**Assignment 5**

**Implement Minimax algorithm for game playing**

**Theory -**

The Minimax algorithm is a fundamental concept in artificial intelligence and game theory. It's a recursive decision-making algorithm used to determine the optimal move for a player in a two-player, zero-sum game—like Tic-Tac-Toe, Checkers, or Chess. A zero-sum game is one where one player's gain is exactly equal to the other player's loss.

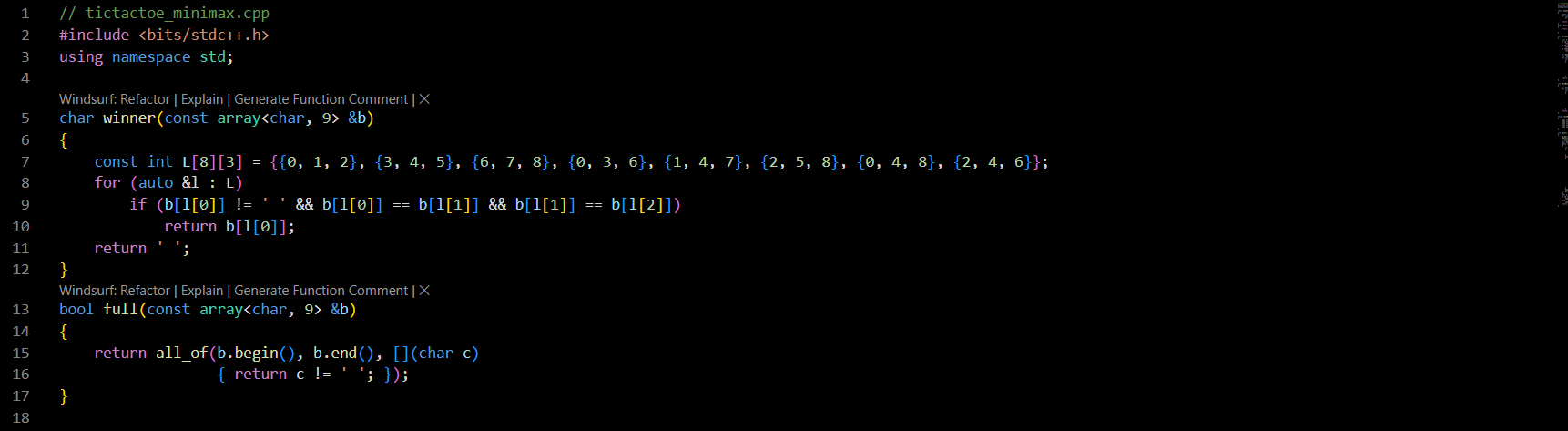
The core idea is to explore a game tree of all possible moves from the current state. The algorithm assumes that both players will play optimally. One player is the Maximizer, who tries to achieve the highest possible score, while the other is the Minimizer, who tries to achieve the lowest possible score.

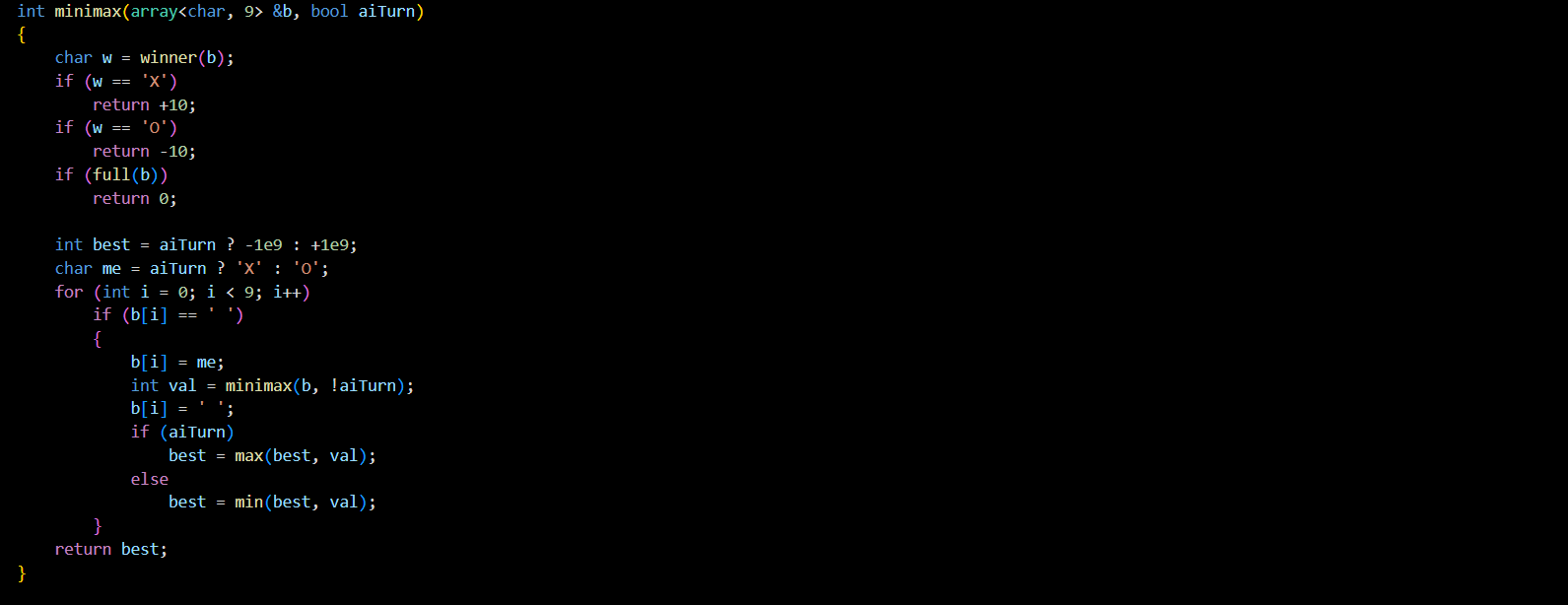
The process works as follows:

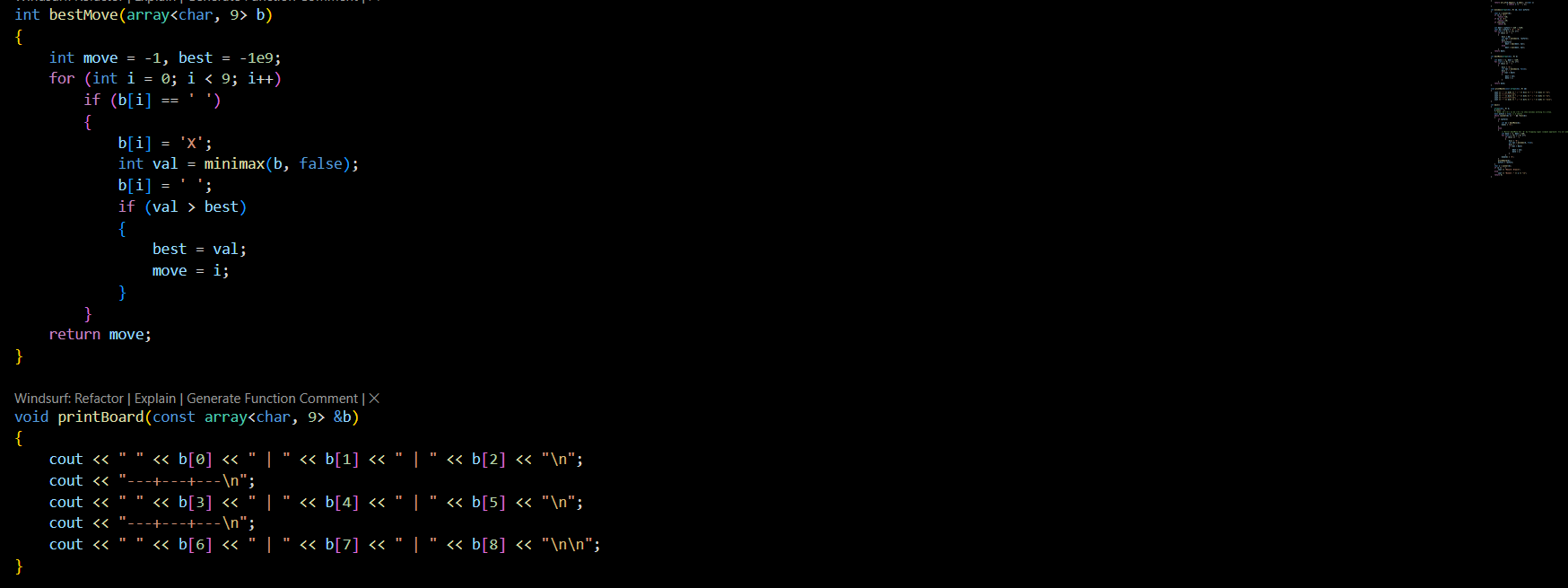
1. Generate the Tree: Starting from the current game state, the algorithm generates all possible future states until it reaches a terminal state (a win, loss, or draw).
2. Assign Scores: Each terminal state is given a score from the Maximizer's perspective. For example:
   * Maximizer Win: +10
   * Minimizer Win: -10
   * Draw: 0
3. Propagate Scores Upward: The algorithm then works its way back up the tree. At each level:
   * The Maximizer will choose the move that leads to the node with the highest score.
   * The Minimizer will choose the move that leads to the node with the lowest score.

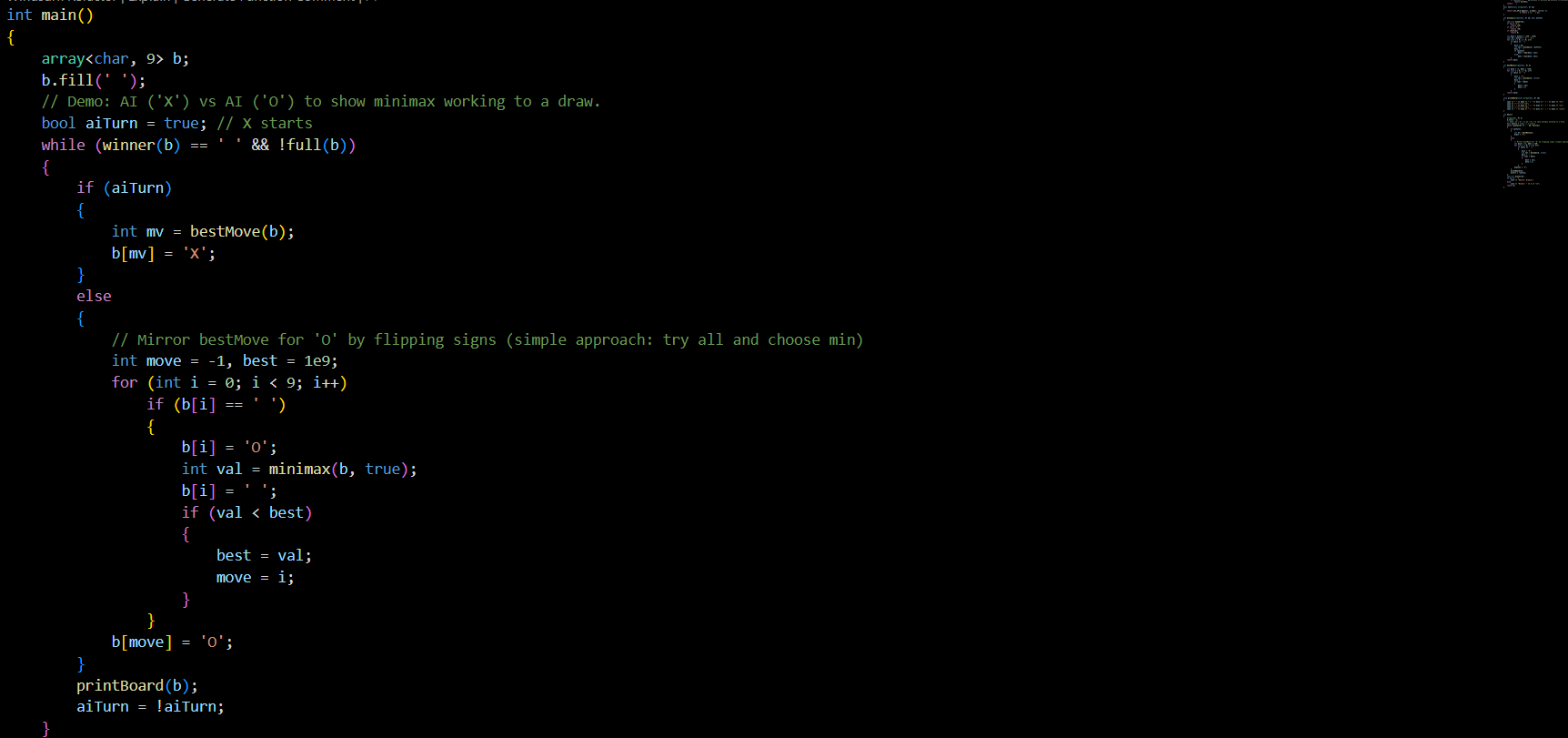
By doing this, the algorithm can determine the best possible move from the current position, assuming the opponent will also play perfectly to counter it.

**Code –**

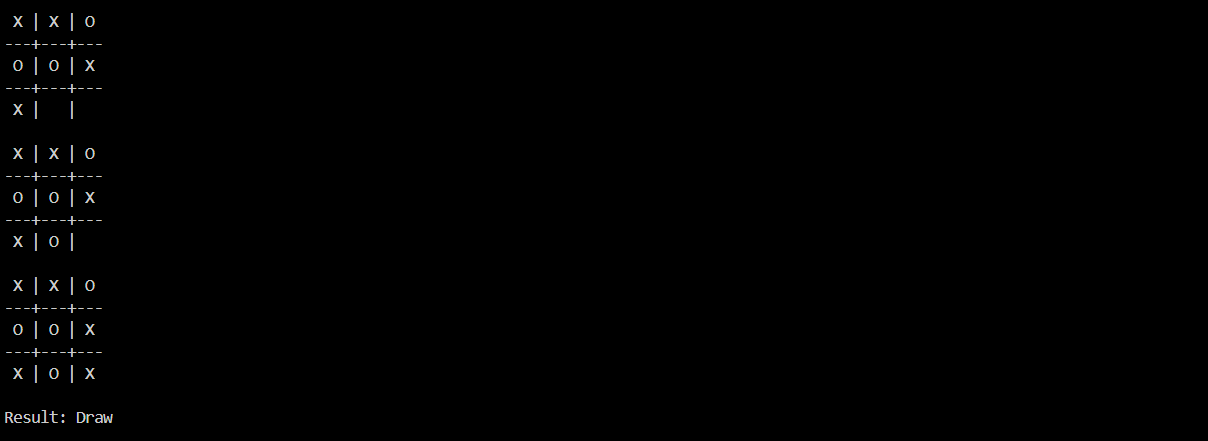
****

****

****

****

**Output –**

****

**Code Explanation -**

This C++ code implements the Minimax algorithm to create an Tic-Tac-Toe AI.

**Helper Functions**

Before the main algorithm, there are some utility functions:

* A function to check for a winner. It iterates through all 8 possible winning lines (3 rows, 3 columns, 2 diagonals) to see if any player has three in a row.
* A function to check if the board is full. This determines if the game has ended in a draw.

**The minimax Function**

This is the recursive heart of the AI. Its goal isn't to find a move but to return the best *score* achievable from a given board state.

* Base Cases: The function first checks if the game is over. If the maximizer ('X') has won, it returns a high score (+10). If the minimizer ('O') has won, it returns a low score (-10). If it's a draw, it returns 0. These are the terminal states.
* Recursive Step: If the game isn't over, the function simulates moves for the current player.
  + If it's the Maximizer's turn (the AI, 'X'), it tries placing an 'X' in every empty square. For each trial, it recursively calls itself to find out the score the Minimizer would achieve from that new board. The Maximizer then chooses the move that results in the highest score.
  + If it's the Minimizer's turn (the opponent, 'O'), it does the same but chooses the move that results in the lowest score.
* Backtracking: After recursively calling itself for a move, the function immediately undoes that move by setting the square back to empty. This is crucial for exploring all branches of the game tree independently.

**The bestMove Function**

While the minimax function finds the best *score*, this function finds the best *move*. It's used by the AI ('X') to decide where to play. It works by iterating through all empty squares on the board. For each empty square, it:

1. Temporarily places its piece ('X') on that square.
2. Calls the minimax function to evaluate the score of the board that results from this move.
3. Undoes the move.
4. It keeps track of the move that leads to the highest score and returns the index of that move as its final decision.

**The main Function**

The main function sets up an empty board and runs a game loop. In this demonstration, it pits two AI players against each other: one playing as 'X' (the Maximizer) and one as 'O' (the Minimizer).

* The 'X' player uses the bestMove function to pick its move.
* The 'O' player has similar logic but is hardcoded to find the move that results in the lowest possible score for 'X'.
* The game continues until there's a winner or a draw. Since both players are using an optimal strategy, the game will always end in a draw, perfectly demonstrating that the Minimax algorithm works.

**Conclusion -**

This program is a perfect illustration of the Minimax algorithm applied to a simple, perfect-information game. It shows how a machine can "look ahead" at all possible outcomes to make an optimal, unbeatable decision. The key takeaway is the recursive, depth-first exploration of the game's future states, with each player trying to optimize their outcome.

While incredibly effective for games like Tic-Tac-Toe, the standard Minimax algorithm becomes computationally expensive for more complex games like Chess, as the number of possible game states grows exponentially. This limitation leads to more advanced techniques like Alpha-Beta Pruning, which optimizes Minimax by ignoring branches of the game tree that don't need to be explored.