

```

/*
 * Classifier.h      header file
 *
 * Author:   Evan Hourigan                      Copyright (c), 2004.
 *           Hobart & William Smith Colleges
 *           Geneva, NY   14456
 *
 *           Email: hourigan@hws.edu
 * This source code may not be reproduced or modified in any way, shape, or
 * form without written consent from the author.
 */

#include <iostream>
#include <string>
#include <ctime>
#include "Array.h"

using namespace std;

const int COOPERATE = 0;
const int DEFECT = 1;

const int NUM_OF_MOVE_PAIRS = 9;
const int NUM_OF_GAMES = 50;

// Starting values are 3 and 6, how about 8 and 20, how about 4 and 21?
const int RULE_LENGTH = 5; // Length of classifier rule
const int NUM_OF_RULES = 11; // Number of classifier rules per individual

// Represent a current_individual-opponent move pair in a classifier rule.
// These values are used in the criteria array in the rule struct. The first
// letter after the underscore in the name of the constant represents the
// move of the current individual, the second letter corresponds to the move of
// the opponent.
const int M_XX = 0; // Match any for individual, match any for opponent
const int M_XC = 1; // Match any for individual, opponent cooperates
const int M_XD = 2; // Match any for individual, opponent defects
const int M_CX = 3; // Individual cooperates, match any for opponent
const int M_DX = 4; // Individual defects, match any for opponent
const int M_CC = 5; // Individual cooperates, opponent cooperates
const int M_CD = 6; // Individual cooperates, opponent defects
const int M_DC = 7; // Individual defects, opponent cooperates
const int M_DD = 8; // Individual defects, opponent defects

// These global variables are being used to track how many times each of the
// four different possible results of each game (i.e. COOPERATE-COOPERATE,
// DEFECT-COOPERATE, COOPERATE-DEFECT, and DEFECT-DEFECT) occur.
extern int cc_count;
extern int cd_count;
extern int dc_count;
extern int dd_count;

extern int games_played;
// This can take a value anywhere from 0 to RULE_LENGTH; used to
// determine whether a full history match can be made to rule sets, or
// only a partial match, which is the case if the current player has
// played less than RULE_LENGTH games with an opponent.

class Classifier { // The "Chromosome" class for the Population template.

// Defines a classifier rule
struct rule {

    int criteria[RULE_LENGTH];
    // Each index in this array corresponds to a move criterion that will

```

```

        // be used to match game histories. Criteria are matched to game
        // history in a sequential progression, with criteria[0] corresponding
        // to the most recent move, and criteria[RULE_LENGTH-1] corresponding
        // to the oldest move.
    int choice;
        // The choice given by this rule (i.e. the move of the current
        // individual if this move corresponds to the current history).
        // Can be one of two values: COOPERATE, or DEFECT
};

rule rules[NUM_OF_RULES]; // The classifier rules for the individual.

int history[RULE_LENGTH];
    // The history of moves (in move pairs) between the current individual
    // and an opponent.

double fitness; // Starting at zero, fitness is computer by playing
                // individuals against each other.

public:

    // For use in parallel implementation
    // Packs all of the current individual's genetic data into a single struct
    // which can be sent to different processes.
    struct Pack {
        rule rules[NUM_OF_RULES];
    };

    // Constructor for a classifier.
    Classifier();

    virtual ~Classifier();

    double getFitness();

    // Takes two move constants and returns true if they match, and returns
    // false if they do not. Typically this function is used to compare move
    // pairs in the game history array to move pairs in the criteria array
    // of the rule struct for the current individual.
    bool matches(int h, int c);

    // Determines if rule passed as parameter matches current history or not
    // by calling matches() for each criterion in the rule with history.
    // Currently not used, since rule matching is done based upon how many
    // games have played, and in some cases, the whole rule does not need to
    // be matched to the history.
    bool ruleMatches(rule r);

    // Returns either COOPERATE or DEFECT, by comparing the history of the
    // current individual to rules in the rule set in order of rules that are
    // most specific to rules that are least specific. Uses the matches()
    // function to determine if a rule matches. If the individual has not
    // yet played RULE_LENGTH games with an opponent (which is kept track of
    // in the current individual's games_played member variable), then only
    // games_played games will be matched to games_played criteria in the
    // rules array.
    int decideMove();

    // Takes move pair constant as parameter and updates the history by
    // deleting the oldest move, shifting the older moves down in position in
    // the array, and recording the most recent move. Also updates fitness

```

```

// value according to the payoff matrix.
void update(int move_pair_constant);

// When this function is called the mutation will work to actually change
// a 'data cell' of genetic data.
void mutate();

void crossover (Classifier& partner);

// Makes an individual into a defector by changing all of the choice
// values in the rule set to DEFECT.
void malign();

static void computeFitnesses(Classifier** data, int size, int method);

// Prints out a screen representation of the data composing the
// current individual
void printIndividual();

// Tests the current individual in a variety of ways and returns an int
// value in the range of 0 (dumb) to 100 (smart), representing the IQ of
// the current individual. Initially the IQ is set at 50, to give each
// individual a fair chance to prove themselves. Then, performs a
// variety of game-behavior tests, and based upon the current individuals
// behavior, increases or decreases the IQ score.
int nicenessIQ();

// For use in parallel implementation
// Replace genetic data of this worst individual object with the data
// passed as newIndividual
void transplant(Pack newIndividual);

// For use in parallel implementation
// Packs current individuals brains into a struct and return it as a
// value of type Classifier::Pack
Pack get();

}; /* end of class Classifier */

```