

5	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47	72	73	74	75	76	77	78	79
	64	65	66	67	68	69	70	71								
3	0x50	0x51	0x52	0x53	0x54	0x55	0x56	0x57	88	89	90	91	92	93	94	95
	80	81	82	83	84	85	86	87								
2	0x60	0x61	0x62	0x63	0x64	0x65	0x66	0x67	104	105	106	107	108	109	110	111
	96	97	98	99	100	101	102	103								
1	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	120	121	122	123	124	125	126	127
	112	113	114	115	116	117	118	119								

Pieces on the board in monsoon are represented as simple bit patterns stored in this array. One bit in the sequence signifies color and three bits signify piece type. Therefore only 1/2 byte of space is required to store a piece on the board. I use the other space in the board elements to maintain several quick-access indexes. These are used to map pieces to locations on the board so that, for example, when I want to do something with every white pawn I do not have to scan the entire board and find them. Instead, I just process the contents of my white pawn list one at a time and get the location of all living white pawns.

In addition to knowing where all the pieces are, of course, the engine must keep track of whose turn to move it is, what castling options are still legal, where en-passant can be played, how close to a 50-move draw it is, etc. All of these are aspects of a position structure (from [chess.h](#)):

```

//
// POSITION
//
typedef struct _POSITION
{
    UINT64 u64NonPawnSig;           // hash signature
    UINT64 u64PawnSig;             // pawn hash signature

    SQUARE rgSquare[128];          // where the pieces are,
                                    // also, the attack table

    ULONG uToMove;                 // whose turn?
    ULONG uFifty;                  // 50 moves w/o progress = draw
    FLAG fCastled[2];              // record when sides have castled
    BITV bvCastleInfo;             // who can castle how?
    COOR cEpSquare;                // en-passant capture square

    COOR cPawns[2][8];             // location of pawns on the board
    ULONG uPawnMaterial[2];        // pawn material of each side

    COOR cNonPawns[2][16];         // location of pieces on the board
    ULONG uNonPawnMaterial[2];     // piece material of each side
    ULONG uNonPawnCount[2][7];    // number of non-pawns / type
                                    // 0 and 1 are the sum,
                                    // 2..6 are per PIECE_TYPE

    ULONG uWhiteSqBishopCount[2]; // num bishops on white squares
    COOR cDanger[2][3];           // pieces in danger (from eval)
    SCORE iKingScore[2];          // king safety score (from eval)
    ULONG uMinMobility[2];        // min piece mobility (from eval)
}
POSITION;

```

Move Generation:

My old strategy for move generation was based on an idea I stole from GNUchess 4.x. Based upon conversations I've had with other programmers, I believe that Diep, Ferret and Quark are all using a similar idea (or have expanded on this idea somehow).

The idea involves using pre-computed tables to derive the next location on the board that a piece can move to given the piece's type, present location and starting location. For example, imagine I have a queen on c3. NextSquare[QUEEN][c3][c3] might return b2. NextSquare[QUEEN][c3][b2] would then return a1. NextSquare[QUEEN][c3][a1] would realize that there are no more squares in the down-left direction for the queen and might then return c2. And so on. (c1, d2, e1...)

But while generating possible moves sometimes we have to change directions before hitting the edge of the board. Like, for example, when we run into a friendly piece and are blocked by it (or when we run into an opponent's piece and generate the possible capture move). In these cases, the NextDir table is used instead of the NextSquare one. NextDir maps a piece type, starting square and present square into the first square in the next direction for that piece. Again, consider our queen on c3. Imagine we are moving in an up-right and are considering e5. We find a friendly pawn. It does not

make sense to continue using NextSquare and look at f6, g7, h8 now, because these squares are blocked by the pawn at e5. Instead we access NextDir[QUEEN][c3][e5] and get back c4, for instance. Both NextDir and NextSquare can indicate a complete loop (no next square) by returning the starting square again.

For knights and kings, NextDir and NextSquare are the same, of course. A move generator based on this technique is very simple to read:

```
cSquare = g_cNextSquare[pShiftPiece][cStart][cStart];
while (cSquare != cStart)
{
    ASSERT(ONBOARD(cSquare));

    xPiece = pos->pSquare[cSquare];

    //
    // Empty square? Add a move.
    //
    if (IS_EMPTY(xPiece))
    {
        AddMove(pos, cStart, cSquare, pPiece, 0);
        cSquare = g_cNextSquare[pShiftPiece][cStart][cSquare];
    }

    //
    // Not empty? Add a capture if its an enemy piece.
    //
    else
    {
        if (OPPOSITE_COLORS(xPiece, pPiece))
        {
            AddMove(pos, cStart, cSquare, pPiece, xPiece, 0);
        }
        cSquare = g_cNextDir[pShiftPiece][cStart][cSquare];
    }
}
```

Now, after all that explanation, I'll tell you why I don't use these tables anymore. In my testing I found that, for my engine, they are slower than a more "direct" way of generating moves. Perhaps this is because of memory bandwidth.

So instead I now use a simple 0x88 loop to generate moves. The move generator lives in [generate.c](#). Here's an example of how it works:

```
const INT g_iBDeltas[] = {
    -15, -17, +15, +17, 0 };

// ...

int x = 0;
COORD c;
PIECE p;

while(g_iBDeltas[x] != 0)
{
    c = cBishop + g_iBDeltas[x];
    while(IS_ON_BOARD(c))
    {
        p = pos->rgSquare[c].pPiece;
        if (IS_EMPTY(p))
        {
            _AddNormalMove(pStack, pos, cBishop, c, 0);
        }
        else
        {
            if (OPPOSITE_COLORS(p, pos->uToMove))
            {
                _AddNormalMove(pStack, pos, cBishop, c, p);
            }
            break;
        }
        c += g_iBDeltas[x];
    }
    x++;
}
```

Move Data Structure

My old codebase used a (somewhat large) struct to represent a move. I would then pass pointers to

moves all over the place in the move generator code. When I rewrote my engine a while back I decided that this was dumb and that I would make my moves 32-bit numbers. This, of course, allows them to be passed directly on the stack by value and eliminates a lot of pointer dereferencing in the generator code. Here's the definition of a MOVE in typhoon, which comes from [chess.h](#):

```
typedef union _MOVE
{
    UINT move;
    struct
    {
        UCHAR cFrom      : 8;
        UCHAR cTo        : 8;
        UCHAR pMoved     : 4;
        UCHAR pCaptured : 4;
        UCHAR pPromoted  : 4;
        UCHAR bvFlags    : 4;
    };
} MOVE;
```

The ": 8" and ": 4" constructs are called "bitfields" and they request that the compiler use eight or four bits to represent a struct member. When you use bitfields it's always good to make sure the compiler generates code that actually listens to your size requirements; it can take some fiddling to make it work. They are not the most portable part of the C language but they sure are handy sometimes.

Vector Delta Tables:

An attack vector table answers the question: given two squares, A and B, what kind of pieces sitting on A would attack B if no blocking pieces were in-between. An attack delta table answers the question: given two squares on a line, A and B, how do I move to get from A to B.

In my engine I use each bit in the contents of the vector table to represent a distinct piece: black pawn, white pawn, bishop, knight, rook, queen, king. The delta table element contents are just an signed number -- the direction of the first step on the ray from the initial square to the destination or zero if no such ray exists.

These tables are indexed in a neat way. At first I thought about implementing these as two dimensional arrays of 64 elements in each dimension (for a total of 4K entries). But after reading some articles on CCC and thinking about the problem a little it became clear that these tables need only be 256 elements in length. They are indexed by subtracting the destination square index from the starting square index and adding 128. This results in a number between 0 and 256.

Tord Romstad's Glaurung engine uses a clever bit of pointer arithmetic to save the step of adding 128. Here he declares a data structure that is indexed the same way (from square - to square + 128) but also declares a pointer to the "middle" of the array. By using positive and negative indexes off this pointer (+/- 128) he can use constructs like `Distance[from - to]`:

```
uint8 Distance_[256];
uint8 *Distance = Distance_+128;
```

Monsoon generates the vector delta tables as part of [program data structure initialization](#). Specifically, see `InitializeVectorDeltaTable`. These tables are used extensively in check detection, move generation, X-ray attack detection, PGN file parsing for book creation, static exchange evaluation, and position evaluation.

Search:

Algorithm

Monsoon uses a derivative of the alpha-beta search called a principal variation search (PVS). PVS is like normal alpha-beta in that it prunes away whole branches of the minimax tree that do not influence the value of the root position. However with PVS only the first move at a given node is searched with a full alpha-beta window. Subsequent moves are searched with a minimal window of $\beta = \alpha + 1$.

Ideally the best move is searched first and all subsequent moves should fail low. With the minimal window, a great deal of work is saved in the process. However if a subsequent move fails high it must then be re-searched with a larger window. Because of good move ordering, such re-searches are rare and the practice of narrowing the search window artificially for subsequent moves saves work. More information about PVS and many other search techniques and game structures can be found at [this website](#). You can also read my (very cluttered) [search.c](#) file for a working implementation.

Nullmove Pruning

A nullmove search is a forward pruning technique used heavily in microcomputer chess based on this idea: moving twice in a row in chess is a strong (though illegal) advantage. To help understand why this is so powerful, imagine you were playing a game of chess and you said to me, "My board position is so strong that you can go ahead and move twice in a row... it won't help you". Because moving twice in a row is such a great advantage, if I indeed could not make any progress against you when you "passed" on your turn I could conclude that my position was terrible.

So in non-root plies computers often try a nullmove (i.e. "pass") before any other real move. This lets the opponent move twice in a row. After this "pass" a recursive search of the position is performed to a reduced depth. If this reduced depth nullmove search fails high then the computer assumes that doing something should be better than doing nothing at all and proceeds to fail high at the node in question without doing further work. This saves a lot of time because the engine did not have to generate any moves or search any move subtrees under this node. All it had to do was a reduced depth search under a "pass".

The amount of reduction in the nullmove search is called R. Typical values of R range between 1 and 3 plies. So, a nullmove R=2 would search the position after a pass to depth - R - 1 ply.

Like all good things, though, nullmove has some dangers. The first is that there are some positions where it is an advantage to pass: *zugzwang* positions. These positions violate the nullmove assumption that "doing something is better than doing nothing". Typically *zugzwang* positions only occur in the late endgame, so most engines simply disable the nullmove search when the position is deemed too late in the game. Another way to deal with this is to "confirm" all nullmove results with a nullmove verification search.

Monsoon uses an idea called adaptive nullmove searching where R in the reduction formula changes depending on the position under consideration. Ernst Heinz describes an adaptive nullmove strategy that selects R based on the remaining depth. It reduces the nullmove search more aggressively when high search depth remains. For more information about Ernst's version of adaptive nullmove see his book *Scalable Search in Computer Chess* or [website](#). I use another idea that scales R based on the number of pieces on the board. This code is at the bottom of my search-support module, [searchsup.c](#).

In the past I've played with two other ideas involving nullmove that I call "quick pruning" and "avoid nullmove". I'll describe each briefly here despite the fact that both are disabled in the current codebase.

"Quick pruning" is the ability to reduce the depth of the nullmove search when it seems clear by positional analysis that the nullmove search is sure to succeed. Imagine that in the position after the nullmove your opponent's score is terrible. Further, imagine that he has a piece hanging or trapped. On top of that, he has no threats to your king. If you run the nullmove search it seems likely that it will result in things getting worse for your opponent. It will fail low at the ply after the nullmove meaning that the nullmove search will fail high and allow you to prune. Quick pruning strives to recognize these positions and cheapen the cost of the nullmove search (or, in drastic cases, avoid the search altogether and just prune).

"Avoid nullmove" is the ability to recognize a position where a nullmove is unlikely to succeed and skipping the nullmove search. For example, imagine that after the nullmove you find that your opponent is in great shape (score way > beta). Imagine you also find that he's about to win one of your pieces. It's likely that your opponent will fail high after the nullmove (which means the nullmove will fail low and not allow you to cutoff). In this case it might make sense to skip the nullmove altogether and save the cost of the reduced depth search.

I'm interested in the results of any experiments you do with these two ideas. I think they are both good but I have never been able to get them to clearly outperform standard nullmove searching and thus have them both commented out.

Search Extensions

In some positions a player's move is forced. Such positions are good candidates for search line extension. My program's search extensions include: intelligent checks, nullmove threat detection, *zugzwang* positions, single legal reply to check, multi-piece checks, forced recaptures, "singular" reply to check, and productive passed pawn pushes in the endgame.

My engine uses the now standard idea of fractional ply move extensions. This means that some extensions may extend the search by less than one full ply.

The extension code is split into two parts in my code: the part that considers the present position for extension and the part that considers the move we are making. The former lives at the top of `search` in [search.c](#) while the latter lives in `ComputeMoveExtension` inside [searchsup.c](#).

Multithreaded Search

When typhoon is running on a multi-processor machine it uses a tree splitting algorithm somewhat similar to PVSplit to search the chess tree in parallel. However instead of just splitting the tree on PV nodes I split the tree anywhere that I think is an "all" node. This includes PV nodes and "alpha" nodes (nodes where all moves will fail low and facilitate a fail high at the previous ply). Specifically, after the first move has been searched I will search the rest in parallel if it's a PV node or if the first N moves all fail low. Right now N=3 but this is a matter of experimentation. Because a little work is

involved to split the tree, you really don't want a fail high at a split node -- it's just a waste of time. Moreover I don't split when there's not enough useful depth left to warrant the extra work. You can see the conditions of where I split the search in the `#ifdef MP` block(s) inside [search.c](#). The code that splits the position lives in [split.c](#).

Getting the parallel search working involved some redesign but it was not anywhere near as hard as some chess programmers lead you to believe. You need a place for individual searcher threads to store different positions, different move stacks and different ply counters. I use a "searcher thread context" structure for this purpose. This can be found in [chess.h](#).

```
typedef struct _SEARCHER_THREAD_CONTEXT
{
    POSITION sPosition;
    MOVE_STACK sMoveStack;
    POSITION_HISTORY sPositionHistory[MAX_PLY_PER_SEARCH];
    MOVE mvKiller[MAX_PLY_PER_SEARCH][2];
    ULONG uPly;
    COUNTERS sCounters;
    FLAG fImAHelper;
    SPLIT_INFO *pSplitInfo;
    ULONG uPositional;
}
SEARCHER_THREAD_CONTEXT;
```

I pass this struct around in `Search`, `QSearch`, `Eval` and `Generate` which were rewritten to use data from the context passed in instead of a global variable. I would recommend either doing something like this or using a class if you are trying to implement a parallel search. I would not recommend TLS (Thread Local Storage, a win32 feature to support per-thread identifiers) because they are slower and non-portable.

Move Ordering:

When searching a move tree, the order you consider moves at a given position is important. Good move ordering reduces the size of the search tree by causing fail highs (beta cutoffs) sooner. Here is monsoon's move ordering scheme:

1. Hash move
2. Winning captures (judged by the SEE)
3. Even captures (judged by SEE)
4. Killer moves
5. Other non-captures (ranked by history value)
6. Losing captures (judged by SEE)

The hash move is the move attached to a hit in the transposition table. The score of a capture move is judged by the static exchange evaluator (SEE): a routine that tries to predict the material gain or loss of a sequence of captures quickly, without moving pieces on the board. A lot of people have asked me how this works. Basically it makes a list of all pieces that attack or defend a certain square and plays out the material exchange to assign it a value. There are a couple of wrinkles: first, either side may opt out of the trade at any point when they are on the move by doing something else. Second, my SEE looks for pieces that can't move because they are pinned to the king. Also after each stage of the trade it checks to see if any new attackers (or defenders) have been uncovered. These things make it slower than some SEEs but also more accurate. The dominant part of the routine itself, a function called `GetAttackers`, is coded in x86 assembly language to maximize speed. You can read it in [x86.asm](#). There's also a fallback routine written in C that lives in [see.c](#).

Killer moves are a typical dynamic move ordering idea where good moves (i.e. refutations or beta cuts) at a certain level (ply / distance from the root) in the search are stored and ordered sooner in subsequent nodes at the same level. My code only allows non-capturing killer moves. The basic idea behind the killer move is that if you have a crushing response at some distance from the root on one branch of the search tree that same move might prove to be crushing (again) on another, different branch. Threatening passed pawn pushes are a good example of this killer move principle. In addition to moves that fail high, my engine sticks mate-threatening responses to null moves into the killer lists. It also considers killers from both the current ply and current ply + 2 when ranking moves.

The history heuristic, like the killer move idea, is very widely used. History values for moves increase as that move proves to be smart (by causing beta cutoffs, for example). Subsequently the move is ordered sooner in the search.

I don't know where else to put this rant so I apologize in advance for the non sequitur. There is a somewhat new idea called "history pruning" that was very confusing to me when I first heard about it. The idea is to build up a per-move percentage that estimates the likelihood of the move to fail high. This percentage is based on how the move has performed in the already searched nodes of the tree. The pruning comes into play when we reduce the depth of search below moves that are unlikely to cut off.

Why am I talking about this now? Well, as you can tell from the description of the pruning idea it has very little to do with the "history heuristic". Specifically: **history heuristic increases a per move counter by an amount related to the depth of search remaining when the cutoff occurred** and

history pruning tries to estimate the likelihood of a move to cut off. Don't confuse them.

Back to move ordering. Typhoon searches losing captures (as judged by the SEE) last.

That's about it for move ordering, basically. If you implement something like the above your engine will do pretty well. I've heard of people trying the killer moves before the winning captures even, the thought being that the hash move and the killer moves are free (i.e. you don't have to generate them) so that if one causes a beta cutoff then you have a very inexpensive (therefore fast) node. I've never tried this but it seems to be a sound idea.

Just to throw a little confusion into the mix I'll confess: I do have a couple of other dynamic ideas that affect the search order. For example, non capturing moves of an en prise piece tend to be considered sooner. Non capturing moves to squares controlled by the enemy are deferred until later. And capturing moves of an en prise piece that appear to be productive are tried sooner in the winning captures list. I also give a small bonus to checking moves so that they are considered before non-checking moves of the same value.

The full move scoring code lives in routines like `_ScoreAllMoves`, `_ScoreQSearchMovesInclChecks`, and `_ScoreQSearchMovesNoChecks` in [generate.c](#). They call the SEE which lives in [see.c](#).

Position Evaluation:

My engine does most of its evaluation work at leaf positions in the search tree. That said, the engine keeps track of piece-square bonuses and material strengths incrementally -- these are part of a POSITION structure and are updated at every move. It also runs a full eval on some internal nodes when it's trying to make intelligent pruning/extension decisions.

Before running a normal evaluation at a leaf position the engine sees if it can use a specially coded endgame evaluation routine.

Before running a full evaluation on a chess position, typhoon tries a "lazy evaluation". That is, given the current alpha..beta bounds and a very rough approximation of the current positions score, if a position "cannot possibly" fall inside the alpha..beta window then return a fail-high (>alpha) or fail-low (eval.c for details).

Once it has been determined that a position will be fully evaluated the first step is to compute a bunch of scores related to the pawn structure. Because many different board configurations share the same pawn structure, I hash the pawn structure scores (and some other data about each pawn structure). This hashtable hitrate is very high, something on the order of 99%. Here's what a pawn hash entry looks like (from [chess.h](#)):

```
typedef struct _PAWN_HASH_ENTRY
{
    volatile ULONG uCheckedOut;
    signed short iScore[2];
    UINT64 u64Key;
    BITBOARD bbPawnLocations[2];
    BITBOARD bbPasserLocations[2];
    BITBOARD bbStationaryPawns[2];
    UCHAR uCountPerFile[2][10];
    UCHAR uNumRammedPawns;
    UCHAR uNumUnmovedPawns[2];
}
PAWN_HASH_ENTRY;
```

As you can see, part of each entry is made up of bitboards. This is the only place in my engine that bitboards are used. Here's [bitboard.c](#) and [x86.asm](#) where some bitboard routines live.

The precise formula and term weights that monsoon uses to evaluate a normal chess position are secret and too complicated to explain in any detail in this document. Most of these terms are not just static bonuses and penalties but are rather scaled based on board position, other terms etc. For example the strength of an outposted knight is related to that knight's distance from the enemy king and the strength of the friendly army. Moreover there is some code in the evaluator that is even harder to explain. For example monsoon counts threats to a king... some of these threats are only counted if other threats exist. The severity of the threat penalty is then scaled based on the strength of the enemy army. Also, monsoon changes the weights of some terms based on game stage. If you're interested in how this stuff works, check out [eval.c](#). I'm also interested in any ideas **you** have about position evaluation -- I am a terrible chessplayer and find writing good eval code to be a challenge.

When assessing piece mobility monsoon does not count squares controlled by the enemy because moving there would hang a piece.

If you're trying to write an eval routine you might be interested in a couple of books: [Pawn Power in Chess](#) by GM Hans Kmoch is the basis of a lot of my pawn evaluator stuff. Also [How to Reassess Your Chess](#) by IM Jeremy Silman is a good all-around evaluation book.

Here's a rough overview of the features cares about. I by no means consider myself to be remotely qualified to write a chess eval function -- I am a weak club chess player and do not play enough. But this list may be a useful starting point:

Pawns

- Pawn duos
- Rammed pawns / closed positions
- Backwards pawns and backwards exposed pawns
- Non-backwards pawn-supports-pawn structures
- Passed pawns
- Candidate passers / pawn majorities
- Isolated and isolated exposed pawns
- Doubled pawns
- Pawn dispersion term
- Connected passers
- Connected passers vs. King and Rook code
- Outside passers
- Pawn shielding the king
- Pawn storming the enemy king
- Unstoppable runners
- Assess control of the center

Kings

- Open files, ranks or diagonals nearby
- Half-open files nearby
- Assess control of squares around the king
- Encourage castling
- Discourage loss of right to castle
- Good to support passers with
- Good to attack the other king with
- Get it to the center in the endgame

Knights

- Enemy king-tropism
- Encourage early development
- Discourage trapped knights
- Encourage outposted knights
- Good to block a passer or a backwards pawn
- Good if there is an open file behind
- Knight mobility
- Knight centrality
- Knight relevance
- Bishop can't drive away the knight
- Stronger in closed positions

Bishops

- Enemy king-tropism
- Bonus for Bishops over Knights in endgames with two pawn wings
- Encourage development
- Bonus for good fianchetto
- Detect trapped bishops
- Good vs. Bad bishops
- Detect an active bad bishop
- Bishop mobility
- Two living bishops
- Bishops of opposite colors in endgame

Rooks

- Connected rooks
- Good on half open or full open
- Good to attack passers with
- Good to point near the enemy king
- Rook mobility (horizontal and vertical)
- Good to support friendly passer
- Good to trap enemy king on the edge of the board
- Good to get up to the 7th/2nd rank
- Detect trapped rooks
- Assess control of open files

Queen

- Enemy king tropism
- Friendly king tropism
- Good to point near the enemy king
- Bad to bring it out too early

Misc

- Address material imbalances (pawns for a minor etc)
- Detect stonewall positions
- Detect edge of board mate threats
- Dislike blocked positions

Transposition Tables:

A transposition table (or hash table) is a large data structure containing score information about subtrees recently scored by search. If the same position, P, can be reached in more than one way, the second time that search reaches position P it can, ideally, access the transposition table and retrieve score information for P (and its subtree) without doing any further work.

Therefore one of the first things a search routine does in an engine with a transposition table upon entering a new position is probe the transposition table to see if it can learn anything about the new position cheaply. Likewise, when the search routine has learned something about a position by the (expensive) recursive searching of its subtree, it stores the information in the transposition table so that it can hopefully be reused later.

Each board position in typhoon has a 64-bit signature associated with it. This signature is built based on the current piece configuration, the side to move, the castling rights, etc... all the items that make up a unique board position. A hashing function maps this position signature to an entry in the transposition table where information about the position can be stored. Of course, since the table is limited in size often two positions will map to the same hash table entry. This is called a collision. It is possible, therefore, that when the search tries to store information about a position it find the slot in the hash table it would like to store in is already used. Or when it attempts to get information about a position, it find information about a different position than it sought.

A few things have to be done to effectively deal with collisions. First, the hash entries have to contain some or all of the signature of the position to which they relate. This is so that when search probes the table for data about position P and finds data about position Q it will know that a collision has occurred and that the data in the hash is irrelevant. To deal with collisions while storing information in the hash a replacement scheme is needed. This is a set of rules that govern when an entry already in the table can be overwritten and when it should not be.

My strategy for dealing with collisions (along with the rest of my hash table code) lives in [hash.c](#). Specifically, check out `_SelectHashEntry` for my replacement scheme.

When I wrote the hash code in monsoon I decided to try to make use of the size of a cpu cache line. My hash table entry is 16 bytes in size and I run monsoon on an Athlon with a 64 byte data cache line. So when I hash a position really select a line that could contain information about that position. I use one of these four entries as an always-store table, one as a depth-priority table and the remaining two as locations to store whatever entries get bumped from either the depth or always table entries.

Other than storing the signature of its position, each hash entry also stores the score data about its position. Score data for a node can say "this node is worth exactly X", "this node is worth at least X" or "this node is worth at most X". These are called exact, lower bound and upper bound scores. It is pretty rare that we know the precise value of a position -- exact evaluations can be stored only after a node has been fully searched and found to be on the PV. Upper and lower bounds, far more frequent, can be stored when the search fails low or high (respectively). The draft of a node tells us to what depth the search evaluated this position before storing its data in the table.

Typhoon uses about 256Mb of memory for its main transposition table. Each table entry is 16 bytes in length and the hash probing function tries four entries at per probe.

The multi-threaded version of my engine locks 1/64th of the hash table at a time to prevent corruption when more than one searcher thread is using the same entry. This is cheap, contention is very low, and seems to work well. The code locks pawn hash entries on an individual basis (not a whole range like the main hash table) but has a provision for other search threads to multi-probe in the pawn hash table after spinning a while so that they do not block waiting on another searcher thread that is evaluating a position with the same pawn configuration. Because of this I tend to have a larger pawn hash table than most engines.

More information about transposition tables and many other aspects of chess programming [can be found here](#). Transposition tables are a breeding ground for chess program bugs. To get one working well is a non-trivial task. The best way to start is to make sure you fully understand alpha-beta searching and what your hash table is doing. Then read the source code for some other programs that do hashing already.

Miscellaneous:

Typhoon has an opening book of over 3 million position based on about 600,000 expert human games in PGN format. I don't know jack about opening theory so I made the book structure have the ability to learn based on the results of games it plays with particular lines. The book is also pretty customizable -- each line has a weight that can be adjusted on the fly. You can also tag one line as an always/never play. The opening library code lives in [book.c](#).

My engine also supports Eugene Nalimov's endgame tablebases and uses all 3, 4, and 5 man files while playing online. To do this I'm using Eugene's code (with permission) by simply pulling it from the Crafty codebase. My engine also has some simple interior node recognizers for wrong color bishop endgames and trivially won KP+K games. These recognizers are based on Thorsten Greiner's excellent program Amy.

Test Suites

I run the [Encyclopedia of Chess Middlegames](#) (ECM), [Win at Chess](#) (WAC) and [Winning Chess Sacrifices](#) (WCSAC) test suites to help measure the impact of changes to the engine and its tactical abilities.

Suite name	Time / move	Score	Date
ECM	20 sec	717 / 879	Aug 2006
WAC	20 sec	296 / 300	Jan 2002

Tournaments

Monsoon played in the [3rd CCT tournament](#) on the Internet Chess Club. Scored 3.0 / 8.0 in a strong field. I'm pretty happy with this result despite the fact that its not particularly good. I had a great time and picked up some new ideas. I even found a big bug in the move ordering. This, coupled with the fact that monsoon's hardware was outclasses by nearly every other engine, made 30th place (of 32) a bit easier to stomach.

Monsoon played in the [4th CCT tournament](#) on the Internet Chess Club. Scored 6.0 / 11.0 in a strong field for 15th out of 47 engines. All in all a fun tournament and a pretty good result.

Monsoon played in the [5th CCT tournament](#). It placed 6th out of 45 engines scoring 6.0 / 9.0, a great result for me. I have not had any time since this tournament in January 2003 to work on the engine, unfortunately.

Thanks!

Thanks to all the members of the [CCC discussion list](#) especially Bob Hyatt (Cray Blitz, [Crafty](#)) and Bruce Moreland (Ferret, [Gerbil](#)) for their patience and willingness to explain chess-programming concepts. Thanks to Eugene Nalimov and Ernst Heniz for their continuing work on high quality endgame tablebases. Thanks to Tim Mann for his work on [xboard / winboard](#). Thanks to Tom Kerrigan ([TSCP](#), Stobor) for publishing TSCP source code which was the first chess engine I read and the reason I became interested in chess programming. Thanks to Thorsten Greiner for writing and publishing the source to his [Amy](#) program and (again) to Bob Hyatt for writing and publishing the source to [Crafty](#). Thanks to Ernst Heinz ([DarkThought](#)) for publishing his excellent research on computer chess.

Thanks to Vincent Diepeveen ([Diep](#)) for his discussions and guidance. Many thanks to Dann Corbit for porting monsoon to Microsoft VC and Intel C++ compilers. Thanks to IM Mark Chapman (ICC handle [agro](#)) for his help with opening book lines and his patient expert analysis of chess positions. Thanks to Peter McKenzie ([LambChop](#)) for discussing chess programming ideas and sharing his thoughts and advice. Finally thanks to Steve Timson (Chester) for sharing his good ideas and listening to my lousy ones... without some of his advice monsoon would surely not be as strong as it is today.