pas bien de ne pas parler de destructeur virtuel plus tot ?

Polymorphisme et substitution

Polymorphisme d'héritage

On a vu que l'héritage publique est une relation EST-UN. Si la classe A dérive de la classe B, alors A EST-UN B. Mais A est également un A. Selon le contexte, on va donc pouvoir manipuler un objet de type A comme si c'était un objet de type A (normal), mais également comme si c'était un objet de type B. C'est cette capacité d'un objet a avoir plusieurs types que l'on appelle le polymorphisme d'héritage.

Dans la sémantique de valeur, nous avons vu qu'il était possible de convertir une valeur d'un type donné dans un autre type.

```
int i { 123 };
double d { i }; // d = 123.0
```

Dans ce cas, même si ces deux valeurs seront identiques à l'affichage, on a bien affaire à des valeurs différentes, définies dans deux ensembles différents (N et R). En particulier, certaines opérations, comme `++` ou `--`, n'auront de sens qu'avec la valeur de type `int` et pas la valeur de type `double`. Les valeurs ne sont pas substituables.

Avec le polymorphisme d'héritage, la situation est différente. Si l'on a une hiérarchie de classes :

```
class Base {};
class Derived : public base {};
```

On peut manipuler un objet de type `Derived` comme si c'était un `Base`. Comment activer le polymorphisme ? On a déjà vu les références, qui permettent de créer une variable “qui fait référence” à une autre variable.

```
Derived d;
Base & b { d }; // création d'une référence
```

Dans ce code, b “fait référence” à un objet de type `Derived`. Pour autant, `b` est de type `Base`. Si on essaie d'appeler une fonction de la classe `Derived` dans `b`, le compilateur ne sait pas que l'objet référencé n'est pas un B (mais il peut... confu comme explications).

On parle de type statique et de type dynamique :

- le type statique d'une variable est le type que l'on utilise pour définir cette variable. Dans le code précédent, la variable `d` est de type statique `Derived`, la variable `b` est de type statique `Base`.
- le type dynamique d'une variable est le type réel d'un objet. Par exemple, dans le code précédent, la variable `b` fait référence à un objet de type dynamique `Derived`.

Remarque : si l'on crée un objet de type `Base`, celui-ci ne pourra jamais être vu comme un objet de type `Derived`.

Cela a un impact sur la façon dont on voit les objets. Si on manipule un objet A qui peut être dérivé (donc qui possède une sémantique d'entité), on ne peut pas savoir s'il peut être en fait un objet de type B, C D, etc.

Principe de substitution de Liskov

: possible de voir selon le contexte une classe ou son parent. Ce notion de voir une même chose différemment selon le contexte est appelé “polymorphisme”. Ici, plus précisément, polymorphisme d'héritage.

Pour rappel, déjà vu d'autres formes de polymorphisme :

- plusieurs fonctions qui ont le même nom, mais des paramètres différents : polymorphisme ;
- plusieurs fonction de même nom et même liste de paramètres,

dans une hiérarchie de classe (cf prochain chapitre) : polymorphisme;

Pour activer le polymorphisme, besoin que la classe soit passé par référence ou pointeur.

```
Base c1;
Derived c2;

Base& poly1 = c1;
Base& poly2 = c2;
```

Dans les 2 cas, quelque soit la classe réelle, contient dans une variable de type Base.

Pointeur unique

Pointeur partagé

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++