11. Compléments de cours (hors programme)

Structures

Les structures permettent de regrouper plusieurs variables de types différents sous un même nom. Cela peut rendre un programme plus simple et plus lisible, mais est aussi une alternative aux pointeurs pour faire retourner plus d'une seule valeur par une fonction. Syntaxe générale :

```
typedef struct {
  bloc de définitions de variables membres;
} nom;
```

Par exemple pour regrouper les propriétés d'une particule :

```
typedef struct {
  double masse, charge;
  int spin2; // spin est entier ou demi-entier, donc spin fois deux est entier
  double* pos; // vecteur position
} particule;
```

Puis on peut déclarer des variables du type `particule` et on a accès à ses membres en utilisant l'opérateur point :

```
particule e1, e2;
e1.masse = 9.1e-31; e1.charge = -1.6e-19; e1.spin2 = 1;
e1.pos = (double*)malloc(3*sizeof(double));
e1.pos[0] = 0.; e1.pos[1] = 0.; e1.pos[2] = 0.;
e2 = e1;  // copie toutes les valeurs de e1 dans e2
e2.pos = (double*)malloc(3*sizeof(double)); // e2.pos ne doit pas pointer sur
e2.pos[0] = 1.; e2.pos[1] = 0.; e2.pos[2] = 0.; // le meme vecteur que e1.pos
```

Remarque : on ne peut pas comparer deux `struct` directement (`e1 == e2` ne marche pas), il faut comparer membre par membre.

Pour le reste on peut utiliser des `struct` comme d'autres types de variables : comme argument d'une fonction, comme type de retour d'une fonction, et on peut définir des pointeurs sur un `struct` (et donc des tableaux de `struct`).

Remarque : si `ps` est un pointeur sur un `struct` qui a un membre `i`, la syntaxe `ps->i` est une alternative à `(*ps).i`.

Exemple : 30 charges positives aléatoires dans une boîte carrée 2D de côté 1 (on néglige la pesanteur) :

```
#include<bibli_fonctions.h>
typedef struct {
  double masse, charge, *pos;
} particule;

particule* part;   // Variables globales, tableau 1D de struct
int Np = 30;

void init_part(double* q) {
  int i;
  part = (particule*)malloc(Np * sizeof(particule));
  for(i = 0; i < Np; i++) {
    part[i].masse = 4.*drand48() + 1.;   // M_i ∈ [m,5m]
    part[i].charge = 2.*drand48() + 1.;  // q_i ∈ [e,3e]
    part[i].pos = &(q[2*i]);    // Les part[i].pos pointent sur les parties
    part[i].pos[0] = drand48(); // pertinentes du long vecteur q
    part[i].pos[1] = drand48(); // Positions aleatoires dans [0,1]*[0,1]
  }
}
```

```
void systeme(double* q, double t, double* qp, int n) {
  int i, j, j0;
  double** r = (double**)malloc(Np * sizeof(double*));
  for(i = 0; i < Np; i++) {   // Matrice des distances entre particules
    r[i] = (double*)malloc(Np * sizeof(double));
    for(j = 0; j < Np; j++)
      if(j > i)
        r[i][j] = sqrt(pow(part[i].pos[0]-part[j].pos[0], 2)
                       + pow(part[i].pos[1]-part[j].pos[1], 2));
      else if(j < i)
        r[i][j] = r[j][i];    // Cette matrice est symetrique
  }
  for(i = 0; i < n/2; i++)
    qp[i] = q[i+n/2];
  for(i = n/2; i < n; i+=2) {
    j0 = (i-n/2)/2;  // Numero de la particule qui correspond a l'indice i
    qp[i] = 0.; qp[i+1] = 0.;
    for(j = 0; j < Np; j++)
      if(j != j0) {            // e^2/(4πε0 m) = 1
        qp[i] += part[j0].charge * part[j].charge / part[j0].masse
                 * (part[j0].pos[0]-part[j].pos[0]) / pow(r[j0][j], 3);
        qp[i+1] += part[j0].charge * part[j].charge / part[j0].masse
                   * (part[j0].pos[1]-part[j].pos[1]) / pow(r[j0][j], 3);
      }
  }
  for(i = 0; i < Np; i++) free(r[i]);
  free(r);
}
```

```
int main() {
  int i, j, n = 4*Np, Nt = 1000;
  double t = 0, tfin = 1, dt = (tfin - t) / (Nt - 1);
  double* q = (double*)malloc(n * sizeof(double));
  fstream fich("particules.res", ios::out);
  srand48(time(NULL)); init_part(q);  // q = (x0, y0, x1, y1, ..., ẋ0, ẏ0, ...)
  for(i = n/2; i < n; i++) q[i] = 0.; // Initialiser vitesses a zero
  for(i = 0; i < Nt; i++) {
    for(j = 0; j < Np; j++)
      fich << part[j].pos[0] << " " << part[j].pos[1] << " ";
    fich << endl;
    rk4(systeme, q, t, dt, n);
    for(j = 0; j < n/2; j++) {        // Collisions elastiques avec parois
      if(q[j] < 0.) { q[j] = -q[j]; q[j+n/2] = -q[j+n/2]; }
      if(q[j] > 1.) { q[j] = 2.-q[j]; q[j+n/2] = -q[j+n/2]; }
      if(j%2 == 0 && sqrt(q[j+n/2]*q[j+n/2]+q[j+n/2+1]*q[j+n/2+1]) > 0.1/dt) {
        q[j+n/2] = 0.; q[j+n/2+1] = 0.; } } }  // On remet a zero les vitesses
  free(q); free(part); fich.close(); ostringstream pyth;       // trop elevees
  pyth << "A = loadtxt('particules.res')\n"
       << "for i in range(" << Nt-1 << "):\n"
       << "  clf()\n"                  // Effacer la figure precedente
       << "  xlim(0,1)\n"
       << "  ylim(0,1)\n"
       << "  for j in range(" << Np << "):\n"
       << "    plot(A[i:i+2, 2*j], A[i:i+2, 2*j+1], linewidth=4,"
       <<          "color=cm.Spectral(j/" << Np << ".))\n"
       << "  pause(0.001)\n";
  make_plot_py(pyth); return 0; }
```

Création de bibliothèques

Si vous voulez utiliser les mêmes fonctions dans plusieurs programmes, il est pratique de les mettre dans une bibliothèque. Il faut alors trois fichiers :

1. Le fichier du programme principal, disons *prog.cpp*, qui contient la fonction `main` et éventuellement des fonctions uniques au programme que vous ne voulez pas mettre dans la bibliothèque.
2. Le fichier (la bibiothèque), disons *fonctions.cpp*, qui contient les fonctions que vous voulez réutiliser.
3. Le fichier d'en-tête (*header file*), disons *fonctions.h*, qui contient les prototypes des fonctions de la bibiothèque.

Pour utiliser la bibliothèque dans votre programme, vous mettez la ligne `#include"fonctions.h"` au début de votre programme et vous compilez avec `g++ fonctions.cpp prog.cpp` et exécutez avec `./a.out`.

(Vous ne pouvez pas utiliser la commande `ccc` dans ce cas ; voir la définition de `ccc` dans le répertoire /public/mphyo/bin pour les options à rajouter dans la commande `g++` si vous voulez utiliser `math.h`, `bibli_fonctions.h` ou Python.)

Exemple simple : le fichier *fonctions.cpp* consiste en la ligne

```
int fonc_triv(int x) { return x; }
```

le fichier *fonctions.h* consiste en la ligne

```
int fonc_triv(int x);
```

et le fichier *prog.cpp* consiste en

```
#include<iostream>  // Difference entre <> et "" est endroit ou compilateur
#include"fonctions.h" // cherche le fichier : repertoire systeme ou courant
using namespace std;
int main() { cout << fonc_triv(2) << endl; return 0; }
```