```cpp
#include<cmath>
#define NUM_AGENTS 2
#define NUM_ACTIONS 4
#define NUM_OBSTACLES 2

// The world is an infinite two-dimensional grid. As such a physical array is not needed, only positions.

// Data component of agent.
template<typename ACTION_TYPE>
struct agent{
    // Current Position of Agent
    int x, y;
    // The list of actions the agent can perform.
    ACTION_TYPE* Actions[NUM_ACTIONS];
};

// State Information
template<typename ACTION_TYPE>
struct state{
    // The goal of the protagonist agent. The goal of the antagonnist agent is to touch the protagonist.
    agent<ACTION_TYPE> Goal;
    // Current Agent
    int CurrentAgent;
    // Which agent is the protagonist.
    int ProtagonistID;
    // The list of impassible spaces.
    agent<ACTION_TYPE> Obstacles[NUM_OBSTACLES];
    // The list of agents in the space.
    agent<ACTION_TYPE> Agents[NUM_AGENTS];
};

// Action Interface
struct action{
    // How much utility is there in performing the action right now?
    virtual int Evaluate(state<action>) = 0;
    // Can the action actually be performed?
    virtual bool Possible(state<action>) = 0;
    // What is the state returned by the action being performed?
    virtual state<action> Perform(state<action>) = 0;
};


int PrevAgent(state<action> s){
    return (s.CurrentAgent-1) % NUM_AGENTS;
}

// Computes the manhattan distance between two points.
int ManhattanDistance(agent<action> a, agent<action> b){
    return abs(b.x - a.x) + abs(b.y - a.y);
}

bool Overlap(agent<action> a, agent<action> b){
        return a.x == b.x && a.y == b.y;
}

// Defines common parts of the interface for moving.
struct moveAction : action {
    int Evaluate(state<action> s){
        int dist; s = Perform(s);
        agent<action> dest, winDest;

        if(PrevAgent(s) == s.ProtagonistID){
            // Protagonist Lose Condition
            for(int i = 0; i < NUM_AGENTS; i++){
                if(i == s.ProtagonistID) continue;
                if(Overlap(s.Agents[PrevAgent(s)],(s.Agents[i]))) return -9999;
            }
            dest = s.Goal;
        } else {
            dest = s.Agents[s.ProtagonistID];
        }
        if(Overlap(s.Agents[PrevAgent(s)],dest)) return 9999;
        dist = ManhattanDistance(s.Agents[PrevAgent(s)],dest);
        return (dist == 0) ?  2000 : 1000/dist;
    }

    bool Possible(state<action> s){
        s = Perform(s);
        for(int i = 0; i < NUM_OBSTACLES; i++)
            if(Overlap(s.Agents[PrevAgent(s)],s.Obstacles[i])) return false;
        return true;
    }
};

// The following four classes represent moving in the four cardinal directions.
struct moveLeft : moveAction{
    state<action> Perform(state<action> s){
        int i = s.CurrentAgent;
        s.Agents[i].x -= 1;
        s.CurrentAgent = (i+1) % NUM_AGENTS;
        return s;
    }
};
struct moveRight : moveAction{
    state<action> Perform(state<action> s){
        int i = s.CurrentAgent;
        s.Agents[i].x += 1;
        s.CurrentAgent = (i+1) % NUM_AGENTS;
        return s;
    }
};
struct moveUp : moveAction{
    state<action> Perform(state<action> s){
        int i = s.CurrentAgent;
        s.Agents[i].y -= 1;
        s.CurrentAgent = (i+1) % NUM_AGENTS;
        return s;
    }
};
struct moveDown : moveAction{
    state<action> Perform(state<action> s){
        int i = s.CurrentAgent;
        s.Agents[i].y += 1;
        s.CurrentAgent = (i+1) % NUM_AGENTS;
        return s;
    }
};
```