#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define COMPUTER 1

#define HUMAN 2

#define SIDE 3

#define COMPUTERMOVE 'O'

#define HUMANMOVE 'X'

// ---------------- Intelligent Moves start

struct Move {

int row, col;

};

char player = 'x', opponent = 'o';

// This function returns true if there are moves

// remaining on the board. It returns false if

// there are no moves left to play.

bool isMovesLeft(char board[3][3])

{

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

if (board[i][j] == '\_')

return true;

return false;

}

// This is the evaluation function

int evaluate(char b[3][3])

{

// Checking for Rows for X or O victory.

for (int row = 0; row < 3; row++) {

if (b[row][0] == b[row][1]

&& b[row][1] == b[row][2]) {

if (b[row][0] == player)

return +10;

else if (b[row][0] == opponent)

return -10;

}

}

// Checking for Columns for X or O victory.

for (int col = 0; col < 3; col++) {

if (b[0][col] == b[1][col]

&& b[1][col] == b[2][col]) {

if (b[0][col] == player)

return +10;

else if (b[0][col] == opponent)

return -10;

}

}

// Checking for Diagonals for X or O victory.

if (b[0][0] == b[1][1] && b[1][1] == b[2][2]) {

if (b[0][0] == player)

return +10;

else if (b[0][0] == opponent)

return -10;

}

if (b[0][2] == b[1][1] && b[1][1] == b[2][0]) {

if (b[0][2] == player)

return +10;

else if (b[0][2] == opponent)

return -10;

}

// Else if none of them have won then return 0

return 0;

}

// This is the minimax function. It considers all

// the possible ways the game can go and returns

// the value of the board

int minimax(char board[3][3], int depth, bool isMax)

{

int score = evaluate(board);

// If Maximizer has won the game return his/her

// evaluated score

if (score == 10)

return score;

// If Minimizer has won the game return his/her

// evaluated score

if (score == -10)

return score;

// If there are no more moves and no winner then

// it is a tie

if (isMovesLeft(board) == false)

return 0;

// If this maximizer's move

if (isMax) {

int best = -1000;

// Traverse all cells

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

// Check if cell is empty

if (board[i][j] == '\_') {

// Make the move

board[i][j] = player;

int val

= minimax(board, depth + 1, !isMax);

if (val > best) {

best = val;

}

// Undo the move

board[i][j] = '\_';

}

}

}

return best;

}

// If this minimizer's move

else {

int best = 1000;

// Traverse all cells

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

// Check if cell is empty

if (board[i][j] == '\_') {

// Make the move

board[i][j] = opponent;

// Call minimax recursively and choose

int val

= minimax(board, depth + 1, !isMax);

if (val < best) {

best = val;

}

// Undo the move

board[i][j] = '\_';

}

}

}

return best;

}

}

// This will return the best possible move for the player

struct Move findBestMove(char board[3][3])

{

int bestVal = -1000;

struct Move bestMove;

bestMove.row = -1;

bestMove.col = -1;

// Traverse all cells, evaluate minimax function for

// all empty cells. And return the cell with optimal

// value.

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

// Check if cell is empty

if (board[i][j] == '\_') {

// Make the move

board[i][j] = player;

// compute evaluation function for this

// move.

int moveVal = minimax(board, 0, false);

// Undo the move

board[i][j] = '\_';

// If the value of the current move is

// more than the best value, then update

// best/

if (moveVal > bestVal) {

bestMove.row = i;

bestMove.col = j;

bestVal = moveVal;

}

}

}

}

// printf("The value of the best Move is : %d\n\n",

// bestVal);

return bestMove;

}

// -----------------------------------Intelligent Moves end

// Function to display the game board

void showBoard(char board[][SIDE])

{

printf("\n\n");

printf("\t\t\t %c | %c | %c \n", board[0][0],

board[0][1], board[0][2]);

printf("\t\t\t--------------\n");

printf("\t\t\t %c | %c | %c \n", board[1][0],

board[1][1], board[1][2]);

printf("\t\t\t--------------\n");

printf("\t\t\t %c | %c | %c \n\n", board[2][0],

board[2][1], board[2][2]);

}

// Function to show the instructions

void showInstructions()

{

printf("\t\t\t Tic-Tac-Toe\n\n");

printf("Choose a cell numbered from 1 to 9 as below "

"and play\n\n");

printf("\t\t\t 1 | 2 | 3 \n");

printf("\t\t\t--------------\n");

printf("\t\t\t 4 | 5 | 6 \n");

printf("\t\t\t--------------\n");

printf("\t\t\t 7 | 8 | 9 \n\n");

printf("-\t-\t-\t-\t-\t-\t-\t-\t-\t-\n\n");

}

// Function to initialise the game

void initialise(char board[][SIDE], int moves[])

{

srand(time(NULL));

// Initially, the board is empty

for (int i = 0; i < SIDE; i++) {

for (int j = 0; j < SIDE; j++)

board[i][j] = ' ';

}

// Fill the moves with numbers

for (int i = 0; i < SIDE \* SIDE; i++)

moves[i] = i;

// Randomize the moves

for (int i = 0; i < SIDE \* SIDE; i++) {

int randIndex = rand() % (SIDE \* SIDE);

int temp = moves[i];

moves[i] = moves[randIndex];

moves[randIndex] = temp;

}

}

// Function to declare the winner of the game

void declareWinner(int whoseTurn)

{

if (whoseTurn == COMPUTER)

printf("COMPUTER has won\n");

else

printf("HUMAN has won\n");

}

// Function to check if any row is crossed with the same

// player's move

int rowCrossed(char board[][SIDE])

{

for (int i = 0; i < SIDE; i++) {

if (board[i][0] == board[i][1]

&& board[i][1] == board[i][2]

&& board[i][0] != ' ')

return 1;

}

return 0;

}

// Function to check if any column is crossed with the same

// player's move

int columnCrossed(char board[][SIDE])

{

for (int i = 0; i < SIDE; i++) {

if (board[0][i] == board[1][i]

&& board[1][i] == board[2][i]

&& board[0][i] != ' ')

return 1;

}

return 0;

}

// Function to check if any diagonal is crossed with the

// same player's move

int diagonalCrossed(char board[][SIDE])

{

if ((board[0][0] == board[1][1]

&& board[1][1] == board[2][2]

&& board[0][0] != ' ')

|| (board[0][2] == board[1][1]

&& board[1][1] == board[2][0]

&& board[0][2] != ' '))

return 1;

return 0;

}

// Function to check if the game is over

int gameOver(char board[][SIDE])

{

return (rowCrossed(board) || columnCrossed(board)

|| diagonalCrossed(board));

}

// Function to play Tic-Tac-Toe

void playTicTacToe(int whoseTurn)

{

// A 3\*3 Tic-Tac-Toe board for playing

char board[SIDE][SIDE];

int moves[SIDE \* SIDE];

// Initialise the game

initialise(board, moves);

// Show the instructions before playing

showInstructions();

int moveIndex = 0, x, y;

// Keep playing until the game is over or it is a draw

while (!gameOver(board) && moveIndex != SIDE \* SIDE) {

if (whoseTurn == COMPUTER) {

char tempBoard[3][3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == 'X') {

tempBoard[i][j] = 'x';

}

else if (board[i][j] == 'O') {

tempBoard[i][j] = 'o';

}

else {

tempBoard[i][j] = '\_';

}

}

}

struct Move thisMove = findBestMove(tempBoard);

x = thisMove.row;

y = thisMove.col;

board[x][y] = COMPUTERMOVE;

printf("COMPUTER has put a %c in cell %d %d\n",

COMPUTERMOVE, x, y);

showBoard(board);

moveIndex++;

whoseTurn = HUMAN;

}

else if (whoseTurn == HUMAN) {

int move;

printf("Enter your move (1-9): ");

scanf("%d", &move);

if (move < 1 || move > 9) {

printf("Invalid input! Please enter a "

"number between 1 and 9.\n");

continue;

}

x = (move - 1) / SIDE;

y = (move - 1) % SIDE;

if (board[x][y] == ' ') {

board[x][y] = HUMANMOVE;

showBoard(board);

moveIndex++;

if (gameOver(board)) {

declareWinner(HUMAN);

return;

}

whoseTurn = COMPUTER;

}

else {

printf("Cell %d is already occupied. Try "

"again.\n",

move);

}

}

}

// If the game has drawn

if (!gameOver(board) && moveIndex == SIDE \* SIDE)

printf("It's a draw\n");

else {

// Toggling the user to declare the actual winner

if (whoseTurn == COMPUTER)

whoseTurn = HUMAN;

else if (whoseTurn == HUMAN)

whoseTurn = COMPUTER;

// Declare the winner

declareWinner(whoseTurn);

}

}

// Driver program

int main()

{

// Let us play the game with COMPUTER starting first

playTicTacToe(COMPUTER);

return 0;

}