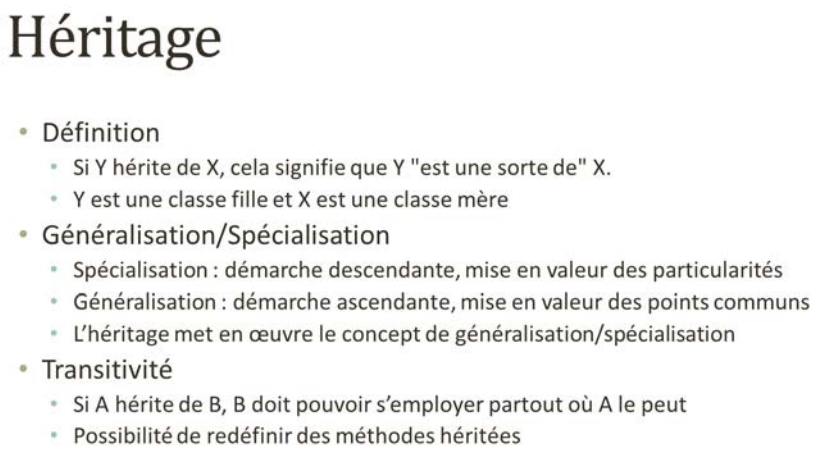
Chapitre 4. Héritage et polymorphisme en C++

Notion d'héritage

Rappels et définition sur l'héritage

La notion d'héritage permet d'exprimer les similitudes entre classes.



Slide 2

Le lien d'héritage correspond aussi à ce qu'on appelle une association « Is-a »
Une classe fille peut aussi être appelée une sous-classe et une classe mère une super-classe.

La spécialisation permet d'étendre les propriétés d'une classe sous forme de sous-classes plus spécifiques.
La généralisation permet de factoriser les propriétés d'un ensemble de classes sous forme d'une super-classe plus abstraite.

Exemple : Un Employé est une Personne (lien d'héritage) ; on peut dire que la classe Employé est une spécialisation de la classe Personne.

Héritage en C++

- S'exprime lors de la déclaration de la classe fille
- La classe héritée ne peut accéder aux membres privés de la classe de base

```
class Point {
  int x,y ; //Attributs non accessibles à la classe fille
public :
  //…
}

class Pixel : public Point {
  Couleur couleur;
public :
enum Couleur {NOIR, ROUGE, BLANC} ;
Pixel (int nAbs, int nOrd, Couleur C) ; //Constructeur
void colorier(Couleur c=NOIR) ; //Méthode
} ;
```

Slide 3

On peut déclarer certains attributs de la classe mère sous forme « protected », mais ce n'est pas forcément très « propre ».

Il y a 3 types d'héritage possibles...

Types d'héritage en C++ 1/2

- Comme pour un membre, une classe de base peut être déclarée private, protected ou public
  - `public` : les membres publics de la classe mère peuvent être utilisés par toute fonction ; ses membres protégés peuvent être utilisés par les membres, les fonctions amies et les classes dérivées de la classe fille.
  - `protected` : les membres publics et protégés de la classe mère ne peuvent être utilisés que par les fonctions amies, les fonctions membres et les fonctions dérivées de la classe fille.
  - `private` : les membres publics et protégés de la classe mère ne peuvent être utilisés que par les fonctions amies et fonctions membres de la classe fille.

slide 4

Types d'héritage en C++ 2/2

Classe de base			Dérivée public		Dérivée protected		Dérivée private	
Statut initial	Accès FMA	Accès User	Nouveau statut	Accès User	Nouveau statut	Accès User	Nouveau statut	Accès User
public	O	O	public	O	protected	N	private	N
protected	O	N	protected	N	protected	N	private	N
private	O	N	private	N	private	N	private	N

FMA : fonctions membres ou amies

Slide 5

Différence entre protected et private lors de la dérivation de la classe héritière.

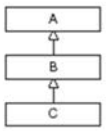
Être vivant
^
Mammifère
^
Félin
^
Chat

Un chat est un félin, est un mammifère, est un être vivant.
Relation à un seul sens : un être vivant n'est pas un chat (au sens informatique/ontologique)
Un chat est un félin seulement si la dérivation a été **public**

Héritage, constructeurs et destructeurs

Constructeurs et destructeurs

- Les constructeurs par défaut sont invoqués implicitement
- A la création :
  - Appel de A() puis B() puis C()
- A la destruction :
  - Appel de ~C() puis ~B() puis ~A()



- Arguments : si un constructeur a des arguments, il faut préciser dans le corps des constructeurs des classes dérivées les arguments à lui transmettre
- Attention : l'héritage du constructeur n'est pas toujours aussi simple, notamment lorsqu'on enrichit un type construit.

Slide 6

Selon le schéma de classe, C hérite de B qui hérite de A, donc, à la création, appel de A() puis B() puis C() (du moins spécifique au plus spécifique) et à la destruction, on inverse : on libère du plus spécifique au moins spécifique.
Attention aux arguments dans les constructeurs : si un constructeur des classes mères a des arguments, il faut préciser dans le corps des constructeurs des classes dérivées (des classes filles) les arguments à lui transmettre.

Attention : l'héritage du constructeur n'est pas toujours aussi simple, notamment lorsqu'on enrichit un type construit. Voir cours de C++ avancé. La norme 2011 permet de clarifier les héritages de constructeurs venant, par exemple, de conteneurs de la librairie standard.

Exemple


Exemple

Slide 7

```
#include<iostream>
using namespace std;

enum Couleur {ROUGE, VERT, BLEU} ;
class Point {
    int x,y ;
public :
    Point(int xx, int yy) : x(xx), y(yy) {}
    const void affiche() {std::cout  << x << " " << y << std::endl;}
};

class Pixel : public Point {
    Couleur coul;
public :
    Pixel(int xx, int yy, Couleur c) : Point(xx,yy), coul(c) {} ;
    void colorier(Couleur c=VERT) ; // Méthode inline
};

int main () {
    Point p(1,2);
    p.affiche(); // (1) 1 2
    Pixel pix(3,4,ROUGE);
    pix.colorier(); // valeur par défaut : VERT
    pix.affiche(); // (2) 3 4 // méthode affiche héritée de Point : ne traite pas la couleur
    return 0;}
```

Remarque : il faudrait redéfinir la méthode affiche pour la classe Pixel afin de pouvoir afficher la couleur en plus des coordonnées.

Attention aux différences Java/C++

En Java, on avait...

- Mot-clé pour définir une classe dérivée
  - extends
- Mot-clé pour empêcher l'héritage
  - final
- Types d'héritages
  - identiques au public du C++

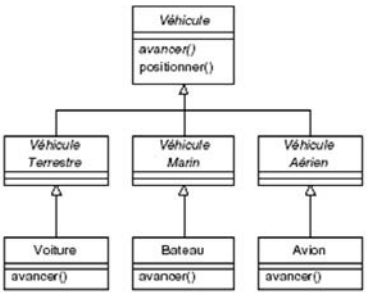
Slide 8

Notion de Polymorphisme

Le polymorphisme est l'aptitude des objets à réagir différemment à un même message.
Cette capacité des objets à prendre plusieurs formes est l'un des principaux concepts de la programmation objet.
Intérêt du polymorphisme : permet par exemple de gérer une collection d'objets de façon homogène, tout en conservant le comportement propre à chaque objet.

Polymorphisme

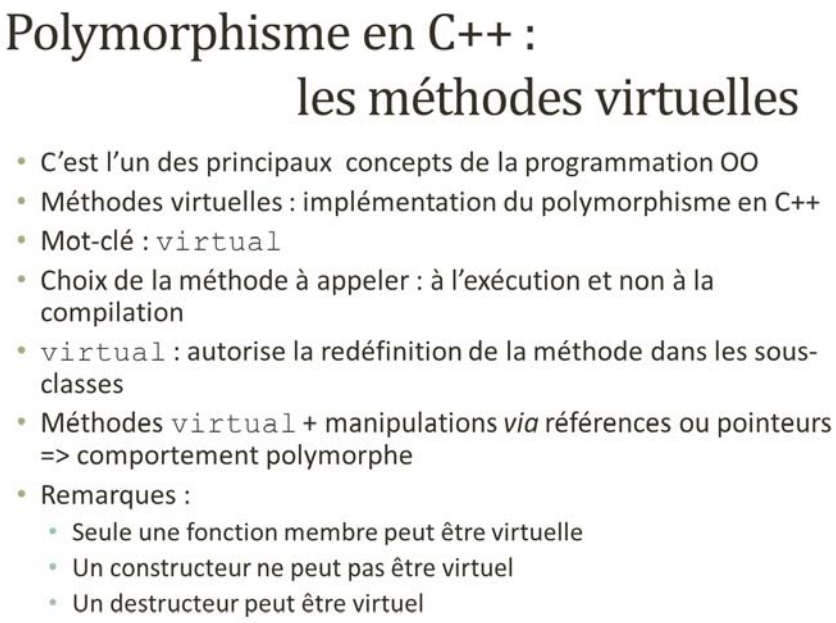
- Aptitude des objets à réagir différemment à un même message
- Intérêt : gestion homogène d'une collection d'objets en conservant des comportements spécifiques
- Possibilité de redéfinir une méthode héritée (≠ surcharge) pour la spécialiser
- Choix de la méthode à appeler : à l'exécution



Slide 9

Attention : redéfinir n'est pas surcharger. La fonction garde la même en-tête (même nombre et même type de paramètres), mais on peut en changer le corps pour la spécialiser. Le choix de la méthode à appeler se fait à l'exécution, en fonction du type de l'objet.
Attention à la différence entre la ligature/ le typage statique (à la compilation) vs le typage dynamique, qui se fait à l'exécution.

Comment pratiquer le polymorphisme en C++ ?



Slide 10

Exemple de polymorphisme :
Les mammifères sont des animaux, les oiseaux également. On peut définir une super-classe Animal.
Ces deux types d'animaux n'ont pas la même façon de manger, mais on utilise le même verbe pour désigner l'action.
Un zoo aura besoin de stocker les données concernant ces animaux et de les nourrir indépendamment de leur espèce.
Afin de pouvoir garder les information sur tous les animaux, on utilisera des pointeurs sur le type Animal, et on définira une méthode virtuelle manger.
Les pointeurs pourront être initialisés indifféremment par tous les héritiers de la classe Animal, et la méthode manger *ad hoc* sera choisie par le compilateur.
Je peux donc virtuellement faire manger un animal sans savoir qui il est ! Le zoo aura un comportement polymorphe.

En C++, on met le polymorphisme en œuvre en utilisant des méthodes virtuelles (donc qu'on peut redéfinir) et en manipulant les objets *via* des pointeurs ou des références.

Le polymorphisme vient du fait qu'un pointeur sur une instance d'une classe de base peut également pointer sur toute instance de sous-classe et comme les méthodes peuvent être redéfinies, leur comportement s'adapte.
Remarque : seule une fonction membre peut être virtuelle. Un constructeur ne peut pas être virtuel, mais un destructeur peut l'être.

Exemple de méthode virtuelle

```
class Point {
  int x,y ; //Attributs

public :
  virtual void affiche() const {std ::cout<<"x=" <<x <<", y=" <<y;}
//…
};

class Pixel : public Point {
  Couleur couleur;

public :
void affiche() const ; //inutile de répéter virtual
//…
} ;

void Pixel::affiche() const {
  Point::affiche() ; //appel de la méthode de la classe de base
  std::cout << ", couleur = " <<couleur;
}
```

Slide 11

Lorsque la classe Pixel hérite de la classe Point, elle hérite des attributs (x et y, entiers) et des méthodes (constructeur, destructeur et void affiche).
Dans la classe Point, la méthode affiche est virtuelle : sa redéfinition dans les classes filles est autorisée.
Dans la classe Pixel, affiche a exactement le même prototype, mais elle est spécialisée : elle appelle la méthode affiche héritée de la classe Point ET affiche l'attribut supplémentaire de la classe Pixel.

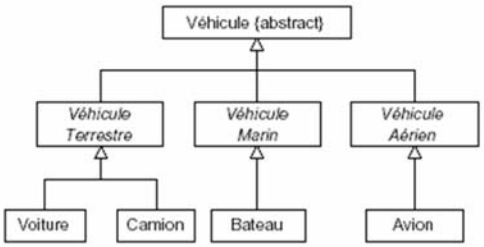
Notion de classe abstraite

Les classes abstraites représentent un concept abstrait qui ne peut être instancié : le niveau de généralisation est trop important pour permettre d'implémenter leur comportement. Elles servent alors comme classes de base (super-classes) dans une hiérarchie d'héritage.
NB : les classes abstraites permettent de regrouper des objets ayant un comportement commun.

Les classes abstraites ne sont pas instanciables, elles forment un cadre de référence pour les descendants (homogénéisation des caractéristiques).

Classes abstraites

- Représentent un concept abstrait qui ne peut être instancié (*e.g.* : véhicule, nourriture,…)
- Utilisées comme classes de base dans une hiérarchie d'héritage
- Permettent de regrouper des objets ayant un comportement commun



Slide 12

Concrètement, en C++

Une classe est abstraite si elle contient au moins une méthode virtuelle pure.
Une méthode virtuelle pure est une méthode virtuelle dont on ne fournit pas l'implémentation, et qui doit donc obligatoirement être redéfinie dans les descendants (c.f. le =0).

Classes abstraites en C++ (1/2)

- Non instanciables : cadre de référence
- Contient au moins une méthode virtuelle pure (pas d'implémentation pour cette méthode, qui devra être redéfinie)

Exemple :

```
class Forme {
        //méthode virtuelle pure :
        virtual void affiche()=0;
        //…
        };
```

slide 13

En conclusion, une classe ne contenant que des méthodes virtuelles pures est une spécification d'interface : elle se contente d'imposer les noms et les signatures des méthodes de ses descendantes sans implémenter de comportement générique.

Classes abstraites en C++ (2/2)

Remarques :

- Une classe ne contenant que des méthodes virtuelles pures est une spécification d'interface
- Ne pas confondre
  - Surcharger : fonctions de même nom avec des listes de paramètres différentes
  - Redéfinir : fonction identique à une fonction héritée
- Si une des classes filles est susceptible d'être manipulée *via* un pointeur sur la classe de base, le destructeur doit être virtuel.

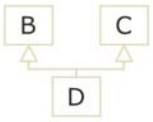
Slide 14

L'héritage multiple en C++

En C++, une seule classe peut hériter des attributs de plusieurs classes.
Avec Java, l'arborescence est stricte, c'est-à-dire qu'une classe donnée ne peut posséder qu'une seule superclasse (l'héritage est dit *simple* contrairement à des langages comme le C++, pour lesquels un héritage dit *multiple* est possible).

Héritage multiple (1/2)

- Une clase peut hériter de plusieurs classes




```
class B {/*…*/};
class C {/*…*/};
class D : public B, public C {/*…*/};
```

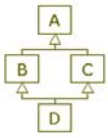
- Problème de nommage : deux attributs de même nom issus de deux classes différentes (`B::a` et `C::a`)
- NB : en Java, l'arborescence est stricte, ce n'est pas le cas en C++

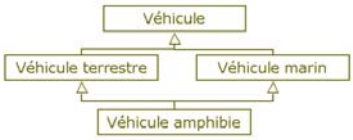
Slide 15

Dans l'exemple, la classe D hérite des attributs et fonctions des classes B et C.
Il y a parfois à résoudre des problèmes d'ambiguïté de noms (voir transp. suivant).

Héritage multiple (2/2)

- Une classe peut hériter plusieurs fois d'une même classe : héritage à répétition





Soit

- Les attributs de A sont dupliqués dans D
  - Accès aux attributs par B::A::attr
- Héritage virtuel (une seule copie des attributs de A)

```
class B : public virtual A {/*…*/};
class C : public virtual A {/*…*/};
class D : public B, public C {/*…*/};
```

Slide 16

Un cas particulier de l'héritage multiple est qu'une même classe peut hériter plusieurs fois d'une classe. On se retrouve alors dans le cas d'un « héritage à répétition »

B::A::attr désigne l'attribut déclaré dans A et correspondant à B.
Dans le 1° cas (attributs dupliqués) :
véhicule amphibie hérite de la méthode deplacer à la fois de véhicule terrestre et de véhicule marin. Il faut donc choisir laquelle sera effectivement appelée quand on voudra déplacer un véhicule amphibie.
La solution est dans le corps de la méthode deplacer: celle-ci est surchargée, et à l'aide de l'opérateur ::, on appelle la méthode deplacer de la super-classe qu'on souhaite.

Le second problème se situe dans la construction même des instances. Revenons aux classes vehicule marin et vehicule terrestre : dans le cas d'un héritage, avant même la construction des attributs, il y a la construction de la partie de l'objet issue des super-classes. Ainsi, pour les classes vehicule marin et vehicule terrestre, il faut construire la partie vehicule avant la partie propre à chaque sous-classe. une instance vehicule amphibie se retrouve avec deux fois la partie vehicule, ce qui n'est pas nécessairement ce qu'on souhaite ici.

Dans le 2° cas (héritage virtuel) :
Idéalement, il faudrait qu'un vehicule amphibie soit constitué d'une seule partie vehicule, d'une sous-partie vehicule terrestre et d'une sous-partie vehicule marin.
Pour résoudre ce problème, il existe l'héritage *virtuel*. L'idée est qu'on hérite d'une classe, mais si par héritage multiple, on se retrouve avec plusieurs fois la même classe, alors on ne la considère qu'une seule fois.

Au niveau de la construction, il faut alors détailler le chaînage de la classe de base (i.e. la super-classe la plus haute dans la hiérarchie) jusqu'aux sous-classes immédiates. Même si les arguments des constructeurs sont répétés pour le constructeur de vehicule, de vehicule terrestre et de vehicule marin, ils ne sont réellement pris en compte que dans la classe de base vehicule. Ensuite, ils ne servent qu'à repérer les arguments significatifs pour chaque sous-classe.