```c
/**
 * tictactoe.c
 *
 * Implements Tic Tac Toe (generalized to a board of size d x d).
 *
 * Usage: tictactoe
 *        tictactoe d
 *
 * if d is not specified, we default to a typical 3x3 board.
 *
 * whereby the board's dimensions are to be d x d,
 * where d must be in [DIM_MIN,DIM_MAX]
 *
 * Some of the input/output functions are generalized from fifteen.c (PS3)
 *
 * To make our comparisons easy,
 *      Player 1 is "O" and is the machine player
 *      Player -1 is "X" and is the human player
 */

#define MACHINE_PLAYER 1
#define MAX_DEPTH 5

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <ctype.h>

// constants
#define DIM_MIN 3
#define DIM_MAX 9

// board
int board[DIM_MAX][DIM_MAX];

// dimensions
int d;

// machine player best move
int best_row;
int best_col;

// prototypes
void init();
void clear();
```

```
49. void draw();
50. int check_for_winner(int b[DIM_MAX][DIM_MAX]);
51. int parse_command_line(int argc, char *argv[]);
52. int read_player_input(int *r, int *c, int player);
53. int minimax(int hypothetical_board[DIM_MAX][DIM_MAX], int player, bool mymove, int depth);
54.
55. int main(int argc, char *argv[])
56. {
57.     int current_player=-1; // X goes first
58.     int winner, row, col, score;
59.     int total_moves = 0;
60.
61.     // parse the command line
62.     if (parse_command_line(argc, argv) != 0)
63.         return 1;
64.
65.     // initialize the board
66.     init();
67.
68.     // accept moves until game is won
69.     while (true)
70.     {
71.         // clear the screen
72.         clear();
73.
74.         // draw the current state of the board
75.         draw();
76.
77.         // check for winners
78.         winner = check_for_winner(board);
79.         if (winner != 0)
80.             {
81.                 printf("VICTORY FOR %s!!!!\n\n",  winner==-1 ?"X":"O");
82.                 break;
83.             }
84.
85.         // check for a tie
86.         if (total_moves == (d*d))
87.             {
88.                 printf("GAME ENDS IN A TIE!!!!\n\n");
89.                 break;
90.             }
91.
92.         if(current_player == MACHINE_PLAYER)
93.         {
94.             //select the machine player's move
95.             score = minimax(board, current_player, true, 0);
96.
```

```
            if (score != -1000)
            {
                row=best_row;
                col=best_col;
            }
        }
        else
        {
            // read in the human player's move
            if (read_player_input(&row, &col, current_player) != 0)
                {
                    continue;
                }
        }

        // check boundaries
        if ((row>=0 && row<d) && (col>=0 && col<d))
        {
            // check board is empty
            if (board[row][col] == 0)
            {
                // put a piece on the board
                board[row][col] = current_player;

                // update the current_player
                current_player=current_player*-1;

                // increment move counter
                total_moves++;
            }
        }
    }
}


int minimax(int hypothetical_board[DIM_MAX][DIM_MAX], int player, bool mymove, int depth)
{
    int i, j, score;

    // if we have gone too deep, return;
    if (depth > MAX_DEPTH)
        return 0;

    // see if someone has won
    int winner = check_for_winner(hypothetical_board);
    if (winner != 0)
    {
```

```
145.         return winner;
146.     }
147.
148.     int move_row = -1;
149.     int move_col = -1;
150.     if (mymove)
151.         score = -2; //Losing moves are preferred to no move
152.     else
153.         score = 2;
154.
155.     // For all possible locations (moves),
156.     for(i=0; i<d; i++)
157.     {
158.         for(j=0; j<d; j++)
159.         {
160.         if(hypothetical_board[i][j] == 0)
161.             {
162.                 // If this is a legal move,
163.                 hypothetical_board[i][j] = player; //Try the move
164.                 int thisScore = minimax(hypothetical_board, -1*player, !mymove, depth+1);
165.
166.                 if (mymove)
167.                 {
168.                     // my move, so maximize the score
169.                     if(thisScore > score) {
170.                         score = thisScore;
171.                         move_row = i;
172.                         move_col = j;
173.                     }
174.                 }
175.                 else
176.                 {
177.                     // not my move, so minimize the score
178.                     if(thisScore < score) {
179.                         score = thisScore;
180.                         move_row = i;
181.                         move_col = j;
182.                     }
183.                 }
184.                 hypothetical_board[i][j] = 0;//Reset board after try
185.             }
186.         }
187.     }
188.     if(move_row == -1) return 0;  // no valid moves, so it is a tie.
189.     best_row = move_row;
190.     best_col = move_col;
191.     return score;
192. }
```

```
193.
194.
195.
196. /**
197.  * read in a line from standard input, parsing as a grid location
198.  */
199.
200. int read_player_input(int *r, int *c, int player)
201. {
202.     // prompt for move
203.     printf("Enter move for player %s (for example, b2):", player==-1 ?"X":"O");
204.
205.     string loc = GetString();
206.     if (loc == NULL)
207.     {
208.         return 1;
209.     }
210.
211.     // convert to integers
212.     *r = tolower(loc[0]) - 'a';
213.     *c = loc[1] - '0';
214.
215.     return 0;
216. }
217.
218. /**
219.  *  Parse the command line
220.  */
221. int parse_command_line(int argc, char *argv[])
222. {
223.     if (argc != 2)
224.     {
225.         // if not specified, assume a 3x3 board
226.         d=3;
227.     }
228.     else
229.     {
230.         // ensure valid dimensions
231.         d = atoi(argv[1]);
232.         if (d < DIM_MIN || d > DIM_MAX)
233.         {
234.             printf("Board must be between %i x %i and %i x %i, inclusive.\n",
235.                 DIM_MIN, DIM_MIN, DIM_MAX, DIM_MAX);
236.             return 1;
237.         }
238.     }
239.     return 0;
240. }
```

```c

/**
 * Clears screen using ANSI escape sequences.
 */
void clear()
{
    printf("\033[2J");
    printf("\033[%d;%dH", 0, 0);
}

/**
 * Initializes the game's board with tiles (numbered 1 through d*d - 1),
 * i.e., fills 2D array with values but does not actually print them).
 */
void init()
{
    int i,j;

    // Set the board to be empty
    for (i=0;i<d;i++)
    {
        for(j=0;j<d;j++)
        {
            board[i][j] = 0;
        }
    }

    best_row=-1000;
    best_col=-1000;
}

/**
 * Prints the board in its current state.
 */
void draw()
{
    int i, j;

    // colorcode for red
    printf("\033[31m");
    printf("   ");
    for(j = 0; j < d; j++)
    {
        printf(" %d  ", j);
    }
    printf("\n");
    for (i = 0; i < d; i++)
    {
```

```
289.         printf("\033[31m %c ", (char)'a'+i);
290.         for (j = 0; j < d; j++)
291.         {
292.             if (board[i][j] != 0)
293.             {
294.                 if (board[i][j] == 1)
295.                 {
296.                     printf("\033[0m O ");
297.                 }
298.                 else
299.                 {
300.                     printf("\033[0m X ");
301.                 }
302.             }
303.             else
304.             {
305.                 printf("   ");
306.             }
307.             if(j<(d-1))
308.                 printf("\033[31m|");
309.         }
310.         if (i<(d-1))
311.         {
312.             printf("\n\033[31m   ");
313.             for(j = 0; j < d-1; j++)
314.             {
315.                 printf("---+");
316.             }
317.             printf("---");
318.         }
319.         printf("\033[0m\n");
320.     }
321.     printf("\n");
322. }
323. 
324. /**
325.  * check to see if someone has won, return the player number of the winner,
326.  * or zero if no winner
327.  */
328. int check_for_winner(int b[DIM_MAX][DIM_MAX])
329. {
330.     // check to see if the board has been won
331.     int i,j;
332.     int prev, winner;
333. 
334.     for(i=0;i<d;i++)
335.     {
336.         // check each row
```

```
337.         prev = b[i][0];
338.         winner = prev;
339.         for(j=1;j<d;j++)
340.         {
341.             if (prev != b[i][j])
342.             {
343.                 winner=0;
344.             }
345.         }
346.         if (winner != 0) return(winner);
347.
348.         // check each column
349.         prev = b[0][i];
350.         winner = prev;
351.         for(j=1;j<d;j++)
352.         {
353.             if (prev != b[j][i])
354.             {
355.                 winner=0;
356.             }
357.         }
358.         if (winner != 0) return(winner);
359.     }
360.
361.     // check diagonals, but only if d is odd
362.     if ((d%2) == 1)
363.     {
364.         prev = b[0][0];
365.         winner = prev;
366.         for(j=1;j<d;j++)
367.         {
368.             if (prev != b[j][j])
369.             {
370.                 winner=0;
371.             }
372.         }
373.         if (winner != 0) return(winner);
374.
375.         prev = b[0][d-1];
376.         winner = prev;
377.         for(j=1;j<d;j++)
378.         {
379.             if (prev != b[j][d-1-j])
380.             {
381.                 winner=0;
382.             }
383.         }
384.         if (winner != 0) return(winner);
```

```
385.     }
386.     return(0);
387. }
```