DEA

« Astrophysique et instrumentations associées »

C++

Programmation orientée objet

2004

Benoît Semelin

Références

Il existe de très nombreux tutoriels C++ téléchargeable gratuitement:

- Pour une liste de tutoriels avec appréciation:

http://c.developpez.com/cours/

- La page personnelle du créateur (en anglais)

http://www.research.att.com/~bs/3rd.html

On y trouve un manuel d'apprentissage téléchageable.

La POO
(Programmation orientée objet)

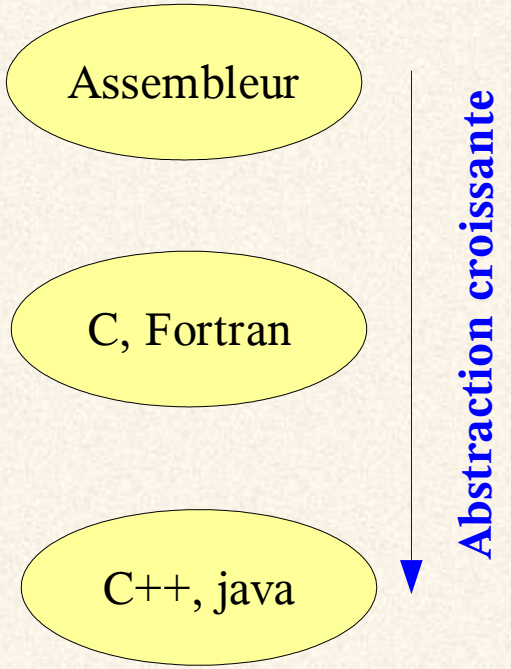
C++, java sont des **langages orientés objet**.

Qu'est-ce que ça veut dire?

- Les objets sont de nouvelles **sortes de variables encore plus générales** que les types dérivés (ou structures).
- Leur caractéristique principale: ils **acceptent des procédures** (fonctions, subroutines) comme composantes.

A quoi ça sert?

- Améliorer la **lisibilité d'un code** en utilisant des schémas plus **proche du raisonnement humain** grâce aux objets.
- Développer un code au sein d'un grand projet (plusieurs développeurs): assurer la **robustesse** et l'**interopérabilité** des différentes parties.



Un exemple d'objet

On veut simuler un système physique (p.e. un disque proto-planétaire) par un ensemble de particules. L'objet de base du programme va être la **particule**.

Rappel: type dérivé (structure)

Définition

```
typedef struct {
     double x;
     double y;
     double z; }
vecteur3D;
```

Utilisation

```
vecteur3D   pos[100],vel[100];
double      dt;

dt=0.01;
pos[50].x += vel[50].x*dt;
```

Déclaration de la **classe** « particule »:

Déclaration (prototype)

```
class particule {
      private:
         vecteur3D   pos,vel;
         double      mass;
      public:
         void adv_pos (double);
};
```

Description des **« méthodes »**

```
void particule::adv_pos (double dt) {
        pos.x += vel.x*dt;
        pos.y += vel.y*dt;
        pos.z += vel.z*dt;
}
```

utilisation

```
int main()
{
   particule part[100];
   double   dtime;
   .....
   for (i=0;i<100;i++)
   {
        part[i].adv_pos(dt);
   }
}
```

La classe particule possède 4 **membres**: 3 variables (pos, vel et mass) et une méthode, ou "fonction " (adv_pos).

Encapsulation de l'objet

```
class particule {
        private:
           vecteur3D   pos,vel;
           double        mass;
        public:
           void adv_pos (double);
};
```

Les membres d'une classe peuvent avoir plusieurs statuts:

- **private**: ils ne sont accessibles qu'aux autres membres de la classe. **C'est le statut par défaut**
- **public**: ils sont accessibles partout où l'objet est déclaré.
- **protected**: ils ne sont accessibles aux membres de la classe et des classes descendantes.

Dans notre exemple, la valeur de mass est privée, elle n'est pas accessible depuis l'extérieur de la classe:

```
int main()
{
   particule part[100];
   double   dtime;

   part[50].mass += 0.01;
}
```

<== n'a pas de sens!

Pourquoi mettre un membre de classe en « private » ?

Précisons tout d'abord que « private » est le statut **par défaut** d'un membre.

- Un membre « private » est protégé contre une modification incontrôlée de sa valeur
- On peut quand même y accéder en définissant des méthodes spéciales:

```
class particule {
        private:
           double       mass;
        public:
           void set_mass (double);
           double get_mass (void);
};

void particule::set_mass(double x) {
mass=x;
}

double particule::get_mass(void){
return mass;
}
```

Quel intérêt ?? Juste éviter les modifs incontrôlées ??

Non!

On pourrait, dans **set_mass**, vérifier que la mass est bien positive! Si un autre programmateur utilise **set_mass**, il n'aura pas besoin de faire le contrôle lui-même!

On obtient un code **plus robuste** pour le développement à plusieurs.

Les fonctions amies d'une classe

Il s'agit d'une autre manière d'**agir sur les membres « private »** d'une classe.

On déclare dans la classe, la liste des **fonctions « amies »**, celles-ci auront un accès illimité aux membres de la classe. Exemple:

```
class particule {
        friend double Ec(int);

         vecteur3D   pos,vel;
         double        mass;
};

double Ec(int i) {
         Ec=0.5*part[i].mass*(pow(part[i].vel.x,2)
           +pow(part[i].vel.y,2)+pow(part[i].vel.z,2));
         return;
}

int main() {
particule part[100];

.....
for(i=0;i<100;i++) { cout << Ec(i) << endl;}
.....
}
```

Il ne faut **pas abuser** des fonctions amies: cela va à l'encontre de l'esprit de la programmation objet.

On peut définir la **classe B** comme **amie** dans la **classe A**: toutes les méthodes de B sont alors amies de A:

```
class particule {
         friend class diagnostic;
         vecteur3D   pos,vel;
         double        mass;
};
class diagnostic {
         public:
         double Ec(int);
         double Ep(int);
};
```

Constructeur et destructeur d'objet

Il existe 2 **méthodes spéciales** pour chaque classe, le **constructeur** et le **destructeur**. Le mon des fonctions correspondantes n'est **pas** libre.

Le constructeur:

Il est appelé **automatiquement** à chaque fois qu'on déclare un objet (instanciation) de la classe.

```
class particule {
        private:
            vecteur3D   pos,vel;
            double      mass;
        public:
            particule (double);    // Constructeur
                                       // pas de type !!
};

particule::particule(double m)
{
        mass = m;
}                                  // pas de ; !!

particule part(1.);              // Déclaration et appel
                                   // du constructeur
```

Le destructeur:

Il est appelé **automatiquement** quand la ***portée*** l'objet s'achève, ou, pour allocation dynamique, au moment de l'appel de ***delete***.

```
class particule {
        private:
            vecteur3D   pos,vel;
            double      mass;
        public:
            ~particule (void);    // Destructeur
                                       // pas de type !!
                                       // pas d'argument !!
};

particule::~particule(double m)
{
        cout << "Une particule détruite! " << endl ;
}                                  // pas de ; !!
```

Surcharge de fonction.

Il est possible de donner **plusieurs** définitions d'une **même** fonction qui diffèrent par le type et le nombre d'arguments. Cela s'appelle la surcharge de fonction. Exemple:

```
#include <string>

void alarme()
{
        cout << "Alarme! " << endl;
}
void alarme(string message}
{
        cout<<"Alarme : " << message << endl;
}
```

Suivant le contexte (type et nombre d'arguments passés à la fonction) le compilateur détermine quelle version appeler.

La surcharge de fonction peut s'appliquer aux fonctions membres de classes.

Surcharge d'opérateur.

```
typedef struct {
      double x;
      double y;
      double z; }
vecteur3D;

class particule {
          private:
          vecteur3D   pos,vel;
          double        mass;
          public:
          particule operator+ (particule);
};

particule particule::operator+(particule r) {
          particule resultat;
          resultat.pos.x=(pos.x*mass+r.pos.x*r.mass)/(mass+r.mass);
          resultat.pos.y=(pos.y*mass+r.pos.y*r.mass)/(mass+r.mass);
          resultat.pos.z=(pos.z*mass+r.pos.z*r.mass)/(mass+r.mass);
          resultat.vel.x=(vel.x*mass+r.vel.x*r.mass)/(mass+r.mass);
          resultat.vel.y=(vel.y*mass+r.vel.y*r.mass)/(mass+r.mass);
          resultat.vel.z=(vel.z*mass+r.vel.z*r.mass)/(mass+r.mass);
          resultat.mass=mass+r.mass;
          return(resultat);
}
int main() {
particule a,b,c;
...
a=b+c;                              /* valide  */
a=b.operator+(c);                /* également valide  */
....
}
```

Il est possible d'**ajouter une définition** (surcharger) aux opérateur classiques (=, +, -, *, /, %, &&, ||, ==, <, ...) pour qu'ils soient capables d'agir sur les objets d'une classe.

A gauche, exemple de surcharge de +, à l'aide d'une méthode de la classe. Il est aussi possible d'utiliser une fonction externe.

L'opérateur = possède une définition par défaut !! Il copie les membres un à un dans les membres correspondants.

Héritage de classe

Il s'agit d'un aspect essentiel de C++. A partir d'une **classe de base** (ou classe mère) on peut définir des **classes dérivées** (ou classes filles):

- Les classes dérivées possèdent automatiquement les membres de la classe mère.
- Les classes dérivées peuvent définir les propres membres, mais aussi redéfinir les méthodes de la classe mère.

```
class particule {
        plublic:
            vecteur3D   pos,vel;
        protected:
            double        mass;
        public:
            void adv_pos (double);
};

class particule_gaz : public particule {
        protected:
            double   temperature;
            double   densite;
        public:
            double pression();
};
```

} Classe mère

} Classe fille:

NB: le membre **mass** de particule_gaz est accessible par les méthodes de particule et particule_gaz, c'est tout!

Fin