partie 4 : Héritage

MOOC Langage C++ INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 classes imbriquées

Il est possible de définir une classe à l'intérieur d'une autre classe.

```
class A {
public:
  classe B { ...};
};
int main(){
  A a;
  A::B b;
  return 0;
}
```

En pratique, cela est très peu utilisé.

2 Héritage et instanciation

Les notions introduites dans cette partie permettent au C++ d'améliorer la réutilisabilité et l'évolutivité des programmes.

2.1 Quelques définitions

2.1.1 Classification

La classification permet de supprimer les détails des instances. Ceci accentue les propriétés de la classe dans son ensemble. Par exemple, "La twingo de V." est une instance d'un "Véhicule à moteur". L'instanciation permet de caractériser "La twingo de V." dans l'ensemble des "Véhicules à moteur".

2.1.2 Généralisation

Cette action supprime les `différences` entre les `catégories` et accentue leurs propriétés `communes`.

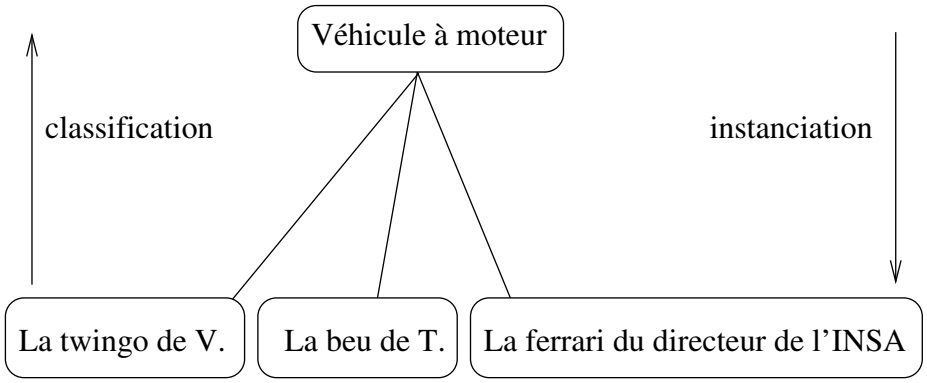

FIGURE 1 – Classification et Instanciation

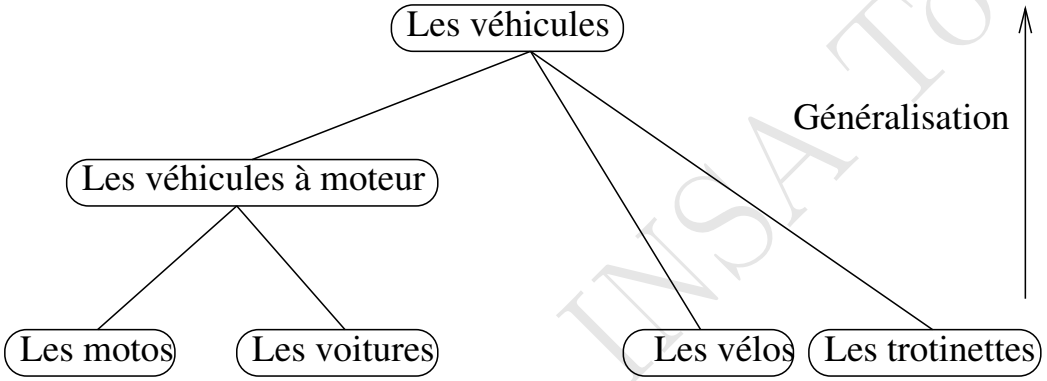

FIGURE 2 – Généralisation

2.1.3 Application au C++

L'héritage est le mécanisme d'abstraction qui permet au C++ de mettre en œuvre la généralisation. Cette relation correspond à la relation "**est un**" ou "**est une sorte de**". D'un point de vue ensembliste, ceci revient à dire que la classe A hérite de la classe B si tout objet de A peut être considéré comme un objet de B ($A \subset B$). Si on s'intéresse aux propriétés des classes, ceci revient à dire que les objets de A vérifient au moins les propriétés de B. Les différentes terminologies rencontrées sont :

- A hérite de B,
- B est un ancêtre de A,
- A est une sous-classe de B,
- B est une superclasse de A,
- A est une classe dérivée de B,
- B est une classe de base de A.

Dans l'exemple de la figure, ceci revient à dire que la classe "Les véhicules" généralise les classes "Les véhicules à moteur" et "Les vélos".

L'héritage en C++ peut être multiple. Dans l'exemple de la figure **??**, la classe "Les véhicules à moteur" est vue comme une spécialisation des classes "Les moteurs" et "Les véhicules".

2.2 Héritage simple

L'héritage implique qu'un nom (membre, opération) n'est pas toujours défini dans la classe mais peut être défini dans un des ancêtres de cette classe. Dans le cas de l'héritage simple, il suffit de

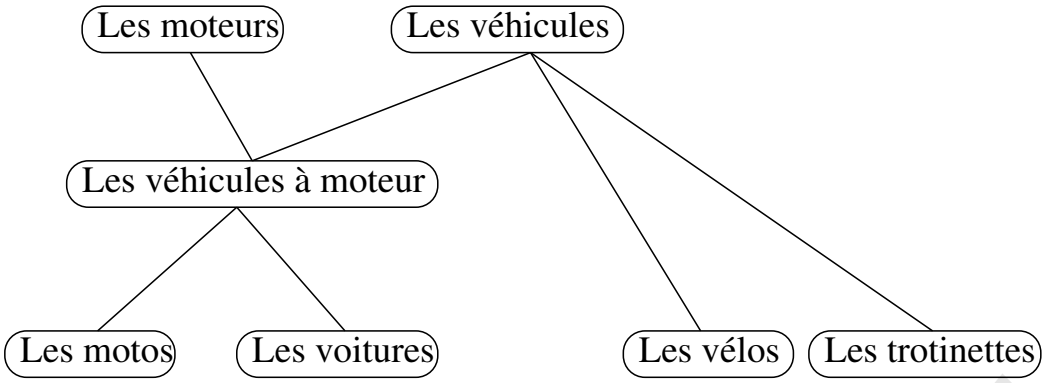

FIGURE 3 – Héritage multiple

parcourir les ancêtres jusqu'à ce que le nom soit défini (pour l'héritage multiple se pose le problème de plusieurs définitions). Cette recherche est effectuée à la compilation, ceci n'a donc pas d'influence sur les performances.

La syntaxe est la suivante :

```
class nom_classe_mère {
...
};
class nom_classe_fille : nom_classe_mère {
...// définition éventuelle de nouveaux membres et opérations
};
```

2.2.1 Contrôle d'accès

On rappelle que les attributs et les méthodes déclarés comme `protected` dans une classe mère sont accessibles dans les classes filles uniquement. La classe dérivée doit aussi préciser le contrôle d'accès des membres hérités. On a :

- dans tous les cas, seule les parties `public` et `protected` sont accessibles aux classes dérivées,
- l'héritage est dit public si les membres `public` et `protected` de la classe de base sont respectivement `public` et `protected` dans la classe dérivée,
- l'héritage est dit privé si les membres `public` et `protected` sont privés dans la classe dérivée,
- l'héritage est dit protégé si les membres `public` sont protégés dans la classe dérivée,
- par défaut, l'héritage est privé pour une classe.

Exemple:

```
class A {
     public:
          int pub;
     protected:
          int pro;
     private:
          int pri;
};
class B: public A {};
// pub public, pro protégé, pri inaccessible
class C: private A {};
// pub et pro privés, pri inaccessible
class D: protected A {
     public:
// pub et pro protégés, pri inaccessible
};
int main{
     A a; // a.pub OK
     B b; // b.pub OK
     C c; // tout est inaccessible
     D d; // tout est inaccessible
     ...
}
```

2.2.2 Constructeur des classes dérivées

Une classe dérivée est constituée de la classe de base à laquelle on ajoute quelque chose. Pour construire l'objet de la classe dérivée, il faudra donc appeler le constructeur de la classe de base, puis construire les données locales à l'objet dérivé. On peut toutefois appeler un ou plusieurs constructeurs particuliers par le mécanisme des listes d'initialisation de membres :

```
class base {
     //...
     public:
          base(int);
          base();
}
class derivee: public base {
     int a;      public:
          derivee(int i): a(i),base(i){ /* .. */};
          derivee(){ /* ... */};
}
```

Le premier constructeur de `derivee` fera appel au constructeur `a(int)` puis `base(int)` avant d'exécuter le code associé au constructeur.

2.2.3 Exemple avec les complexes

Un complexe (`math_complexe`) est représenté par un couple de flottants de manière générale (`float_pair`). Un couple de flottants peut aussi servir à d'autres choses en mathématiques (point géométrique : `math_point`, vecteur par rapport à l'origine : `math_vecteur`, etc). De plus, on peut extraire un certain nombre de propriétés communes pour ces différentes utilisations : la saisie (read), l'affichage (write), la norme (norme), les opérateurs += et -= , etc (voir figure **??**).

La multiplication ne peut pas être définie dans `float_pair` car elle n'a pas de définition générale permettant à cet opérateur d'être directement utilisé dans les classes dérivées. Par contre, l'opérateur de multiplication a un sens pour les `math_complexe`.

À partir de `math_complexe`, on peut dériver une nouvelle classe de complexes offrant des services tels que la protection des opérations, la sauvegarde, etc.

Exemple:

Voici un code possible :

```
#include <assert.h>
#include <iostream>
#include <math.h>
using namespace std; class float_pair {
public:
     float_pair() {};
     float_pair(float x,float y=0): r(x),i(y) {}
     float_pair(float_pair &autre): r(autre.r),i(autre.i){}
     float first() const { return r;}
     float second() const { return i;}
     float norme() {
          return sqrt (r*r + i*i);
     }
     float_pair& operator += (const float_pair & autre) {
          set(r+autre.r,r+autre.i);
          return (*this);
```

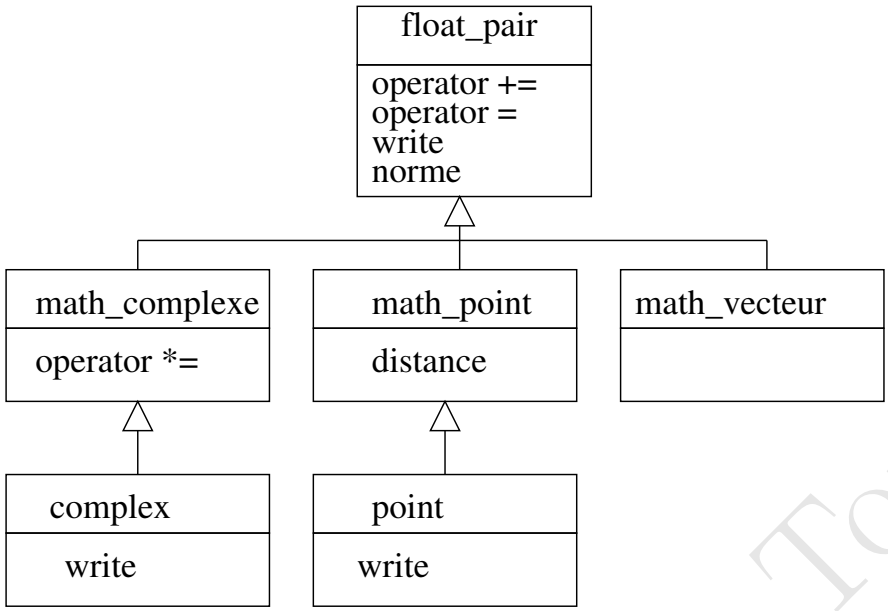

FIGURE 4 – Exemple pour les complexes

```
      }
      float_pair& operator -= (const float_pair & autre) {
            set(r-autre.r,i-autre.i);
            return (*this);
      }
protected :
      void set(float x,float y){r=x;i=y;}
      float r,i;
};
```

Les fonctions et opérateurs extérieurs à `float_pair` :

```
int operator == (const float_pair& un, const float_pair& autre) {
      return (un.first()== autre.first()) && (un.second() == autre.second());
}
float_pair operator+(float_pair& un,float_pair&autre) {
      return float_pair(un.first()+autre.first(), un.second()+autre.second());
}
float_pair operator-(float_pair& un,float_pair&autre) {
      return float_pair(un.first()-autre.first(),un.second()-autre.second());
}
```

La classe math_complexe :

```
class math_complexe : public float_pair {
public:
      math_complexe(math_complexe& autre) :float_pair(autre) {}
      math_complexe() : float_pair() {}
      math_complexe(float x,float y=0) : float_pair(x,y) {}
      float reel() const {//**Accès aux définitions de la classe de base
            return float_pair::first();}
      float imag() const {//**Accès aux définitions de la classe de base
```

```
        return float_pair::second();}
    math_complexe& operator *= (const math_complexe& autre) {
        r = (r*autre.r-i*autre.i);
        i = (r*autre.i+i*autre.r);
        return (*this);
    }
};
```

Les fonctions et opérateurs extérieurs à math_complexe :

```
  /* ==, +, et - sont les mêmes */
math_complexe operator*(math_complexe& un,math_complexe&autre) {
    return math_complexe(un.reel()*autre.reel()- un.imag()*autre.imag(),
    un.reel()*autre.imag()+un.imag()*autre.reel());
}
math_complexe operator-(math_complexe& un){
    return math_complexe(-un.reel(),-un.imag());}
```

Par polymorphisme d'inclusion, `math_complexe` est incluse dans `float_pair`, il est possible :

- d'écrire '`math_complexe z(1,1);`
  `float_pair mt1=z,mt2(z),mt3; mt3=z;`'.
- d'utiliser l'opérateur '`+=`' membre de `float_pair` (défini avec un paramètre de type `float_pair`) pour les complexes :
  '`math_complexe z(1,1); z += z; z -= z;`'
  Ce qui est équivalent à l'appel '`z.operator+=(z);`'.
- d'utiliser l'opérateur '`+`' (défini avec deux paramètres dans `float_pair`) avec des paramètres de type `math_complexe` :
  '`float_pair x; math_complexe z(1,1); x= z + z;`'
  On notera que le résultat est un `float_pair`. Un `math_complexe` peut être vu comme un `float_pair`, mais pas le contraire.

Par contre, il n'est pas correct :

- d'écrire '`float_pair mt(1,1);math_complexe z3; z3=mt;`'
  `z3=mt` équivaut à écrire `z3.operator =(mt)`, c'est donc l'égalité entre `math_complexe` qui est utilisée. Il faudrait donc pouvoir transformer `mt` en `math_complexe`, ce qui n'existe pas par défaut
- d'écrire '`z=z+z;`'.
  On utilise le "+" des `float_pair`. `z` est vu comme un `float_pair` grâce à l'héritage. Par contre, le résultat est un `float_pair` que l'on ne peut pas transformer en `math_complexe` pour l'opérateur d'affectation.

Pour cela, il faut surcharger par un paramètre '`float_pair`' :

1. l'opérateur d'affectation dans `math_complexe`
   '`math_complexe& operator= (float_pair& x){...}`'
2. ou le constructeur de '`math_complexe`'
   '`math_complexe(float_pair& x):float_pair(x),...{...}`'

La solution 2 est la meilleure car elle résoud en même temps le problème d'utilisation de l'opérateur '`+`' :

'`math_complexe z(1,1), t = z + z, u(z+z);`'
'`float_pair mt(1,1);math_complexe z1=mt,z2(mt);`'

2.3 Héritage multiple

On parle d'héritage multiple lorsqu'une même classe est définie par l'héritage direct de plusieurs autres classes. Au niveau de la syntaxe, cela revient à donner une liste des classes héritées avec le statut (private, etc) de l'héritage.

Exemple:

```
class vehicules_moteur: public vehicules, public moteur {
     //...
}
```

Avec l'héritage multiple, il est possible d'hériter de plusieurs membres de même nom et de même prototype. Dans ce cas, à l'utilisation il faut préciser la classe d'origine avec l'opérateur de portée (" : :") si l'on veut utiliser un membre particulier. Si l'opérateur de portée n'est pas utilisé, le C++ tentera de faire un choix du membre à utiliser. La stratégie utilisée diffère selon les langages objet. En C++, le membre le plus proche dans l'arbre de descendance sera choisi, s'il demeure une ambiguïté, il y a rejet (cas sans domination). En résumé, il vaut mieux utiliser l'opérateur de portée.

3 Les amis

Dans certain cas, les règles de contrôle d'accès ne sont pas suffisantes :

— on veut qu'une fonction ait accès à tous les membres de la classe sans pour autant être membre de la classe,

— une fonction d'une classe doit pouvoir accéder aux membres d'une autre classe.

Dans ces exemples, on va définir la fonction comme amie (`friend`) de la classe. **Exemple:**

```
class matrix {
...
     friend vector multiply (...);
};
class vector {
...
     friend vector multiply (...);
};
vector multiply(...) { ... }
```

L'utilisation du mot clé `friend` est limité au même namespace. Autrement dit, des amis doivent être dans le même espace de noms. On peut aussi dire que toutes les méthodes d'une classe A peuvent accéder aux attributs privés d'une classe B en déclarant la classe A ami :" friend class A ; " dans la classe B.

4 Fonctions et variables statiques

4.1 Variables de classe

Des attributs peuvent être définis comme statiques. Cela signifie que ces données seront partagées par tous les objets de la classe. On parle alors de variables de classe.

Exemple:

```
class x{
     private:
```

```
          static int cpt; // utilisation pour compter les instances
          ...
     public:
          x() {ctp++;} // compteur global incrémenté à chaque création
          ...
};
int x::cpt=0; // déclaration extérieure obligatoire
```

Les champs statiques doivent aussi être déclarés à l'extérieure de la classe en utilisant l'opérateur de portée, peu importe à ce niveau le contrôle d'accès sur l'attribut. Dans l'exemple, la valeur du compteur "cpt" donne le nombre d'instances de la classe x créée dans le programme utilisant cette classe.

4.2 Fonctions statiques

Nous avons vu dans le chapitre précédent comment définir des variables d'instance, on peut faire de même avec les méthodes d'une classe. Ces méthodes s'appliquent donc à la classe et sont indépendantes d'une instance particulière de la classe. Elles ne peuvent en particulier manipuler que les variables d'instance d'une classe. L'appel à ces méthodes ne se fera pas en utilisant une instance particulière mais le nom de la classe elle-même, associé à l'opérateur de portée. **Exemple:**

```
class test {
...
     static void mon_nom(){
          cout <<"je suis la classe test"<<endl;
     }
...
};
int main() {
     test t1;
     test::mon_nom();
... }
```

5 Fonctions membres constantes

La notion de constante est élargie aux classes. On peut notamment préciser qu'un paramètre d'une méthode sera traitée comme une constante. Ceci a l'avantage de permettre au compilateur de vérifier pour le programmeur qu'un paramètre défini comme paramètre d'entrée n'est pas modifié dans une méthode.

```
class point {
  int x,y;
  public:
   point(int i,int j):x(i),y(j){}
   void affiche(){
     cout << '('<<x<<','<<y<<')'<<endl;
   }
};
void coucou(const int i, const point p){
// erreur de compilation si on essaye de modifier i,
// par exemple i=2
cout << i  ;
...
}
```

```
int main(){
 int k=2;
 point p1(2,4);
 coucou(k,p1);
}
```

Il fait de même pour un objet passé en paramètre. Néanmoins, si dans la fonction manipulant l'objet, on fait appel à une méthode de l'objet, il faut aussi garantir que cette méthode ne modifie pas l'objet. Le compilateur ne sait pas faire cette vérification seul, il va falloir explicitement lui dire que c'est une fonction membre constante. Ce qui donne dans l'exemple précédent :

```
class point {
  int x,y;
  public:
    point(int i,int j):x(i),y(j){}
    void affiche() const { /* le const interdit par la
    suite de modifier les attributs x et y */
      cout << '('<<x<<','<<y<<')'<<endl;
    }
};
void coucou(const int i, const point p){
     /* erreur de compilation si on essaye de modifier i,
     par exemple i=2 */
     cout << i  ;
     p.affiche(); /* ok car dans la classe point il est
     précisé que point ne modifie pas l'objet*/
}
int main(){
  int k=2;
  point p1(2,4);
  coucou(k,p1);
}
```

Remarque:
On peut assouplir la vérification faite sur la modification de certains attributs par une fonction membre constante avec le qualificatif "`mutable`".