#ifndef _BOARD_H
#define _BOARD_H

#include <string>
#include "tvector.h"

using namespace std;

class Board
{
public:
    enum Player{X, O, empty};

    Board();

    bool isFull() const;
    bool isClear(int index) const;
    bool isWin(Player p) const;
    void print() const;
    int clearCount() const; // # clear locations

    string toString() const;
    int size() const;

    void place(int index, Player p);
    void unplace(int index);

private:
    tvector<Player> mySquares;
    string myString;
    int myCount; // # pieces played

    bool horizWin(int index, Player p) const;
    bool vertWin(int index, Player p) const;
};
#endif
```cpp
#include <iostream>
#include <string>
using namespace std;

#include "board.h"
#include "prompt.h"

/**
 * simple illustration of minimax/game backtracking program
 * for Tic-Tac-Toe. See board.h/board.cpp for tic-tac-toe
 * infrastructure. This program plays game until the end,
 * so there's no so-called board-evaluation function needed.
 * In this code the computer is always X and the human O
 * 
 * author: Owen Astrachan
 * history: revised 4/15/2002 to clean up separate
 * human/computer move functions, now there's one function
 */

class Game
{
public:
    Game();
    void display(); // show the current game board
    void play(); // play a game, start to finish

private:
    Board myBoard;
    int myCount; // bookkeeping statistics

    int bestMove(Board::Player p, int& move);
    int getHumanMove();

    bool scoreIsBetter(int score, int best, Board::Player p) const
    {
        if (p == Board::X)
        {
            if (score > best)
            {
                return true;
            }
            else
            {
                if (score < best)
                {
                    return true;
                }
                return false;
            }
        }
    }

    Board::Player opposite(Board::Player p) const
    // post: return opposite of p, i.e., X for O and O for X
    {
        if (p == Board::X) {
            return Board::O;
        }
        return Board::X;
    }

    int Game::bestScore(Board::Player p) const
    // post: return the best possible score for player p
    {
        if (p == Board::X) {
            return COMPUTER_WIN;
        }
        return HUMAN_WIN;
    }

    static int COMPUTER_WIN; // a big number
    static int DRAW; // a middle number
    static int HUMAN_WIN; // a low number

    int Game::COMPUTER_WIN = 10;
    int Game::DRAW = 0;
    int Game::HUMAN_WIN = -10;

    Game::Game()
    :
        myCount(0)
    {
    }

    void Game::display()
    // post: game board displayed
    {
        cout << "current board ————" << endl;
        myBoard.print();
        cout << "———" << endl;
    }

    Board::Player Game::opposite(Board::Player p) const
    // post: return opposite of p, i.e., X for O and O for X
    {
        if (p == Board::X) {
            return Board::O;
        }
        return Board::X;
    }

    int Game::getHumanMove()
    // post: move entered by human player is returned
    {
        int move;
        while (true)
        {
            move = PromptInRange("Enter Square:", 0, myBoard.size() - 1);
            if (myBoard.isClear(move)) {
                return move;
            }
            cout << "Illegal move" << endl;
        }
    }

    int Game::bestMove(Board::Player p, int & move)
    // post: return best score for player p, this
    // post: move is set to the move that achieves this score
    {
        if (myBoard.isFull()) {
            // game is a draw
            myCount++;
            return Game::DRAW;
        }

        if (myBoard.isWin(p)) {
            // win for p?
            myCount++;
            return bestScore(p);
        }

        static int bestScore(Board::Player p) const
        {
            if (p == Board::X) {
                return COMPUTER_WIN;
            }
            return HUMAN_WIN;
        }

        int k;
        int best = bestScore(opposite(p)); // assume opponent does well
        int score;
        int dontCareMove;
        for (k = 0; k < myBoard.size(); k++) {
            if (myBoard.isClear(k)) {
                // can we move here?
                myBoard.place(k, p);
                score = bestMove(opposite(p), dontCareMove);
                myBoard.unplace(k);
            }
        }
    }
};
```

```
if (scoreIsBetter(score, best, p)) {
    best = score;
    move = k;
}

return best;

void Game::play() {
    // post: game is played
    int move;
    int score;

    // loop until game over: draw or someone wins
    Board::Player player = Board::X;
    while (true) {
        // determine computer move
        myCount = 0;
        score = bestMove(player, move);
        myBoard.place(move, player);
        display();
        cout << "# boards looked at" << myCount << endl;
        if (myBoard.isWin(player) || myBoard.isFull()) {
            break;
        }

        // determine human move
        player = opposite(player);
        move = getHumanMove();
        myBoard.place(move, player);
        display();
        if (myBoard.isWin(player) || myBoard.isFull()) {
            break;
        }
        player = opposite(player);
    }
}

int main() {
    Game g;
    g.play();
    return 0;
}