Adam & Eve Simulator with Genomes

Onat Bas, SWE501, Assn 2

Implementation

This project is implemented in object oriented way. Each human is an instance of a Class called Human. Reproducing(mating) between humans are implemented using the techique used by natural crossover. Each Human has a certain DNA, which can be represented as a n-byte long array.

DNA

DNA is handled by Genome class. This class is responsible for creating-allocating parts in the dna chain. This class will also provide the user an object through which the implementor can access to the data stored in the dna.

Human DNA

The structural human DNA is created-handled by the Class HumanGenome. At it's constructor, HumanGenome class allocates certain data types it's parent class, Genome. This class also keeps the access objects provided by Genome, so they can be accessed/altered at a later time. The data stored for a human are these:

- Gender (male/female)
- Hair color (RGB)
- Eye color (RGB)
- Likely to weigh (kg)
- Likely to be tall as(cm)

Human

Human, a child class of HumanGenome, has also a string field for name. New humans can be created by providing at least two parent Humans. By doing this, the child Human will obtain the dna chains of it's parent and do cross-over on them and do simple modifications (which we call mutation) to create potentially not-existing features in parents. For example:

```
Parent 1' DNA string: 13_D@ ??+D?
Parent 2' DNA string: ??_D? ??_DM
Child's DNA string  : ??_DM ??+D?
```

After new gene is created, new human can immediately obtain these new properties, which are usually slightly different than it's parents.

Conclusion

Population Limit

Even in a really small simulation such as this, one can observe the effect of restraining a population too much. In a very restricted population, in time, some genomes simply die off, disappear from the gene pool, never to be seen again. After that a certain set of properties will simply dominate all the population, eliminating everyone that's different, resulting in a confined population.

Even though new genomes are created by Mutation, they still are minorities and since we don't have a performance function implemented to increase the chance of mutants increase, they'll just die off again.

Generations

As generations pass, the diversity in the population also increase, making the gene pool richer and richer. The result of this is the number of mutations per birth is few. Producing a very different dna/human requires many mutations, and that can only happen with a lot of mutations, thus more generations.

Result

Usage:

```
$ ./sampler
usage: ./sampler "birth_count" "population_limit"
```

Here's a sample output for 100000 generatios and 100 population size limit.

```
        $ ./sampler 10000 100
Adam                  (Male)           183cm       65kg
```

```
Eve                  (Female)          148cm       44kg
This is the final population (10000 generations, 100 people):
Darren               (Male)            203cm       196kg
Alfric               (Male)            146cm       84kg
Snowdon              (Female)          163cm       68kg
Smithies             (Male)            194cm       132kg
Darren               (Male)            139cm       132kg
Robbins              (Male)            137cm       8kg
Harrington           (Male)            139cm       196kg
.
.
.
Huxley               (Male)            207cm       68kg
Gabriel              (Male)            139cm       192kg
Snowdon              (Male)            203cm       192kg
Smithies             (Male)            155cm       192kg
Crawford             (Female)          72cm        212kg
Clare                (Male)            235cm       206kg
Crawford             (Male)            147cm       64kg
Robbins              (Female)          202cm       224kg
```

Sourcecode:

Main

main.cxx

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include "Genome.hxx"
#include "GeneEncapsulator.hxx"
#include "Human.hxx"
#include "HelperFunctions.hxx"
#include <fstream>
using namespace std;

int main(int argc, char const *argv[])
{
        if (argc < 3)
        {
                cout << "usage: ./sampler <birth_count> <population_limit>" << endl;
                return 0;
        }
        const int pop_limit = stoi(argv[2]);
        const int birth_count = stoi(argv[1]);

        vector<Human *> humans;

        Human *adam = new Human("Adam");
        Human *eve = new Human("Eve");
        adam->setIsMale(true);
        eve->setIsMale(false);
        traceHuman(adam, std::cout);
        traceHuman(eve, std::cout);
        std::cout << endl;

        humans.push_back(adam);
        humans.push_back(eve);


        for (int i = 0 ; i < birth_count ; i++){

                Human *baby = new Human(
                                getRandName(),
                                *humans[rand() % humans.size()],
                                *humans[rand() % humans.size()]
                        );

                humans.push_back(baby);

                while (humans.size() > pop_limit)
                {
                        humans.erase(humans.begin()); // Oldest dies off.
                }
        }

        std::cout << "This is the final population (" << birth_count << " generations, " << pop_limit << " people): " << end
        for (int i = 0 ; i < humans.size() ; i++)
        {
                traceHuman(humans[i], std::cout);
        }


        return 0;
}
```

Headers

Human.hxx

```
	#pragma once

#include <Human.hxx>
#include <HumanGenome.hxx>
#include <iostream>

using namespace std;


class Human : protected HumanGenome
{
	string _name;


public:
	Human(string name);
	Human(string name, Human &parent1, Human &parent2);
	string getName();


double getHeight()
{
	return geneticHeight->get();
}
double getWeight()
{
	return geneticWeight->get();
}
double getEyecolor()
{
	return eyeColor->get();
}
double getHaircolor()
{
	return haircolor->get();
}
bool isMale()
{
	char got = gender->get() >> 7;
	return got == 0 ? false : true;
}

void setHeight(double height)
{
	geneticHeight->set(height);
}
void setWeight(double weight)
{
	geneticWeight->set(weight);
}
void setEyecolor(double color)
{
	eyeColor->set(color);
}
void setHaircolor(double color)
{
	haircolor->set(color);
}
void setIsMale(bool value)
{
	gender->set(value ? 0xFF : 0x00);
}


};
```

HumanGenome.hxx

```
	#pragma once

#include "Genome.hxx"
#include "GeneEncapsulator.hxx"
#include <string>

using namespace std;

class HumanGenome : protected Genome{
protected:

	GeneEncapsulator *gender;
	GeneEncapsulator *eyeColor;
	GeneEncapsulator *haircolor;
	GeneEncapsulator *geneticHeight;
	GeneEncapsulator *geneticWeight;

	void allocateGenes();

	HumanGenome(string dna1, string dna2);
	HumanGenome();
};
```

Genome.hxx

```
	#pragma once
```

```
#include <iostream>
#include <vector>

#include "GeneEncapsulator.hxx"
#include <string>
#include <sstream>

class Genome
{
private:
        int allocate(size_t size);

protected:
        std::vector<unsigned char> genes;

public:
        template <class T>
        GeneEncapsulator<T> add()
        {
                int offset = allocate(sizeof(T));
                return GeneEncapsulator<T>(&genes, offset);
        };

        std::string getGenomeAsString()
        {
                std::stringstream s;
                for (unsigned char gene : genes)
                        s << gene;
                return s.str();
}

protected:
        void initializeByCrossover(std::string dna1, std::string dna2);


};
```

Source files

Human.cxx

```
        #include "Human.hxx"
#include <stdlib.h>
#include <iostream>

Human::Human(string name) : _name(name)
{
        bool f= (rand()%0xff) < 0x7F;
        setIsMale(f);
        setEyecolor(rand()%255);
        setHaircolor(rand()%255);
        setHeight(rand()%100 + 125);
        setWeight(rand()%100 + 35);
}

Human::Human(string name, Human &parent1, Human &parent2) : _name(name)
{
//      if (parent1.isMale() == parent2.isMale()) // if same gender
//              cout << "That's biologically impossible, but
// this is a computer program so who cares?" << endl;

        initializeByCrossover(parent1.getGenomeAsString(), parent2.getGenomeAsString());
        int seed;

        for (int i = 0 ; i < genes.size() ; i++)
        {
                seed = rand() % 25500; // 4 percent change of mutation for each gene
                if (seed < 255)
                {
                        char mutation = (char)seed;
                        mutation++; // So it has only one 1 in it's binary decimals, rest is 0.
                        genes[i] += mutation;
                }
        }
}

string Human::getName()
{
        return _name;
}
```

HumanGenome.cxx

```
        #include "HumanGenome.hxx"

HumanGenome::HumanGenome(string dna1, string dna2)
{
        allocateGenes();
        initializeByCrossover(dna1, dna2);
}

HumanGenome::HumanGenome()
{
        allocateGenes();
}

void HumanGenome::allocateGenes(){
```

```
        gender = new GeneEncapsulator(NULL, 0);
        *gender = add();

        eyeColor = new GeneEncapsulator(NULL, 0);
        *eyeColor = add();

        haircolor = new GeneEncapsulator(NULL, 0);
        *haircolor = add();

        geneticHeight = new GeneEncapsulator(NULL, 0);
        *geneticHeight = add();

        geneticWeight = new GeneEncapsulator(NULL, 0);
        *geneticWeight = add();
}
```

Genome.cxx

```
        #include "Genome.hxx"
#include <stdlib.h>

using namespace std;

int Genome::allocate(size_t size){
        int position = genes.size();
        while(size-- > 0)
        {
                genes.push_back((unsigned char)0);
        }
        return position;
}


void Genome::initializeByCrossover(std::string dna1, std::string dna2){
        for (int i = 0 ; i < genes.size() ; i++)
        {
                unsigned char seed  = (unsigned char)(rand() % 255);

                genes[i] = (unsigned char)(dna1[i] & seed) +
                                        (unsigned char)(dna2[i] & (~seed));
        }
}
```