Fonctions virtuelles

Polymorphisme d'héritage : voir une classe avec différents types. Principe de Liskov : pouvoir substituer une classe par une autre.

Principe d'encapsulation

Voyons avec une variable membre :

```
class Base {
public:
    int i {};
};

class Derived : public Base {
public:
    int j {};
};
```

Si on souhaite utiliser ces classes, on peut accéder aux membres de la façon suivante :

```
Derived d;
d.j; // ok
d.i; // ok, par héritage

Base b;
b.i;

Base& b1 = d;
b1.i; // ok
b1.j; // erreur, type statique est Base, j n'existe pas dans ce contexte
```

On voit ici un problème majeur : si on définit des variables membres publiques et qu'on les utilise en dehors des classes, on a besoin de

connaître le type réel pour utiliser ces membres. Autrement dit, si on crée une classe Derived, on a besoin de savoir que l'on a créé ce type d'objet pour accéder à la variable membre j.

On ne respecte pas le principe de substitution de Liskov, puisque que si on a besoin de connaître le type réel de l'objet, on perd la substituabilité. La classe ne respecte pas Liskov.

On comprend mieux un autre principe expliqué plus tôt, le principe d'encapsulation. Si on pense en termes de données encapsulées, on rencontre un problème de substituabilité. Il faut donc penser en termes de services rendus (càs l'ensemble de fonction public) et non en termes de données. On dit que “on encapsule des services, pas des données”.

Nous allons voir que, contrairement aux variables membres, les fonctions membres peuvent avoir un comportement qui s'adapte au type réel des objets (type dynamique) et non au type statique (le type de la variable qui contient l'objet).

Fonctions virtuelles

Rappel : type statique et dynamique

```
Derived d;
Base& b = d;
```

Type réel de l'objet (ie le type d'objet créé en mémoire) = Derived. Le type static (ie de la variable, càd le type vu du point de vu compilateur et donc les fonctions et variables membres que l'on peut appeler) = Base&.

- masquage de fonctions
- destructeur virtuel
- MyClass::foo pour appeler explictement le parent

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++