



C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 - May 2007

Assignment 1: Technique points

Lisa (Ling) Liu

Main steps in chess program



- Board representation
- Compute legal moves
- Find best move in the legal moves

Board representation



Real board

a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

0x88 representation

```
70 71 72 73 74 75 76 77 | 78 79 7a 7b 7c 7d 7e 7f
60 61 62 63 64 65 66 67 | 68 69 6a 6b 6c 6d 6e 6f
50 51 52 53 54 55 56 57 | 58 59 5a 5b 5c 5d 5e 5f
40 41 42 43 44 45 46 47 | 48 49 4a 4b 4c 4d 4e 4f
30 31 32 33 34 35 36 37 | 38 39 3a 3b 3c 3d 3e 3f
20 21 22 23 24 25 26 27 | 28 29 2a 2b 2c 2d 2e 2f
10 11 12 13 14 15 16 17 | 18 19 1a 1b 1c 1d 1e 1f
 0  1  2  3  4  5  6  7 |  8  9  a  b  c  d  e  f
```

Check if a position is legal



1. Get the index of the position

$\text{index} = \text{rank} * 4 + \text{file}$ ($\text{index} = \text{rank} \ll 4 \mid \text{file}$)

2. Check $0 \leq \text{index} < 128$

if (! (index & 0x80))

3. Check index in the left part of 0x88 board
(corresponding to the real board)

if (! (index & 0x88))

The corresponding code



```
public class Square
{
    private int _file;
    private int _rank;
    private int _index;
    private Piece _piece;
    ...
}

public abstract class Piece
{
    protected Square SSquare;
    ...
}
```

```
public class Board
{
    private constant int SQUARE_COUNT = 128;
    private static Square[] squares;
    static Game ()
    {
        squares = new Square[SQUARE_COUNT];}
    }
    static Square GetSquare (index)
    {
        if (index & 0x88)
            return squares[index];
    }
    ...
}
```

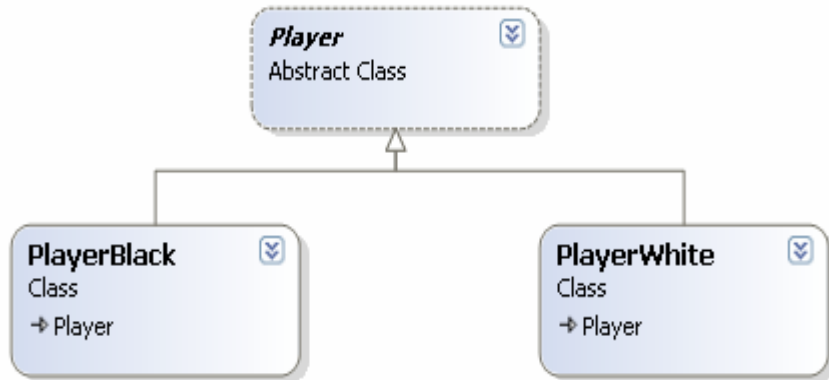
Calculate legal moves for each piece



1. Find the movement offset for each piece according to the rules
 - pawn (fixed offsets)
 - if has not moved
 - White player, offset = {32}
 - Black player, offset = {-32}
 - if has moved
 - Forward offset
 - White player: offset = {16}
 - Black player: offset = {-16}
 - Attack left, right
 - White player: offset = {15, 17}
 - Black player: offset = {-15, 17}

- Knight (fixed offset)
 - {14, -14, 18, -18, 31, -31, 33, -33};
- King (fixed offset)
 - {1, -1, 15, -15, 16, -16, 17, -17}
- Rook (iterated offset)
 - {1, -1, 16, -16}
- Bishop (iterated offset)
 - {15, -15, 17, -17}
- Queen (iterated offset)
 - {1, -1, 15, -15, 16, -16, 17, -17}

Abstract class player



properties:
BishopIteratedOffsets,
KnightOffsets,
RookIteratedOffsets,
QueenIteratedOffsets,
KingOffsets

class PlayerWhite

properties:
PawnForwardOffset,
PawnRightAttackOffset,
PawnLeftAttackOffset

class PlayerBlack

properties:
PawnForwardOffset,
PawnRightAttackOffset,
PawnLeftAttackOffset

Get legal moves of each piece (step 1)



Compute moves that are not out of the board

Bishop, Queen, Rook mode:

```
void GenerateMovesIterated (ref Moves moves, Piece piece, Player
player, int[] Offsets)
{
    foreach (int offset in Offsets)
    {
        index = piece.Square.Index;
        index += offset;
        while ( (square = Board.GetSquare(index))!=null )
        {
            if (...)
                create move and moves.add(move);
        }
    }
}
```

Compute moves that are not out of the board

Knight mode:

```
void GenerateMovesFixed (ref Moves moves, Piece piece, Player
player, int[] Offsets)
{
    foreach (int offset in Offsets)
    {
        index = piece.Square.Index;
        index += offset;
        if ( (square = Board.GetSquare(index))!=null )
        {
            if (...)
                create move and moves.add(move);
        }
    }
}
```

Compute moves that are not out of the board

King mode:

```
void GenerateMovesKing (ref Moves moves, Piece piece, Player
player, int[] Offsets)
{
    GenerateMovesFixed (ref Moves moves, Piece piece, Player
player, int[] Offsets);
    if (canCastleKingSide)
        ...
    if (canCastleQueenSide)
        ...
}
```

Compute moves that are not out of the board

Pawn mode:

a bit complicated!

Get legal moves of each piece (step 2)



```
foreach (Move move in Moves
{
    Move undoMove = move.Piece.MoveTo (Move.To)
    if (move.Piece.Player.IsInCheck)
        Moves.Remove (move);
    undoMove.Undo();
}
```

```
public class Move
{
    private Square _from, _to;
    private Piece _capturedPiece;
    private int _capturedPieceIndex
    Enum.Name _name;
    ...
    public void Undo ()...
}
```

```
public abstract class Player
{
    ...
    public bool IsInCheck
    {
        return _king.Square.CanBeMovedToBy(this.
opponent);
    }
}
```

FindBestMove



Evaluation function

Minmax