Micro-Max, a 133-line Chess Source

Newly released beta version!

There now is a Winboard version that can handle bigger boards, and a Fairy-Max engine that can run under it (download)

The GUI knows the rules of Capablanca Chess, while Fairy-Max can play a wide variety of board sizes and piece types, as it allows user-defined pieces.

My original aim was to write a chess program smaller than 1024 characters. I could not do it, so far. Even when I dropped the nitty gritty details of the FIDE rules, like castling and en-passant capture, I could not get the size much below 1200 characters.

So I shifted my aim somewhat, and wrote something less minimalistic in up to 2000 characters of source-code. This gave me enough space to implement a (hash) transposition table, checking of the legality of the input moves, and full FIDE rules. Except for under-promotions, which I considered a dull input problem, since the program is unlikely to ever find itself in a situation where it would be useful to play one.

(For real purists: a close-to-minimal version that does understand full FIDE rules including under-promotion can be found here. It measures 1433 characters. The under-promotions are implemented in a single line that wastes 32 characters. To play one, type 1, 2, or 3 as the 5th character of the input move, for promotion to R, B, or N, respectively. If you type nothing, or 0, promotion is to Q.)

As far as I am aware, this still makes micro-Max the smallest C Chess program in existence. A close competitor for this honor, Toledo, measures 2168 characters. Despite its smaller size, micro-Max seems to beat Toledo easily.

On these pages various aspects of micro-Max are described:

    - Pawn Promotions
- Search Algorithm
    - Alpha-Beta Minimax
    - Quiescence Search
    - Do and Undo Move
    - Iterative Deepening
    - Move Sorting
- Transposition Table
    - Hashing
    - Replacement
- Evaluation
    - Material
    - Piece-Square Table
    - Checkmate and Stalemate
    - Delay Penalty
- The Interface
    - Move Legality Checking

An overview of the meaning of all variables in the program can be found here. To make it easier for those who want to study the algorithm, there now also is a version that uses more meaningful (and much longer...) variable names.

Downloading

If you want to try micro-Max on your PC, you can copy-paste the source and compile it yourself. Details on how to do this can be found here.

Future

There are still plenty places where I can scavenge a few characters off the source code. (E.g. `A->K` in stead of `A[0].K`, and `a->X&M^M` in stead of `(a->X&M)!=M`, and perhaps combining the two Zobrist keys in a 64-bit type.) The castling code is also rather dumb and bulky. I hope to be able to compact the code enough (without loss of functionality) to make room for new features, in particular null-move threat detection. I will post the progress of this project regularly on separate pages, so that it does not mess up the tutorial on micro-Max 3.2. If some clearly defined feature is added to future versions of micro-Max, the page explaining it will be included in the index above.

Below is the complete source code of micro-Max 3.2. (Click on the various code lines to go directly to their explanation.) If you want to copy-paste it, it is recommended you do it from here, because if I correct a small bug or typo I am generally too lazy to do it on all other pages where the source occurs. So I do it here, and on the page where the particular feature needing the correction is discussed and highlighted.

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
```

```
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct  _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e4,C=799,Q,N,i;        /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                         /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                           /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                         /* board: half of 16x8+dummy*/
T[1035],                                        /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                         /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                                /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                 /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                      /* first empty or match     */
 a+=i?j:0;                                      /* dummy A[0] if miss & full*/
 if(a->K)                                       /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                         /* examine stored data      */
  if(d>=n)                                      /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;              /* use if window compatible */
   d=n-1;                                       /* or use as iter. start    */
  }X&=~M;Y=a->Y;                                /*       with best-move hint */
  Y=d?Y:0;                                      /* don't try best at d=0    */
 }else d=X=Y=0;                                 /* start iter., no best yet */
 N++;                                           /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                       /* iterative deepening loop */
 {x=B=X;                                        /* start scan at prev. best */
  Y|=8&Y>>4;                                    /* request try noncastl. 1st*/
  m=d>1?-I:e;                                   /* unconsidered:static eval */
  do{u=b[x];                                    /* scan board looking for   */
   if(u&k)                                      /*  own piece (inefficient!)*/
   {r=p=u&7;                                    /* p = piece type (set r>0) */
    j=o[p+16];                                  /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                     /* loop over directions o[] */
    {A:                                         /* resume normal after best */
     y=x;F=G=S;                                 /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                 /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                          /* sneak in prev. best move */
      if(y&M)break;                             /* board edge hit           */
      if(p<3&y==E)H=y^16;                       /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;       /* capt. own, bad pawn mode */
      i=99*w[t&7];                              /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;        /* K capt. or bad castling  */
      if(m>=l)goto C;                           /* abort on fail high       */
……
      if(h=d-(y!=z))                            /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                   /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;               /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}              /* castling: put R & score  */
       if(p<3)                                  /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+             /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);          /*         squares plus bias */
```

```
        if(y+r+1&S){b[y]|=7;i+=C;}              /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,      /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);           /* J,Z: hash keys           */
       v-=v>e;                                  /* delayed-gain penalty     */
       if(z==9)                                 /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
        v=m;                                    /* (prevent fail-lows on    */
       }                                        /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
      }                                         /*           if non castling */
      t+=p<5;                                   /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
         ||(u&~24)==36&j==7&&                   /* virgin K moving sideways,*/
         G&M&&b[G=(x|7)-(r>>1&7)]&32            /* 1st, virgin R in corner G*/
         &&!(b[G^1]|b[G^2])                     /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                               /* unfake capt., enable e.p.*/
     }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                           /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                         /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,S,1)/2;          /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                 /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                 /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;               /* empty, type (limit/exact)*/
  }                                             /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                               /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                       /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                           /* or think up one    */
  F(i,0,U)A[i].K=0;                                 /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                /* check legality & do*/
 }
}
```

In the following pages you will find a detailed discussion of the various features of Micro-Max, and how they are implemented.

Previous Next

Basic Data Representations

Piece Encoding

Pieces are encoded as integers, which (like anything else on a binary digital computer) are eventually bitpatterns. The encoding makes use of the binary representation by assigning a specific meaning to some bits of the integer encoding: the '8'-bit is used (i.e. set) to indicate a white piece, the '16'-bit indicates a black piece. This makes it easy to test if a piece *u* belongs to white, by bitwise anding with 8 (`u&8`). The code uses a variable *k* to indicate the side that should move, which has the value 8 on white's turn, or 16 for black, so that `u&k` tells us if a piece belongs to the side to move. The '32'-bit is used to indicate that a piece is in the original position (the 'virgin bit'). (Since this is only important for Kings and Rooks in connection with castling, Micro-Max doesn't bother to set the virgin bit on pawns.

The three low-order bits can be interpreted as a number 0-7 that indicates the piece type, and have a meaning that is independent of piece color. This makes tables that do not have to distingush by color (like piece value `w[u&7]`) half as small. Tables that would like to make this distinction (like the character to used to representthe piece on printout) can simply be indexed with the 4 low-order bits (`n[u&15]`).

Upstream moving pawns are considered different pieces than downstream moving pawns (type 1 and 2, respectively). Each side has only one of the types of pawns, moving in the appropriate direction, although the engine would not frown at putting pieces on the board that move 'backwards'. The encoding is further chosen such that tests if a piece belongs to a certain group is easy: the sliding pieces (BRQ) are giving the highest type numbers (5,6,7), so that crawling pieces can be recognized by piece type *p* (`=u&7`) `p<5`. Pieces that are awarded for moving towards the center of the board (everything but RQ) are recognized by `p<6`, pawns by `p<3`. The complete list thus is {1,2,3,4,5,6,7} = {P+,P-,N,K,B,R,Q}. Type 0 is not used, so it can indicate empty squares on the board.

The following highlights places that show how the piece encoding is used in Micro-Max.

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
```

```
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                        /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
```

```
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop         */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board       */
  p=c;W((*p++=getchar())>10);                        /* read input line   */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one   */
  F(i,0,U)A[i].K=0;                                  /* clear hash table  */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

The initial boad setup might deserve some explanation: the piece types in the back of the array `o[]` are copied to the 1st and 8th rank of the board. On the 1st rank 40 = 8 + 32 is added, which sets the 'white' and the 'virgin' bit. On the 8th rank an extra 8 is added, for a total of 48 = 16 + 32, the 'black' and 'virgin' bits. The 18 = 16 + 2 and 9 = 8 + 1 are the encodings for a black P- and a white P+, respectively.

Previous Next

Basic Representations: the Board

Square Numbering

Squares are designated by a 'square-number'. The '0x88' system is used, meaning that the the board actually has ranks of 16 squares, but only the first 8 squares of each rank are valid and correspond to the 8 files of the board. The two-dimensional board is mapped into a one-dimensional array, rank after rank. Due to this the 4 lowest-order bits of the square number indicate the file, and the higher-order bits the rank. This makes it easy to recognize moves that fall off the board. On the left and they move to invalid square numbers (e.g. from the h-file to the 'i-file'), that have the '8'-bit set (the i-p file have numbers 8-15). Furthermore, all moves that move off the board at the front or back move to invalid rank, and have the 128 bit set (0x80, in hexadecimal). The validity of a square-number can thus be tested by bitwise anding with 0x88 (=136). This constant (which has given the system its name) is held in the global variable *M*, since it is used quite a lot, thus saving a lot of characters. (If the aim would not be source size, but speed, one would of course do the opposite!)

The Board Array

The principal data structure of the program thus is the board array `b[129]`, a one-dimensional array indexed by square-number. Because of the invalid square-numbers, only half of the array is used to store the current chess position. Blocks of 8 used and 8 unused elements alternate, the highest valid square-number is 0x77=119. A dummy square with square-number 0x80=128 (a value held in the global variable S) is included in the array. This square number is sometimes used to indicate an invalid square (e.g. in the variable that indicates on which square e.p. capture is possible when no e.p. captures are possible), to make sure there are no crashes if the program inadvertently stores something there (saving us the trouble of testing the validity of the square number before we store).

Printing the Board

Despite the non-contiguous mapping of the chess board into the `b[]` array, it is quite easy to step through all squares on the board, skipping the invalid ones. The expression to step from a field *x*to the next field, which can, for instance, be used in th increment part of a for-loop, is `x = x+9 & ~8;`. Due to adding 9 = 8 + 1 in stead of 1 to a square number of a square in the 'a'-'g' file, its '8'-bit gets set, which is taken care of by the `& ~8`, which clears it again. But doing the same in the 'h' file also gives a carry into the '8'-bit due to the addition of 1. This resets the '8'-bit, and passes the carry through to the more-significant bits, i.e. the rank number. (The `& ~8` then has no effect.) If we want the board scan to 'wrap around' from h8 back to a1, we can and with `~M` in stead of `~8`, clearing the '128'-bit together with the '8'-bit when the rank number overflows.

In the line that prints out the board (in `main()`) things are done a little differently, because the fact that we 'overflow' to a new rank also has to result in the printing of a new-line. Thus it uses a normal for-loop (with `i++` increment, but tests the '8'-bit to see if we run off the board inside the `printf()` to know what to print (a new-line character, 10, or the representation of the piece on the square, `n[b[i]&15]`). If this tests true, an extra `i += 7` (which always evaluates non-zero, and thus considered true as well) positions us back in front of the next valid square.

Below the declaration and use of the board are highlighted. (Where the original source contained macro's for for or while loops, these are shown expanded for readability.)

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
```

```
/* - quiescence search with recaptures                                  */
/* - recapture extensions                                               */
/* - (internal) iterative deepening                                     */
/* - best-move-first 'sorting'                                          */
/* - a hash table storing score and best move                           */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking  */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */
```

```
      if(h=d-(y!=z))                          /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                 /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;            /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}            /* castling: put R & score  */
       if(p<3)                                /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+           /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);        /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}            /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                /* delayed-gain penalty     */
       if(z==9)                               /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                   /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                  /* (prevent fail-lows on    */
       }                                      /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                       /*          if non castling */
      t+=p<5;                                 /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                     /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32         /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                  /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                             /* unfake capt., enable e.p.*/
     }W(!t);                                  /* if not capt. continue ray*/
  }}}while((x=x+9&~M)-B);                     /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                       /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;        /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)               /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;               /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;             /* empty, type (limit/exact)*/
  }                                           /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 for(i=0;i<8;i++)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop         */
 {for(i=0;i<121;i++)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board       */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Move Generation: Simple Moves

Three Nested Loops

Move generation is done by three nested loops:

- over the pieces
- over all directions for that piece
- over all squares in that direction

For crawling pieces (PNK) the loop over squares is usually terminated after the first iteration, although Pawns and Kings (castling!) sometimes also make more than one step in a particular direction. For any piece this inner loop can be aborted because we run off the board or into an obstructing piece, although for obstructing enemy pieces the move might still be legal. Before discussing the nitty-gritty details of the implementation in micro-Max below, we give the general outline of the basic move generator:

```
static int o[] = {
        -16,-15,17,0,                   /* downstream pawn        */
        16,15,17,0,                     /* upstream pawn          */
        1,-1,16,-16,0,                  /* rook                   */
        1,-1,16,-16,15,-15,17,-17,0,    /* king, queen and bishop */
        14,-14,18,-18,31,-31,33,-33,0,  /* knight                 */
        -1,3,21,12,16,7,12              /* directory              */
      };

x = 0;
do{                                     /* LOOP OVER PIECES                                    */
  u = b[x];                             /* contents of square                                  */
  if(u&k)                               /* only do something if it contains a piece of mine    */
  { p = u&7;                            /* extract piece type                                  */
    j = o[p+30];                        /* look up first direction it moves in directory       */
    while(r=o[++j])                     /* LOOP OVER DIRECTIONS r taken from vector list       */
    { y = x;                            /* start ray at current piece location                 */
      do{                               /* LOOP OVER SQUARES                                   */
        y += r;                         /* add one step                                        */
        if(y&M) break;                  /* terminate ray if off board                          */
        t = b[y];                       /* contents of target square                           */
        if(t&k) break;                  /* terminate ray if own piece                          */
        if(p<3&&!(r&7)!=!t) break;      /* terminate if pawn and inappropiate mode             */
        ....                            /* valid move available: u@x -> t@y. Use it!           */
        t += p<5;                       /* make sure t!=0 for crawling pieces                  */
      }while(!t)                        /* ray continues if move was not capture               */
    }
  }
}while(i=i+9&~M);                       /* next square of board                                */
```

The Loop over Pieces

In micro-Max the loop over pieces is implemented as a loop over all valid squares of the board, searching for pieces of the side to move. This is rather inefficient, especially if the board gets more empty during the game. Even in the beginning only on 25% of the squares it finds own pieces. The board array is a direct way of finding out what we find where we go, (a question that has to be answered frequently once we try to move a piece), but it leaves us unfortunately unaware of where our pieces are. For that a 'piece list', a table listing the square number and piece type for each piece we (still) have, would be better suited. The outer loop of the move generator would then simply scan this piece list, and find a piece every time, rather than only occasionally

finding a piece on a board square. Because of the minimalist approach we decided to sacrifice the piece list. This is the best of two evils: sacrificing the board array would be completely disastrous, because for every attempted move the program could only find out if the move was to an occupied square by scanning the piece lists of both sides. And there are many more attempted moves than there are pieces to move.
For reasons that will become apparent later, the board scan does not begin in a corner, but can begin anywhere on the board, as indicated by variable *B*. It then wraps around until it reaches that starting square again.

The Loop over Directions

If the scan of the board discovers a friendly piece, we start generating moves for this piece. To this end we loop through a list of move vectors, one for every direction a piece of that type can move in. An initialized global array `o[]` contains the move vectors *r* (numbers that have to be added to the current square to make one step in that direction). This list is scanned by the loop index *j* until a 0 is encountered (which obviously is an invalid move vector, since it would make no step at all). The move-vector lists for all piece types are all placed in the same array, one after the other, separated by zeroes. Lists are combined where possible: in the list for a Queen (which is also used for the King) the straight directions all preceed the diagonal directions, so that the final part of the list can be used for a Bishop, by starting it in the middle. For each piece type we need to know where to start scanning the `o[]` array, this information is stored further in the `o[]` array itself, so that a piece of type *p* finds its starting direction index *j* at `o[p+30]`. We could say that `o[]` summarizes most of the rules for chess.

To save some characters in the initializer for the list, we make use of the fact that allowed directions usually come in pairs of opposite directions. The list contains then only contains the positive one, and in the loop over directions each step vector is tried with both signs. Unless the piece is a Pawn, of course. This explains the obfuscated expression `p>2&r<0?-r:-o[++j]` for the next direction *r*: first the negative version is tried, and the second time such a negative value is negated and used again.

The Loop over Squares

Once a move vector *r* is selected, the inner loop of the move generator repeatedly adds this to the location of the piece, to sweep the destination square *y* over the board along a ray. Each new *y* is tested for being on the board (`y&M==0`), and if it is, the piece *t* on the destination square (`t=b[y]`) is checked. If it is one of our own or if we fall off the board, we break out of the loop immediately. If the piece belongs to the opponent, the move is a capture, and in general valid. So in that case we finish that iteration of the loop, but the test at the end of this do-while loop only continues looping if there was no capture (`t==0`, written compactly as `!t`).

For crawling pieces the loop has to terminate after the first iteration irrespective of the fact if the last move was a capture or not. To this end we *fake* a capture for those pieces (`p<5`) at the end of the loop, by incrementing t. We still have to deal with the complication that some of the moves generated for pawns are forbidden. In particular straight pawn moves can't be captures, and diagonal moves can't be non-captures. We therefore break out of the loop at the beginning if we detect that the emptyness of the target sqaure (quantified by the expression `!t`) matches the straightness of the move vector (quantified as the expression `!(r&7)`, i.e. if the least significant bits of the move vector, the file displacement, equals zero or not).

Below the basic move-generator code is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
```

```
/* - a hash table storing score and best move                              */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking     */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                        /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    while(r=p>2&r<0?-r:-o[++j])                /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
```

```
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }while(!t);                               /* if not capt. continue ray*/
   }}}while((x=x+9&~M)-B);                     /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Previous Next

Move Generation: En Passant

The basic move generator only plays chess as it used to be several hundred years ago, before e.p. capture and castling where invented. It does not even allow pawns to move two squares from their initial position. These 'advanced moves' unfortunately add a relatively large amount of complexity to the move generator, because they violate the basic philosophy of most chess moves. In castling two pieces move, in e.p. capture one does not capture the piece by stepping in its place. In addition, these moves cannot be performed at just any time, and we have to test all the conditions that have to be obeyed.

Pawn's Double Move

The discussion of the pawn's double move was deferred to this page, because it is intimately tied to the e.p. capture. The basic move generator terminates the inner loop for crawling pieces after the first iteration, by faking a capture through incrementing the captured piece *t*. Pawns on the 2nd/7th row have to be given one extra iteration, though. This is simply done by decrementing *t* back to its original value if the target square *y* of the current iteration is on the 3rd/6th row. To test this for both colors simultaneaously the expressions `p<3&6*k+(y&V)==S` is used. It effectively compares the rank number of the pawn, `(y&V)` with the rank from which they have to make the extra step. Which rank this is, is derived from the color *k* of the pawn: by multiplying this by 6 we obtain either 0x30 or 0x60, which are subtracted from `S=0x80` to optain the sought rank numbers 0x50 (2nd) and 0x20 (6th). In addition the field from which the move was extended is remembered in *F*, which normally holds the invalid square code *S*, for passing it to the next ply so that the move generator there knows where it can try an e.p. capture.

E.p. Capture

The e.p. square is thus passed as a parameter *E* to the next level of the search tree. If a pawn tries to move to this square, (`p<3 & y==E`) it must be through a diagonal step (it cannot come from the square to which the opponent just moved!). For e.p. capture this move does have to be executed, and we have to be able to make a piece disappear that is not in the target square *y* when the move is later performed. To this end the square of the captured piece is held in a separete variable *H*, which is normally set to *y*, but for e.p. capture is set to the 'companion square' `E^16`, by flipping the lowest bit of the rank number.

Below the code lines involved in Pawn double move and e.p. capture are highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
```

```
/* - best-move-first 'sorting'                                          */
/* - a hash table storing score and best move                           */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking  */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
```

```
       b[G]=b[H]=b[x]=0;b[y]=u&31;             /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Previous Next

Move Generation: Castling

King Capture

Before we discuss castling, it is good to point out that micro-Max is a King-capture engine: At any level it generates all pseudo-legal moves, including those that put his own King in check. Since this is a condition that depends on which moves the opponent has, it seems more logical to test it on the next ply, when we generate the opponent's moves anyway. Usually all opponent's moves are generated before micro-Max starts to think about any of them, and if a King capture is found the branch is aborted and the caller is informed (by the move's score) that the move was an illegal one.

Castling and e.p. Capture

Castling and e.p. capture share an underlying philosophy: a piece (Pawn or King) moves farther than it is ordinarily allowed to, and if the square it skips in doing so is under attack by the enemy, it can be captured there anyway. For castling this of course results in the move being declared illegal, since you are not allowed to expose your King to capture.
Micro-Max exploits this similarity: the field that is skipped over with these moves is communicated to the next ply as the recursion argument *E*, to indicate that something can be captured there. Castling and Pawn e.p. can be easily distinguished by the contents `b[E]`of the e.p. square: after castling it contains a Rook, after a Pawn double move it is empty. King captures never have to be executed. They lead to termination of the move generation altogether, because the previous move was apparently already an illegal one and we are working on a phantom branch of the search tree. This holds just as much for the 'e.p.' capture of a King after castling, as for ordinary king captures. In fact we can also cover the rule that you are not allowed to castle when in check easily by the same mechanism, by not only testing if the subsequent move goes to the e.p. square, but also to either square next to it `(y-E<2&E-y<2)`.
Thus, in line with the King-capture philosophy, several conditions for castling are checked in hindsight, on the reply move.

Castling Rights

Enough remains to be checked during move generation itself, though. Castling applies to a King moving sideways (recognized by the direction number *j*). We have to figure out what the corner associated with the direction (King or Queen side) is, and remember it in *G*. This variable at the same time flags that we have a castling move, for any non-castling it remains set to the dummy square code *S*.
We then have to test if both King and Rook have not been moved before (from their virgin bits), and indeed are present, and if the two squares next to the Rook are empty. The King ends on one of these squares, so this also checks if the King does not capture something on castling. If the King captured something next to it, this would terminate the ray after the King's first side step and thus prevent castling. Finally, the lateral King ray must be extended like this only once, tested by checking if the previous move already was castling through *G*, before calculating the new *G*. A lot of testing, but if we pass it all, we do exactly the same as for the pawn-double-move

extension: unfake the terminating capture and remember the e.p. square in *F*. The only difference is that *G* has also been set as a side-effect of the tests. If the test fail, it does not matter that *G* has been set, because the ray then terminates and at the start of a new ray *G* is re-initialized as the dummy.

Below the code lines that implement castling are highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
```

```
   {r=p=u&7;                                  /* p = piece type (set r>0) */
    j=o[p+16];                                /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                   /* loop over directions o[] */
    {A:                                       /* resume normal after best */
     y=x;F=G=S;                               /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                               /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                        /* sneak in prev. best move */
      if(y&M)break;                           /* board edge hit           */
      if(p<3&y==E)H=y^16;                     /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;     /* capt. own, bad pawn mode */
      i=99*w[t&7];                            /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;      /* K capt. or bad castling  */
      if(m>=l)goto C;                         /* abort on fail high       */

      if(h=d-(y!=z))                          /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                 /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;             /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}            /* castling: put R & score  */
       if(p<3)                                /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+           /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);        /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}            /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);         /* J,Z: hash keys           */
       v-=v>e;                                /* delayed-gain penalty     */
       if(z==9)                               /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                   /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                  /* (prevent fail-lows on    */
       }                                      /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;   /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}             /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}              /* update max, mark with S  */
      }                                       /*          if non castling */
      t+=p<5;                                 /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                     /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32         /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                  /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                             /* unfake capt., enable e.p.*/
     }W(!t);                                  /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                         /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                       /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;        /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)               /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;               /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;             /* empty, type (limit/exact)*/
  }                                           /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;
```

```
 W(1)                                                  /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                          /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                              /* or think up one    */
  F(i,0,U)A[i].K=0;                                    /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                   /* check legality & do*/
 }
}
```

Note that castling also requires some extra code in the do_move and undo_move section, to move the Rook in addition to the King, (which uses the normal mechanism). In the undo part a Rook of the proper color is 'synthesized' through `k+36`, (the 36 = 32 + 4 being the code for Rook plus the virgin bit), and put back in the corner *G*. The latter is set to the dummy square for non-castlings, hence this code does not have to be executed conditionally (hich is good for size as wel as speed!).

Previous Next

Previous Next

Move Generation: Promotions

Queening

Micro-Max does not implement minor promotions. I considered it too unlikely that it would ever need one.
To detect if a Pawn has reached last rank, we test if continuation of the move by one more step would move it to a nonexistent rank. The expression `y+r+1&S` evaluates true if this is the case (S=0x80). The `+1` is needed because otherwise captures to the a-file, that would cause a carry into the rank number would be treated wrong.

Below the code performing Queening is highlighted.

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
```

```
 struct _*a=A;

                                        /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                         /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);              /* first empty or match     */
 a+=i?j:0;                              /* dummy A[0] if miss & full*/
 if(a->K)                               /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                 /* examine stored data      */
  if(d>=n)                              /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;      /* use if window compatible */
   d=n-1;                               /* or use as iter. start    */
  }X&=~M;Y=a->Y;                        /*      with best-move hint */
  Y=d?Y:0;                              /* don't try best at d=0    */
 }else d=X=Y=0;                         /* start iter., no best yet */
 N++;                                   /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)               /* iterative deepening loop */
 {x=B=X;                                /* start scan at prev. best */
  Y|=8&Y>>4;                            /* request try noncastl. 1st*/
  m=d>1?-I:e;                           /* unconsidered:static eval */
  do{u=b[x];                            /* scan board looking for   */
   if(u&k)                              /*  own piece (inefficient!)*/
   {r=p=u&7;                            /* p = piece type (set r>0) */
    j=o[p+16];                          /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])             /* loop over directions o[] */
    {A:                                 /* resume normal after best */
     y=x;F=G=S;                         /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                         /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                  /* sneak in prev. best move */
      if(y&M)break;                     /* board edge hit           */
      if(p<3&y==E)H=y^16;               /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;/* capt. own, bad pawn mode */
      i=99*w[t&7];                      /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;/* K capt. or bad castling  */
      if(m>=l)goto C;                   /* abort on fail high       */

      if(h=d-(y!=z))                    /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;           /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;      /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}      /* castling: put R & score  */
       if(p<3)                          /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+     /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);  /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}      /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,/* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);   /* J,Z: hash keys           */
       v-=v>e;                          /* delayed-gain penalty     */
       if(z==9)                         /* called as move-legality  */
       {if(v!=-I&x==K&y==L)             /*   checker: if move found */
        {Q=-e-i;O=F;return l;}           /*   & not in check, signal */
        v=m;                            /* (prevent fail-lows on    */
       }                                /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;/* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}       /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}        /* update max, mark with S  */
      }                                 /*          if non castling */
      t+=p<5;                           /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S               /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&          /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32   /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])            /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                       /* unfake capt., enable e.p.*/
     }W(!t);                            /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                   /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                 /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;  /* best loses K: (stale)mate*/
```

```
  if(!a->K|(a->X&M)!=M|a->D<=d)               /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;               /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;             /* empty, type (limit/exact)*/
  }                                           /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Search: Minimax

Minimax

The search is done by the recursively called routine `D()`. This is the only subroutine in micro-Max, all other code was in-lined to save characters on calls and declarations. But for the recursive search this is not possible, of course. `D()` has a lot of arguments; the ones governing the search are: the side to move *k* which has to be toggled (as `24-k`) for each ply, the alpha-beta window (q,l), and the 'depth-left' argument *n*. For the rest its code is rather conventional (this section supposes you are familiar with the principles of alpha-beta minimax, and just discusses the way it is implemented in Micro-Max. If not, an excellent explanation of general principles of computer chess can be found on Bruce Moreland's web site):

```
int D(int k, int q, int l, int n)
{
        int m;                          /* score of best move so far */

        m = n>1 ? -I : EVAL();

        INITIALIZE_MOVE_GENERATION;
        DO{{{
                GENERATE_NEXT_MOVE;
                if(m>=l) goto C;
                h = n-1;                /* new depth left */
                if(h)
                {       DO_MOVE;
                        v = -D(24-k, -l, q>m?-q:-m, h);
                        UNDO_MOVE;
                        if(v>m){m=v;}
                }
        }}} WHILE_MOVES_LEFT;

    C:  return m;
}
```

where the macros implied by the capital writing are not implemented as macros but just a condensed way of indicating the code sections discussed elsewhere. The three nested loops of the move generator are symbolyzed by triple braces.

Soft Fail

The search uses a 'soft fail', i.e. if the score of the state does not fall within the (q,l) window it returns the upper or lower bound it found, not merely *q* or *l*. Although this difference would not affect the operation of the bare alpha-beta search, this will be beneficial once we start using a hash table for storing the results: we might encounter the same position later in the search with a different window, where the original score bound might

produce a cutoff. So always report the sharpest bound on the move value you know, even if for this node itself it does not matter.

A consequence is that the recursive call always has to look at both *q* and *m* to determine the upper limit of the new search window (`q>m ? -q : -m`). I don't want to tinker with *q* (upping it when a better maximum is found), which would be slightly more efficient, because when we implement iterative deepening the next iteration has to be aware of the original *q*. I did not feel that this was worth throwing a new variable at, that you have to declare, initialize, etc...

Depth Control

The meaning of the depth control argument *n* is such that if `n==1` all moves are generated, but none is actually tried (so that this position is an end leaf of the search tree). This might seem strange at first, why generate moves that we are not going to look at? Well, in the first place, even for moves that are not looked at, King-captures are still detected. So you might view calling `D()` with `n==1` just a code-efficient way of detecting if the move generated on the previous ply was truly a legal move, rather than just 'pseudo-legal'. But when we implement quiescence search we will allow trying out some moves in this situation, and in that case the moves have to be generated in order to judge if they are worth looking at. An end-leaf then occurs only for `n==1` if none of the moves was found to be worth considering.

If `n==1` many, if not all moves will stay unconsidered. Such moves receive the static evaluation score `EVAL()`, under the assumption that they will not significantly change the position. (If they did, they should have been considered!) To this end *m* is initialized to `EVAL()`, rather than -infinity, if `n==1`. In practice, `EVAL()` is not really a function, but is immediately available as the argument *e*, because the evaluation is so rudimentary and simplistic that it can be differentially updated along with doing the moves in the path that led to the current poition.

Fail-High Cutoffs

The detection of the 'fail high' cutoff (`if(m>=l)`) might seem to occur in an unusual place: a more conventional location would be to put it inside the `if(v>m)` update of the maximum, where it would be executed less frequently. In addition it would also immediately discover the fail high, rather than after generation of one more move (which subsequently is not looked at). In its current place it also acts for intercepting King captures, which become evident during the move generation, without first considering replies to such a King capture (which would be really costly). Also, the static evaluation score can now already causes the fail high, before any moves have to be considered at all. For opimum efficiency the test for fail high could be made in all these places, the chosen solution sacrifices a little efficiency to compactness of the code.

Below the code that implements the recursive alpha-beta search is highlighted: (note that *d* ends up as *n*.) The move generation is symbolized by its outer loop, although in fact it is three nested loops.

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
```

```
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
```

```
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop         */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board       */
  p=c;W((*p++=getchar())>10);                        /* read input line   */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one   */
  F(i,0,U)A[i].K=0;                                  /* clear hash table  */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next Version 4

Search: Quiescence

Quiescence Search

Static evaluation only gives meaningful scores if the situation is quiet, i.e. there are no hanging pieces that the opponent will take on his next move, and thus might as well not have been there at all. To prevent the static evaluation to be used in situations like that, the program always considers moves that capture the piece that last moved, no matter how much search depth was left. This leads to a complete finishing of a sequence of exchange captures to the same square that might be in progress at what would otherwise have been the end of the tree.

Of course there are still many tactical play-lines that are overlooked this way, because the capture sequence on one square might have important consequences for the safety of other squares (e.g. a piece pinned on the Queen might have been used to recapture something). For all the fancy tactics that distinguish a chess player from someone that merely knows the rules of the game the program fully depends on the full-width part of the search. But exchange sequences in progress are so common that these are a bare minimum one should have in the QS, or very severe horizon problems will occur. A good next candidate (not implemented in micro-Max) would be to also always consider all lower-takes-higher captures, this makes it much more difficult for the program to try to push forced losses because of trapped pieces or fork threats beyond the horizon by intervening exchanges or sacrifices.

Recapture

To implement the exchange sequences, the target field *y* of the last performed move is passed to the next level as the argument *z*. Moves that go to this square are then not counted at all in the ply depth: we don't reduce the depth remaining by one (`h=n-1;`), but only do so if the move was not a recapture (`h=n-(y!=z);`) and otherwise keep the original depth. Note that this is not only done at the end of the tree, for a true QS, but anywhere in the tree. This way of implementing causes a QS as well as recapture extensions. (To only get the QS we would have to write

```
h=n-1;
if(h>0|y==z){...}
```

Note further that this 'recapture' captures anything that moved, not just anything that captured (which might be unnecessary, and therefore wasteful and bad...).

Weak Spots

The worst shortcoming of the QS, however, is not that it is blissfully unaware of pinnings, discovered threats or overloaded defenders, but that it judges the quiescence only one-sided: if the side to move does find that all the moves it wants to consider are worse than the static evaluation, it assumes the position is quiet and uses the static score. It does not test at all if the opponent still has tricks up his sleeve. In particular, the opponent could have just attacked our Queen (or King!) with a defended Pawn, and if capturing that Pawn is no good, we still assume we are OK because we revert to static evaluation. We just *assume* that the Queen can be saved because it is our move (which for a Queen is indeed more likely than for a King, but nevertheless...).

The listing below is nearly the same as that on the previous page. Note in particular the line `if(h=d-(x!=y))`, that makes the depth of the subtree dependent on if the current move goes to the same square as the previous one, ('recapture') or not.

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
```

```
/* version 3.2 (2000 characters) features:                                  */
/* - recursive negamax search                                               */
/* - quiescence search with recaptures                                      */
/* - recapture extensions                                                   */
/* - (internal) iterative deepening                                         */
/* - best-move-first 'sorting'                                              */
/* - a hash table storing score and best move                               */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking      */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
```

```
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;      /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;             /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Search: Do and Undo Move

Doing a Move

The move generation delivers the start and target squares *x* and *y*, and the capture square *H*. Performing the move is simply clearing the capture square, putting the moving piece (*u*) in the target square, and clearing the start square. In that order, since the capture square and target square usually coincide. The virgin bit is stripped off the piece code, to indicate that the piece has moved. If the move was a castling, we have to move the Rook as well, (from *G* to *F*), since start and target square only give the move of the King in that case.

And Taking it Back

Undoing the move is just as easy: Clear the target square and put the captured piece *t* back (in that order!), and put the moving piece back on its start square. Because the moving piece is remembered in the variable *u*, we don't have to read it from the board, and automatically restore the virgin bit to its state from before the move (and turn a promotion Queen back into a Pawn). In this case, we *always* move a Rook back as well, from *F* to *G*. We can afford that, because for non-castlings *G* is the dummy square, and we don't care if there is a Rook there. The square *F* that is cleared by this Rook move is usually also the dummy square. Only in the case of an e.p. capture it contains a valid square number, but then that square was empty already, and clearing it changes nothing.

Below the code that performs the moves and takes them back is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
>#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */
```

```
char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                         /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                           /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                          /* initial piece setup      */
b[129],                                         /* board: half of 16x8+dummy*/
T[1035],                                        /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                         /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                                /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                 /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                  /* first empty or match     */
 a+=i?j:0;                                      /* dummy A[0] if miss & full*/
 if(a->K)                                       /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                         /* examine stored data      */
  if(d>=n)                                      /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;              /* use if window compatible */
   d=n-1;                                       /* or use as iter. start    */
  }X&=~M;Y=a->Y;                                /*      with best-move hint */
  Y=d?Y:0;                                      /* don't try best at d=0    */
 }else d=X=Y=0;                                 /* start iter., no best yet */
 N++;                                           /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                       /* iterative deepening loop */
 {x=B=X;                                        /* start scan at prev. best */
  Y|=8&Y>>4;                                    /* request try noncastl. 1st*/
  m=d>1?-I:e;                                   /* unconsidered:static eval */
  do{u=b[x];                                    /* scan board looking for   */
   if(u&k)                                      /*  own piece (inefficient!)*/
   {r=p=u&7;                                    /* p = piece type (set r>0) */
    j=o[p+16];                                  /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                     /* loop over directions o[] */
    {A:                                         /* resume normal after best */
     y=x;F=G=S;                                 /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                 /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                          /* sneak in prev. best move */
      if(y&M)break;                             /* board edge hit           */
      if(p<3&y==E)H=y^16;                       /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;       /* capt. own, bad pawn mode */
      i=99*w[t&7];                              /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;        /* K capt. or bad castling  */
      if(m>=l)goto C;                           /* abort on fail high       */

      if(h=d-(y!=z))                            /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                   /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;               /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}              /* castling: put R & score  */
       if(p<3)                                  /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+             /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);          /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}              /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,     /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);           /* J,Z: hash keys           */
       v-=v>e;                                  /* delayed-gain penalty     */
       if(z==9)                                 /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
        v=m;                                    /* (prevent fail-lows on    */
```

```
       }                                        /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
      }                                         /*          if non castling */
      t+=p<5;                                   /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32           /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                    /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                               /* unfake capt., enable e.p.*/
     }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                           /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                         /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;          /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                 /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                 /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;               /* empty, type (limit/exact)*/
  }                                             /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Previous Next Version 4

Search: Iterative Deepening

Move ordering

The efficiency of the alpha-beta search is strongly enhanced (in terms of the number of nodes searched) if the best move is tried first, and trying idiotic moves first (such as both Queens getting berserk) can easily lead to an explosion of the search. Rather than aggressively starting a search of the required depth, proceed carefully and first invest some time doing a search of shallower depth. Then use the result to determine the order in which you are going to try the moves in the full search. The time you save because of the good move ordering pays back many times the time you invested in the shallower search.

Iterative Deepening

This concept, known as internal iterative deepening, works like a charm, because the completely stupid full-width search in principle tries anything, even the most idiotic moves like taking a defended Pawn with a Queen. Such moves can cause tremendous tactical complications if you refrain from taking the Queen back immediately, a move that a one-ply search would already recommend. So the power of iterative deepening is that it immediately finds the solutions for problems that do have trivial solutions, so that in the deeper searches their subtrees are quickly and decisively pruned though the alpha-beta mechnism. Going into the search 'depth first', on the other hand, tends to dive deep into idiotic variations before realizing they were idiotic, when accidentally hitting upon the proper refutation.

Of course one could implement a satic way to find out what likely refutations are for idiotic moves, but there are many levels of idiocy. Some, like taking a defended piece with a higher one, are refuted in the next ply, but if the apparently defending piece happened to be a pinned one, it is the opponent who turns out to be the idiot on ply number two. Judging this statically becomes progressively more difficult. With iterative deepening the results vary wildly with depth until the entire relevant tactics are within the horizon (e.g. including the capture of the piece on which the defender was pinned), and stabilize after that.

To implement the iterative deepening there is (of course) a loop around the search code, that executes it with a depth left *d* running from 1 to `n-1`, rather than just for `d=n-1`, like this:

```
int D(int k, int q, int l, int n)
{
        int m,                          /* score of best move so far    */
            d=0,                        /* iteratively improved depth   */

        while(++d<n)
        {       m = d>1 ? -I : EVAL();                  /* if d==1 no moves will be tried       */

                INITIALIZE_MOVE_GENERATION(B);          /* start with moves that go from B      */
                DO{{{
                        GENERATE_NEXT_MOVE;
```

```
                        IF_KING_CAPTURE m=I;
                        if(m>=l) goto C;
                        h = d-1;                        /* new depth left                       */

                        if(h)
                        {       DO_MOVE;
                                v = -D(24-k, -l, q>m?-q:-m, h);
                                UNDO_MOVE;

                                if(v>m) { m=v; }        /*   account maximum                    */
                        }
                }}} WHILE_MOVES_LEFT;

            C:  ;
        }
        return m;
}
```

The information returned by the pevious, shallower searches can be used in two ways. The score obtained can be used to set the window more appropriately, leading to a more efficient search. In addition the order of looking at the moves can be inspired by it. Micro-Max uses only the second method, in a minimal way: it tries the best move from the previous iteration first, but all other moves in the order they hppen to be generated. There are no manipulations with the window.

Maximum Depth

In internal nodes the maximum depth is set by the caller, but at the root we want to deepen until time runs out. How many ply that is depends strongly on the position and the stage of the game. To take account of this micro-Max deepens in the root node (which is recognized because it is the only instance of `D()` that is called with the invalid square code 8 for the 'recapture square' *z*) until a certain number of nodes, counted in *N*, has been reached. In the sample listing this is after 10 million nodes (1e7), to change the average time per move this number should be changed. To prevent problems with stack overflow in a tree that does not branch, an absolute maximum of 98 ply is set to the depth. One could also set a fixed depth here, by simply writing `while(d++<n)`), and calling `D()` from main with *n* set to the required value (saving many characters...). But then micro-Max starts to play very fast (and poorly) in the end-game.

Game Ends

Iterative deepening makes no sense if the search has already reached the legal end of the game, through checkmate. If mate-in-three is certain, nothing can be gained by looking deeper, we are not interested anymore in mate-in-four. So if the score tells us that a checkmate is discovered, we immediately set the depth *d* to the maximum (i.e. 99), so that no new iterations are started.

Below the code that implements the iterative deepening is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */
```

```
#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 while(d++<n|z==8&N<1e7&d<98)                  /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
```

```
              ((x+2)&M||b[x+2]!=u)-1);          /*        squares plus bias */
       if(y+r+1&S){b[y]|=7;i+=C;}               /* promote p to Q, add score*/
      }
      v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,      /* recursive eval. of reply */
           J+J(0),Z+J(8)+G-S,F,y,h);            /* J,Z: hash keys           */
      v-=v>e;                                  /* delayed-gain penalty     */
      if(z==9)                                 /* called as move-legality  */
      {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
       {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
       v=m;                                    /* (prevent fail-lows on    */
      }                                        /*   K-capt. replies)       */
      b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
      if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
      if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
     }                                         /*          if non castling */
     t+=p<5;                                   /* fake capt. for nonsliding*/
     if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
         ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
         G&M&&b[G=(x|7)-(r>>1&7)]&32           /* 1st, virgin R in corner G*/
         &&!(b[G^1]|b[G^2])                    /* 2 empty sqrs. next to R  */
     ){F=y;t--;}                               /* unfake capt., enable e.p.*/
    }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next Version 4

Previous Next

Search: Move Sorting

Move Ordering

Searching the tree successively deeper is just a waste of time if we don't somehow use what we learned. Micro-Max only uses one piece of information, namely what the best move of the previous iteration was, for the purpose of searching that first.

To implement this, apart from remembering the score of the best move in *m*, we also keep track of what this best move was at any level, in the local variables *X* and *Y*. When a new iteration starts, we sneak in this best move before any of the other moves is considered. To this end the outer loop of the move generator, that runs through all pieces, is modified to start at the piece that could do the best move previously, rather than in the corner of the board. A local variable *B* keeps track of where we started the board scan, so that we know when we're done (remember the scan wraps around). This makes it easier to slip in the best move before all others, because the piece and square of origin are already that of the best move.

After generating the first target square *y* for this piece, we replace it by the target square of the previous best move. After searching the sub-tree for this move, we intercept the control flow, (based on the '8'-bit of *Y* being set) recording the score as the maximum found (since this was the first move we considered we don't have to compare with any previous best) and clearing the '8'-bit of *Y* to indicate that the best move is now considered. We then jump back to the beginning of the loop-over-directions of the move generator, to redo everything for the move that would have been originally generated as the first move for that piece.

Somewhat later, possibly even immediately, the best move will be generated a second time, in the course of normal move generation. To prevent it from being evaluated again (which would be quite inefficient, since it is likely to still be the best move, which will not trivially fail), we would have to test every move for being this best one (prepending an `if(x!=B|y!=Z)` before the code that considers the generated moves). Once transposition tables are implemented we don't have to bother with this, since re-searching this move will take the value of the original search immediately from the transposition table, an acceptable overhead.

```
int D(int k, int q, int l, int n)
{
        int m,                          /* score of best move so far    */
            d=0,                        /* iteratively improved depth   */
            X=0,Y=0,                    /* best move                    */
            B,Z;                        /* move to skip, already done   */

        while(++d<>n)
        {       m = d>1 ? -I : EVAL();                  /* note there is no best to try if d==1 */
                B = X;                                  /* start move generation with piece here*/
                Z = 8;                                  /* nothing to skip yet: invalid square  */
                Y |= 8&Y>>4;                            /* copy 128-bit of Y to 8-bit, to ask   */
                                                        /* priority for a valid prev. best move */
            A:
```

```
                INITIALIZE_MOVE_GENERATION(B);          /* start with moves that go from B      */
                DO{{{
                        GENERATE_NEXT_MOVE;
                        if(Y&8) y=Y;                    /* overrule generated move with best    */

                        IF_KING_CAPTURE m=I;
                        if(m>=l) goto C;
                        h = d-1;                        /* new depth left                       */

                        if(x-B|y-Z)                     /* skip if already done as best         */
                        if(h)
                        {       DO_MOVE;
                                v = -D(24-k, -l, q>m?-q:-m, h);
                                UNDO_MOVE;

                                if(Y&8)
                                {       m=v;            /* first is always best, m was -I       */
                                        Z=y;            /* set this move for future skipping    */
                                        Y=y|S;          /* clears 8-bit now best is tried       */
                                        goto A;         /* try normal moves                     */
                                }
                                if(v>m)
                                {       m=v;
                                        X=x;Y=y|S;      /* remember best, set 128-bit of Y to   */
                                }                       /*   indicate a best move is available  */
                        }
                }}} WHILE_MOVES_LEFT;

            C:  ;
        }

        return m;
}
```

If the best move from the previous iteration turns out not so good after all, searching other moves of the same piece (as will be done next) often does provide a new best move quickly: In many cases the best move is to withdraw a threatened piece, and if one escape route turns out bad, another might still be good.

Castling Problems

To properly perform a castling move, the variables *F* and *G* for the Rook part have to be set, and this is only the case if the move is generated in the normal way: These variables are not remembered together with the best move. So castling can not be eligible for being sorted in front. This nuisance is solved by deriving the 'S'-bit in *Y*, which indicates that there is a best move that should be tried first in the next iteration, from the casting variable *G*. This prevents setting the 'S'-bit (i.e. the '128'-bit) when the best move is a castling. The 'S'-bit is copied to the '8'-bit of the same variable before the beginning of the next iteration, and acts as a request to sneak in the move. So castlings will not be sorted first.

However, the generation of moves will start at the King, in that case. Because the first directions in which moves are generated are the lateral ones, the castling will be found quite early, though.

Below the code that controls move ordering (by determining the best move, and make it jump the queue in the next iteration) is highlighted:

```
/************************************************************************/
/*                              micro-Max,                              */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/************************************************************************/
/* version 3.2 (2000 characters) features:                              */
/* - recursive negamax search                                           */
/* - quiescence search with recaptures                                  */
```

```
/* - recapture extensions                                              */
/* - (internal) iterative deepening                                    */
/* - best-move-first 'sorting'                                         */
/* - a hash table storing score and best move                          */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                     /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */
```

```
      if(h=d-(y!=z))                          /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                 /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;            /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}            /* castling: put R & score  */
       if(p<3)                                /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+           /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);        /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}            /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                /* delayed-gain penalty     */
       if(z==9)                               /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                   /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                  /* (prevent fail-lows on    */
       }                                      /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;   /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}             /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}              /* update max, mark with S  */
      }                                       /*          if non castling */
      t+=p<5;                                 /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                     /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32         /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                  /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                             /* unfake capt., enable e.p.*/
     }W(!t);                                  /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                         /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                       /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;        /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)               /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;               /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;             /* empty, type (limit/exact)*/
  }                                           /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Hash Table: Lookup

Transpositions

The same position can occur in many places in the search tree. Not only can positions be reached by the same set of moves played in a different order, pieces can also reach the same destination square through a different route. This in particular happens with the King in the end-game. The purpose of the hash table is to store the useful information about a position, so that the computer does not have to search it again if it is encountered a second time.

The Hash Table Entry

An entry of the hash table in micro-Max is a `struct _`. It contains a more or less unique (4-byte) key *K* derived from the position, in order to identify it. The actual information that is stored contains of the score *V* found for that position (4 bytes), the best move *X,Y* (2 bytes) and the depth *D* of the search at which this score was found. Since the size is rounded to a multiple of 4 bytes, (for memory-word alignment), the entry occupies 12 bytes in memory, and wastes some space.

In an Alpha-Beta search most nodes are not completely searched, and as a consequence the scores are only upper or lower bounds. This is indicated by the bits in the field *X* that are not used in valid square codes, the '8'- and the 'S'-bit. (In the destination field *Y* these bits are used to indicate if the move was a castling.) The '8'-bit is set if a result is above alpha, the 'S'-bit if it is below beta, so that exact results (within the window) have both bits set. Exact results, after all, are both upper and lower limit.

The Zobrist Keys

To make sure we make no retrieval errors, two 32-bit Zobrist-like hash keys are used. They are differentially updated, and passed along up the tree from the root as arguments *J* and *Z* to `D()`. The Zobrist scheme requires a table of random numbers for each piece-square pair. In micro-Max this table is packed by having the 4-byte random numbers overlap three bytes out of four, i.e. we have a table `T[1035]` of (about) nr_of_pieces x nr_of_squares *bytes*, and acces that table like it were integers, retreiving four bytes at a time. (Bad for memory alignment, but much better than having L1-cache misses because the table is too big...) Since this is done a number of times, the rather awkward syntax is spelled out in a macros J and K. Since `T[]` is indexed by square number, the holes in the 0x88 board are used for storing the random numbers for the other color. The two Zobrist keys swap the numbers for the black and white pieces.

The low-order bits of One key is used to determine where we store (and look for) the position in the table, the other key is stored in the table as the id *K*. If the entry is already occupied by another position that happened to map to the same address (but with different *K*) - something which will hapen quite often if the table fills up - we try to put it in the next entry ('rehash') up to eight times. Together, this gives a 1 in 2^53 probability for mis-identification: 1 in 2^32 that two different positions have the same id number, 8 in 2^24 (=16M) that they are sought in the same place (because a position can be stored in 8 locations of a table of 16M entries). This seems

small enough: even with 16M entries stored there are 2^47 pairs of entries, so the probability that 2 positions from a full table are accidentally confused is 1 in 64. OK, so it can occasionally happen...

Castling, E.P., and Side to Move

Castling rights, the ability to make an e.p. capture somewhere on the board, and who is to move are all part of the 'state' of a chess game, and two positions that differ in this respect sould never be confused, even if they have all pieces on the same location. The Zobrist keys would only take care of the latter aspect. The castling rights are incorporated by adding `G-S` (which is 0 for non-castlings) to one of the keys in the differential update. (OK, so it does consider a position reached by castling as different from the same one that was reached by normal moves, who cares?) The e.p. capture and side to move, which are not lasting features, are xor'ed to the key directly before retreival, (i.e. not differentially updated), as `(k*E^J)`. The product `k*E` produces values that differ at least 8 from each other for any legal combination of *k* and *E*, so there can be no confusion even after re-hashing. (Note that neither *k* nor *E* can be zero, and k is a multiple of 8.)

Below the code that implements the hashing scheme is highlighted:

```
/*****************************************************************************/
/*                              micro-Max,                                   */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/*****************************************************************************/
/* version 3.2 (2000 characters) features:                                   */
/* - recursive negamax search                                                */
/* - quiescence search with recaptures                                       */
/* - recapture extensions                                                    */
/* - (internal) iterative deepening                                          */
/* - best-move-first 'sorting'                                               */
/* - a hash table storing score and best move                                */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking     */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
```

```
 {d=a->D;v=a->V;X=a->X;                          /* examine stored data      */
  if(d>=n)                                      /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;              /* use if window compatible */
   d=n-1;                                       /* or use as iter. start    */
  }X&=~M;Y=a->Y;                                /*      with best-move hint */
  Y=d?Y:0;                                      /* don't try best at d=0    */
 }else d=X=Y=0;                                 /* start iter., no best yet */
 N++;                                           /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                       /* iterative deepening loop */
 {x=B=X;                                        /* start scan at prev. best */
  Y|=8&Y>>4;                                    /* request try noncastl. 1st*/
  m=d>1?-I:e;                                   /* unconsidered:static eval */
  do{u=b[x];                                    /* scan board looking for   */
   if(u&k)                                      /*  own piece (inefficient!)*/
   {r=p=u&7;                                    /* p = piece type (set r>0) */
    j=o[p+16];                                  /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                     /* loop over directions o[] */
    {A:                                         /* resume normal after best */
     y=x;F=G=S;                                 /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                 /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                          /* sneak in prev. best move */
      if(y&M)break;                             /* board edge hit           */
      if(p<3&y==E)H=y^16;                       /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;       /* capt. own, bad pawn mode */
      i=99*w[t&7];                              /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                           /* abort on fail high       */

      if(h=d-(y!=z))                            /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                   /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;               /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}              /* castling: put R & score  */
       if(p<3)                                  /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+             /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);          /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}              /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,     /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);           /* J,Z: hash keys           */
       v-=v>e;                                  /* delayed-gain penalty     */
       if(z==9)                                 /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
        v=m;                                    /* (prevent fail-lows on    */
       }                                        /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
      }                                         /*          if non castling */
      t+=p<5;                                   /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32           /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                    /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                               /* unfake capt., enable e.p.*/
     }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                           /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                         /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;          /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                 /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                 /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;               /* empty, type (limit/exact)*/
  }                                             /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
```

```
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next Version 4

Previous Next

Hash Table: Hits

Using the Retrieved Data

Upon entering `D()` for scoring a new position, we first look in the hash table to see what we know about this position. A retrieved table entry that was calculated at sufficient depth that contains an appropriate type result (i.e. exact within the window, or the proper bound type when outside the current window) is immediately used by returning the value without any further search.

If the position was found in the table, it can also be of insifficient depth, or it can be incompatibility with the current window (e.g. an upper bound above beta, where we need lower bounds). In both cases we use the best move from the table as a first try in the search that we now have to do. We then start the iterative deepening at the depth of the table result, or the requested depth if that was lower (and we could not accept the table value because of bad window). After all, the only thing used from previous iterations is the best move, and we already have one from the table. (This might be unjustified if the table value was a fail low, where all moves considered but all were deemed so bad that they were not even worth figuring out how bad. We then might have no idea at all what the best move is. A move from fail high at least must have some merit... The behavior when the the hash value is incompatible with the window is clearly a weak spot in micro-Max, much more advanced algorithms are concievable, deciding with which depth to start the iteration based on the positioning of the table value compared to the current window, and the likelyhood this implies for the 'best move' to be a useful guess.)

If the depth of the table result was that of QS, we don't even use the best move, because there might be none (in which case the table score represents the static evaluation). If the position was not in the table at all, (the entry is empty, as seen from `a->K` being zero), or if the table is 'full' (i.e. in the 8 accessible places, in which case the dummy entry `A[0]` was retrieved, also empty) iteration starts at a depth of zero, with no best move. Just as in normal iterative deepening without a hash table.

Below the code that implements the hash-table retrieval is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
```

```
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
```

```
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop         */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board       */
  p=c;W((*p++=getchar())>10);                        /* read input line   */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one   */
  F(i,0,U)A[i].K=0;                                  /* clear hash table  */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Previous Next

Hash Table: Replacement

Clearing

The hash table is completely cleared before every move. This is good for debugging, because it makes the move independent of the history that might have filled the hash table. It also means that the Zobrist keys don't have to be conserved between moves, and they can start at any value.

Replacement

Micro-Max uses a very simple replacement scheme. For every position only one result is stored. If this result is not exact, it is always replaced, no matter what its depth was. If it was an exact result, it is only replaced by results of better or equal depth. Also equal, because it is more recent, and therefore might be more reliable: a re-search at the same depth does not necessarily return the same result as before, because the hash table is learning all the time (deepening the entry for all positions), so a newer search might benefit from hits to deeper, and therefore more reliable, other positions.

This scheme worked dramatically better than any other I tried in my benchmark, the KPK end-game. Especially clinging on to deep non-exact results produced very poor performance: the bounds stored in the table might have no use after a major re-adjustment of the score (e.g. because the promotion gets within the horizon), after which the new situation has to be searched without the benefit of the hash table, because it is crammed with useless data that refuses to make way. And without the aid of the hashing of shallow nodes the search will never get as deep as before, so it will never get to replace the obsolete entries.

By the way, there is no reason to assume that 'deep' results are more valuable than shallower searches: it is more work to perform such a deep search if it was not in the table, but you won't need to do it that often, while shallow searches are quickly done, but need to be done in very any nodes deep in the tree. In the end that kind of balances.

Below the code that implements the hash-table replacement is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */
```

```
#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;             /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
```

```
              ((x+2)&M||b[x+2]!=u)-1);          /*        squares plus bias */
       if(y+r+1&S){b[y]|=7;i+=C;}               /* promote p to Q, add score*/
      }
      v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,      /* recursive eval. of reply */
           J+J(0),Z+J(8)+G-S,F,y,h);            /* J,Z: hash keys           */
      v-=v>e;                                  /* delayed-gain penalty     */
      if(z==9)                                 /* called as move-legality  */
      {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
       {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
       v=m;                                    /* (prevent fail-lows on    */
      }                                        /*   K-capt. replies)       */
      b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
      if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
      if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
     }                                         /*          if non castling */
     t+=p<5;                                   /* fake capt. for nonsliding*/
     if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
         ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
         G&M&&b[G=(x|7)-(r>>1&7)]&32           /* 1st, virgin R in corner G*/
         &&!(b[G^1]|b[G^2])                    /* 2 empty sqrs. next to R  */
     ){F=y;t--;}                               /* unfake capt., enable e.p.*/
    }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  for(i=0;i<U;i++)A[i].K=0;                          /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Evaluation: Basics

Permanent and Transient

The evaluation consists of two kind of terms. Permanent terms are functions of the position, like material. These can be differentially updated. Transient terms are bonuses or penalties given to encourage or prevent certain behavior. For instance, in the middle game we could put a penalty on King moves. This is not entirely consistent: if the same position can be reached through 2 King moves (say after checks) or 3 King moves (with the opponent also wasting one move in checking) that position would be considered better when it is reached by the first sequence. In a consistent evaluation the score is only dependent on position, not on history. It would not be so easy, though, to make a function of position that discourages King moves, or moving the same piece twice in the opening.

In Micro-Max the evaluation is updated differentially all the way from the start position. The inconsistent transient terms could accumulate to a large value over the game, leading to undesired effects. Even a small penalty on King moves could raise the evaluation to a large score if one of the opponents was forced to do only King moves (because he has just King and some blocked Pawns), while the other can aimlessly push his Bishop around. If the game lasts long enough, the imbalance created by this can raise the score even above that for checkmate, so that the winning side will start to avoid the checkmate because he thinks it a bad deal, and judge it better to pursue the advantage of forcing more King moves out of the opponent!

Material

To avoid such problems, the update of the evaluation score is split in two parts. The permanent part *i* is added to the evaluation *e* on any move, both in the search tree and at the game level. It consists only of material, acording to the usual 1:3:3:5:9-rule. These values are stored in the initialized array `w[]`. The value of a Pawn is set to 99 (to save 1 character over 100...), and the value retrieved from the array is multiplied by that (to avoid a list of multi-digit numbers in the initializer).
A King is represented by a negative value in `w[]`, so that it can be easily tested for, and capture of it can result in special action: immediate return of an 'infinity' score I. It would not suffice to give the King merely a huge value and let the normal desire of the engine to avoid losses protect the King without such special action: the engine would then see no harm in exchange of Kings, and answer check with check!

Center Control

The positional score (see next page) is not implemented consistently, because it is not subtracted when a piece is captured. This might lead to strange play, where the computer tries to earn some positional improvement with a piece that is lost anyway. It prefers to have it captured on a good square, rather than a bad one! In practce this is no big problem, since the desire to avoid the loss in the first place avoids situations like this. But due to the inconsistency, the center score is considered a transient term, and accounted in *v*. Such transient terms are used to update *e* in the tree, but not the evauation score *Q* at the game level.

Pawn Structure

The evaluation of Pawn structure is very minimal. It just discourages moving pawns if there is no other Pawn (of the same color) two squares to the left or right from its initial position. The idea is that in that case moving the Pawn irreversably gives away control over the square(s) it defended. To make the rule consistent you would also have to encourage moves that do cover a square that was not defended by other Pawns before. But in practice this is less important: you always can do that later, when really forced by the tactical situation, while there is no way to move the Pawn back if you regret its move. A worse inconsitency is the fact that no penalty is given when Pawn control over a square is lost due to capture of that Pawn. Anyway, because of all these inconsistencies Pawn structure is treated as a transient term. The main disadvantage of having so many transient terms is that all these aspects are not contributing to the score at the game level, so that the engine will never go for a draw (e.g. if it can force stalemate) if it is equal in material, but badly behind positionally. But in practice stalemates occur only on a bare King, when there is enough material advantage to want to avoid them.

The thing that is most lacking in the evaluation of Pawn structure is doubled Pawns. Probably the most efficient way to make the program better at avoiding doubled Pawns is to give a penalty for blocking your own Pawn by standing directly in front of it with whatever piece. Putting his own pieces in front of his Pawns is the most common source of doubled Pawns. It usually leads to bad Pawn structure even if the piece can not be captured, because the blocked Pawn usually ends up as a severely backward Pawn.

Special Moves

Castling and Queening had special conditional statements to perform them anyway, making it easy to award them appropriately. The score for Queening - being a material advantage - is permanent, that for castling transient. The score for Queening is a little bit off, because it uses a constant (*C*) that was needed anyway and had nearly the correct value (799, in stead of (9-1)*99 = 792). Never mind the inconsistency: you won't queen often enough to have it build up to a large value. If you are the only side that can force Queening the game is quickly won, if both sides queen, the inconsitencies cancel.

Below the code that implements the various evaluation terms is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
```

```
    6,3,5,7,4,5,3,6},                       /* initial piece setup      */
b[129],                                     /* board: half of 16x8+dummy*/
T[1035],                                    /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                     /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                            /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                             /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);              /* first empty or match     */
 a+=i?j:0;                                  /* dummy A[0] if miss & full*/
 if(a->K)                                   /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                     /* examine stored data      */
  if(d>=n)                                  /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;          /* use if window compatible */
   d=n-1;                                   /* or use as iter. start    */
  }X&=~M;Y=a->Y;                            /*      with best-move hint */
  Y=d?Y:0;                                  /* don't try best at d=0    */
 }else d=X=Y=0;                             /* start iter., no best yet */
 N++;                                       /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                   /* iterative deepening loop */
 {x=B=X;                                    /* start scan at prev. best */
  Y|=8&Y>>4;                                /* request try noncastl. 1st*/
  m=d>1?-I:e;                               /* unconsidered:static eval */
  do{u=b[x];                                /* scan board looking for   */
   if(u&k)                                  /*  own piece (inefficient!)*/
   {r=p=u&7;                                /* p = piece type (set r>0) */
    j=o[p+16];                              /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                 /* loop over directions o[] */
    {A:                                     /* resume normal after best */
     y=x;F=G=S;                             /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                             /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                      /* sneak in prev. best move */
      if(y&M)break;                         /* board edge hit           */
      if(p<3&y==E)H=y^16;                   /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;   /* capt. own, bad pawn mode */
      i=99*w[t&7];                          /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;    /* K capt. or bad castling  */
      if(m>=l)goto C;                       /* abort on fail high       */

      if(h=d-(y!=z))                        /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;               /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;          /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}          /* castling: put R & score  */
       if(p<3)                              /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+         /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);      /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}          /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,  /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);       /* J,Z: hash keys           */
       v-=v>e;                              /* delayed-gain penalty     */
       if(z==9)                             /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                 /*   checker: if move found */
        {Q=-e-i;O=F;return l;}               /*   & not in check, signal */
        v=m;                                /* (prevent fail-lows on    */
       }                                    /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;  /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}           /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}            /* update max, mark with S  */
```

```
      }                                     /*          if non castling */
      t+=p<5;                               /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                   /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&              /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32       /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                           /* unfake capt., enable e.p.*/
     }W(!t);                                /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                       /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                     /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;      /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)             /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;             /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;           /* empty, type (limit/exact)*/
  }                                         /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

Previous Next

Evaluation: Piece-Square Tables

Positional Points for Pieces in Good Places

To make for at least vaguely sensical positional play in the opening and mid-game, the program needs a clue as to where its pieces are best deployed. A simple way that is in common use to do this are 'piece-square' tables, arrays that for each piece type tabulate the desirability to have that piece on a certain square. In Micro-Max all piece-square tables are condensed into a single one, used for all pieces except Rook and Queen. The latter don't care where the stand, and thus need no table at all. This 'çenter-positional-score' table, since it is indexed by square-number, also occupies blocks of 8 values separated by spaces of 8 unused elements (for the invalid square-numbers). It thus fits nicely in the unused elements of `b[]`, interleaving the game board with the positional table. i.e. the positional score for square x can be found in `b[x+8]`, while the piece on it is found in `b[x]`.

The value to be on a particular square decreases faster if we approach the board edge. This has the effect that in an end-game against a bare King, the engine finds it important to keep that King against the edge or in the corner, even if it has to move his own King slightly off center to achieve this. Unfortunately Kings are also attracted by the center in the middle game...

Below the code lines that implement the piece-square table are highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.1 (1997 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
```

```
    7,-1,11,6,8,3,6,                        /* 1st dir. in o[] per piece*/
    6,3,5,7,4,5,3,6},                       /* initial piece setup      */
b[129],                                     /* board: half of 16x8+dummy*/
T[1035],                                    /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                     /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=9,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                            /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                             /* try 8 consec. locations  */
 W((h=A[++j].K)&&h-Z&&--i);                  /* first empty or match     */
 a+=i?j:0;                                  /* dummy A[0] if miss & full*/
 if(a->K)                                   /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                     /* examine stored data      */
  if(d>=n)                                  /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;          /* use if window compatible */
   d=n-1;                                   /* or use as iter. start    */
  }X&=~M;Y=a->Y;                            /*      with best-move hint */
  Y=d?Y:0;                                  /* don't try best at d=0    */
 }else d=X=Y=0;                             /* start iter., no best yet */
 N++;                                       /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                   /* iterative deepening loop */
 {x=B=X;                                    /* start scan at prev. best */
  Y|=8&Y>>4;                                /* request try noncastl. 1st*/
  m=d>1?-I:e;                               /* unconsidered:static eval */
  do{u=b[x];                                /* scan board looking for   */
   if(u&k)                                  /*  own piece (inefficient!)*/
   {r=p=u&7;                                /* p = piece type (set r>0) */
    j=o[p+16];                              /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                 /* loop over directions o[] */
    {A:                                     /* resume normal after best */
     y=x;F=G=S;                             /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                             /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                      /* sneak in prev. best move */
      if(y&M)break;                         /* board edge hit           */
      if(p<3&y==E)H=y^16;                   /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;   /* capt. own, bad pawn mode */
      i=99*w[t&7];                          /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;    /* K capt. or bad castling  */
      if(m>=l)goto C;                       /* abort on fail high       */

      if(h=d-(y!=z))                        /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;               /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;          /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}          /* castling: put R & score  */
       if(p<3)                              /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+         /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);      /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}          /* promote p to Q, add score*/
       }
       v=-D(24-k,-l,m>q?-m:-q,-e-v-i,       /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);       /* J,Z: hash keys           */
       v-=v>e+50;                           /* delayed-gain penalty     */
       if(z==9)                             /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                 /*   checker: if move found */
        {Q=-e-i;O=F;return l;}               /*   & not in check, signal */
        v=m;                                /* (prevent fail-lows on    */
       }                                    /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t; /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}           /* best=1st done,redo normal*/
```

```
       if(v>m){m=v;X=x;Y=y|S&G;}              /* update max, mark with S  */
      }                                       /*          if non castling */
      t+=p<5;                                 /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                     /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32         /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                  /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                             /* unfake capt., enable e.p.*/
     }W(!t);                                  /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                         /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                       /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;        /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)               /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;               /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;             /* empty, type (limit/exact)*/
  }                                           /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 for(i=0;i<8;i++)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  for(j=0;j<8;j++)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                  /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                               /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                       /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                           /* or think up one    */
  F(i,0,U)A[i].K=0;                                 /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                /* check legality & do*/
 }
}
```

Previous Next

Evaluation: Mate

Checkmate

In end leaves King capture is recognized and awarded the highest possible score ('infinity') +I. It is not known there if the King of the side to move is in check, this requires at least a one-ply search. So checkmate is not recognized in end leaves. It is a general weakness of micro-Max that it does not recognize threats in end leaves, and if there is to be any solution the best way would be to implement general threat detection, for instance by allowing null move (early) in QS, and using the null-move score rather than the static evaluation as a reference. It did not fit in 2048 characters (yet...), but is at the top of the wish list!

So checkmate is only visible in a search of at last one ply. In such a search the best score *m* starts as `-I`, and if all generated moves result in King capture on the next ply, it stays at `-I`. From this the engine knows that there were no legal moves. The only thing left to do in that case, is to see if we are in check. To this end `D()` is called in the current position with the other side to move, and zero depth (i.e. `n==1`), and a dummy 'current evaluation' of zero. If we are checkmated, this call will return +I, otherwise the 'evaluation' 0. We return minus half of that as the evaluation of the position, i.e. the score for being checkmated is -I/2. A stalemate produces a zero score. (Note that `n==1` only searches recaptures, and won't find any because we pass it the invalid square number S as recapture square. So it just runs the move generator, which aborts on encountering a King capture.)

Stalemate

From the description above, you might think that stalemate detection is foolproof. But alpha-beta, and in particular the beta cutoffs, interfere with proper operation of the described mechanism. Stalemate violates one of the basic premises of alpha beta: if you found a move for the opponent that makes you previous move bad enough, reject that previous move straight away, because searching further for the opponent can only make your move worse. Not so with stalemate: suppose I am badly behind, and have already found a move a few ply earlier that preserves the status quo (for as long as it lasts...). Now I am trying another move, say BxP, and the opponent in reply takes my Bishop. Bad move, I didn't need to lose the Bishop. So the opponents RxB fails high, and I am not looking further. That was too bad, because as it was he could not only take RxB, but also RxK: the Bishop with which I took the Pawn was in fact pinned to the King. Of course losing the King is even worse than losing the Bishop, so the cutoff was justified. ...Unless I had no other moves and this was a stalemate (W:Kh1,Bg1; B:Kg3,Re1,Pe3). Then losing a King is much better than losing the Bishop, because it plugged my last legal move on the ply before. If the opponent's move is too good, that is just too bad for him, because it does not mean that my move was bad, it means that it was illegal. And that can be good, if stalemate makes me happy!

To prevent this problem beta cutoffs should not be allowed in the reply to the first move of the opponent, i.e. as long as that opponent had no legal moves yet. One should always continue the search (if only at depth zero) to see if there is a King capture. Micro-Max ignores this problem, and might thus no and then overlook a stalemating of a non-bare King. This probably has zero-impact on his playing strength, the stalemate detection is only of practical importance to avoid stalemating a bare King with overwhelming material advantage (as in KQK), because you confuse stalemate with checkmate.

Below the code that implements the checkmate and stalemate scoring is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
```

```
/* - best-move-first 'sorting'                                          */
/* - a hash table storing score and best move                           */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking  */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e4,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
      if(m>=l)goto C;                          /* abort on fail high       */

      if(h=d-(y!=z))                           /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                  /* center positional pts.   */
```

```
       b[G]=b[H]=b[x]=0;b[y]=u&31;             /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}             /* castling: put R & score  */
       if(p<3)                                 /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+            /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);         /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}             /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,    /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);          /* J,Z: hash keys           */
       v-=v>e;                                 /* delayed-gain penalty     */
       if(z==9)                                /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                    /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                 /*   & not in check, signal */
        v=m;                                   /* (prevent fail-lows on    */
       }                                       /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;    /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}              /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}               /* update max, mark with S  */
      }                                        /*          if non castling */
      t+=p<5;                                  /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                      /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                 /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32          /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                   /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                              /* unfake capt., enable e.p.*/
     }W(!t);                                   /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                          /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                        /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,S,1)/2;         /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;              /* empty, type (limit/exact)*/
  }                                            /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop         */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board       */
  p=c;W((*p++=getchar())>10);                        /* read input line   */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one   */
  F(i,0,U)A[i].K=0;                                  /* clear hash table  */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next Version 4

Previous Next

Evaluation: Aging

Why hurry?

At first glance it does not seem very important to prefer mate in one over mate in two. A win is a win, isn't it? Well, actually, it isn't...

If a computer prefers mate in two over mate in one, on the next move it will be faced with the same choice again, and what then will be mate in two is now mate in three, and so on. In short, a program that does not prefer the short route, might never get to its destination. Of course implementation of the 50-move rule might make the program see the light after 49 moves, in this particular example. But often the 50-move rule is similar to painting yoursef in a corner: by the time the paint seems awfully close, there is no way out anymore.

Take the KBNK end-game as an example. To enforce the checkmate might take 33 moves (66 ply), and lies usually far beyond the horizon. If we would have a method to directly judge how far each position is from the mate (as in an end-game TableBase) life would be easy and even the shallowest search would be able to run towards the checkmate. But in the absence of a TB we only have an approximate idea how far the bare King is removed from its doom, like its distance from the edge or to the proper corner. In that case the path towards checkmate might not see a monotonous improvement (for the winning side) of the approximate measure for progress. It is as if we have to climb a mountain range: to get to the next, higher summit we have to cross a mountain pass first. In a fogg, just looking at the slope of the local terrain, we wouldn't know in which direction to go. We need at least enough visibility to see the peak across the valley. But it would be stupid to assume that that is the final destination, just because it is the only peak in view. If we would extend our picnic until we can just make that peak before dark, we are in for a cold night indeed!

In computer chess the search delivers the required visibility. But the path might go up for 5 moves to a score +20, than downhill up to move 10, to reach scores above +20 only from move 15 onwards. If the engine can see 10 moves ahead, the tempory setback between move 5 and 10 would not be able to prevent it from finding the path to enlightning. But only when it first goes to the peak at +20 moves, and searches from there. If it would be in no hurry to take the first of those 5 steps towards +20, because there are no paths within its horizon that can reach a score better than +20, and among the many paths that do end at this +20 position in a leaf node a large fraction just fools around for 5 moves before getting to the point, it might fool around forever. Because after wasting the first move on a detour path, it can see upto move 11, extending its time left for fooling around by one more move. After move 40 it will really get worried that all paths suddenly seem to lead to draw...

The Delay Penalty

To prevent such problems it is important that the engine prefers the fastest path to the best position it can foresee. In Micro-Max the drive for this is provided by applying a small penalty (1 centipawn) for any move that you (have to) delay in improving your situation. To see if improvement is possible, it looks at the difference of the score of the best move and the current evaluation. If this is positive and above a certain threshold, one point is deducted from the score. Since the opponent in such cases foresees that it can be forced into a situation that

gets worse, delaying tactics are in his favor, and he does not receive a penalty. So every move needed to reach an attractive position depreciates that position by 1 point.

Adjusting the Window

I you adjust the value of a move *after* the search, you have to be careful to precompensate this adjustment in the (alpha, beta) window. If the move will be chosen as best will be decided after the adjustment, and during the search the adjustment has not been made yet. This is unavoidable, because it depends on the outcome of the search if the adjustment will be made. So if scores larger than current_eval will be depreciated by 1 point, window limits larger than current_eval will have to be increased by one point. It will lead to horrible blunders if you make the window too small, even by 1 point. Making the window too large only wastes time, but not very much if it is only by 1 point. So Micro-Max only increases beta (*l*), which will be the next ply's -alpha, by 1 point in that case.

Below the code that implements the delay penalty is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.2 (2000 characters) features:                                */
/* - recursive negamax search                                             */
/* - quiescence search with recaptures                                    */
/* - recapture extensions                                                 */
/* - (internal) iterative deepening                                       */
/* - best-move-first 'sorting'                                            */
/* - a hash table storing score and best move                             */
/* - full FIDE rules (expt minor promotion) and move-legality checking    */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
```

```
  if(d>=n)                                  /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;           /* use if window compatible */
   d=n-1;                                   /* or use as iter. start    */
  }X&=~M;Y=a->Y;                            /*      with best-move hint */
  Y=d?Y:0;                                  /* don't try best at d=0    */
 }else d=X=Y=0;                             /* start iter., no best yet */
 N++;                                       /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                   /* iterative deepening loop */
 {x=B=X;                                    /* start scan at prev. best */
  Y|=8&Y>>4;                                /* request try noncastl. 1st*/
  m=d>1?-I:e;                               /* unconsidered:static eval */
  do{u=b[x];                                /* scan board looking for   */
   if(u&k)                                  /*  own piece (inefficient!)*/
   {r=p=u&7;                                /* p = piece type (set r>0) */
    j=o[p+16];                              /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                 /* loop over directions o[] */
    {A:                                     /* resume normal after best */
     y=x;F=G=S;                             /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                             /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                      /* sneak in prev. best move */
      if(y&M)break;                         /* board edge hit           */
      if(p<3&y==E)H=y^16;                   /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;   /* capt. own, bad pawn mode */
      i=99*w[t&7];                          /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;    /* K capt. or bad castling  */
      if(m>=l)goto C;                       /* abort on fail high       */

      if(h=d-(y!=z))                        /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;               /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;          /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}          /* castling: put R & score  */
       if(p<3)                              /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+         /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);      /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}          /* promote p to Q, add score*/
       }
       v=-D(24-k,-l-(l>e),m>q?-m:-q,-e-v-i,  /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);       /* J,Z: hash keys           */
       v-=v>e;                              /* delayed-gain penalty     */
       if(z==9)                             /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                 /*   checker: if move found */
        {Q=-e-i;O=F;return l;}               /*   & not in check, signal */
        v=m;                                /* (prevent fail-lows on    */
       }                                    /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;  /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}           /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}            /* update max, mark with S  */
      }                                     /*          if non castling */
      t+=p<5;                               /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                   /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&              /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32       /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                           /* unfake capt., enable e.p.*/
     }W(!t);                                /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                       /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                     /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,Z,S,z,1)/2;      /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)             /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;             /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;           /* empty, type (limit/exact)*/
  }                                         /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
```

```
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```

Previous Next

The Interface: Move-Legality Checking

Reading the input move

The input code just reads characters upto a newline, and then combines first with second and third with fourth to get the move *K,L*. The conversion from ASCII to a square number involves a constant that is so close to the value of a Queen minus Pawn (799 in stead of (9-1)*99 = 792) that is serves a double use and is stored in a variabe *C*. Before parsing the move micro-Max checks, however, if the input consisted of a single newline, in which case it thinks up a move itself by calling `D()` with the invalid sqauare umber 8 in the capture square *z*. This also puts a move in the global variables*K,L*.

At any move the program is thus ready to accept either an input move in algabraic notation, or to start thinking and playing his own (on an empty input line). This makes it easy to set up a starting position, change sides during the game, or have the computer play against itself.

Doing the Moves

To execute the moves at the game level, be it from user input or from the engine, the same code is used as for do_move in the tree search. In fact, the search routine is called to do this. The difference is that the move is not taken back, as it would be in a normal search! To this end the recursive search routine has an exit point just before the normal move undo. It is used if the routine was called with the invalid square number 9 as recapture square *z*, and the move that was just searched equals the move held in global variables*K,L*.

Avoiding Check

To make sure that moves that put the King in check are not accepted, the call to `D()` that performs the move should search to a depth of one ply, (called with `n==2`) to make sure that all replies are considered and checked for King capture (or illegal castling). Fail-high cutoffs on ply 2 must be prevented, or we could miss some King captures. To this end *m* is artificially kept at minus infinity (`-I`) during this search.

E.p. capture

The move-legality check also has to know if e.p. capture is allowed (and where). To this end the e.p. square is saved in global variable *O* when a move is accepted, and passed to the next call to `D()` (both for thinking up a move, and for later checking/performing it). The same is done with the differentially updated evaluation score, in *Q*.

Below the code that implements the move-legality checking is highlighted:

```
/**************************************************************************/
/*                              micro-Max,                                */
/* A chess program smaller than 2KB (of non-blank source), by H.G. Muller  */
/**************************************************************************/
/* version 3.1 (1997 characters) features:                                */
```

```
/* - recursive negamax search                                          */
/* - quiescence search with recaptures                                 */
/* - recapture extensions                                              */
/* - (internal) iterative deepening                                    */
/* - best-move-first 'sorting'                                         */
/* - a hash table storing score and best move                          */
/* - full FIDE rules (expt minor ptomotion) and move-legality checking */

#define F(I,S,N) for(I=S;I<N;I++)
#define W(A) while(A)
>#define K(A,B) *(int*)(T+A+(B&8)+S*(B&7))
#define J(A) K(y+A,b[y])-K(x+A,u)-K(H+A,t)

#define U 16777224
struct _ {int K,V;char X,Y,D;} A[U];           /* hash table, 16M+8 entries*/

int V=112,M=136,S=128,I=8e3,C=799,Q,N,i;       /* V=0x70=rank mask, M=0x88 */

char O,K,L,
w[]={0,1,1,3,-1,3,5,9},                        /* relative piece values    */
o[]={-16,-15,-17,0,1,16,0,1,16,15,17,0,14,18,31,33,0, /* step-vector lists */
     7,-1,11,6,8,3,6,                          /* 1st dir. in o[] per piece*/
     6,3,5,7,4,5,3,6},                         /* initial piece setup      */
b[129],                                        /* board: half of 16x8+dummy*/
T[1035],                                       /* hash translation table   */

n[]=".?+nkbrq?*?NKBRQ";                        /* piece symbols on printout*/

D(k,q,l,e,J,Z,E,z,n)    /* recursive minimax search, k=moving side, n=depth*/
int k,q,l,e,J,Z,E,z,n;  /* (q,l)=window, e=current eval. score, E=e.p. sqr.*/
{                       /* e=score, z=prev.dest; J,Z=hashkeys; return score*/
 int j,r,m,v,d,h,i=8,F,G;
 char t,p,u,x,y,X,Y,H,B;
 struct _*a=A;
                                               /* lookup pos. in hash table*/
 j=(k*E^J)&U-9;                                /* try 8 consec. locations  */
 while((h=A[++j].K)&&h-Z&&--i);                 /* first empty or match     */
 a+=i?j:0;                                     /* dummy A[0] if miss & full*/
 if(a->K)                                      /* hit: pos. is in hash tab */
 {d=a->D;v=a->V;X=a->X;                        /* examine stored data      */
  if(d>=n)                                     /* if depth sufficient:     */
  {if(v>=l|X&S&&v<=q|X&8)return v;             /* use if window compatible */
   d=n-1;                                      /* or use as iter. start    */
  }X&=~M;Y=a->Y;                               /*      with best-move hint */
  Y=d?Y:0;                                     /* don't try best at d=0    */
 }else d=X=Y=0;                                /* start iter., no best yet */
 N++;                                          /* node count (for timing)  */
 W(d++<n|z==8&N<1e7&d<98)                      /* iterative deepening loop */
 {x=B=X;                                       /* start scan at prev. best */
  Y|=8&Y>>4;                                   /* request try noncastl. 1st*/
  m=d>1?-I:e;                                  /* unconsidered:static eval */
  do{u=b[x];                                   /* scan board looking for   */
   if(u&k)                                     /*  own piece (inefficient!)*/
   {r=p=u&7;                                   /* p = piece type (set r>0) */
    j=o[p+16];                                 /* first step vector f.piece*/
    W(r=p>2&r<0?-r:-o[++j])                    /* loop over directions o[] */
    {A:                                        /* resume normal after best */
     y=x;F=G=S;                                /* (x,y)=move, (F,G)=castl.R*/
     do{H=y+=r;                                /* y traverses ray          */
      if(Y&8)H=y=Y&~M;                         /* sneak in prev. best move */
      if(y&M)break;                            /* board edge hit           */
      if(p<3&y==E)H=y^16;                      /* shift capt.sqr. H if e.p.*/
      t=b[H];if(t&k|p<3&!(r&7)!=!t)break;      /* capt. own, bad pawn mode */
      i=99*w[t&7];                             /* value of capt. piece t   */
      if(i<0||E-S&&b[E]&&y-E<2&E-y<2)m=I;       /* K capt. or bad castling  */
```

```
      if(m>=l)goto C;                           /* abort on fail high       */

      if(h=d-(y!=z))                            /* remaining depth(-recapt.)*/
      {v=p<6?b[x+8]-b[y+8]:0;                   /* center positional pts.   */
       b[G]=b[H]=b[x]=0;b[y]=u&31;              /* do move, strip virgin-bit*/
       if(!(G&M)){b[F]=k+6;v+=30;}              /* castling: put R & score  */
       if(p<3)                                  /* pawns:                   */
       {v-=9*(((x-2)&M||b[x-2]!=u)+             /* structure, undefended    */
              ((x+2)&M||b[x+2]!=u)-1);          /*        squares plus bias */
        if(y+r+1&S){b[y]|=7;i+=C;}              /* promote p to Q, add score*/
       }
       v=-D(24-k,-l,m>q?-m:-q,-e-v-i,           /* recursive eval. of reply */
            J+J(0),Z+J(8)+G-S,F,y,h);           /* J,Z: hash keys           */
       v-=v>e+50;                               /* delayed-gain penalty     */
       if(z==9)                                 /* called as move-legality  */
       {if(v!=-I&x==K&y==L)                     /*   checker: if move found */
        {Q=-e-i;O=F;return l;}                  /*   & not in check, signal */
        v=m;                                    /* (prevent fail-lows on    */
       }                                        /*   K-capt. replies)       */
       b[G]=k+38;b[F]=b[y]=0;b[x]=u;b[H]=t;     /* undo move,G can be dummy */
       if(Y&8){m=v;Y&=~8;goto A;}               /* best=1st done,redo normal*/
       if(v>m){m=v;X=x;Y=y|S&G;}                /* update max, mark with S  */
      }                                         /*          if non castling */
      t+=p<5;                                   /* fake capt. for nonsliding*/
      if(p<3&6*k+(y&V)==S                       /* pawn on 3rd/6th, or      */
          ||(u&~24)==36&j==7&&                  /* virgin K moving sideways,*/
          G&M&&b[G=(x|7)-(r>>1&7)]&32           /* 1st, virgin R in corner G*/
          &&!(b[G^1]|b[G^2])                    /* 2 empty sqrs. next to R  */
      ){F=y;t--;}                               /* unfake capt., enable e.p.*/
     }W(!t);                                    /* if not capt. continue ray*/
  }}}W((x=x+9&~M)-B);                           /* next sqr. of board, wrap */
C:if(m>I/4|m<-I/4)d=99;                         /* mate is indep. of depth  */
  m=m+I?m:-D(24-k,-I,I,0,J,K,S,z,1)/2;          /* best loses K: (stale)mate*/
  if(!a->K|(a->X&M)!=M|a->D<=d)                 /* if new/better type/depth:*/
  {a->K=Z;a->V=m;a->D=d;A->K=0;                 /* store in hash,dummy stays*/
   a->X=X|8*(m>q)|S*(m<l);a->Y=Y;               /* empty, type (limit/exact)*/
  }                                             /*    encoded in X S,8 bits */
/*if(z==8)printf("%2d ply, %9d searched, %6d by (%2x,%2x)\n",d-1,N,m,X,Y&0x77);*/
 }
 if(z&8){K=X;L=Y&~M;}
 return m;
}

main()
{
 int j,k=8,*p,c[9];

 F(i,0,8)
 {b[i]=(b[i+V]=o[i+24]+40)+8;b[i+16]=18;b[i+96]=9;   /* initial board setup*/
  F(j,0,8)b[16*j+i+8]=(i-4)*(i-4)+(j-3.5)*(j-3.5);   /* center-pts table   */
 }                                                   /*(in unused half b[])*/
 F(i,M,1035)T[i]=random()>>9;

 W(1)                                                /* play loop          */
 {F(i,0,121)printf(" %c",i&8&&(i+=7)?10:n[b[i]&15]); /* print board        */
  p=c;W((*p++=getchar())>10);                        /* read input line    */
  N=0;
  if(*c-10){K=c[0]-16*c[1]+C;L=c[2]-16*c[3]+C;}else  /* parse entered move */
   D(k,-I,I,Q,1,1,O,8,0);                            /* or think up one    */
  F(i,0,U)A[i].K=0;                                  /* clear hash table   */
  if(D(k,-I,I,Q,1,1,O,9,2)==I)k^=24;                 /* check legality & do*/
 }
}
```