ENGS110: Introduction to Programming Final Project

Chess Game Documentation

Lusine Khachatryan, Anna Grigoryan

May 16, 2026

Introduction

This project is a fully playable chess game developed as our final ENGS110: Introduction to Programming assignment. Its core concept is a human player competing against an AI opponent in a standard game of chess, complete with all official chess rules.

The application is split into two distinct layers that communicate with each other at runtime:

- **Backend (C)** — Handles all chess logic, including board state, move generation and validation, special rules, and the AI engine. It is compiled as a standalone executable.
- **Frontend (Python / Pygame)** — Provides the graphical user interface. It renders the board and pieces, captures mouse input, and drives the overall game loop.

The two layers communicate exclusively through standard input and output (`stdin/stdout`). The Python frontend launches the C engine as a subprocess and exchanges messages with it using a simple text-based ASCII protocol.

Application Functionality

When we started this project, the minimum requirement was a working chess game with a basic AI that picks any legal move. We ended up going quite a bit further than that.

The final application supports every official chess rule that matters in a real game, and the AI plays with genuine strategy rather than random moves. This section walks through everything the application can do, roughly in order from most obvious to most nuanced.

Setup Screen: Choosing Your Side and Difficulty

Before the game starts, the player is greeted with a simple setup screen (Figure 1). Here you choose two things:

- **Side:** Play as White or Black. When Black is chosen, the board automatically flips so your pieces are always at the bottom of the screen.
- **Difficulty:** Easy, Medium, or Hard, which maps to AI search depths of 2, 4, and 6 half-moves respectively.

Pressing R at any point during the game returns you to this screen without restarting the program.

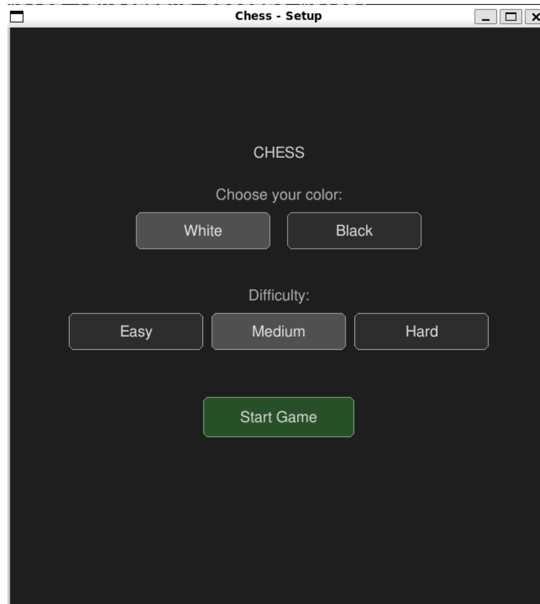

Figure 1: The setup screen where the player chooses their colour and difficulty.

Board Representation and Turn-Based Gameplay

The game follows standard chess rules: White moves first, then Black, alternating until the game ends. The board is an 8×8 grid, with squares labelled by algebraic notation (`a1`–`h8`). A status bar at the bottom of the window always shows whose turn it is and the current move number (Figure 2).

When a piece is clicked, all of its legal destination squares are highlighted in green, so the player always knows exactly where they can go. The last move played is also highlighted in a subtle yellow, which makes it easy to see what the AI just did.

Figure 2: The board after a few moves, with the status bar showing whose turn it is, and yellow highlights showing last move.

Move Input and Validation

The player interacts entirely with the mouse via three intuitive steps:

1. Click a piece to select it; its legal moves will appear as green dots.
2. Click a highlighted square to move there.
3. Click elsewhere, or click the same piece again, to deselect it.

Under the hood, every attempted move is sent to the C engine for validation before it is applied. The engine generates all legal moves for the selected piece and only accepts a move if it appears in that list. This architecture ensures it is impossible to make an illegal move through the GUI.

Capturing Pieces

Capturing works exactly as it does in standard chess: moving your piece onto a square occupied by an opponent's piece removes it from the board. The only captures that require special handling are *en passant* and captures that resolve a check state. All other captures are processed automatically by the normal move validation system.

Check Detection

Whenever a move is made, the engine checks whether the opponent's king is under attack. If it is, the game enters a **check** state: the status bar displays *"[Colour] is in CHECK"*, and the king's square is highlighted in red as a visual warning (Figure 3).

Figure 3: The king highlighted in red when in check.

A player in check *must* resolve it on their very next move, or else the engine will strictly reject any move that leaves the king in harm's way.

Checkmate and Stalemate

The game ends automatically when the engine detects a terminal position:

- **Checkmate:** The current player is in check and has no legal moves. A game-over overlay appears displaying the winner's name (Figure 4).
- **Stalemate:** The current player is *not* in check but has no legal moves. This results in an immediate draw, and the overlay text reads *"Stalemate! It's a draw"* (Figure 5).

Once the game concludes, the board state freezes, and the player can press R to clear the board and return to the setup menu.

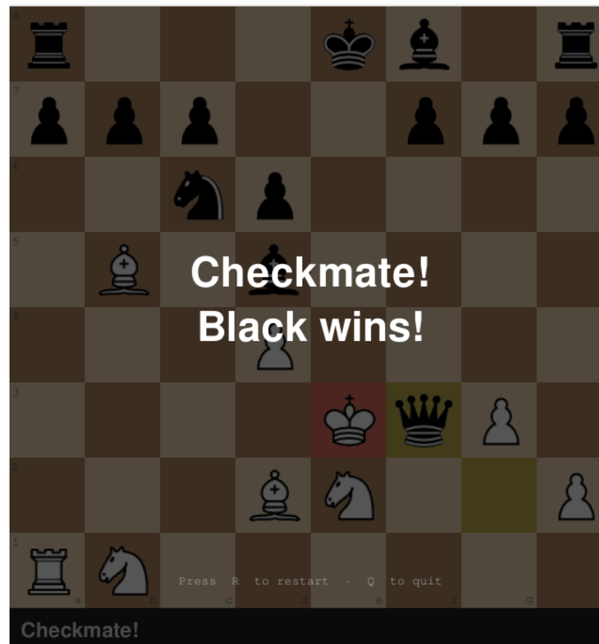

Figure 4: The checkmate overlay displayed when the game ends.

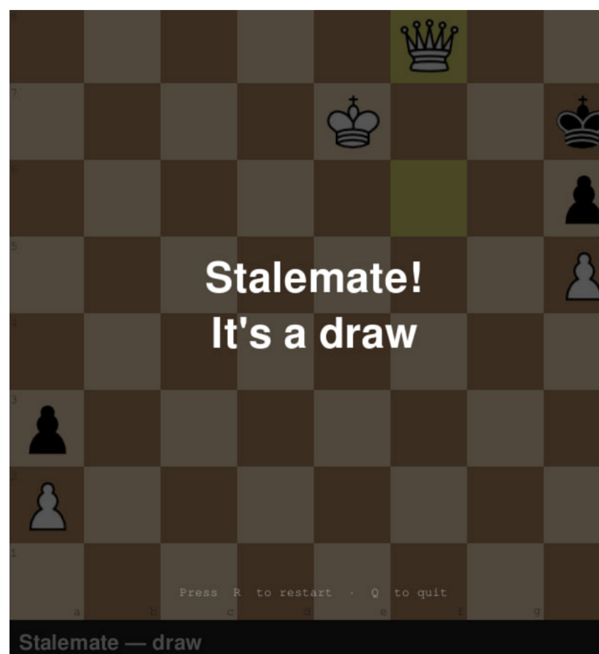

Figure 5: The stalement overlay displayed when the game ends.

Castling

Castling is fully supported for both kingside (short) and queenside (long). When the player moves their king two squares to either side, the engine automatically moves the corresponding rook to its correct destination square within the same animation sequence (Figure 6).

Castling rights are dynamically tracked within the board state and are permanently revoked as soon as the king or the respective rook moves, matching official chess regulations.

Figure 6: Kingside castling: the king moves from e1 to g1, and the rook jumps from h1 to f1.

En Passant

En passant is one of the trickier rules in chess: if a pawn moves two squares forward and lands directly beside an opponent's pawn, that opponent's pawn may capture it *as if it had only moved one square*. This capture is strictly legal **only** on the very next move.

The engine tracks the *en passant target square* dynamically. After a double pawn push, the skipped square is recorded; if the opponent chooses not to capture en passant immediately, that right expires right away (Figure 7).

Pawn Promotion

When a pawn reaches the opposite back rank, it must be promoted. The application pauses and displays a clean promotion dialog offering four choices: Queen, Rook, Bishop, or Knight (Figure 8). The game resumes only after a selection is made.

Both the human player and the AI handle promotion. The AI always opts for a Queen (statistically the strongest piece), while the human is free to choose any piece to fit their tactical scenario.

50-Move Draw Rule

To prevent games from dragging on indefinitely in deadlocked positions, the engine enforces the 50-move rule: if 50 consecutive full moves pass without a pawn move or a piece capture, the

Figure 7: En passant: the capturing pawn moves diagonally and the captured pawn disappears.

Figure 8: The promotion dialog that appears when a pawn reaches the back rank.

game is declared a draw. The internal halfmove clock resets to zero whenever a pawn moves or a piece is captured, and a draw triggers automatically as soon as the counter reaches 100 half-moves.

Figure 9: The 50 move draw dialog that appears after 50 move rule is hit.

AI Opponent: Minimax with Alpha-Beta Pruning

The AI engine serves as the centerpiece of this project, expanding far beyond our initial baseline goal of generating random legal moves. It uses the classic Minimax algorithm paired with Alpha-Beta pruning to build a structured game-tree search.

How Minimax Works

Minimax models a standard two-player zero-sum game layout: White tries to *maximise* the board evaluation score, while Black tries to *minimise* it. The algorithm builds a search tree out to a fixed depth, evaluating each leaf position with a static evaluation function (material balance combined with positional square bonuses), then propagates scores back up to determine the optimal path at the root level.

Alpha-Beta Pruning

A standard minimax search at a depth of 6 half-moves evaluates an unmanageable number of positions. Alpha-beta pruning makes this process highly efficient by maintaining two bounds during the tree traversal:

- α — the best score the *maximising* player (White) can already guarantee.
- β — the best score the *minimising* player (Black) can already guarantee.

Whenever $\alpha \geq \beta$, the current branch is abandoned immediately; neither player would ever choose to enter that sub-tree because a better guaranteed alternative already exists elsewhere. This optimization eliminates a massive portion of the evaluation tree without changing the final output, allowing the engine to search much deeper in the same amount of time.

For a comprehensive breakdown of this logic, refer to the GeeksforGeeks Alpha-Beta Pruning Guide. Figure 10 maps this pruning behavior on a simplified sample tree.

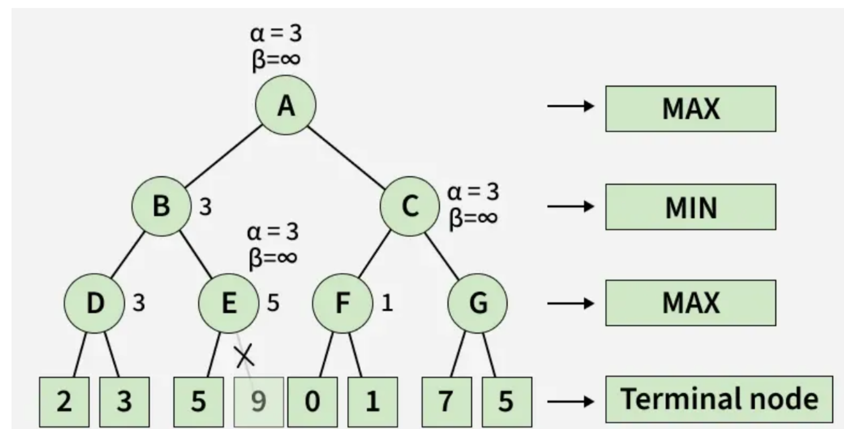

Figure 10: An alpha-beta pruning example. Greyed-out branches are never evaluated because the bounds α and β already render them tactically irrelevant.

Difficulty Levels

The search depth is configured on the initial setup screen according to the user's preference:

Difficulty Option	Search Depth (Half-Moves)
Easy	2
Medium	4
Hard	6

At depth 6, the AI demonstrates strong tactical awareness and routinely uncovers complex move combinations several turns ahead.

"AI is thinking..." Indicator

To keep the user interface smooth and responsive during intensive engine calculations, the AI processing runs on a dedicated background thread. The status bar displays an animated *"AI is thinking..."* prompt during this window, and the board interface locks to prevent any accidental input from interrupting the live search path (Figure 11).

Figure 11: The animated status bar while the AI is computing its move.

Communication

As already mentioned, our program uses a Python frontend and a C backend. However, it is crucial to highlight how these two communicate. They communicate using input/output streams. When C compiles the code, Python (`protocol.py`) uses `subprocess.Popen` to launch the C binary as a background process. Thus, they communicate via `stdin`/`stdout`, which are connected to each other.

For example, when Python wants the C engine to do something, it writes a string (like `"MOVE e2e4\n"`) straight to the C engine's `stdin` pipe and then waits to read the C `stdout` pipe. On the other hand, when the backend (C) is working, it runs an infinite `while(1)` loop inside `main()` (core `main.c` logic). It reads these commands using `fgets`, determines what to do, and prints the result back to `stdout` (`"MOVED e2e4"`).

In C, output is buffered. Therefore, after `main.c` prints the reply, it must call `fflush(stdout)` to force the text through the pipe immediately. This process happens continuously throughout the game, ensuring seamless communication between the backend and frontend.

Examples of main.c code:

```
// INIT
// Resets the board to the starting position and confirms it's ready.
static void handle_init(void) {
    board_init(&game);
    printf("OK\n");
    fflush(stdout); // Always flush stdout so the Python frontend sees the
    reply immediately!
}

// main - read-dispatch loop
// This is the entry point for the C engine. It runs in an infinite loop,
// waiting for text commands from the standard input (stdin) sent by the Python
    frontend.
int main(void) {
    board_init(&game);
```

```
14
15     char line[256];
16     // fgets blocks (waits) until a new line of text comes in
17     while (fgets(line, sizeof(line), stdin)) {
18         // Strip trailing newline / carriage-return so we just have the raw
    command
19         int len = (int)strlen(line);
20         while (len > 0 && (line[len-1] == '\n' || line[len-1] == '\r'))
21             line[--len] = '\0';
22
23         // Dispatch to the correct handler based on the first word.
24         // "line + 6" means skip the first 6 characters (e.g. "DEPTH " is 6
    chars)
25         // so we pass only the argument part to the handler function.
26         if      (strcmp(line, "INIT")         == 0) handle_init();
27         else if (strncmp(line, "DEPTH ",  6) == 0) handle_depth(line + 6);  //
    passes e.g. "4"
28         else if (strncmp(line, "MOVES ",  6) == 0) handle_moves(line + 6);  //
    passes e.g. "e2"
29         else if (strncmp(line, "MOVE ",   5) == 0) handle_move(line + 5);   //
    passes e.g. "e2e4"
30         else if (strcmp(line, "AI_MOVE")      == 0) handle_ai_move();
31         else if (strcmp(line, "STATUS")       == 0) handle_status();
32         else if (strcmp(line, "QUIT")         == 0) break; // Exit the loop and
    close the engine
33         else {
34             printf("ERROR unknown command\n");
35             fflush(stdout);
36         }
37     }
38
39     return 0;
40 }
```

Example of protocol.py code:

```
1  class ChessProtocol:
2      def __init__(self, binary_path: str):
3          # Check if the C backend has been compiled first
4          if not os.path.exists(binary_path):
5              raise FileNotFoundError(
6                  f"Chess binary not found at '{binary_path}'.\n"
7                  "Run:  cd backend && make"
8              )
9          # We start the C engine as a background process (subprocess).
10         # We use pipes so Python can talk to the C engine's standard input (
    stdin)
11         # and listen to its standard output (stdout).
12         self.proc = subprocess.Popen(
13             [binary_path],
14             stdin=subprocess.PIPE,
15             stdout=subprocess.PIPE,
16             stderr=subprocess.DEVNULL,
17             text=True,
18             bufsize=1,
19         )
20
21     def _send(self, command: str) -> str:
22         # Send a text command to the C engine and press "Enter" (\n)
23         self.proc.stdin.write(command + "\n")
24         self.proc.stdin.flush() # Make sure it sends immediately
25
26         # Wait for the engine to reply with a single line of text
```

```
        reply = self.proc.stdout.readline()
        return reply.strip()
```

Logic of board.c and board.h

First and foremost is the board representation. For an easier workflow, we represented our board using an 8×8 integer array, where the White pieces are positive numbers and Black pieces are negative. Then we defined a struct `Board` (using `typedef` to avoid writing the full `struct` keyword every time), which serves as a container for the state of the game. It holds the 8×8 array, whose turn it is, castling rights, the 50-move clock, and the en-passant target square.

```
// Board layout: board[rank][file]
//   rank 0 = rank 1 (white's back rank)
//   rank 7 = rank 8 (black's back rank)
//   file 0 = 'a', file 7 = 'h'
// Piece encoding: positive = white, negative = black
// e.g. WHITE QUEEN = 5, BLACK QUEEN = -5
typedef struct {
    int squares[8][8];

    int turn; // WHITE or BLACK
    // Castling rights - set to 1 (allowed) at the start, set to 0 if the king
    // or that rook has moved. Once lost, castling rights can't be recovered.
    int white_castle_k; // Can white still castle kingside (short)?
    int white_castle_q; // Can white still castle queenside (long)?
    int black_castle_k;
    int black_castle_q;

    // En-passant target square - if a pawn just did a double push, this stores
    // the square it skipped over (where an enemy pawn could capture it).
    // Set to -1 if there's no en-passant available this turn.
    int ep_rank;
    int ep_file;

    // Counts moves since the last capture or pawn move (used for 50-move draw
    rule)
    int halfmove_clock;
    // Counts the number of full moves (increments after Black moves)
    int fullmove_number;
} Board;
```

Evaluation (eval.c)

Moving forward, we need the AI to decide who is winning. For that, we write evaluation code, which assigns arbitrary point values to pieces and uses “Piece-Square Tables” (bonus points for putting pieces on good squares, like knights in the center).

Knights in the center reward demonstration code:

```
static const int PST_KNIGHT[8][8] = {
    {-50,-40,-30,-30,-30,-30,-40,-50 },
    {-40,-20,  0,  0,  0,  0,-20,-40 },
    {-30,  0, 10, 15, 15, 10,  0,-30 },
    {-30,  5, 15, 20, 20, 15,  5,-30 },
    {-30,  0, 15, 20, 20, 15,  0,-30 },
    {-30,  5, 10, 15, 15, 10,  5,-30 },
    {-40,-20,  0,  5,  5,  0,-20,-40 },
    {-50,-40,-30,-30,-30,-30,-40,-50 },
};
```

Who is winning?

```c
// Evaluates the current board position and returns a score.
// A positive score means White is winning, a negative score means Black is winning.
int evaluate(const Board *b) {
    int score = 0;

    for (int r = 0; r < 8; r++) {
        for (int f = 0; f < 8; f++) {
            int piece = b->squares[r][f];
            if (piece == EMPTY) continue; // Skip empty squares, nothing to evaluate here

            int color = board_color(piece);
            int type  = board_type(piece);

            // A piece's value is its base material value (e.g. 900 for Queen)
            // PLUS its positional value from the piece-square table (is it on a good square?)
            int val   = MAT[type] + pst_value(type, r, f, color);

            // White pieces add to score (White wants a higher positive score)
            // Black pieces subtract from score (Black wants a lower negative score)
            // By multiplying with color (1 for White, -1 for Black), we get the correct sign!
            score += color * val;
        }
    }

    return score;
}
```

The AI Logic (ai.c)

As already mentioned, the engine uses the Minimax algorithm with Alpha-beta Pruning. In `ai.c`, we make a copy of the board to test the moves before applying them to the real board. Then it recursively explores future moves up to a certain depth (e.g., 4 moves ahead). As explained, alpha-beta pruning stops evaluating moves that are obviously bad, which significantly speeds up the AI.

Example (maximizing player's turn):

```c
if (maximizing) {
    int best = -INF;
    for (int i = 0; i < n; i++) {
        Board tmp;
        // Make a copy of the board so we can test the move without changing the real board
        copy_board(&tmp, b);
        apply_move(&tmp, &moves[i]);

        // Recursively call minimax for the opponent's turn (they are trying to minimize)
        int score = minimax(&tmp, depth - 1, alpha, beta, 0);
        if (score > best)  best  = score;
        if (score > alpha) alpha = score; // Update alpha (the best score White can guarantee)

```

```
14         // If alpha is greater than or equal to beta, it means Black will never
     let us reach this position
15         // because they have a better alternative already. So we can stop
    checking the rest of the moves here.
16         if (alpha >= beta) break;   // beta cut-off
17     }
18     return best;
19 }
```

The Rules of the Chess Game (moves.c)

Here, all the moves of a standard chess game are defined. The main components are the `apply_move` and `generate_legal_moves` functions, while the other functions serve as helpers.

The `generate_legal_moves()` function acts like a filter. It first calls `generate_pseudo_moves()`, which generates every physically possible move based solely on piece movement rules. Then it loops through all the pseudo-moves, creating a temporary copy of the board for each one, playing the move, and checking if the king is placed in check. If the king remains safe, the move is considered legal and valid.

The `apply_move()` function is what modifies the actual board and updates the game state. It moves the piece from the `from` square to the `to` square, and whatever occupied the destination square is automatically overwritten (if captured). Next, the function checks if the moving piece was a pawn or if the `to` square contained an enemy piece. If either condition is true, it resets the `halfmove_clock` to 0; otherwise, it increments it by 1.

Then it updates special rules:

- **En Passant:** If a pawn moves diagonally into an empty destination square, the program identifies it as an en passant capture and deletes the enemy pawn located directly beside the moving pawn.
- **Castling:** If the king moves exactly 2 squares left or right, the program detects castling and shifts the correct rook to the opposite side of the king. Additionally, moving the king sets both of its castling rights (kingside and queenside) to 0 (false). Moving a rook sets the castling right for that specific side to 0.
- **Double Pawn Push:** If a pawn moves 2 squares forward, the program calculates the skipped square and stores it in the en-passant variables. For any other move type, these variables are reset to -1.

Finally, the `apply_move` function multiplies the turn variable by -1 to switch sides and increments the full-move counter.

Special Chess Rules Further Explained:

- **Check & Checkmate:** Check is determined by scanning the board to see if any enemy piece can attack the square where the King currently resides. Checkmate occurs when `generate_legal_moves` returns 0 available moves and the King is currently in check.
- **En Passant Tracking:** When a pawn moves forward 2 squares, the board struct records the skipped square in `ep_rank` and `ep_file`. On the very next turn, if an enemy pawn attacks that skipped square, the system allows the capture of the pawn that just moved.
- **Pawn Promotion:** When generating moves for a pawn reaching the 8th or 1st rank, the program generates 4 separate promotion cases for that square (queen, rook, knight, and bishop). When the move is applied, the pawn is replaced by the chosen piece value.

- **50-Move Draw Rule:** Every time a player makes a move, the `halfmove_clock` increments by 1. However, if a pawn moves or any piece is captured, it resets to 0. If this clock hits 100 (50 full moves), the game ends in a draw to prevent endless cycles.

Interface

Our user interface is written via Pygame. The crucial part here is converting the board layout to Pygame's rendering logic. First, it converts the 8×8 array representing the board into an 800×800 pixel window. Since Pygame draws starting from the top-left corner, whereas in chess rank 0 is located at the bottom, the renderer must mathematically invert the y-axis. Furthermore, every time Pygame updates the frame, the `draw()` function paints the screen from the bottom layer to the top layer. It also displays visual move hints using green and red dots.

Testing

Testing was something we took seriously from fairly early on in the project. Because the application is split into two completely separate programs (the C engine and the Python frontend), we ended up with two distinct test suites — one for each layer — plus a round of manual testing through the actual GUI once everything was wired together. This section walks through all three.

Testing Strategy

Our overall approach was to test each layer in isolation first, and then verify the full system manually.

- **Unit tests for the C engine** (`test_chess.c`): Test individual functions directly, with no Python or GUI involved.
- **Unit tests for the protocol layer** (`test_protocol.py`): Test the Python class that talks to the engine, using a *mock* subprocess so that the compiled binary is not even needed.
- **Manual testing**: Play the game and verify that every special rule works correctly through the graphical interface.

The reason we kept the test suites seperate is because if something breaks in the C engine, the Python tests still pass (and vice versa), which makes it much easier to pinpoint exactly where a bug is located.

Unit Tests: C Backend (`test_chess.c`)

The backend test suite is a C program that connects the same modules used in the real game (`board.c`, `moves.c`, `eval.c`, `ai.c`). It uses a minimal, hand-written assertion helper:

```
void assert_test(int cond, char msg[]) {
    tests_run++;
    if (cond) {
        tests_passed++;
    } else {
        printf("FAIL: %s\n", msg);
    }
}
```

No external testing framework was used. Just a counter, a condition, and a clear message. At the end, the program prints how many tests passed out of how many ran, and exits with a non-zero code if anything failed (which makes it easy to integrate cleanly with `make`).

We organised the tests into seven logical groups, described below.

1. Coordinate Helpers

The engine uses algebraic notation internally (`"e4"`, `"h8"`, etc.), so before testing any chess logic we verified that the conversion functions operate correctly.

Test	Input	Expected Output
`str_to_rank("a1")`	`"a1"`	0
`str_to_rank("a8")`	`"a8"`	7
`str_to_rank("e4")`	`"e4"`	3
`str_to_file("a1")`	`"a1"`	0
`str_to_file("h1")`	`"h1"`	7
`sq_to_str(0, 0)`	Rank 0, File 0	`"a1"`
`sq_to_str(7, 7)`	Rank 7, File 7	`"h8"`

2. Initial Board State

A newly initialised board should always be perfectly symmetric and legal:

- White has exactly 20 legal moves (16 pawn moves + 4 knight moves).
- Neither king is in check.
- The static evaluation returns 0.

3. Pawn Moves and En Passant

We tested that after executing `e2-e4`, the pawn lands safely on the correct square, `e2` is empty, and the en passant target square is explicitly set to `e3`. We then simulated playing `e7-e5`, `d2-d4` and confirmed that the resulting `exd4` capture is legal, verifying that the en passant expiry logic works correctly (the en passant is only available for one move).

4. Legal Move Generation

Scenario	Expected Result
Black to move after `1. e4`	20 legal moves
White knights at start	4 moves total (2 knights $\times$ 2 squares each)
Lone white king on `e1`, black king on `e8`	5 moves (`d1`, `d2`, `e2`, `f1`, `f2`)
Scholar's Mate reached	0 legal moves for Black

The Scholar's Mate (Mankakan Mat) test is particularly useful because it exercises the full execution pipeline: move generation, check detection, and the terminal-node condition all at once.

5. Apply Move

These tests verify that the `apply_move()` function correctly changes the board state:

- The turn alternates correctly from White to Black and back after each move.
- A pawn capture successfully places the capturing pawn on the destination square and clears the origin.

- Kingside castling correctly places the king on `g1`, the rook on `f1`, and safely clears `e1` and `h1`.

6. Attack and Check Detection

We set up isolated board positions and verified the `is_attacked()` and `is_in_check()` functions piece by piece.

Setup	Expected Behavior
White rook on `e4`	Attacks `e8` and `a4`; does NOT attack `d5`
White bishop on `e4`	Attacks `h7` and `b1`; does NOT attack `e5`
White knight on `e4`	Attacks `d6` and `f6`; does NOT attack `e5`
Black queen on `e8`, white king on `e1`	White king IS in check
Same position, white rook on `e4` blocking	White king is NOT in check

7. Evaluation Function

- Removing the white queen: Score is negative (Black is ahead on material).
- Removing the same rook from both sides: Score stays at 0 (board remains symmetric).
- Removing White's queen and both rooks: Score drops to a heavily negative value.
- Removing only Black's rook: Score becomes positive (White is ahead).

8. AI `find_best_move()`

These were the most satisfying tests to write. We verified three critical behaviors:

1. **It returns a move at all** — Coordinates are strictly valid (0–7), and the function returns 1.
2. **It takes a free queen** — We placed a white rook on `e5` and an undefended black queen on `d5`, set the search depth to 6, and confirmed the AI systematically picks `e5-d5`.
3. **It returns 0 on checkmate** — After reaching Scholar's Mate, calling `find_best_move()` for Black gracefully returns 0 (no move available) instead of crashing or returning garbage data.

9. Running the Backend Tests

To run the backend C tests, run this command in the backend folder:

```
make test_c
```

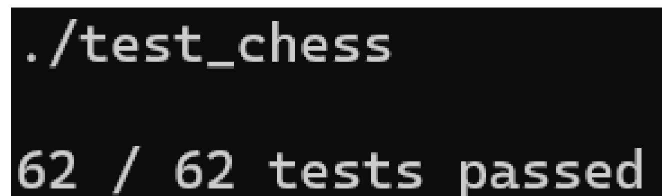

Figure 12: All backend unit tests passing successfully.

Unit Tests: Protocol (test_protocol.py)

The Python test suite targets ChessProtocol, which is the class responsible for launching the C engine as a subprocess and exchanging messages with it over standard input/output channels.

We used unittest.mock to replace the real engine subprocess with a fake one. This means the tests do not need the compiled binary at all, we simply tell the mock what to "reply" and verify that our code parses it correctly.

```
def _make_protocol(responses: list) -> ChessProtocol:
    mock_proc = MagicMock()
    mock_proc.stdout.readline.side_effect = [r + "\n" for r in responses]
    with patch("os.path.exists", return_value=True), \
         patch("subprocess.Popen", return_value=mock_proc):
        proto = ChessProtocol("/fake/chess")
    return proto
```

Every test calls this helper utility with a list of engine replies, then tests a specific method of the protocol layer. The suite is cleanly organised into one TestCase class per command.

Class	What is tested
TestInit	Missing binary raises FileNotFoundError; Popen is invoked with the exact path.
TestInitCommand	INIT → OK returns True; INIT → ERROR returns False; verifies "INIT\n" is written.
TestSetDepth	Depths 2, 4, 6 produce the matching "DEPTH n\n" command; non-OK replies return False.
TestGetMoves	Multi-square strings parsed to a list; empty replies return []; unexpected prefixes handle gracefully.
TestMakeMove	Legal moves return (True, reply); illegal moves return (False, "ILLEGAL"); promotion flags are forwarded verbatim.
TestAiMove	Normal moves parse to (from, to, None); promotions parse the trailing piece character; handles malformed strings without crashing.
TestStatus	All five states (playing, check, checkmate, stalemate, draw) parse cleanly; incomplete replies safely return (None, None).
TestQuit	"QUIT\n" is sent down the pipe; BrokenPipeError and OSError instances are caught and swallowed cleanly during shutdown.

Running the Protocol Tests

To run the protocol test, run this in the backend folder:

```
make test_py
```

Manual Testing

Once both automated test suites were passing, we played through the game manually to verify that all the rules worked correctly end-to-end.

```
Popen is called with the exact binary path supplied. ... ok
test_raises_when_binary_missing (test_protocol.TestInit)
FileNotFoundError if the binary path does not exist. ... ok
test_returns_false_on_error_reply (test_protocol.TestInitCommand) ... ok
test_returns_true_on_ok (test_protocol.TestInitCommand) ... ok
test_sends_init_command (test_protocol.TestInitCommand) ... ok
test_all_promotion_pieces_formatted (test_protocol.TestMakeMove)
Each promotion letter is forwarded verbatim to the engine. ... ok
test_illegal_move_returns_false (test_protocol.TestMakeMove) ... ok
test_no_promotion_suffix_when_none (test_protocol.TestMakeMove) ... ok
test_sends_correct_move_string (test_protocol.TestMakeMove) ... ok
test_success_returns_true_and_reply (test_protocol.TestMakeMove) ... ok
test_calls_terminate_on_process (test_protocol.TestQuit)
proc.terminate() is always called for cleanup. ... ok
test_sends_quit_command (test_protocol.TestQuit) ... ok
test_terminate_raises_is_silenced (test_protocol.TestQuit) ... ok
test_tolerates_broken_pipe (test_protocol.TestQuit)
quit() must not raise even if the process is already dead. ... ok
test_depth_1_easy (test_protocol.TestSetDepth) ... ok
test_depth_medium (test_protocol.TestSetDepth) ... ok
test_returns_false_on_non_ok (test_protocol.TestSetDepth) ... ok
test_returns_true_on_ok (test_protocol.TestSetDepth) ... ok
test_sends_correct_depth_value (test_protocol.TestSetDepth) ... ok
test_parses_black_turn (test_protocol.TestStatus) ... ok
test_parses_checkmate (test_protocol.TestStatus) ... ok
test_parses_playing_status (test_protocol.TestStatus) ... ok
test_parses_stalemate (test_protocol.TestStatus) ... ok
test_returns_none_on_malformed_reply (test_protocol.TestStatus) ... ok
test_returns_none_on_partial_reply (test_protocol.TestStatus)
STATUS with only 2 tokens (missing state) → (None, None). ... ok
test_sends_status_command (test_protocol.TestStatus) ... ok

----------------------------------------------------------------------
Ran 36 tests in 0.105s

OK
```

Figure 13: All Python protocol unit tests passing.

Feature	**How We Tested It**	**Expected Visual Result**	**Status**
Turn alternation	Played consecutive moves while observing the status bar.	Status flips between “White to move” and “Black to move”.	✓
Move highlights	Clicked each piece type across various open positions.	Green dots appear only on valid destination squares.	✓
Capturing	Moved an active piece onto an occupied enemy square.	Opponent piece disappears; capturing piece updates place.	✓
Check	Directed an active piece to directly attack the enemy king.	King square turns red; status bar reads “in CHECK”.	✓
Checkmate	Played a checkmate against the AI	Game-over overlay displays; board interactions freeze.	✓
Stalemate	Played so much that resulted in a stalement	Draw overlay displays on screen.	✓
Castling (Kingside)	Cleared `f1`/`g1`, shifted the king two squares rightward.	King lands on `g1`; rook automatically jumps to `f1`.	✓
Castling (Queenside)	Cleared lines, shifted the king two squares leftward.	King lands on `c1`; rook automatically jumps to `d1`.	✓
En Passant	Simulated a double-pawn push adjacent to an enemy pawn.	Captured pawn disappears from its row upon diagonal move.	✓
Promotion	Put an active pawn to the final rank.	Selection window appears; chosen asset swaps in.	✓
50-Move Draw	Played 50 full turns with no captures or pawn advancements.	Draw overlay triggers showing the rule confirmation.	✓
Board Flip	Initiated a match playing as Black.	Board flips; Black assets generate at the bottom.	✓
AI Difficulty	Tested performance across Easy, Medium, and Hard profiles.	Clear strength variance; Hard requires more search time.	✓
Restart Cycle	Pressed the `R` key midway through an active match.	UI returns to the main setup screen.	✓