Chess Engine Development Plan (C++ with Bitmasks)

Project Overview

Build a functional chess engine using bitboard representation for piece positions and move generation. This approach is efficient and scalable, allowing for fast move generation and evaluation.

Phase 1: Foundation & Board Representation

1.1 Bitboard Basics

- **Concept**: Use 64-bit integers (uint64_t) to represent piece positions on an 8x8 board
- **File & Rank System**:
  - Files: a-h (columns 0-7)
  - Ranks: 1-8 (rows 0-7)
  - Bit index: `file + rank * 8` (0-63)
- **Board State**: 12 bitboards total (6 piece types × 2 colors)

1.2 Core Data Structures

```
- Bitboard: typedef uint64_t Bitboard;
- Position: struct containing 12 bitboards (pawns, knights, bishops, rooks, queens, kings for both colors)
- Move: struct with from square, to square, move type (capture, castle, promotion, en passant)
- Color: enum {WHITE, BLACK};
- Piece: enum {PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING};
```

1.3 Utility Functions

- `setBit(board, square)`: Set bit at square
- `clearBit(board, square)`: Clear bit at square
- `getBit(board, square)`: Get bit at square
- `popCount(board)`: Count set bits
- `bitScanForward(board)`: Find lowest set bit
- `bitScanReverse(board)`: Find highest set bit
- Square/coordinate conversion functions
- Bitboard printing for debugging

Phase 2: Board Management

2.1 FEN Parsing

- Parse Forsyth-Edwards Notation to set up board positions
- Store castle rights, en passant target, halfmove clock, fullmove number

2.2 Move Representation & Storage

- Create Move data type (from, to, flags for special moves)
- Move list/vector container for legal moves

2.3 Move Making & Unmaking

- `makeMove()`: Apply move to position, update all bitboards
- `unmakeMove()`: Reverse move to previous position
- Store captured piece info for undo functionality

2.4 Board State Tracking

- Track whose turn it is
- Track castle rights for both sides
- Track en passant target square
- Track move history for repetition detection

Phase 3: Move Generation

3.1 Piece Attack/Move Bitboards

- **Sliding Pieces** (Bishops, Rooks, Queens):
  - Implement magic bitboards or occupancy-based move generation
  - Pre-compute movement rays
  - Account for piece blocking
- **Non-sliding Pieces** (Knights, Kings):
  - Pre-computed jump tables (knight moves from each square)
  - Pre-computed king move tables
- **Pawns**:
  - Direction depends on color
  - Attacks diagonally (capture)

- Moves forward one square (or two from starting position)
- En passant handling

3.2 Move Generation Strategy

- **Pseudo-legal move generation**: Generate all candidate moves without checking for checks
- **Legal move filtering**: Remove moves that leave the king in check

3.3 Move Generation Function

- `generateMoves(position, color, moves)`: Fill moves vector with all legal moves
- `generateCaptures(position, color, moves)`: Only capturing moves (for quiescence search)
- `isLegal(move, position)`: Check if a move is legal

3.4 Special Cases

- Castling: Check king/rook haven't moved, no pieces between, king not in check
- En passant: Pawn capture on specific en passant target square
- Pawn promotion: Generate 4 moves per pawn reaching last rank

---

Phase 4: Board Evaluation

4.1 Material Count

- Assign piece values: Pawn=1, Knight=3, Bishop=3.25, Rook=5, Queen=9, King=∞
- Quick evaluation based on captured material

4.2 Positional Factors

- Piece-square tables: Bonus/penalty based on piece position
- Center control
- Pawn structure (passed pawns, doubled pawns, isolated pawns)
- King safety (castling status, shelter)
- Piece mobility (number of squares a piece can move to)

4.3 Evaluation Function

- `evaluate(position)`: Return score from white's perspective
- Positive score = white advantage, negative = black advantage
- Comprehensive but computationally feasible

Phase 5: Search Algorithm

5.1 Minimax with Alpha-Beta Pruning

- Recursive search function exploring game tree
- `search(position, depth, alpha, beta)`: Returns best move evaluation
- Alpha-beta pruning to eliminate branches

5.2 Search Enhancements

- **Transposition Table**: Cache evaluated positions to avoid re-computation
  - Map: Hash(position) → (depth, score, flag)
  - Flag: exact, lower bound, upper bound
- **Move Ordering**: Search promising moves first to maximize pruning
  - Killer moves (moves that caused cutoff at sibling nodes)
  - History heuristic (frequently good moves)
  - MVV-LVA ordering (Most Valuable Victim - Least Valuable Attacker)
- **Iterative Deepening**: Search depth 1, then 2, then 3, etc.
  - Allows for time-based termination
  - Improves move ordering with deeper iterations

5.3 Quiescence Search

- Search capture sequences to stable positions
- Prevent horizon effect (missing tactical blows just beyond search depth)

5.4 Search Control

- `getBestMove(position, timeLimit)`: Find best move within time constraint
- Depth limiting or time-based cutoff
- Update transposition table during search

Phase 6: User Interface & I/O

6.1 Input Parsing

- Parse standard algebraic notation (e.g., e2e4, e1g1)
- Or parse long algebraic notation from user input

6.2 Game Loop

- Initialize board with starting position (or FEN)
- Display board state
- Get move (from user or engine)
- Validate & apply move
- Check for game end (checkmate, stalemate, draws)
- Repeat until game over

6.3 Engine Interface

- Simple console-based interaction (user vs engine)
- Optional: UCI (Universal Chess Interface) protocol support for compatibility with chess GUIs

6.4 Output/Display

- Pretty-print board (ASCII art or Unicode pieces)
- Display legal moves
- Show evaluation score
- Announce check/checkmate/stalemate

Phase 7: Testing & Optimization

7.1 Testing

- Perft testing: Verify move generation correctness
    - Count nodes at each depth, compare to known values
- Test special cases: castling, en passant, promotion
- Test checkmate/stalemate detection
- Verify evaluation symmetry

7.2 Profiling & Optimization

- Profile code to find bottlenecks
- Optimize bitboard operations (bit manipulation is critical)
- Optimize move generation (called most frequently)
- Consider SIMD operations if needed

7.3 Strength Evaluation

- Play against other engines or known positions
- Benchmark search speed (nodes per second)
- Evaluate elo rating if possible

Implementation Order (Recommended)

1. **Bitboard utilities** and basic board representation
2. **FEN parsing** and board display
3. **Pawn and knight move generation**
4. **Sliding piece move generation** (rooks, bishops, queens)
5. **Castling and en passant** handling
6. **Check detection** and legal move filtering
7. **Basic evaluation function**
8. **Minimax search** with alpha-beta pruning
9. **Transposition table**
10. **Move ordering heuristics**
11. **Iterative deepening** and time management
12. **Quiescence search**
13. **Console UI** and game loop
14. **Perft testing** and optimization
15. **Advanced features** (opening book, endgame tables, UCI protocol)

Key Performance Considerations

- **Move Generation**: Should be extremely fast (1-100M moves/sec)
- **Evaluation**: Balance speed vs accuracy
- **Memory**: Transposition table size affects strength and memory usage
- **Bit Manipulation**: Use compiler intrinsics for popcount, bit scans
- **Cache Efficiency**: Hot-path optimization for search function

---

Advanced Features (Optional)

- Opening book (pre-analyzed opening positions)

- Endgame tablebases (perfect play for endgames)
- Principal variation (PV) tracking
- Aspiration windows (narrow alpha-beta window)
- Null move pruning
- Late move reductions
- UCI protocol support

Resources & References

- "Bitboards" on chessprogramming.org
- "Move Generation" techniques and magic bitboards
- Alpha-beta pruning tutorials
- Transposition tables guide
- PERFT test suites and positions