them we calculate the entry index as follows:
Table Entry Index = Hash mod TableSize
This way when we search the table to see if a certain Hash exists we know there is only one place it could be stored Table[Hash mod TableSize]
That's it for this article on the Transposition Table.