1 evaluate(int*board)

What it does

This function basically evaluates the current state of the game by checking if the game is a tie, or the ai won, or the the user won and based on that it returns a value. The value for a win is 1, a tie is 0 and a lose is -1.

```
int evaluate(int* board) {
        // checking row-wise
        for(int i = 0; i < 9; i += 3) {
                if(board[i] == board[i + 1] && board[i + 1] == board[i + 2]) {
                        if(board[i] == ai)
                                return score.win;
                        if(board[i] == human)
                                return score.lose;
                }
        }
```

Figure 1: Checking row-wise

Here, it's checking if three columns of any row have the same symbol. If this condition evaluates to true, then if checks if the symbol is that of the human or ai, if it's of **ai** then **score.win**, which is 1, is returned otherwise **score.lose**, which is -1, is returned.

```
  // checking column-wise
  for(int i = 0; i < 3; i++) {
          if(board[i] == board[i + 3] && board[i + 3] == board[i + 6]) {
                  if(board[i] == ai)
                          return score.win;
                  if(board[i] == human)
                          return score.lose;
          }
  }
```

Figure 2: Checking column-wise

Here, it's checking if three rows of any column have the same symbol. If this condition.....(I guess you know the rest).

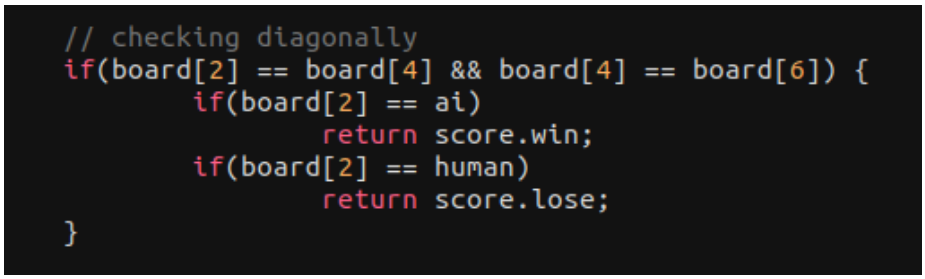

Figure 3: Checking the diagonal that connects the top-right corner to the bottom left corner(of the board)

Caption is self-explanatory. The rest is just like the above two.

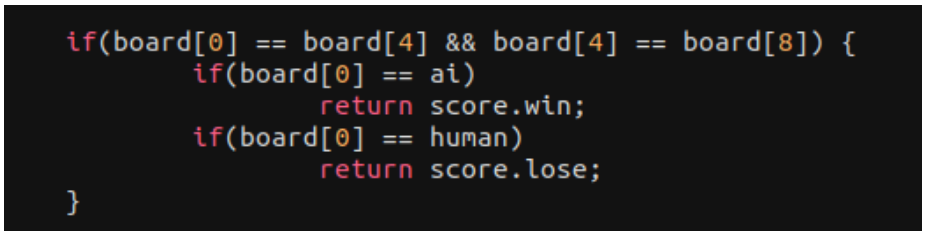

Figure 4: Checking the other diagonal

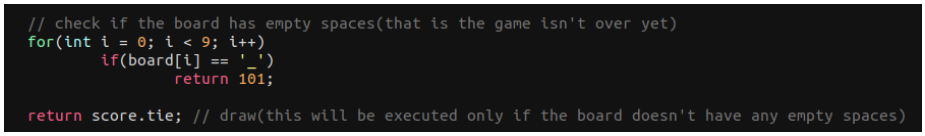

Figure 5: This snippet checks if the game is over or not

If the game is over, then the board won't have any empty spaces. If the game is not over, then the board will have empty spaces. **And this snippet will be executed iff none of the above snippets are executed. It will return *score.tie* which is** 0**(meaning that the game is over because the board is out of empty spaces) otherwise will return** 101**(again this could be any number other than** 0**,** -1 **and** 1**).**

2 bestPos(int*board)

What it does

This function works for the ai. It basically uses the **evaluate()** method to first check if the game whether the game is over(because there is no point in determining the best position if the game is over).

```
void bestPos(int*board) {
        // before determining the best position check if the game is over
        int result = evaluate(board);
        if(result != 101)
                return;
```

Figure 6: Checking if the game if over or not

```
int move;
int bestScore = -INF;
int score;
```

Figure 7: Variables

You may/maynot include this part(these points) in the slide. your wish.

- **move** is the best move that the ai will make.
- **bestScore** is the best score of a position(it's initially initialized to $-\infty$).
- **score** is the score of the position where ai will place its symbol.

Coming to the actual code section:

```
for(int i = 0; i < 9; i++) { // to find the best move, we need to first check if the board has empty space in it
        if(board[i] == '_') { // checking for an empty space
                board[i] = ai; // temporarily place AI's symbol in that empty space
                score = minimax(board, 0, false); // we are the maximizing player
                board[i] = '_'; // restoring the position
                if(score > bestScore) { // set the bestScore to the score that is greater than bestScore
                        bestScore = score;
                        move = i; // if we get a new best score then the position where we got that new best score is the best position
                }
        }
}

board[move] = ai;
```

Figure 8: bestPos's main section

Here, bestPos(), runs a loop(for the whole board) and **temporarily** places **ai's** symbol on the board and performs **minimax** on the board by calling **minimax()** for the current state of the board. **minimax()** returns a certain score for this state of the board and it gets recorded into the **score** variable. Next, that **temporary** ai symbol is removed(notice **board[i] = '_'**).
After this, **score** is compared with **bestScore**. Since **score** > **bestScore** evaluates to true, so **score** gets recorded in **bestScore** and the position for which we got a new best score, which is **i**, also gets recorded into the **move** variable.

And the above paragraph is repeated for the entire board, until we get a best score(which will not have any other score greater than it) and its corresponding position recorded in **move**.

After the loop terminates, this recorded **move**, which is the best move because it has a corresponding **bestScore**, is the new position where ai should place its symbol.

3 minimax(int*board, int depth, bool isMax)

What it does

It starts with **minimax(board, 0, false)** because, we are the **Max** player and ai is the **Min** player(always trying to minimize our winning chances).

Checking if the game if over

```
// checking if the game is over
int result = evaluate(board);
if(result != 101) { // base case
        if(result == score.win)
                return score.win;
        if(result == score.tie)
                return score.tie;
        if(result == score.lose)
                return score.lose;
}
```

Figure 9: Checking the state of the game

```
// Min's turn(ai's turn)
else {
        int bestScore = INF;
        int score;
        // check for an empty space in the board
        for(int i = 0; i < 9; i++) {
                if(board[i] == '_') {
                        board[i] = human; // this is how I understand this line, ai is assigning this position to us
                        score = minimax(board, depth + 1, true);
                        board[i] = '_'; // restoring this board position

                        if(score < bestScore)
                                bestScore = score; // min(score, bestScore);
                }
        }

        best_score = bestScore;
}
```

Figure 10: Min's turn

Since, we started off with **minimax(board, 0, false)** meaning it's **not Max's turn** that is it's **Min's** turn, let us see **Min's** turn.

Again, the same thing that we did in **bestPos()**, here too we run a loop for the whole board, but here we **temporarily place** the **human's symbol**, because **minimax** is the sidekick of the ai and it will always try to minimize the chance of human triumph. Now, after termporarily placing human's symbol on the board, it calls **minimax(board, depth + 1, true)** meaning it's Max's(that is ai's) turn now.

```
int best_score;

// Max's turn
if(isMax) {
        int bestScore = -INF;
        int score;
        // check for an empty space in the board
        for(int i = 0; i < 9; i++) { // this part is like recursive DFS
                if(board[i] == '_') {
                        board[i] = ai;
                        score = minimax(board, depth + 1, false);
                        board[i] = '_';

                        if(score > bestScore)
                                bestScore = score; // max(score, bestScore);
                }
        }

        best_score = bestScore;
}
```

Figure 11: Max's turn(**Note**: this code snippet comes before Min's snippet)

Now, this snippet is exactly like that the snippet in **bestPos()**, here we again temporarily place ai's symbol on the board and then call the **minimax()** function as **minimax(board, depth+1, false)**.

Again, this turn taking goes on until the snippet in this image evaluates to true(that is, this is the base case of this recursive function):

```
// checking if the game is over
int result = evaluate(board);
if(result != 101) { // base case
        if(result == score.win)
                return score.win;
        if(result == score.tie)
                return score.tie;
        if(result == score.lose)
                return score.lose;
}
```

Figure 12: Checking the state of the game

The following image explains a how this function works for a game of tic-tac-toe which has just progressed mid-way.

I am sharing it with this pdf.

In summary, this is what it actually looks like:

Once, the human player places his/her symbol on the board, the ai plays the game with itself(as an ai first, and then as a human, that's why it places the symbols **temporarily** because they are removed once minimax is performed) and decides which move would be best to reduce the chances of winning for the human.