**An obvious place to start**

In chess, both players know where the pieces are, they alternate moves, and they are free to make any legal move.  The object of the game is to checkmate the other player, to avoid being checkmated, or to achieve a draw if that's the best thing given the circumstances.

A chess program selects moves via use of a *search* function.  A search function is a function that is passed information about the game, and tries to find the best move for side that the program is playing.

An obvious sort of search function to use is a *tree-searching* function.  A game of chess can be considered as a (usually) large n-ary tree ("n-ary" means that from each node in the tree, there are some arbitrary number of branches leading to other nodes).  The position that is on the board now is the *root position* or *root node*.  Positions that can be reached in one move from the root position are reached by *branches* from the root position.  These positions are called *successor positions* or *successor nodes*, Each of these successor positions has a series of branches emanating from it, each of which represents a legal move from that position.

The chess tree is very bushy (usually about 35 branches from each position), and very deep.

It would be possible to represent every conceivable chess game in a giant n-ary tree, then perform a tree-search that finds the game that would be produced if both sides were to play the best possible moves.  An obvious algorithm to use to do this is called *min-max search*.

It is possible to use min-max search to solve (completely understand) simple games like tic-tac-toe.  The game-tree in tic-tac-toe is not very bushy and it's not very deep, so the entire tree can be traversed, the game can be completely understood, and a move can be made in any position that is guaranteed to be the best move.

It is possible to do this with chess in a mathematical sense, but it is not possible to do on any computer that is likely to exist now or even in the far future.  Even so, it is still possible to use min-max search as the basis of a program that plays chess.  Rather than min-maxing the entire chess tree, it is very possible to min-max the first few moves from a given start position.  Since the positions at the leaves of the tree aren't necessarily searched all the way out to mate or forced draw, a heuristic function, traditionally called "Evaluate", is used to assign values to these positions.  These values are guesses, although chess programmers hope that they are educated guesses.

**The Evaluate function, in the context of min-max**

I am not going to go into a lot of detail about this function here.  I am going to define it, so I can refer to it in later sections.  Evaluate is a function that first returns an exact value for the position, if possible, and an heuristic value if an exact value is not available.  It can be defined one of two ways:

1) The function returns a very large positive value if Black is checkmated, a very large negative value if White is checkmated, and a constant value, probably zero or something near zero, if the game is drawn now (for instance if the side to move is stalemated, or if there are bare kings).  If the position doesn't represent the end of the game, an heuristic value is returned.  I won't go into detail about how the heuristic value is determined, but suffice to say that material balance is a primary consideration (the value will tend to be large if White has much more material on the board), but other positional considerations (pawn structure, king safety, strong pieces, etc.) are detected and accounted for.  The value returned by the heuristic function will always be positive if White has won or is winning, negative if Black has won or is winning, and around zero if the game is even or is a draw.

2) This function works exactly the same as the other function, only the function returns a positive value if the side to move *in the current node* is "ahead", and a negative value if the side currently to move is "behind".

**How min-max works**

Min-max is a pair of functions that look almost the same, or one function that has duplicated logic.  There is a better function that does the same thing with less code, but in response to an email criticism of this page, I will first describe the "pure" (inelegant) min-max function.  Here's the code:

int MinMax(int depth)

{

    if (SideToMove() == WHITE)    // White is the "maximizing" player.

        return Max(depth);

    else                          // Black is the "minimizing" player.

        return Min(depth);

}

int Max(int depth)

{

    int best = -INFINITY;

    if (depth <= 0)

        return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();

        val = Min(depth - 1);

        UnmakeMove();

        if (val > best)

            best = val;  
    }

    return best;

}

int Min(int depth)

{

    int best = INFINITY;  // <-- Note that this is different than in "Max".

    if (depth <= 0)

        return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();

        val = Max(depth - 1);

        UnmakeMove();

        if (val < best)  // <-- Note that this is different than in "Max".

            best = val;  
    }

    return best;

}

The code is called something like this:

val = MinMax(5);

This will return the value of the position, given 5-plies of look-ahead.

The "Evaluate" function used in the above is the one described in my first definition -- it always returns the value of the position from White's point of view.

I'll briefly describe how the function operates.  Let's say that at the root position (the position on the board now), it's White's turn to move.  The Max function is called, and all of White's legal moves are generated.  In each resulting position, the "Min" function is called.  The "Min" function scores the position and returns a value.  Since it is White to move, and White wants a more positive score if possible, the move with the largest score is selected as best, and the value of this move is returned.

The "Min" function works in reverse.  The "Min" function is called when it's Black's turn to move, and black wants a more negative score, so the move with the most negative score is selected.

These functions are dual recursive, meaning that they call each other until the desired search depth is reached.  When the functions "bottom out", they return the result of the "Evaluate" function.

If you call "MinMax" with a depth value of 1, essentially all that happens is that the "Evaluate" function is called after each legal move, and the position that results in the "best" value for the side to move is selected.  If depth is greater than one, the other side gets a chance to respond, and chose its best move.

The above shouldn't be hard to understand, but it's a lot of code, and there is a better way.

**The nega-max function**

Nega-max is just min-max with an optimization.  The "Evaluate" function returns scores via my second definition -- it returns scores that are positive if the side to move at the current node is ahead, and everything else is also viewed from the perspective of the side to move.  When the value is returned, it is negated, because it is now being viewed from the perspective of the other side.  Here is the code:

int NegaMax(int depth)

{

    int best = -INFINITY;

    if (depth <= 0)

        return Evaluate();

    GenerateLegalMoves();

    while (MovesLeft()) {

        MakeNextMove();

        val = -NegaMax(depth - 1); // Note the minus sign here.

        UnmakeMove();

        if (val > best)

            best = val;  
    }

    return best;

}

The function negates the return value in order to reflect the change and perspective that results when the side to move is changed.  So let's say that you call this from the root with White to move.  If there is no remaining depth, "Evaluate" returns its value from the White perspective.  If there is depth remaining, the successor positions are generated, and the function recurses once for each of those.  Each of these calls evaluates the position from the Black perspective, meaning that the score is larger if Black is doing better.  When the values are returned, they are negated, so that they may be evaluated from White's point of view.

This function traverses the same nodes as "min-max" in the same order, and produces the same result.  It's much less code, which means that there is less opportunity to create a bug due to code replication, and the code is easier to maintain.