La programmation orientée objet (POO) en C++
Quatrième partie : Héritage et polymorphisme

Thierry Vaira

BTS SN Option IR

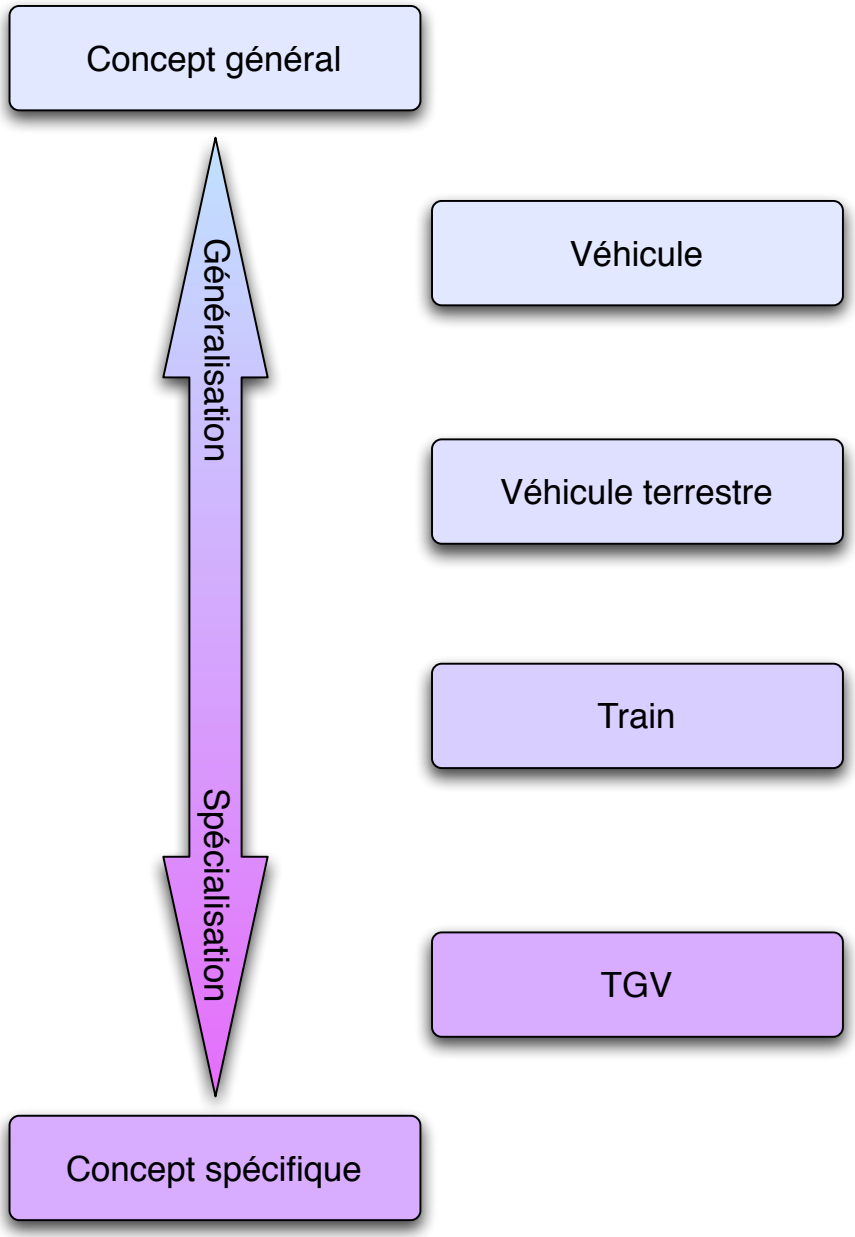
v.1.1 - 5 octobre 2018

Sommaire

Héritage : Introduction (1/5)

- L'**héritage** est un **concept fondamental de la programmation orientée objet** (POO).
- Le principe de l'héritage est en quelque sorte le même que celui d'un arbre généalogique.
- Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.
- L'héritage permet **d'ajouter des éléments à une classe existante pour en obtenir une nouvelle plus précise**.
- Il permet donc **la spécialisation ou la dérivation de types**.

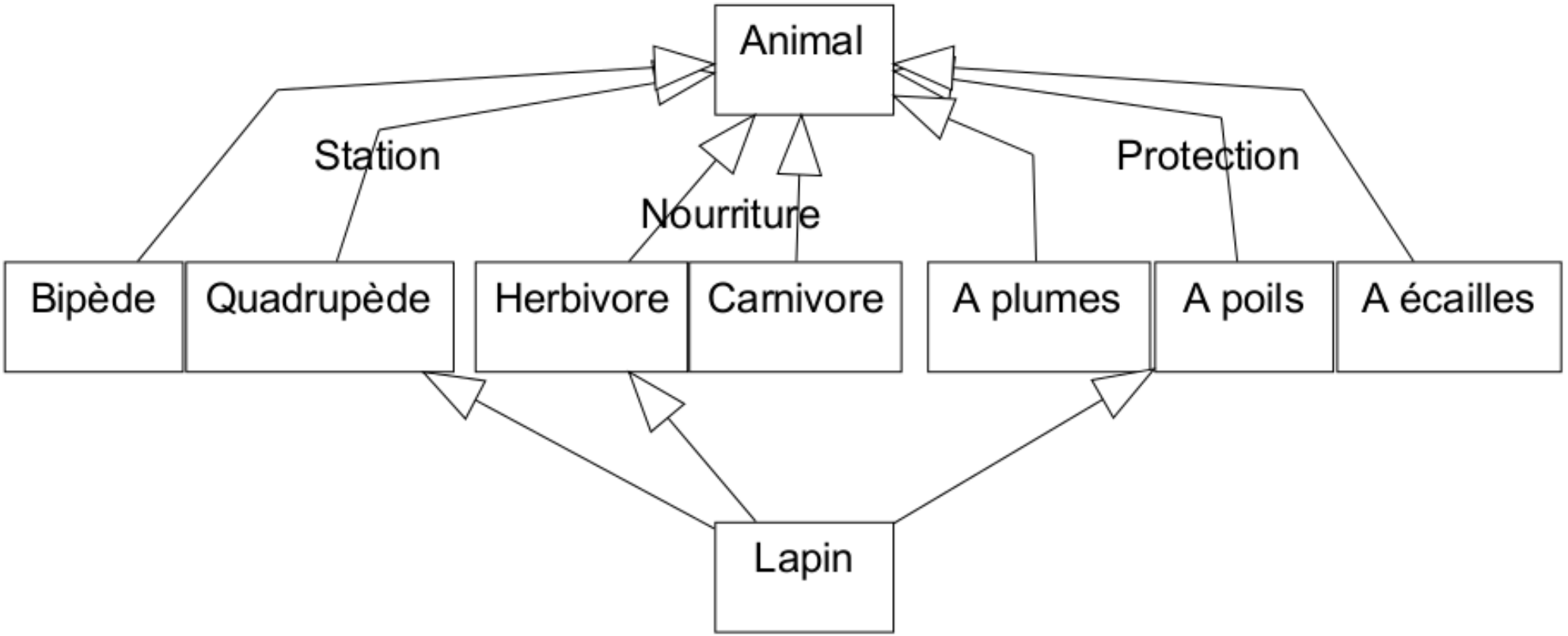
Introduction (2/5)



Introduction (3/5)

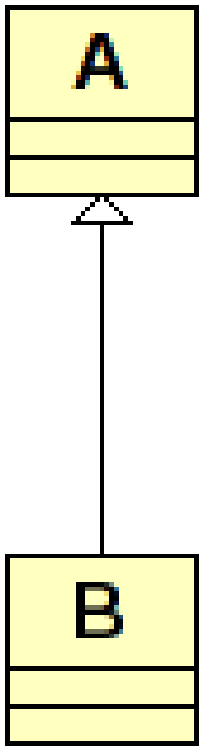
- L'**héritage** permet la **réutilisation** en **spécialisant** des **concepts plus abstraits**
- L'héritage consiste à **affiner la définition d'un concept** en :
  - **Ajoutant** des **propriétés**
  - **Ajoutant** des éléments de **comportement**
  - **Redéfinissant** des éléments de **comportement existants**
- En C++, on appelle :
  - **Classe de base** ou **classe mère**, la classe **plus abstraite**
  - **Classe dérivée** ou **classe fille**, la classe **plus spécifique**

Introduction (4/5)



Introduction (5/5)

- B **hérite** de A : "un B **est** un A avec des choses en plus".
- Toutes les instances de B sont aussi des instances de A.
- On dit que :
  - B dérive de A ou B est une spécialisation de A.
  - A est une généralisation de B.



Coder l'héritage en C++

UML	C++

	`class A`
	`{`
	
	`};`
	
	`class B : public A`
	`{`
	
	`};`

Propriétés de l'héritage

➠ L'héritage est **une relation entre classes** qui a les propriétés suivantes :

- si B hérite de A et si C hérite de B alors C hérite de A
- une classe ne peut hériter d'elle-même
- si A hérite de B, B n'hérite pas de A
- il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C
- le C++ permet à une classe C d'hériter des propriétés des classes A et B (**héritage multiple**)

Redéfinition

➠ En utilisant l'héritage, il est possible :

- **d'ajouter des caractéristiques**
- **d'utiliser les caractéristiques héritées**
- **de redéfinir les comportements (méthodes héritées)**

➠ Il ne faut pas confondre la **redéfinition (*overriding*)** et la surdéfinition ou surcharge (*overloading*) :

- Une **surdéfinition (ou surcharge)** permet **d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente**.
- Une **redéfinition (*overriding*)** permet **de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer**. Elle doit avoir une signature rigoureusement identique à la méthode parente.

Visibilité des membres

➠ Pour le **type d'accès aux membres** (attributs et méthodes), Il faut maintenant tenir compte de la situation d'**héritage** :

- **public** (`public`) : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (`private`) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **non par ceux d'une autre classe même dérivée**.
- **protégé** (`protected`) : les membres protégés d'une classe ne sont accessibles que par les objets de cette classe et **par ceux d'une classe dérivée**.

Type d'héritage

➠ Même si l'héritage **public** est largement le plus utilisé, il existe 3 types d'héritages :

mode de dérivation	Statut dans la classe de base	Statut dans la classe dérivée
public	public	public
	protected	protected
	private	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
private	public	private
	protected	private
	private	inaccessible

Constructeurs

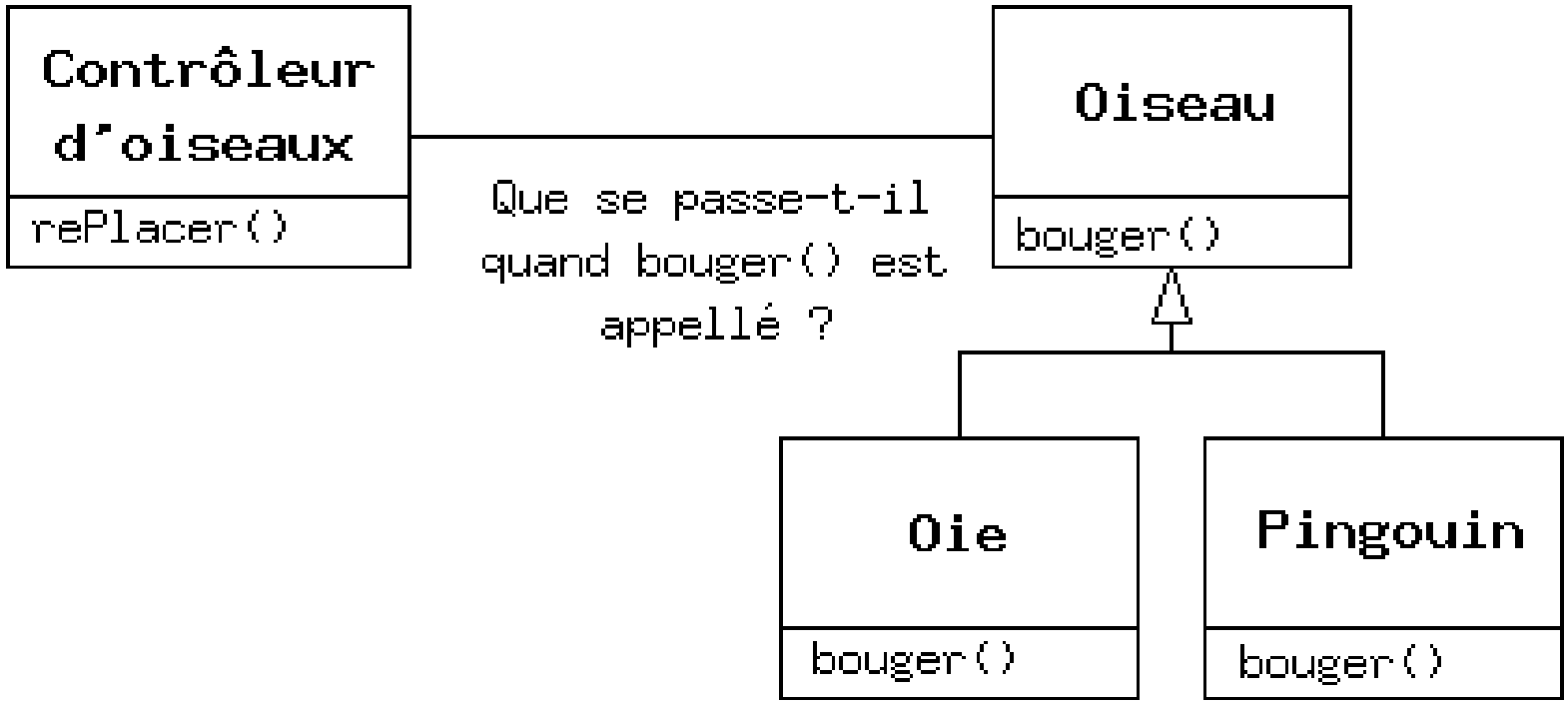
- Le **constructeur de la classe dérivée** doit **intégralement** prendre en charge la **construction de l'objet**
  - Il peut s'appuyer sur le **constructeur de la classe de base**, en y faisant **appel**
  - L'**appel au constructeur de la classe mère** doit être le **premier dans la liste d'initialisation**
    - `B::B() : A()` pour l'appel au constructeur de la classe mère A sans paramètre
    - `B::B(int a, int b, int c) : A(a,b)` pour l'appel au constructeur de la classe mère A avec 2 paramètres
    - Si aucun appel n'est explicité, le compilateur ajoute un **appel implicite au constructeur par défaut s'il existe**

Définitions du polymorphisme

- Le **polymorphisme** est un moyen de manipuler des objets hétéroclites de la même manière, pourvu qu'ils disposent d'une interface commune.
- Un **objet polymorphe** est un objet susceptible de prendre plusieurs formes pendant l'exécution.
- Le **polymorphisme** représente la capacité du système à choisir dynamiquement la méthode qui correspond au type de l'objet en cours de manipulation.
- Le **polymorphisme** est implémenté **en C++** avec **les fonctions virtuelles (`virtual`) et l'héritage**.

Comportement Polymorphe (1/4)

- Par exemple, dans le diagramme suivant, l'objet `Contrôleur d'oiseaux` travaille seulement avec des objets `Oiseau` génériques, et ne sait pas de quel type ils sont. C'est pratique du point de vue de `Contrôleur d'oiseaux`, car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'`Oiseau` avec lequel il travaille, ou le comportement de cet `Oiseau`.



Comportement Polymorphe (2/4)

- Comment se fait-il donc que, lorsque `bouger()` est appelé tout en ignorant le type spécifique de l'`Oiseau`, on obtienne le bon comportement (une `Oie` court, vole ou nage, et un `Pingouin` court ou nage) ?
- La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une **association prédéfinie** : le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter.
- En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Comportement Polymorphe (3/4)

- Pour résoudre ce problème, les langages orientés objet utilisent le concept d'**association tardive**. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour, mais il ne sait pas exactement quel est le code à exécuter.
- Pour créer une association tardive, le compilateur C++ insère une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet. Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Comportement Polymorphe (4/4)

- On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé **`virtual`**.
- On n'a pas besoin de comprendre les mécanismes de `virtual` pour l'utiliser, mais sans lui on ne peut pas faire de la programmation orientée objet en C++.
- En C++, on doit se souvenir d'ajouter le mot-clé `virtual` (devant une méthode) parce que, par défaut, les fonctions membres ne sont pas liées dynamiquement. Les fonctions virtuelles permettent d'exprimer des différences de comportement entre des classes de la même famille.
- Ces différences sont ce qui engendre un **comportement polymorphe**.

Exemple : comportement non polymorphe (1/3)

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

class Forme {
   public:
      Forme() { cout << "constructeur Forme <|- "; }
      void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
   public:
      Cercle() { cout << "Cercle\n"; }
      void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
      Triangle() { cout << "Triangle\n"; }
      void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Exemple : comportement non polymorphe (2/3)

```
void faireQuelqueChose(Forme &f)
{
  f.dessiner(); // dessine une Forme
}

int main()
{
   Cercle c;
   Triangle t;

   faireQuelqueChose(c); // avec un cercle
   faireQuelqueChose(t); // avec un triangle

   return 0;
}
```

L'exécution du programme d'essai nous montre que nous n'obtenons pas un comportement polymorphe puisque c'est la méthode `dessiner()` de la classe `Forme` qui est appelée :

Exemple : comportement non polymorphe (3/3)

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine ... une forme ?
je dessine ... une forme ?
```

Exemple : comportement polymorphe (1/3)

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

class Forme {
   public:
      Forme() { cout << "constructeur Forme <- "; }
      // la méthode dessiner sera virtuelle et fournira un comportement polymorphe
      virtual void dessiner() { cout << "je dessine ... une forme ?\n"; }
};

class Cercle : public Forme {
   public:
      Cercle() { cout << "Cercle\n"; }
      void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
      Triangle() { cout << "Triangle\n"; }
      void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Exemple : comportement polymorphe (2/3)

```
void faireQuelqueChose(Forme &f)
{
  f.dessiner(); // dessine une Forme
}

int main()
{
   Cercle c;
   Triangle t;

   faireQuelqueChose(c); // avec un cercle
   faireQuelqueChose(t); // avec un triangle

   return 0;
}
```

L'exécution du programme d'essai nous montre maintenant que nous obtenons un comportement polymorphe puisque c'est la "bonne" méthode `dessiner()` qui est appelée :

Exemple : comportement polymorphe (3/3)

```
constructeur Forme <|- Cercle
constructeur Forme <|- Triangle
je dessine un Cercle !
je dessine un Triangle !
```

Conclusion : Le C++ permet, par le polymorphisme, que des objets de types différents (`Cercle`, `Triangle`, ...) répondent différemment à un même appel de fonction (`dessiner()`).

Appel de destructeurs : problème

```
class A {
   private:    int *p;
   public:
    A()  {p = new int[4]; cout << "A()";}
    ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
   private:    int *q;
   public:
    B()  {q = new int[64]; cout<< "B() et q=" << q;}
    ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
     A *pA = new B(); delete pA; // A()B() et q=0x100170 ~A()
}
```

L'affichage met en évidence un problème de "fuite" de mémoire. Le destructeur de B n'est jamais appelé !

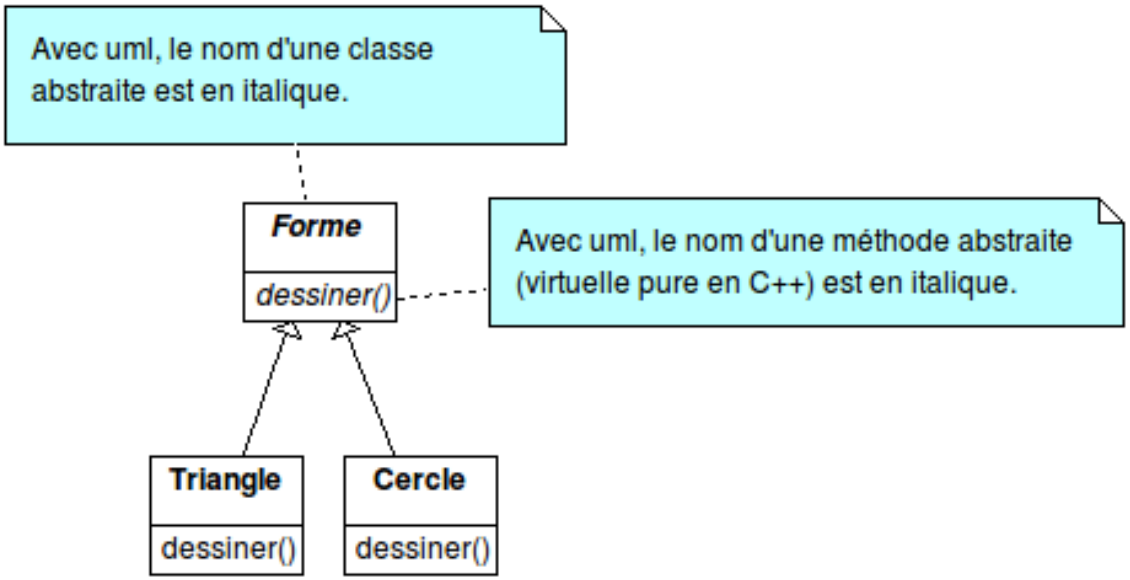
Appel de destructeurs : solution

```
class A {
   private:    int *p;
   public:
    A()  {p = new int[4]; cout << "A()";}
    virtual ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
   private:    int *q;
   public:
    B()  {q = new int[64]; cout<< "B() et q=" << q;}
    ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
      A *pA = new B(); delete pA; // A()B() et q=0x100170 ~B() ~A()
}
```

Avec la déclaration du destructeur de A en virtuel (`virtual`), le destructeur de B est correctement appelé.

Classe abstraite : Introduction

- Quelquefois, la **modélisation d'un concept très général** conduit à laisser des **« trous » dans l'implémentation**.
- Une classe abstraite permet d'introduire certaines méthodes dont on ne peut encore donner aucune définition.
- Par exemple, on ne sait pas programmer ce que doit faire la fonction `dessiner()` dans le contexte de `Forme`. On veut juste s'assurer de sa présence dans toutes les classes filles, sans devoir la définir dans la classe parente.



Classe abstraite en C++

- Une classe est dite **abstraite** si **elle contient au moins une fonction virtuelle pure**.
- Une fonction membre est dite **virtuelle pure** lorsqu'elle est déclarée de la façon suivante :
  `virtual type nomMethode(paramètres) = 0;`
- **On ne peut pas instancier d'objet à partir d'une classe abstraite**.
- Mais on le peut à partir d'une **classe dérivée** à condition qu'elle **définisse complètement la méthode vituelle pure**.

Exemple : classe abstraite

```
// La classe Forme ne peut pas être instanciée : elle est dite abstraite
class Forme {
   public:
      Forme() {}
      // la méthode dessiner est virtuelle pure et ne possède aucune définition
      virtual void dessiner() = 0; // cela oblige tous les descendants à contenir
            une méthode dessiner()
};

// Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure
     devient instanciable
class Cercle : public Forme {
   public:
      Cercle() {}
      void dessiner() { cout << "je dessine un Cercle !\n"; }
};

class Triangle : public Forme {
   public:
      Triangle() {}
      void dessiner() { cout << "je dessine un Triangle !\n"; }
};
```

Le transtypage en C++

- Le transtypage (*cast* ou conversion de type) en C++ permet la conversion d'un type vers un autre.
- Nouvelle syntaxe de l'opérateur traditionnel :
  - En C : `(nouveau type)(expression à transtyper);`
  - En C++ : `type(expression à transtyper);`
- Pour l'héritage :
  - transtypage « ascendant » (*upcast*) : changer un type vers son type de base (ne pose pas de problème)
  - transtypage « descendant » (*downcast*) : conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée.

Nouveaux opérateurs de transtypage en C++

- static_cast : Opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère `const` ou `volatile`.
- const_cast : Opérateur spécialisé et limité au traitement des `const` et `volatile`
- dynamic_cast : Opérateur spécialisé et limité au traitement des *downcast*.
- reinterpret_cast : Opérateur spécialisé dans le traitement des conversions de pointeurs peu portables.
- La syntaxe est la suivante :
  `op_cast<expression type>(expression à transtyper);`
- où op prend l'une des valeurs (`static`, `const`, `dynamic` ou `reinterpret`)

Transtypage « ascendant »

Traiter un type dérivé comme s'il était son type de base est appelé transtypage ascendant, surtypage ou généralisation (*upcasting*). Cela ne pose donc aucun problème.

Exemple : transtypage « ascendant »

```
class Forme {};
class Cercle : public Forme {};
class Triangle : public Forme {} ;

void faireQuelqueChose(Forme &f) { f.dessiner(); }

...

Cercle c;

// Un Cercle est ici passé à une fonction qui attend une Forme.
// Comme un Cercle est une Forme, il peut être traité comme tel par
    faireQuelqueChose()
faireQuelqueChose(c);
```

Transtypage « descendant »

Conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée.

Exemple : transtypage « descendant » (*downcast*)

```
class Animal { public: virtual ~Animal() {} };
class Chien : public Animal {};
class Chat : public Animal {};

int main() {
   Animal* a = new Chat; // Transtypage ascendant

   // On essaye de le transtyper en Chat* :
   Chat* c1 = a; // Erreur : invalid conversion from 'Animal*' to 'Chat*'

   Chat* c2 = dynamic_cast<Chat *>(a); // Valide : Transtypage descendant
}
```