

```

#include <iostream>
#include <fstream>
#include <vector>
#include "utils.h"

#define DURATION 360

class MatriceContact : public Utils::Mat8x8<mfloat>, public Utils::Singleton<MatriceContact>
{
public:
    friend class Utils::Singleton<MatriceContact>;
};

class Groupe
{
protected:
    uint m_indice;        // indice du groupe
    mfloat m_frac;       // fraction de la population de ce sexe contenue dans le groupe
    mfloat m_infect;     // fraction de la population de ce groupe infectée
    mfloat m_duree_moy;  // durée moyenne de l'infection dans ce groupe, en jours

public:
    Groupe( uint i, mfloat frac, mfloat I0, mfloat D )
        : m_indice(i), m_frac(frac), m_infect(I0), m_duree_moy(D)
    {
        if( frac < mfloat(0.0) )
        {
            throw new std::invalid_argument( "Fraction de population négative, c'est
fâcheux." );
        }
    }

    Groupe() {};
    ~Groupe() {};

    uint getIndice() const { return m_indice; }
    mfloat getFrac() const { return m_frac; }
    mfloat getI() const { return m_infect; }
    mfloat getNI() const { return m_infect * m_frac; }

    void tour_suivant();
};

struct tour
{
    mfloat femmes_infect;
    mfloat hommes_infect;
    uint date;

    tour()
    {
        femmes_infect = mfloat(0.0);
        hommes_infect = mfloat(0.0);
    }
};

```

```

        date = 0;
    }
};

class Population : public Utils::Singleton<Population>
{
    Groupe* m_groupes[8];
    Groupe* m_groupes_copie[8];
    // Les groupes, lors de la mise à jour,
    // ont besoin les uns des autres ; on leur transmet une copie des groupes
    // et non les groupes eux-mêmes, pour éviter de calculer les nouvelles
    // valeurs en utilisant les __nouvelles__ valeurs des autres groupes
    // (si ces groupes ont été mis à jour en premiers)
    uint n;
    std::vector<tour> m_graphe;

    void creer_groupes();
    void copier_groupes(); // Copie m_groupes dans m_groupes_copie
    void remplir_matrice_contact();

    virtual ~Population()
    {
        for( int i = 0; i < 8; ++i )
        {
            delete m_groupes[i];
            m_groupes[i] = 0;
        }
    }

    Population() : n( DURATION )
    {
        m_graphe.resize( n );
        creer_groupes();      copier_groupes();
        remplir_matrice_contact();
    }

public:

    // renvoie le i-eme groupe (1<=i<=8)
    const Groupe& operator[] (uint i) const
    {
        // Flemme de gérer i > 8 et i = 0
        return *m_groupes_copie[i-1];    // On renvoie la COPIE
    }

    // Effectue la simulation sur n jours, remplissant
    // m_graphe avec les résultats à chaque tour
    void run();

    void print();

    friend class Utils::Singleton<Population>;
};

```

```

void Groupe::tour_suivant()
{
    static const MatriceContact& mc = MatriceContact::getSingleton();
    static const Population& pop = Population::getSingleton();
    static const mfloat one = mfloat(1.0);

    mfloat deltaNI = mfloat(0.0), deltaI = mfloat(0.0);

    for( int j=1; j <= 8; ++j )
    {
        mfloat Lij = mc(m_indice, j);
        deltaNI += Lij * pop[j].getNI() * (one-m_infect);
    }
    deltaI = deltaNI / m_frac;
    deltaI -= m_infect/m_duree_moy;

    m_infect += deltaI;

    // On divise par m_frac (i.e, Ni), car on veut delta(I) et pas delta(NI)
    // (on calcule I/D séparément, au lieu de faire NI/D, puis de le diviser
    // par N (perte de précision))
}

void Population::remplir_matrice_contact()
{
    static MatriceContact& mc = MatriceContact::getSingleton();

    for( int i = 1; i <= 8; ++i )
    {
        for( int j = 1; j <= 8; ++j )
        {
            // NB : en multipliant ces valeurs par 4 (!), on obtient une stabilisation de
            l'épidémie à un palier haut
            if( (i+j)%2 == 0 )
            {
                // Même sexe
                mc(i,j) = mfloat(0.0);
            }
            else if( j >= 7 )
            {
                // Très actifs sexuellement, mais
                // ont des symptômes apparents : moins attractifs
                mc(i,j) = mfloat(0.002);
            }
            else if( j >= 7 )
            {
                // Pas très actifs sexuellement, et
                // ont des symptômes apparents
                mc(i,j) = mfloat(0.0002);
            }
            else if( j >= 3 )
            {
                // Très actifs sexuellement, et
                // n'ont pas de symptômes apparents
                mc(i,j) = mfloat(0.01);
            }
        }
    }
}

```

```

    }
    else
    {
        // Pas très actifs sexuellement, mais
        // n'ont pas de symptômes apparents
        mc(i,j) = mfloat(0.005);
    }
}

}

for( int i = 1; i <= 8; ++i )
{
    for( int j = 1; j <= 7; ++j )
        std::cout << mc(i,j) << "&\t";
    std::cout << mc(i,8) << "\t \\\\" << std::endl;
}
}

void Population::creer_groupes()
{
    mfloat Fi, I0i; mfloat Di;

    Fi = mfloat(0.25);

    // À faire, éventuellement : lecture depuis un fichier (idem pour valeurs Lij)

    m_groupes[0] = new Groupe( 1, mfloat(0.25), mfloat(0.0), mfloat(30.0) );
    m_groupes[1] = new Groupe( 2, mfloat(0.3), mfloat(0.0), mfloat(25.0) );
    m_groupes[2] = new Groupe( 3, mfloat(0.25), mfloat(0.05), mfloat(30.0) );
    m_groupes[3] = new Groupe( 4, mfloat(0.2), mfloat(0.00), mfloat(25.0) );
    m_groupes[4] = new Groupe( 5, mfloat(0.25), mfloat(0.0), mfloat(10.0) );
    m_groupes[5] = new Groupe( 6, mfloat(0.3), mfloat(0.0), mfloat(10.0) );
    m_groupes[6] = new Groupe( 7, mfloat(0.25), mfloat(0.0), mfloat(10.0) );
    m_groupes[7] = new Groupe( 8, mfloat(0.2), mfloat(0.0), mfloat(10.0) );

    for( int i = 0; i < 8; ++i )
    {
        /*
        //      std::cin >> Fi >> I0i >> Di;

        Fi = 0.25;
        I0i = 0.5;
        Di = 0.1;

        m_groupes[i] = new Groupe( i+1, Fi, I0i, Di*30.0 );
        // Di*30 : Di a été donné en mois, on le veut en jours
        */

        m_groupes_copie[i] = new Groupe();
    }
}

void Population::copier_groupes()
{

```

```

    for( int i = 0; i < 8; ++i )
    {
        *m_groupes_copie[i] = *m_groupes[i];
        // rien de très dur, pas d'autoaffectation à craindre
    }
}

void Population::run()
{
    for( uint k = 0; k < n; ++k )
    {
        m_graphe[k].date = k;
        for( int i = 0; i < 8; ++i )
        {
            if( (i+1)%2 == 0 )
                m_graphe[k].femmes_infect += m_groupes_copie[i]->getNI();
            else
                m_graphe[k].hommes_infect += m_groupes_copie[i]->getNI();

            m_groupes[i]->tour_suivant();
        }
        copier_groupes();
    }
}

void Population::print()
{
    std::cout << "Jour\t\tHommes\t\t\tFemmes" << std::endl;

    for( uint k = 0; k < n; ++k )
    {
        std::cout << k << "\t\t"
            << m_graphe[k].hommes_infect.get_mpf_t() << "\t\t"
            << m_graphe[k].femmes_infect.get_mpf_t() << "\t\t"
            << std::endl;
    }

    std::ofstream ofh("plothommes.dat");
    std::ofstream off("plotfemmes.dat");
    for( uint k = 0; k < n; ++k )
    {
        ofh << k << " " << m_graphe[k].hommes_infect.get_mpf_t() << std::endl;
        off << k << " " << m_graphe[k].femmes_infect.get_mpf_t() << std::endl;
    }
}

int main()
{
    Population::getSingleton().run();
    Population::getSingleton().print();
    return EXIT_SUCCESS;
}

```