

```

/*
 * Population.h      template class definition 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.
 */

/* A shell for a simple version of the genetic algorithm. The class
   that to replace "Chromosome" in the template must have:
   -- copy constructor (default might be OK)
   -- member function   double fitness()
   -- member function   void mutate()
   -- member function   void crossover(const Chromosome&)
   -- member function   void computeFitnesses(Chromosome **data, int size)
To use this shell, make a population object, P, and then call
P.create(). An optional parameter gives the population size.
You can call P.setMutationProbability() and P.setCrossoverProbability()
to set parameters of the algorithm. Call P.getBestIndividual(),
P.getAverageFitness(), and P.getMaximumFitness() to get data about
the current generation. Call P.breed() to make the next generation.
You can get the i-th individual in the current population by calling
P.getIndividual(i). Note that the pointers returned by P.getIndividual()
and P.getBestIndividual() should not be deleted and that they become
invalid when P.breed() or P.create() is called.
 */

#include <cstdlib>
#include <string>
inline int randInt(int N) { // Random int x where 0 <= x < N
    return std::rand() % N;
}
inline double randReal() { // Random real x where 0.0 <= x < 1.0
    return ( std::rand() / (RAND_MAX + 1.0));
}

template<class Chromosome, class ChromosomePacked>
class Population {

    Chromosome **data;           // Pointers to individuals in the population
    double *fitness;
    int populationSize;
    double crossoverProbability; // Probability that an individual will
                                // engage in crossover
    double mutationProbability;  // Probability that an individual will
                                // experience mutation
    double malignProbability;    // Probability that an individual will be
                                // made into a defector
    double bestFitness;          // Highest seen fitness in population
    double worstFitness;         // Lowest seen fitness in population
    double averageFitness;       // Average fitness of population
    double totalFitness;
/*
    int saveBest = 1; // Number of "best" individuals that should be saved
                    // each generation. (start with one) */
    Chromosome *bestIndividual;  // Pointer to the "best" individual in the
                                // population
    Chromosome *worstIndividual; // Pointer to the "worst" individual in
                                // the population
    Chromosome **bestIndividuals; // Dynamic array for holding the specified
                                // number of best individuals to be saved

    int selectByFitness() {
        // Choose a random individual, with probability of selection
        // proportional to fitness. (Not very efficient -- I should
        // maybe sort according to fitness and use binary search.)
        double cut = totalFitness * randReal();
        int i = 0;

```

```

        double total = 0;
        while (true) {
            if (i == populationSize - 1)
                return i;
            total += fitness[i];
            if (total > cut)
                return i;
            i++;
        }
    }

void getCurrentStats() {
    // Compute fitnesses for all individuals, and get various
    // summary statistics.
    bestFitness = -1e300;
    worstFitness = 1e300;
    totalFitness = 0;
    Chromosome::computeFitnesses(data, populationSize);
    for (int i = 0; i < populationSize; i++) {
        fitness[i] = data[i]->getFitness();
        totalFitness += fitness[i];
        if (fitness[i] > bestFitness) {
            bestFitness = fitness[i];
            bestIndividual = data[i];
        }
        else if (fitness[i] < worstFitness) {
            worstFitness = fitness[i];
            worstIndividual = data[i];
        }
    }
    averageFitness = totalFitness / populationSize;
}

public:

    // Used in parallel implementation
    // Holds genetica data for MIGRATION_SIZE individuals
    struct PopulationPack {
        ChromosomePacked individual[MIGRATION_SIZE];
        int migrationGeneration; // Generation at which migration
                                // is taking place
    };

    // Used in parallel implementation
    // Holds statistics for STATS_GEN_SPAN generations
    struct StatisticsPack {
        // Generation at which the indices of these variables are valid
        int currentGeneration[STATS_GEN_SPAN];
        // Niceness of the population
        double populationNiceness[STATS_GEN_SPAN];
        // Average IQ of the population
        double populationAvgNiceIQ[STATS_GEN_SPAN];
        // Average fitness of the population
        double populationAvgFitness[STATS_GEN_SPAN];
    };

    Population() {
        data = 0;
        populationSize = 0;
        crossoverProbability = 0.75;
        mutationProbability = 0.1;
        malignProbability = 0.1;
    }
    void create(int size = 100) {
        if (size <= 0) {
            cerr << "Population size must be positive.\n";
            exit(1);
        }
        if (data) {
            for (int i = 0; i < populationSize; i++)

```

```

        delete data[i];
    }
    if (data && size != populationSize) {
        delete[] data;
        delete[] fitness;
        data = 0;
        fitness = 0;
    }
    if (!data) {
        data = new (Chromosome*)[size];
        fitness = new double[size];
    }
    populationSize = size;
    for (int i = 0; i < size; i++)
        data[i] = new Chromosome;
    getCurrentStats();
}
~Population() {
    if (data) {
        for (int i = 0; i < populationSize; i++)
            delete data[i];
        delete[] data;
        delete[] fitness;
    }
}
int getPopulationSize() {
    return populationSize;
}
Chromosome *getIndividual(int i) {
    return (data && i >= 0 && i < populationSize)? data[i] : 0;
}

double getMaximumFitness() {
    return data? bestFitness : 0;
}
double getAverageFitness() {
    return data? averageFitness : 0;
}
double getMinimumFitness() {
    return data? worstFitness : 0;
}
Chromosome *getBestIndividual() {
    return data? bestIndividual : 0;
}
Chromosome *getWorstIndividual() {
    return data? worstIndividual : 0;
}
void setMutationProbability(double p) { // Default is 0.1
    mutationProbability = p;
}
double getMutationProbability() {
    return mutationProbability;
}
void setCrossoverProbability(double p) { // Default is 0.75
    crossoverProbability = p;
}
double getCrossoverProbability() {
    return crossoverProbability;
}

// Used in parallel implementation
// Imports individuals packed in imm into current population
void doImmigration(PopulationPack imm) {
    for (int i = 0; i < MIGRATION_SIZE; i++) {
        int x = rand()%POPULATION_SIZE;
        data[x]->transplant(imm.individual[i]);
    }
}

// Used in parallel implementation
// Exports individuals from current population and returns an object

```

```

// of type PopulationPack
PopulationPack doEmigration() {
    PopulationPack em;
    for (int i = 0; i < MIGRATION_SIZE; i++) {
        int x = rand()%POPULATION_SIZE;
        em.individual[i] = data[x]->get();
    }
    return em;
}

void breed() {
    // This method produces a new generation.
    if (!data) {
        cerr << "Attempt to breed, when no population has been created.\n";
        exit(1);
    }
    Chromosome **newData = new (Chromosome*)[populationSize];

    newData[0] = new Chromosome(*bestIndividual);
    // Copy the best individual in the current generation. This
    // individual will not be mutated or crossed-over, to ensure that
    // the maximum fitness does not decrease from one generation to
    // the next.
    for (int i = 1; i < populationSize; i++) {
        // Copy individuals from the current population, with selection
        // probability proportional to fitness.
        int who = selectByFitness();
        newData[i] = new Chromosome(*data[who]);
    }

    for (int i = 0; i < populationSize; i++) {
        delete data[i];
    }

    delete[] data;
    data = newData; // Replace current population with new population.
    for (int i = 1; i < populationSize - 1; i += 2) {
        // Consider pairs of individuals in the new population, starting
        // with index 1 to avoid the best individual from the past
        // generation. Do a crossover on the pair of individuals, with a
        // probability given by crossoverProbability. Note that the
        // crossover() function is assumed to modify both its parameter
        // object and the object that receives the crossover message.
        if (randReal() < crossoverProbability)
            data[i]->crossover(*data[i+1]);
    }

    for (int i = 1; i < populationSize; i++) {
        // Consider each individual, starting with index 1 to avoid the
        // best individual from the past generation. Call mutate()
        // with a probability given by mutationProbability.
        if (randReal() < mutationProbability)
            data[i]->mutate();
        if (randReal() < malignProbability)
            data[i]->malign();
    }
    getCurrentStats(); // Compute statistics for new generation.
}

};

```