Chess Engine: 0x88 Board Representation & Core Types

Overview

This chess engine uses the **0x88 board representation**, a clever trick for efficient board bounds checking. This section establishes the fundamental data structures and utility functions that the rest of the engine builds upon.

The 0x88 Board Representation

Core Concept

```
pub type Sq = usize; // Square index type (0..127)
const BOARD_SIZE: usize = 128; // Using 0x88 representation
```

Instead of using a standard 64-square array (8×8), this engine uses a 128-element array. Here's why this is brilliant:

- **Standard approach**: Square indices 0-63 for an 8×8 board
- **0x88 approach**: Square indices spread across 0-127, using only specific indices

The Magic of 0x88

The key insight is in the binary representation: - Valid squares have indices where `(index & 0x88) == 0` - 0x88 in binary is `10001000` - This means valid squares must have: - Bit 7 = 0 (values < 128) - Bit 3 = 0 (file values 0-7)

Square Indexing Formula

```
fn sq(rank: i32, file: i32) -> Sq {
    ((rank << 4) | file) as usize
}
```

This creates indices like: - a1 (rank=0, file=0): `(0 << 4) | 0 = 0x00 = 0` - h1 (rank=0, file=7): `(0 << 4) | 7 = 0x07 = 7` - a8 (rank=7, file=0): `(7 << 4) | 0 = 0x70 = 112` - h8 (rank=7, file=7): `(7 << 4) | 7 = 0x77 = 119`

Why This Matters

The 0x88 trick enables ultra-fast bounds checking:

```
fn on_board(s: Sq) -> bool {
    (s & 0x88) == 0
}
```

When you move a piece by adding offsets (e.g., knight moves +31, +33, etc.), invalid moves naturally produce indices with bits 3 or 7 set, failing the bounds check without explicit range testing.

Piece Representation

The Piece Enum

```
#[derive(Copy, Clone, PartialEq, Eq, Debug)]
enum Piece {
    Empty,
    WP, WN, WB, WR, WQ, WK,  // White pieces
    BP, BN, BB, BR, BQ, BK,  // Black pieces
}
```

Key Design Decisions: - `Copy` trait: Pieces are simple values that can be copied cheaply - `PartialEq/Eq`: Enables piece comparison (e.g., `piece == Piece::Empty`) - Separate enum values for each color simplifies color checking

Piece Utility Methods

Character Conversion:

```
fn from_char(c: char) -> Piece  // 'P' -> Piece::WP, 'p' -> Piece::BP
fn to_char(self) -> char        // Piece::WP -> 'P', Empty -> '.'
```

This follows standard chess notation where uppercase = white, lowercase = black.

Color Detection:

```
fn is_white(self) -> bool  // Pattern matches white pieces
fn is_black(self) -> bool  // Pattern matches black pieces
fn is_empty(self) -> bool  // Checks if square is empty
```

These use Rust's pattern matching for efficient, branch-free checks.

Coordinate Translation

Algebraic Notation Conversion

```
fn sq_to_alg(s: Sq) -> String {
    let r = (s >> 4) as i32;  // Extract rank (high nibble)
    let f = (s & 15) as i32;   // Extract file (low nibble)
    // Converts to "e2", "d7", etc.
}

fn alg_to_sq(s: &str) -> Option<Sq> {
    // Parses "e2" -> square index
    // Returns None for invalid input
}
```

Bit Manipulation Details: - `s >> 4`: Right shift by 4 bits extracts the rank (divides by 16) - `s & 15`: Masks lower 4 bits to extract file (modulo 16) - Files map to 'a'-'h', ranks to '1'-'8'

Why This Foundation Matters

This section establishes:

1. **Efficient Board Access**: O(1) bounds checking without comparisons
2. **Clean Abstractions**: Type-safe square indices prevent bugs
3. **Standard Interfaces**: FEN/algebraic notation compatibility
4. **Memory Efficiency**: Despite using 128 slots, only 64 are active
5. **Performance**: Bit operations are faster than arithmetic comparisons

Practical Usage Patterns

When the engine evaluates moves:

```
// Example: Checking if a knight move is valid
let knight_offsets = [31, 33, 18, -14, -31, -33, -18, 14];
for offset in knight_offsets {
    let target = (current_square as i32 + offset) as Sq;
    if on_board(target) {  // Lightning-fast bounds check!
        // Process valid move
    }
}
```

Technical Notes for Rust Learners

Memory Layout

- The board array has 128 elements but only uses indices where `(i & 0x88) == 0`
- This wastes 64 array slots but gains massive performance in bounds checking
- Modern CPUs handle this sparse array well due to caching

Type Safety

- Using `type Sq = usize` creates a type alias, not a new type
- This provides semantic clarity without runtime overhead
- Consider using a newtype pattern (`struct Sq(usize)`) for stronger type safety in production code

Performance Characteristics

- Bounds checking: Single AND operation + comparison (2 CPU instructions)
- Traditional approach: 4 comparisons (rank >= 0, rank < 8, file >= 0, file < 8)
- This adds up over millions of move evaluations in search trees

Next Section Preview: The Board State section builds on this foundation to manage the complete game state, including castling rights, en passant, and move history.