

POO avec C++

Chapitre 4 L'héritage et polymorphisme

4. Table des matières

- 1. Héritage: classe dérivée**
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple



4.1 Classes dérivées (intro)

Warrior
<i>attack;</i> <i>defense;</i>
interact();

Thief
<i>attack;</i> <i>defense;</i>
interact(); steal();

Wizard
<i>attack;</i> <i>defense;</i> <i>mana;</i>
interact();

Necromancer
<i>attack;</i> <i>defense;</i> <i>mana;</i>
interact(); riseUndead();



4.1 Classes dérivées (intro)

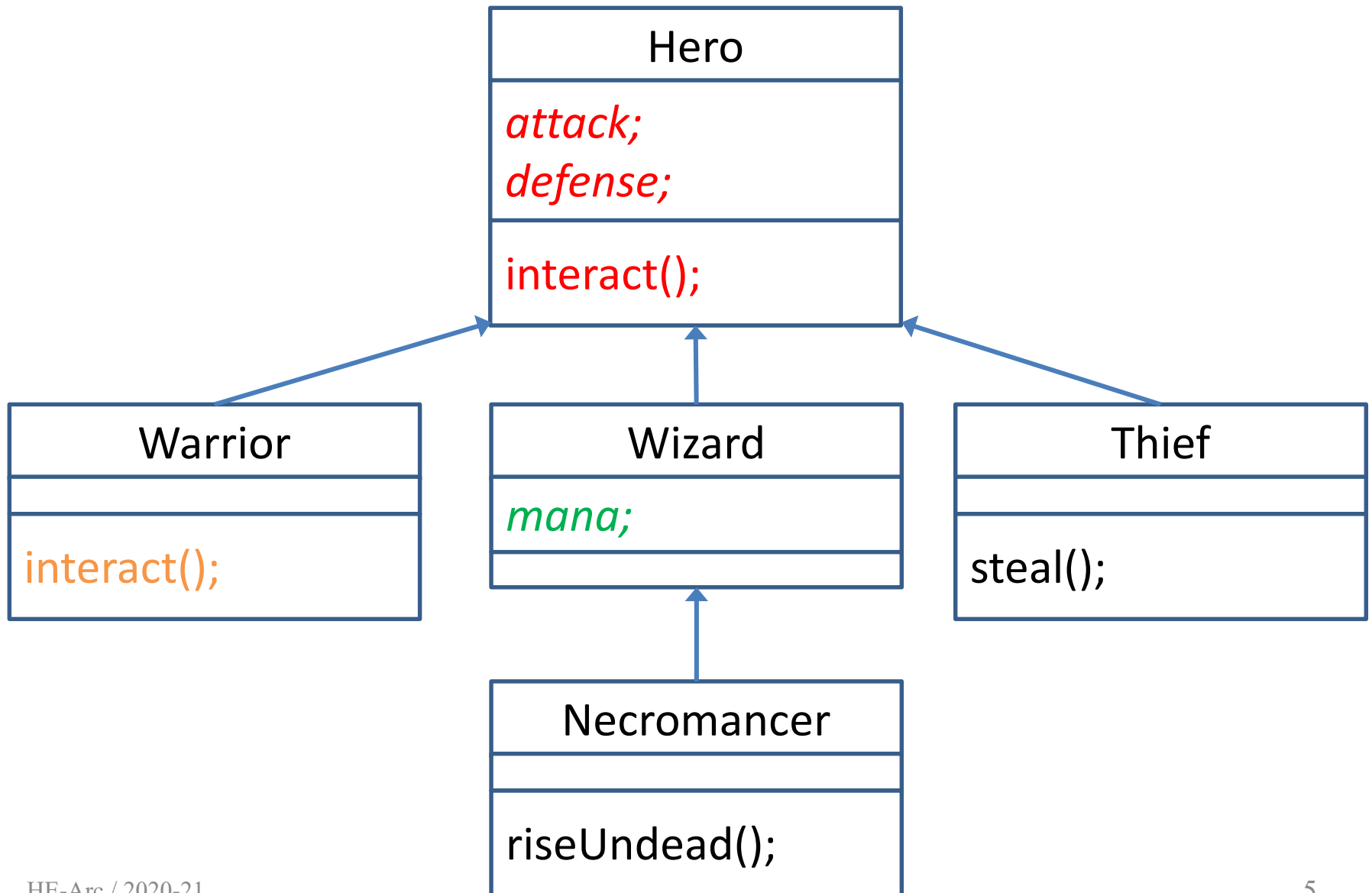
Warrior
<i>attack;</i> <i>defense;</i>
<i>interact();</i>

Thief
<i>attack;</i> <i>defense;</i>
<i>interact();</i> <i>steal();</i>

Wizard
<i>attack;</i> <i>defense;</i> <i>mana;</i>
<i>interact();</i>

Necromancer
<i>attack;</i> <i>defense;</i> <i>mana;</i>
<i>interact();</i> <i>riseUndead();</i>

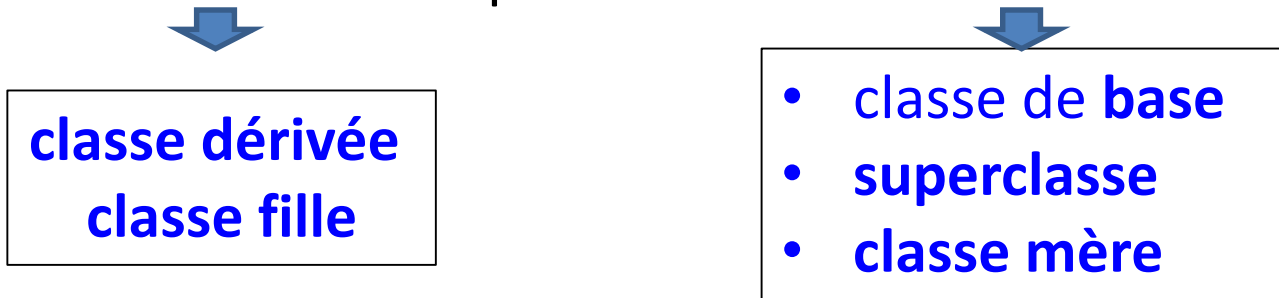
4.1 Classes dérivées (intro)



4.1 Classes dérivées

Héritage

Création d'une **nouvelle classe** à partir d'une **classe existante**

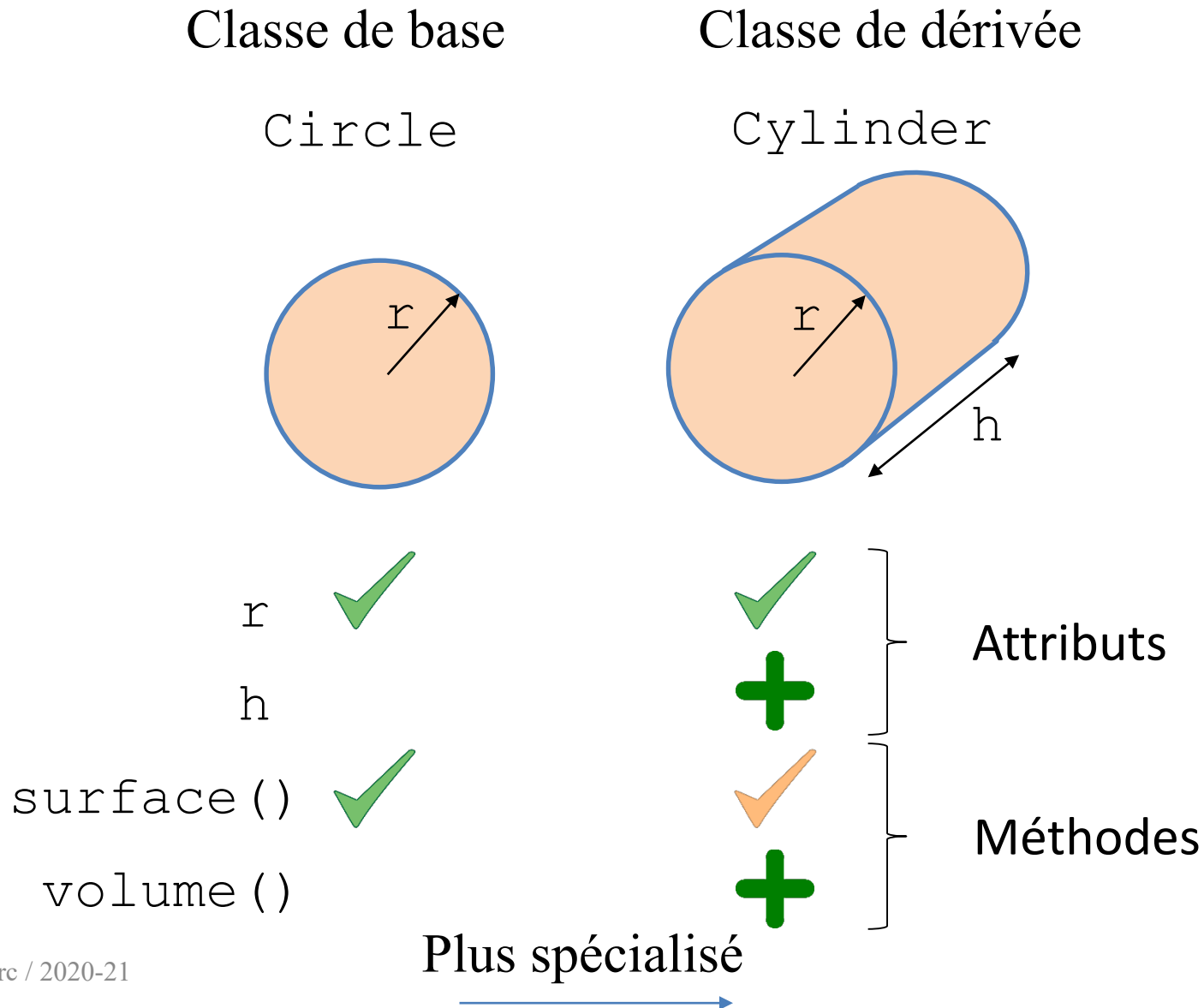


La classe **dérivée** **hérite** des membres de la classe de base

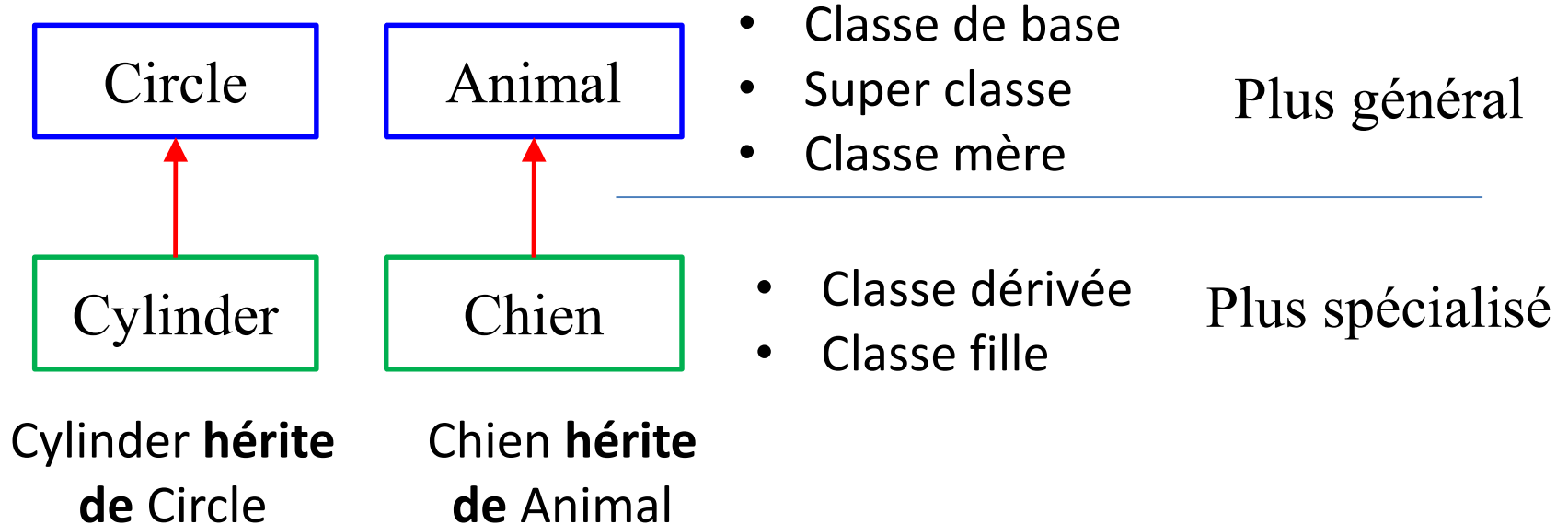
- On peut:
 - **Ajouter** des membres
 - **Redéfinir** des méthodes (spécialiser)

Permet notamment d'**éviter la répétition de code**

4.1 Premier exemple



4.1 Interface de Cylinder → Circle



La flèche → indique une **généralisation**.

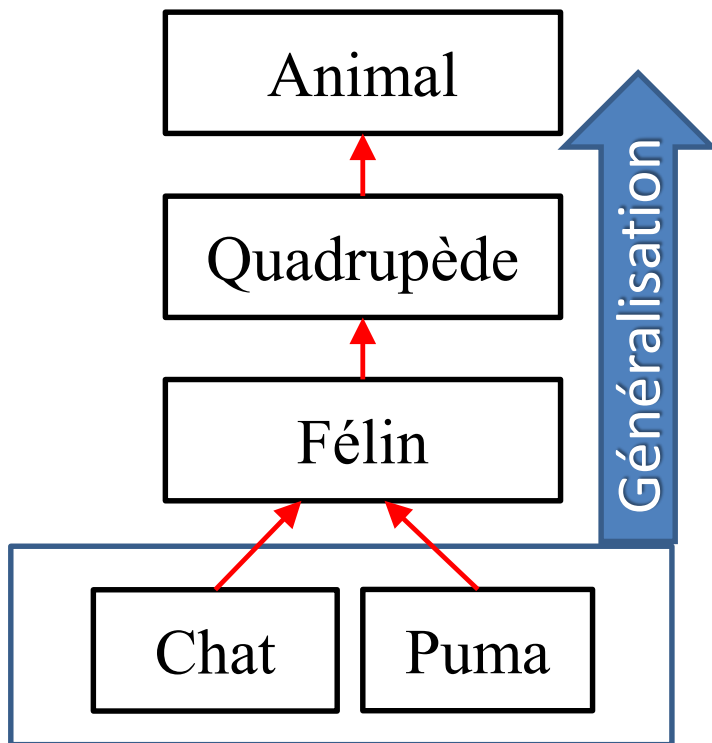
- Un animal est plus général qu'un chien
- Un cercle est plus général qu'un cylindre

Dans l'autre sens on parle de **spécialisation**.

- Un chien est plus spécialisé qu'un animal

4.1 Processus de Généralisation

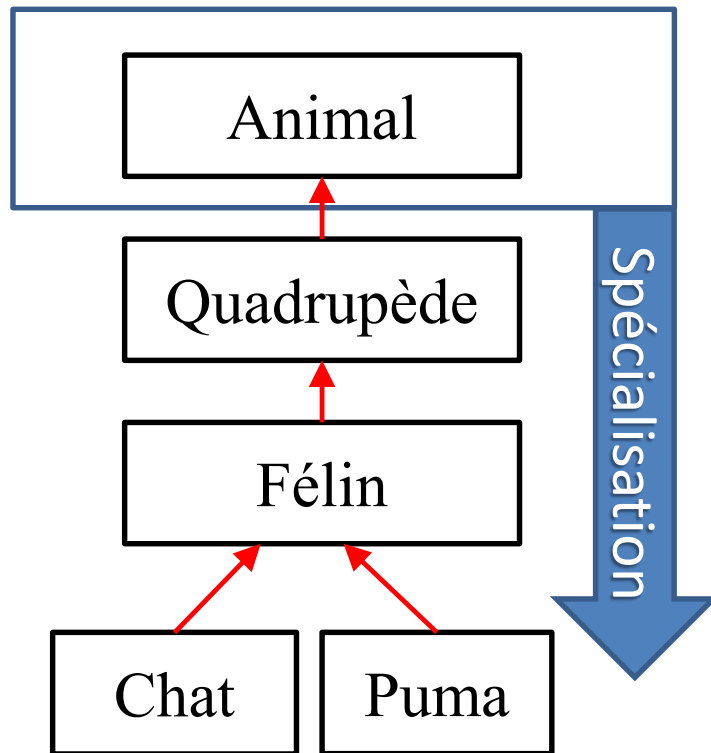
On part du plus spécialisé (bas) et on va au plus général (haut)



- La flèche → indique une **généralisation**.
- Chaque niveau d'abstraction devient de plus en plus général.
- Factorisation des éléments communs à une classe.
- Démarche difficile

4.1 Processus de Spécialisation

On part du plus général (haut) et on va au plus spécialisé (bas)



- **Spécialisation:** Opération inverse de la généralisation
- Chaque niveau d'abstraction devient de plus en plus spécifique.
- Extension des caractéristiques propres
- Démarche plus simple

4.1 Héritage et composition/agrégation

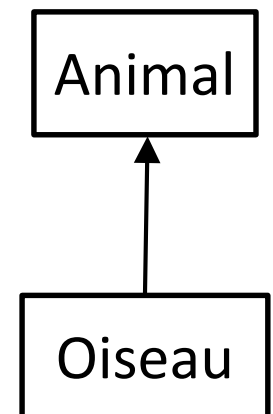
Relation entre classes: Héritage, composition, agrégation:

Deux type principaux de relations:

- Héritage: "Un X **est un** Y, **avec des choses en plus**"
- Composition / agrégation: "Un X **a un** Y"

A) Héritage:

- Un oiseau **est un** animal **avec des choses en plus**
=> La classe Oiseau **hérite** de la classe Animal

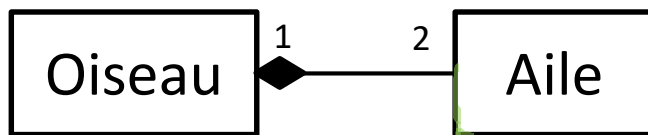


4.1 Héritage et composition/agrégation

B) Composition: Relation **forte** (pas l'un sans l'autre)

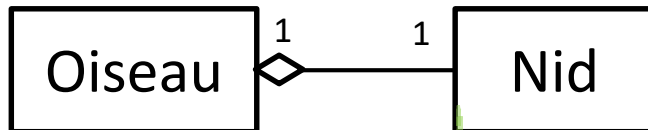
Un **oiseau** **a deux** ailes

La classe `Oiseau` **a** pour attributs 2 instances de la classe `Aile`

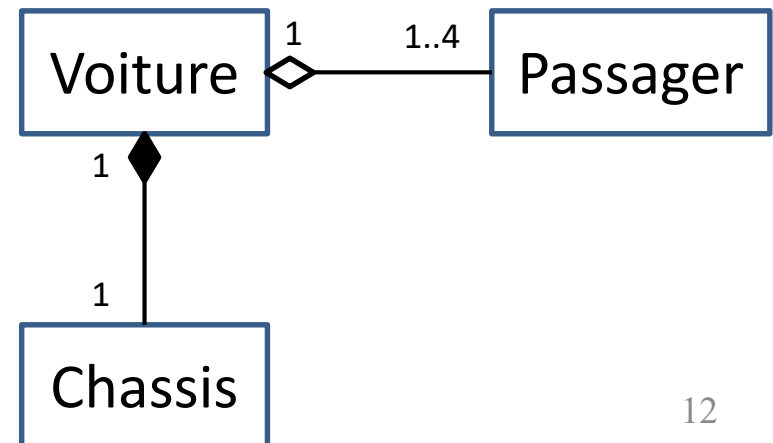


C) Agrégation: Relation **faible**

Un **oiseau** **a un** nid



Composition et agrégation



4.1 Interface et Implémentation de Circle

circle.h

```
#pragma once

class Circle
{
public:
    Circle(double _r=0) : r(_r) {}
    double getRadius() const {return r;}
    void setRadius(double _r) {r=_r;}
    double surface() {return r*r*3.14;}

private:
    double r=0;
};
```

4.1 Interface de Cylinder → Circle

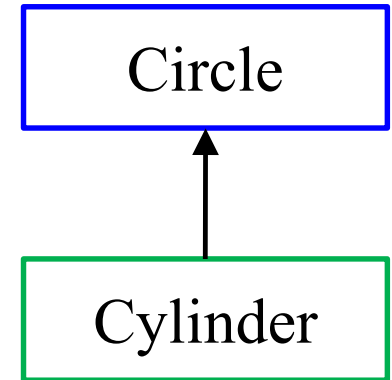
cylinder.h

```
#pragma once

#include "circle.h"

class Cylinder : public Circle
{
public:
    Cylinder(double=0, double=0);
    double surface();
    double volume();
private:
    double h=0;
};
```

Schéma UML:



4.1 Commentaires

```
class Cylinder : public Circle {}
```



Héritage

The diagram illustrates the components of the C++ class declaration `class Cylinder : public Circle {}`. A black arrow points from the box labeled 'Héritage' to the colon (`:`) in the code. A blue arrow points from the box labeled 'Mode de dérivation' to the keyword `public` in the code.

Mode de dérivation

: Indique que **Cylinder** hérite de **Circle**.

Mode de dérivation

public : Héritage public, le plus courant.

private : Valeur par défaut (`class Cylindre : Cercle {}`)

4.1 Commentaires

```
class Cylinder : public Circle {}
```

- Héritage => une instance de `Cylinder` dispose de tous les membres de la classe `Circle`, avec :
 - 1 attribut supplémentaire **h** (hauteur du cylindre)
 - 1 méthode nouvelle **volume()**
 - 1 méthode **redéfinie** **surface()** (formule différente)
- Comme la méthode **surface()** de **Circle** a la même signature que **surface()** de **Cylinder**, elle n'est plus accessible dans la classe **Cylinder**. Sauf avec `Circle::surface()`.
- La méthode **surface()** a été **redéfinie** (pas une surcharge).

4.1 Implémentation de Cylinder → Circle

Cylinder.cpp

```
#include "cylinder.h"
```

```
Cylinder::Cylinder(double radius, double height)
    : Circle(radius)
```

```
{
    h=height;
}
```

Appel du constructeur
du parent

```
double Cylinder::surface()
```

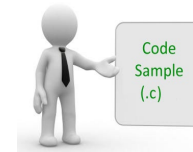
```
{
    double r = this->getRadius();
    return 2 * 3.14 * r * (r + h);
}
```

L'attribut **r** de Circle est
private => getRadius()

Ici, La variable locale **r** masque l'attribut **r** de Circle

4.1 Commentaires

- Dans la méthode `surface()` de `Cylinder`, la variable locale `r` masque l'attribut du même nom (hérité de `Circle`).
- Comme l'attribut `r` a été déclaré `private` dans `Circle`, la classe `Cylinder` n'a **pas accès** à ce membre (bien qu'il soit présent, car hérité de `Circle`).
- Un droit d'accès supplémentaire a été introduit pour pallier à ce problème: `protected`



01_HeritageCircleCylinder

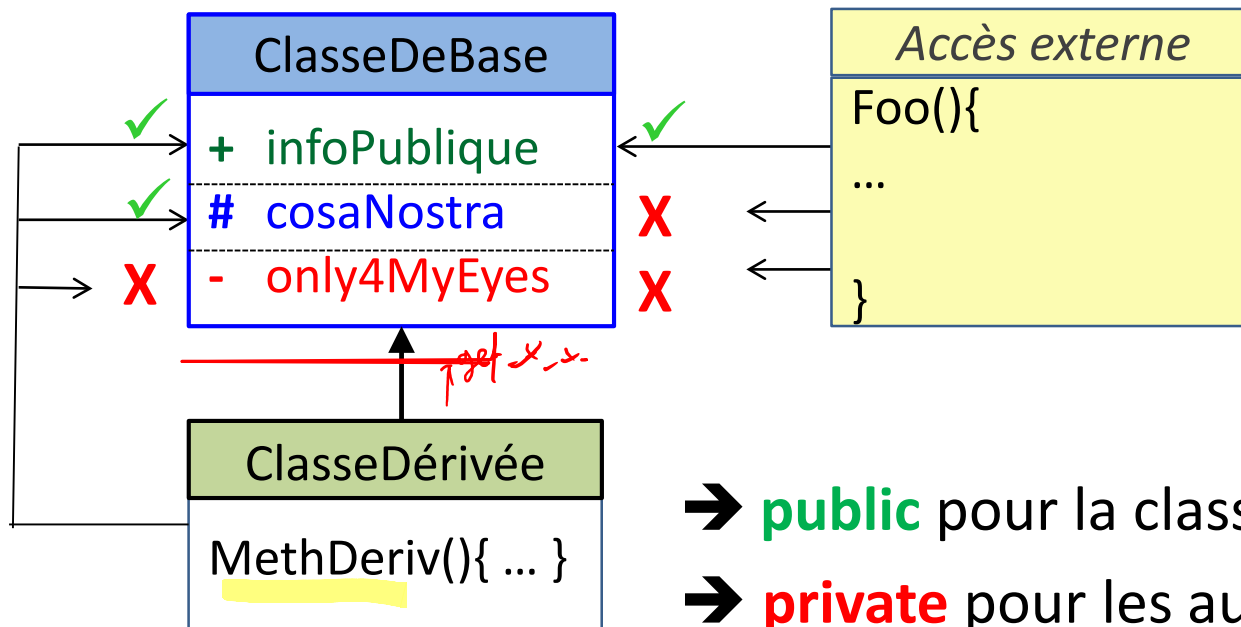
4. Table des matières

1. Héritage: classe dérivée
- 2. Contrôle d'accès**
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple

4.2 Contrôle d'accès

Le contrôle d'accès protected facilite l'implémentation de classes dérivées. Il permet aux classes dérivées, uniquement, (et à elles seules) d'accéder directement aux membres hérités.

Un membre protected (#) sera considéré:



➔ **public** pour la classe dérivée

➔ **private** pour les autres utilisateurs

4.2 Nouvelle déclaration de la classe `Circle`

Circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
    public:
        Circle(double _r=0) : r(_r){}
        double getRadius() const {return r;}
        void    setRadius(double _r) {r=_r;}
        double surface() {return r*r*3.14;}

    protected:
        double r;
};

#endif // CIRCLE_H
```

4.2 Nouvelle implémentation de Cylinder

Cylinder.cpp

```
#include "cylinder.h"
```

```
Cylinder::Cylinder( double _r, double _h) : Circle(_r)
{
    h=_h;
}
```

```
double Cylinder::surface()
{
    return 2 * 3.14 * r * (r + h);

    //return 2* Cercle::surface() + h * 3.14 * 2 * r;
}
```

r est maintenant accessible
grâce à **protected**

```
Cylinder c(5,15);
```

5	surface()	15	surface()	volume()
---	-----------	----	-----------	----------

4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
- 3. Mode de dérivation**
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple

4.3 Mode de dérivation

- Le **mode de dérivation** permet de définir l'accessibilité des membres hérités. Il est précisé avant le nom de la superclasse.

```
class Cylinder : public Circle { ... }
```

- L'héritage n'augmente jamais l'accessibilité d'un membre.**
- Dans tous les cas, les membres `private` de la classe de base seront inaccessibles, mais présents dans la classe dérivée.

Modes de dérivation

private les membres publics et protégés deviennent privés.
Mode par défaut !

protected les membres publics deviennent protégés

public les membres conservent leur accessibilité

4.3 Mode de dérivation (II)

Mode de dérivation	Accessibilité des membres dans la classe de base	Accessibilité des membres hérités dans la classe dérivée
public	public	public
	protected	protected
	private	Inaccessible
	Inaccessible	
protected	public	protected
	protected	
	private	Inaccessible
	Inaccessible	
private	public	private
	protected	
	private	Inaccessible
	Inaccessible	

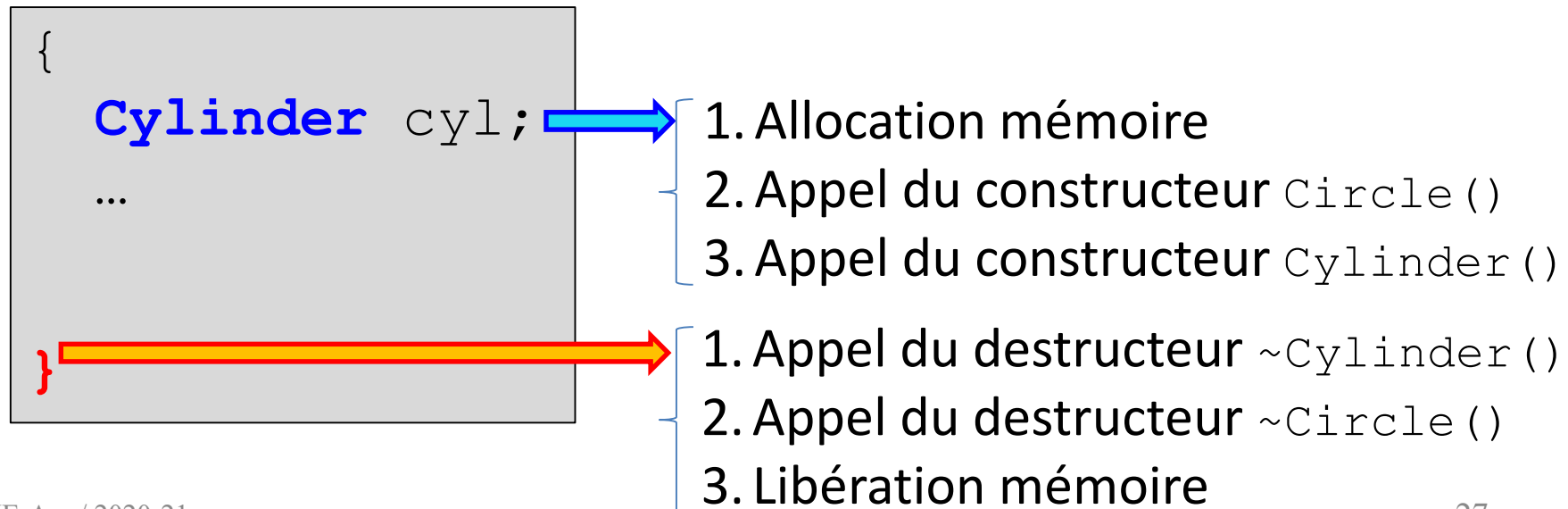
4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
- 4. Hiérarchie de classes**
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple

4.4 Création/Destruction des objets

- Lors de **l'instanciation d'une classe dérivée**, un objet de la superclasse est d'abord alloué et initialisé (appel d'un constructeur, celui par défaut si rien n'est précisé)
- À la **destruction d'un objet**, cette séquence est inversée (exécution destructeur, libération objet dérivé, libération objet de base)

Exemple:



4.4 Hiérarchie de classes: construction/destruction

```
Cylindre canette (16,64);
```

Appel **explicite** des constructeurs de base

```
#include "cylinder.h"
Cylinder::Cylinder(double _r, double _h) : Circle(_r)
{
    this->h=_h;
}
```

Appel explicite

Appel **implicite** des constructeurs de base

```
#include "cylinder.h"
Cylinder::Cylinder(double _r, double _h)
{
    this->r=_r; // impossible si r est private!
    this->h=_h;
}
```

*Appel implicite
de Circle()*





4.4 Hiérarchie de classes

- On peut faire **plusieurs** dérivations successives;
- Dans ce cas, la classe de base commune à toutes les classes est appelée **classe racine**;
- L'intérêt est que toutes les classes héritant de cette classe racine **disposeront de ses membres**;
- Disposer d'une classe racine est particulièrement intéressant pour utiliser le **polymorphisme** (voir plus loin).



4.4 Accès aux méthodes de la superclasse

- L'accès aux méthodes de la classe de base dépend du *mode de dérivation* (public, protected, ou private)
- Si une méthode est **redéfinie** dans une classe dérivée, celle de la superclasse est alors **masquée**. Elle peut cependant être appelée.

Par exemple:

```
Baseclass::display();
```

Cette syntaxe est **utile** pour utiliser le code de la classe de base avant d'exécuter celui propre à la classe dérivée;

- Pour les **constructeurs**, on utilise la syntaxe des listes d'initialisation: s'il n'y a aucun appel explicite, c'est le constructeur par défaut qui sera appelé



Série 4.1

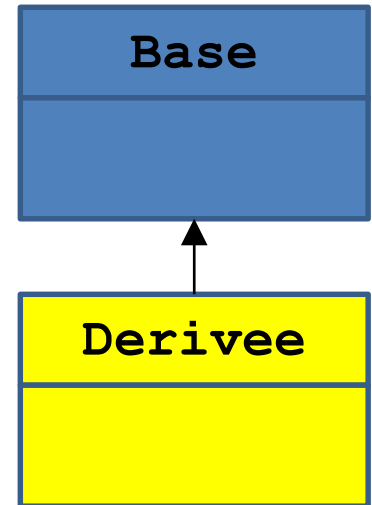
4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
- 5. Conversions de types**
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple

4.5 Conversion standard vers une **classe de base**

Si la classe D dérive publiquement de la classe B

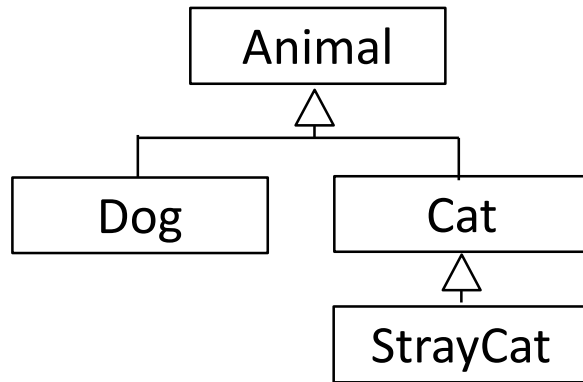
- ⇒ les membres de B sont membres de D.
- ⇒ les membres `public` et `protected` de B peuvent être atteints à travers un objet D.
- ⇒ Toutes les fonctions non `private`, disponibles pour B le sont pour un objet de type D.



Que peut-on dire de la conversion entre les objets instanciés de ces classes ?

```
Base    oB;
Derivee oD;
oB = oD; ?
oD = oB; ?
```


Conversion entre objets de type statique

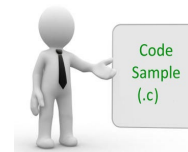


```
cat = dog;  
dog = cat;  
animal = cat;  
cat = animal;  
cat = straycat;  
straycat = cat;  
animal = straycat;
```

Ces instructions sont licites ssi on peut **toujours** répondre par l'affirmative à la question: **G = D**

"le type de la *rightvalue* (D) EST-IL toujours un type de la *left value* (G) ?"

1. un chien(D) EST-IL un chat(G) ?
2. un chat(D) EST-IL un chien(G) ?
3. un chat(D) EST-IL un animal (G) ?
4. un animal(D) EST-IL toujours un chat (G) ?
5. un chat sauvage(D) EST-IL un chat(G) ?
6. un chat(D) EST-IL un chat sauvage(G) ?
7. un chat sauvage(D) EST-IL un animal(G) ?

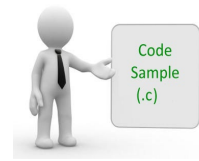
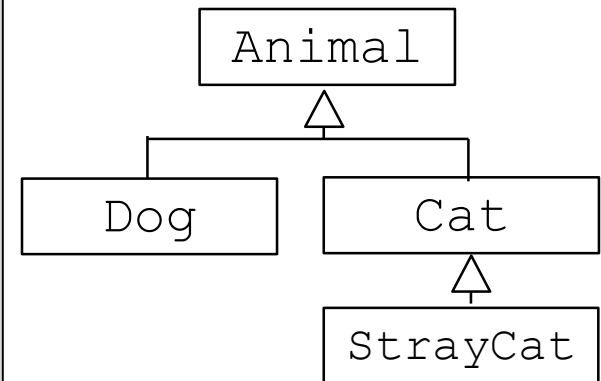


Conversion entre objets de type statique

Les instructions suivantes sont licites ssi on peut toujours répondre par l'affirmative à la question:

"le type du membre de droite EST-IL un type du membre de gauche ?"

cat = dog;	✗	// oD1 = oD2
dog = cat;	✗	// oD2 = oD1
animal = cat;	✓	// oB = oD
cat = animal;	✗	// oD = oB
cat = straycat;	✓	// oD = oDD
straycat = cat;	✗	// oDD = oD
animal = straycat;	✓	// oB = oDD



Classe de Base = classe dérivée ; ✓

02_AnimalsNonVirtual

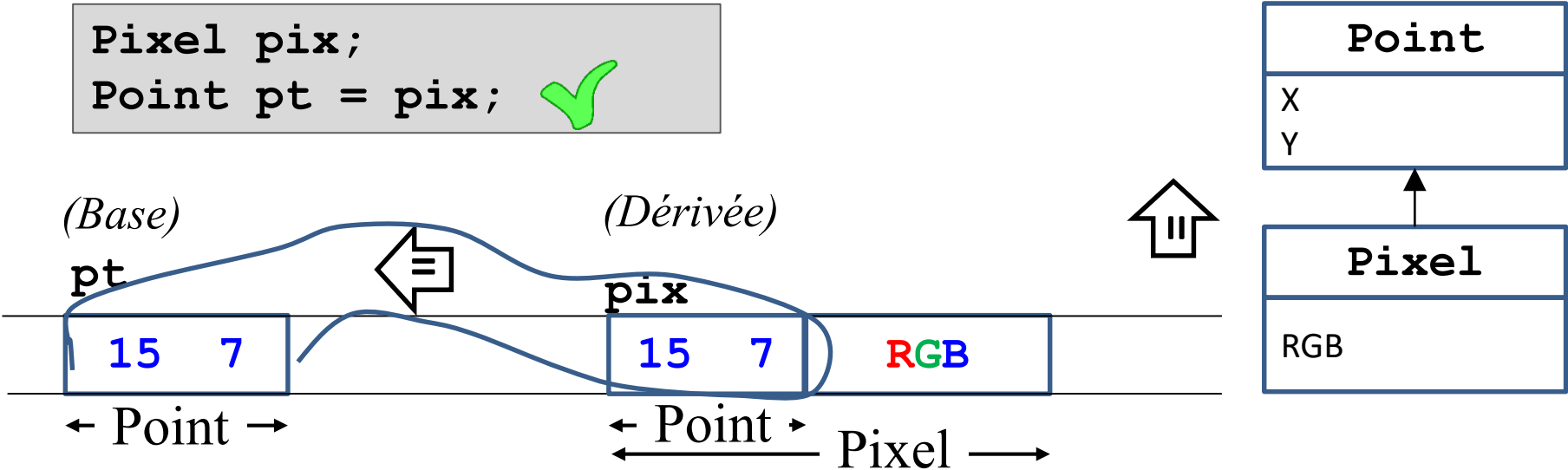
4.5 Conversion standard vers une **classe de base**

La **conversion automatique** d'un objet de la classe dérivée en un objet de la classe de base.

$Base \leftarrow Dérivée$

Exemple:

```
Pixel pix;  
Point pt = pix; ✓
```



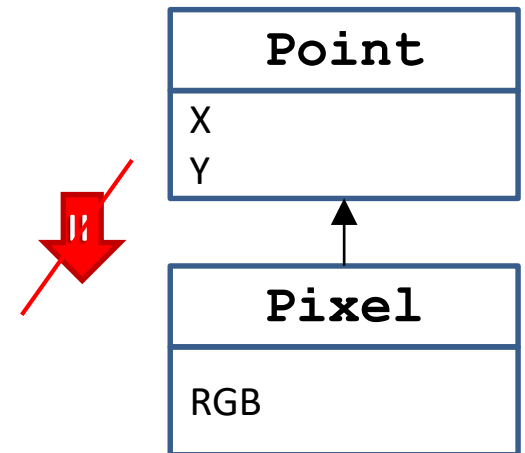
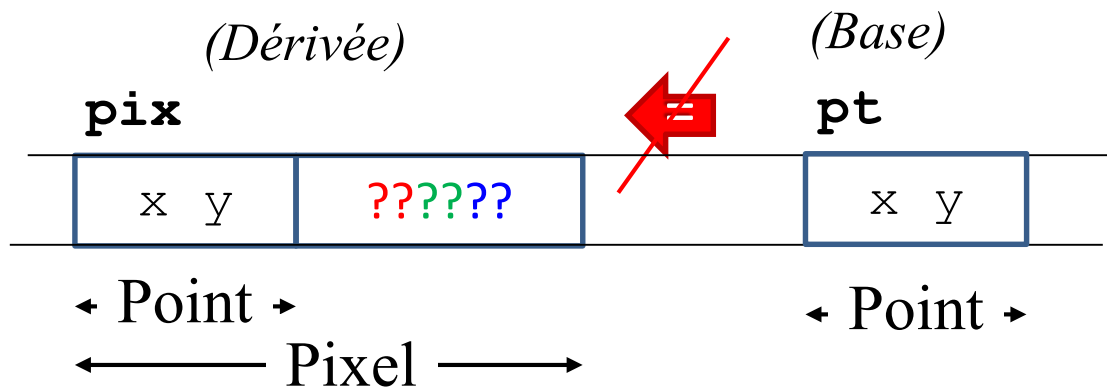
On peut dire qu'un chien est toujours un animal

4.5 Conversion standard ~~vers une classe dérivée~~

La **conversion automatique** d'un objet de la classe de base en un objet de la classe dérivée n'est **pas possible**! *Dérivée ← Base*

Exemple:

```
Point pt;  
Pixel pix = pt; ❌ Erreur de compilation
```



On ne peut pas dire qu'un animal est toujours un chien

Conversions standards

upcasting

...d'une classe dérivée D vers
une classe de base B

Deux cas:

1. Objets:

Base \leftarrow *Dérivée*

D oD ;

B oB = oD ; 

2. Pointeurs ou références

*Base** \leftarrow *Dérivée**

D* poD = new D ;

B* poB = poD ; 

downcasting


...d'une classe de base vers
une classe de dérivée D.

Deux cas:

1. Objets:

Dérivée \leftarrow *Base*

B oB ;

D oD = oB ; 

2. Pointeurs ou références

*Dérivée** \leftarrow *Base**

B* poB = new B ;

D* poD = (D*) poB ;

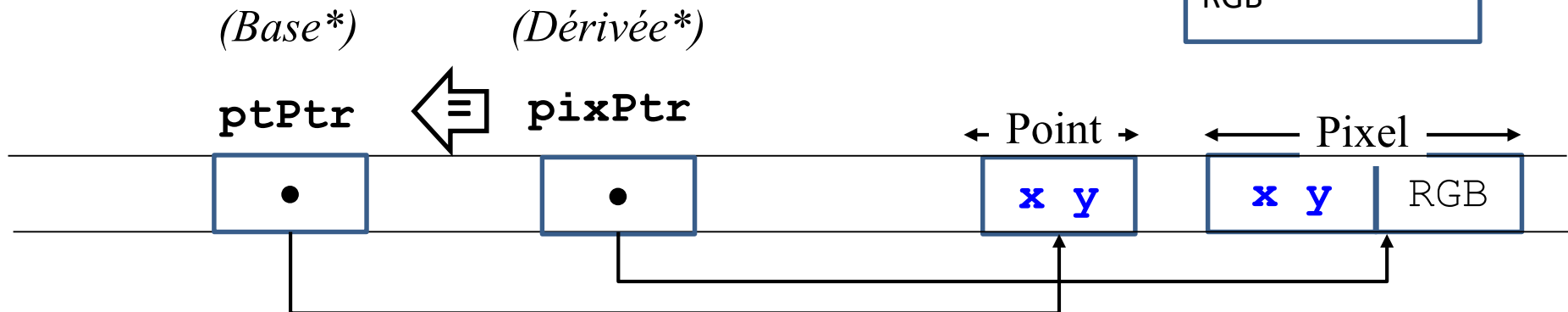
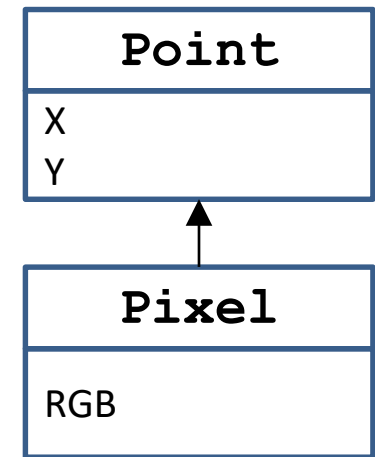
4.5 Conversion standard de pointeurs sur une classe de base

La **conversion automatique** d'un pointeur sur un objet de la classe dérivée en un pointeur sur un objet de la classe de base.

$$Base^* \leftarrow Dérivée^*$$

Exemple:

```
Pixel *pixPtr = new Pixel;  
Point *ptPtr = new Point;  
ptPtr = pixPtr; ✓
```



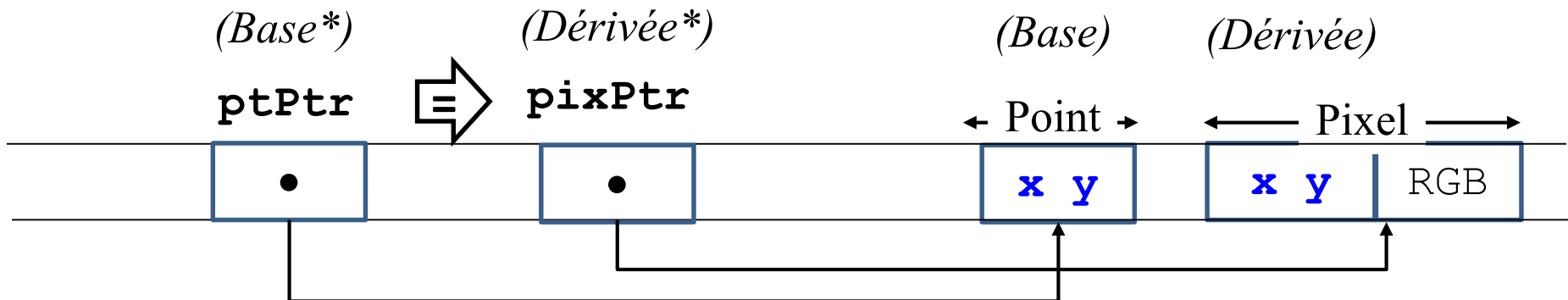
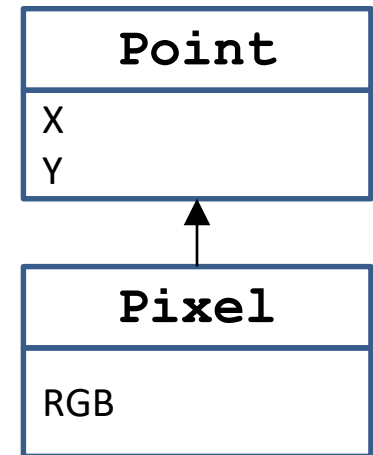
4.5 Conversion standard de pointeurs sur une classe dérivée

La **conversion automatique** d'un pointeur sur un objet de la classe de base en un pointeur sur un objet de la classe dérivée.

$$\text{Dérivée}^* \leftarrow \text{Base}^*$$

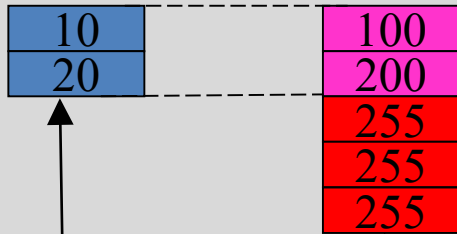
Exemple:

```
Pixel *pixPtr = new Pixel;  
Point *ptPtr = new Point;  
pixPtr = ptPtr; ✗ Erreur de compilation  
  
pixPtr = (Pixel*) ptPtr; ✓
```



Situation initiale

Base **b** Derived **d**



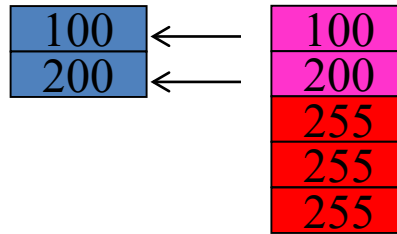
Base* **bPtr**



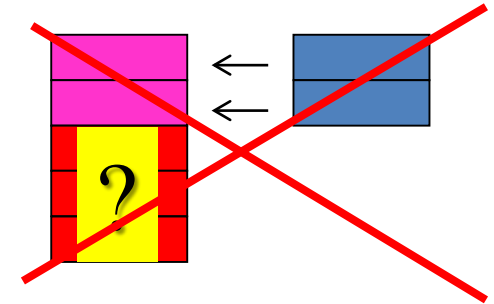
Derived* **dPtr=&d**

Affectations

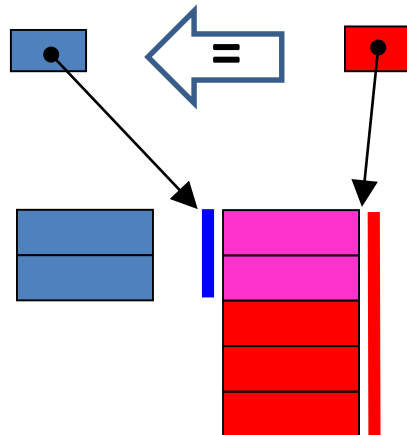
b = d;



d = b; ~~compilation~~

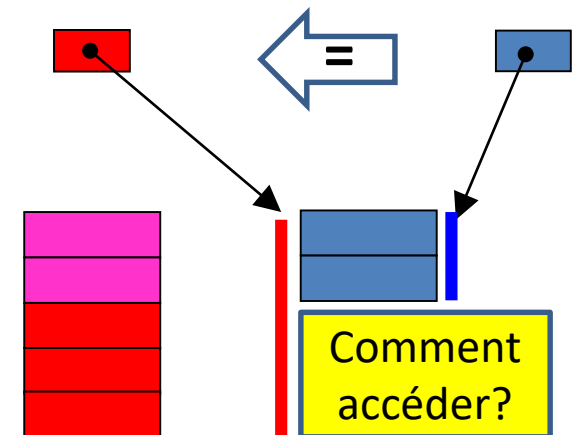


bPtr = dPtr;



Un pointeur sur une classe mère peut pointer sur une classe fille

dPtr = (Derived*) bPtr;



Un pointeur sur une classe fille peut pointer sur une classe mère

Résumé (code)

```
class B
{
...
};
```

```
class D: public B
{
...
};
```

```
B oB;
```

```
D oD;
```

```
oB = oD; ✓
```

```
oD = oB; ✗ Erreur compilation
```

```
B *poB = new B;
```

```
D *poD = new D;
```

```
poB = poD; ✓
```

```
poD = poB; ✗ Err. Compil.
```

```
poD = (D*) poB; ✓
```

Cast obligatoire

Résumé (principe)

...d'une classe dérivée D vers
une classe de base B.

Deux cas:

1. Objets:

$Base \leftarrow Dérivée$



animal \leftarrow chien

2. Pointeurs ou références

$Base^* \leftarrow Dérivée^*$



...d'une classe de base vers
une classe de dérivée D.

Deux cas:

1. Objets:

$Dérivée \leftarrow Base$



chien ~~\leftarrow~~ animal

2. Pointeurs ou références

$Dérivée^* \leftarrow Base^*$



On peut utiliser un pointeur
sur une superclasse pour
pointer sur un objet d'une
classe dérivée !

4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
- 6. Type statique et Type dynamique**
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
9. Héritage multiple

4.6 Type statique vs dynamique

Quel est le type de:

- `oB` ? `B` ?
- `poB` ? `B*` ou `D*` ?
- `roB` ? `B` ou `D` ?

```
class B
{ ... };

class D:public B
{ ... };

D oD;
B oB = oD;
B* poB = &oD;
B& roB = oD;
```

Type statique (static binding)

C'est ainsi que c'est déclaré (connu à la compilation) `D oD;` ou `B oB;`

Type dynamique (dynamic binding)

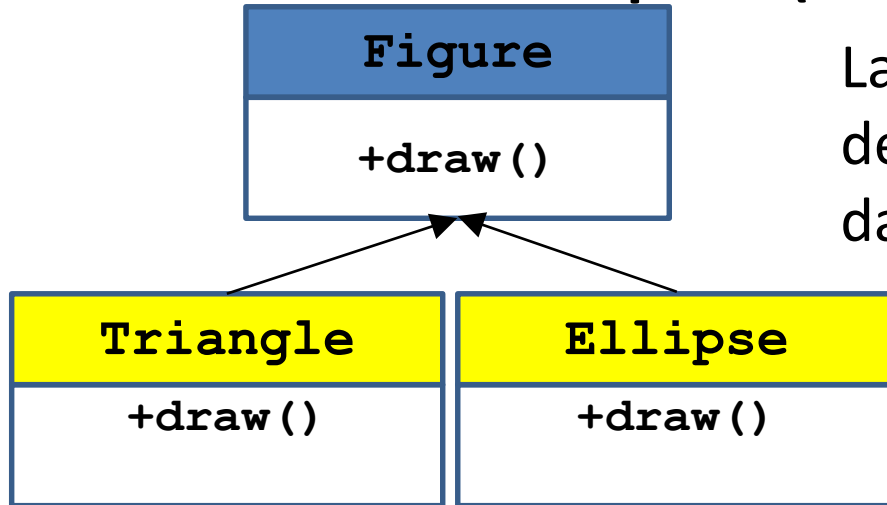
C'est le type des objets effectivement pointés (connu à l'exécution)

	Type statique	Type dynamique
<code>oB</code>	<code>B</code>	
<code>poB</code>	<code>B*</code>	<code>D*</code>
<code>roB</code>	<code>B&</code>	<code>D</code>

4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
- 7. Fonctions virtuelles et Polymorphisme**
8. Classes abstraites
9. Héritage multiple

4.7 Exemple (non polymorphe)

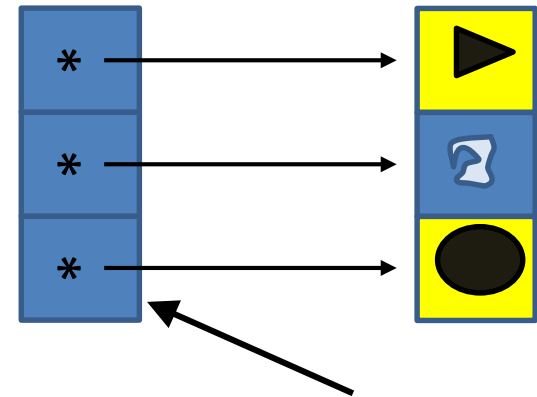


La méthode `draw()` **doit** être définie dans `Figure`, et redéfinie dans `Triangle` **et** `Ellipse`.

```
Figure *image[3];
image[0] = new Triangle();
image[1] = new Figure();
image[2] = new Ellipse();

for (int i = 0; i < 3; i++)
    image[i]->draw() ;
```

Quelle méthode sera appelée ?

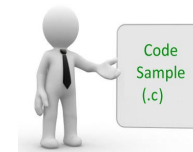


Accès selon le type statique !

`Figure::draw()`

`Figure::draw()`

`Figure::draw()`

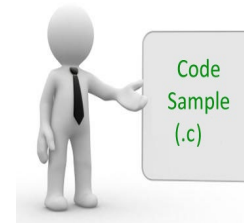


4.7 Fonctions virtuelles

- Dans l'exemple précédent, un pointeur sur une Figure permet de pointer indifféremment un Triangle, un Rectangle ou une Ellipse.
- Comment l'appel de la méthode `draw()` est-il résolu ?
- Solution : **Polymorphisme / fonctions virtuelles**



_vptr Table fncts virtuelles



07_OccupationMémoire

4.7 Fonctions virtuelles

C++ peut choisir la méthode à appeler selon la nature de l'objet pointé à l'exécution du programme (**type dynamique**).

Pour cela, la méthode doit être qualifiée par le mot-clé **virtual** dans la classe de base.

```
virtual void draw();
```



Bonne pratique^(C++11): suffixer la méthode redéfinie avec **override**

```
void draw() override;
```

*Toute classe comportant une fonction virtuelle devra avoir un **destructeur virtuel** sinon il risque d'y avoir une fuite de mémoire (destruction partielle)*



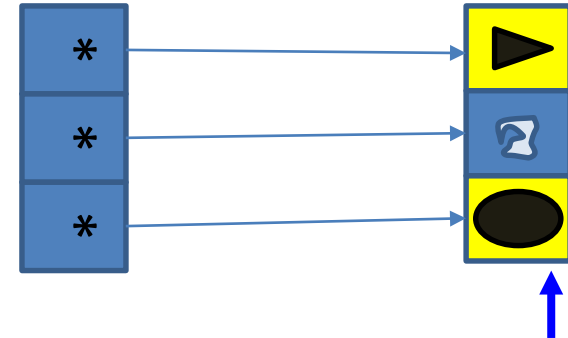
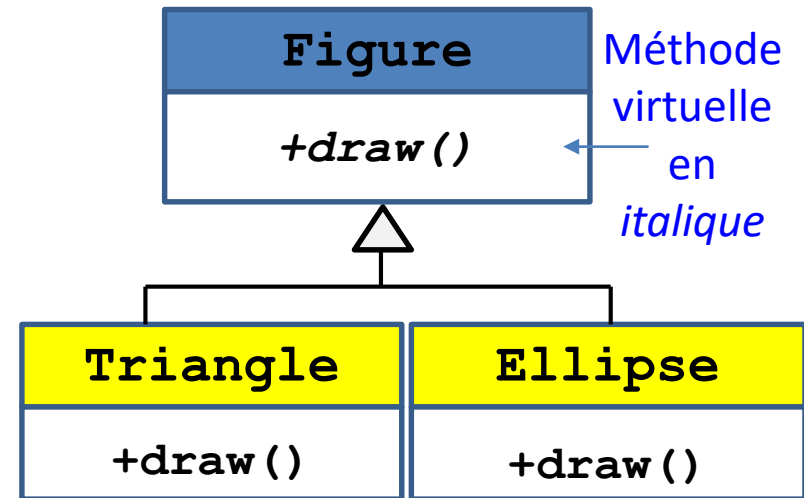
Questions
4.2Q

4.7 Polymorphisme

```
class Figure
{virtual void draw();}

class Triangle: public Figure
{void draw() override;}

main() {
    Figure *image[3];
    image[0] = new Triangle();
    image[1] = new Figure();
    image[2] = new Ellipse();
    ...
    for (int i = 0; i < 3; i++)
        image[i]->draw();
}
```



Accès via le **type dynamique!**

Triangle::draw()

Figure::draw()

Ellipse::draw()



4.7 Polymorphisme

Capacité de prendre plusieurs formes

πολύ·μορφος

- Le polymorphisme est un point essentiel de la programmation objet après l'abstraction de données (encapsulation) et la notion d'héritage.
- Il permet d'appeler la **méthode** d'un objet sans se soucier de son type statique, et qu'elle **s'adapte au type dynamique**.
- Il est implémenté en C++ par les **fonctions virtuelles** (`virtual`).

4.7 Polymorphisme

- Le **polymorphisme** est la faculté pour des objets de différents types (classes) de répondre à des appels de méthodes portant le même nom, chacun correspondant à un code spécifique à chaque type. **Le programmeur n'a pas besoin de connaître à l'avance le type de l'objet**, il pourra être déterminé à l'exécution (type dynamique).
- Un type (classe) possédant des fonctions virtuelles est nommé **type polymorphique**.
- Pour obtenir un **comportement polymorphique en C++**, les **fonctions membres** appelées doivent être **virtuelles** et les **objets** doivent être **manipulés avec des pointeurs ou des références**

4.7 Polymorphisme

Pour que le polymorphisme soit **effectif**, il faut une fonction f qui soit:

- une **fonction membre** d'une classe B,
- **redéfinie dans des classes dérivées** de B,
- **appelée à travers des pointeurs ou des références**
(sur des objets de B ou de classes dérivées de B),
- **déclarée comme fonction virtuelle**
(sa déclaration est précédée du mot clé *virtual*) .

Ainsi, si f est **appelée à travers un pointeur ou une référence** sur un objet de classe C, le choix de la fonction effectivement exécutée (parmi les diverses redéfinitions de f se fera d'après le **type dynamique** de cet objet.

4.7 Destructeurs virtuels

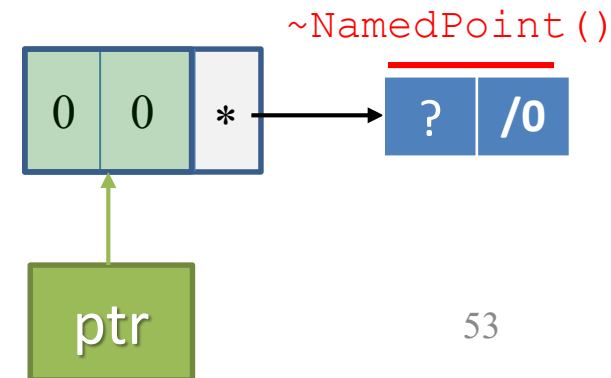
Si une classe a un comportement polymorphe (fonction virtuelle), son destructeur doit aussi être virtuel, mot clé: **virtual**.

Sinon, risque de *memory leak* :

```
class Point
{
    virtual ~Point() {}
};

class NamedPoint: public Point
{
    ~NamedPoint() { delete[] name; }
};
```

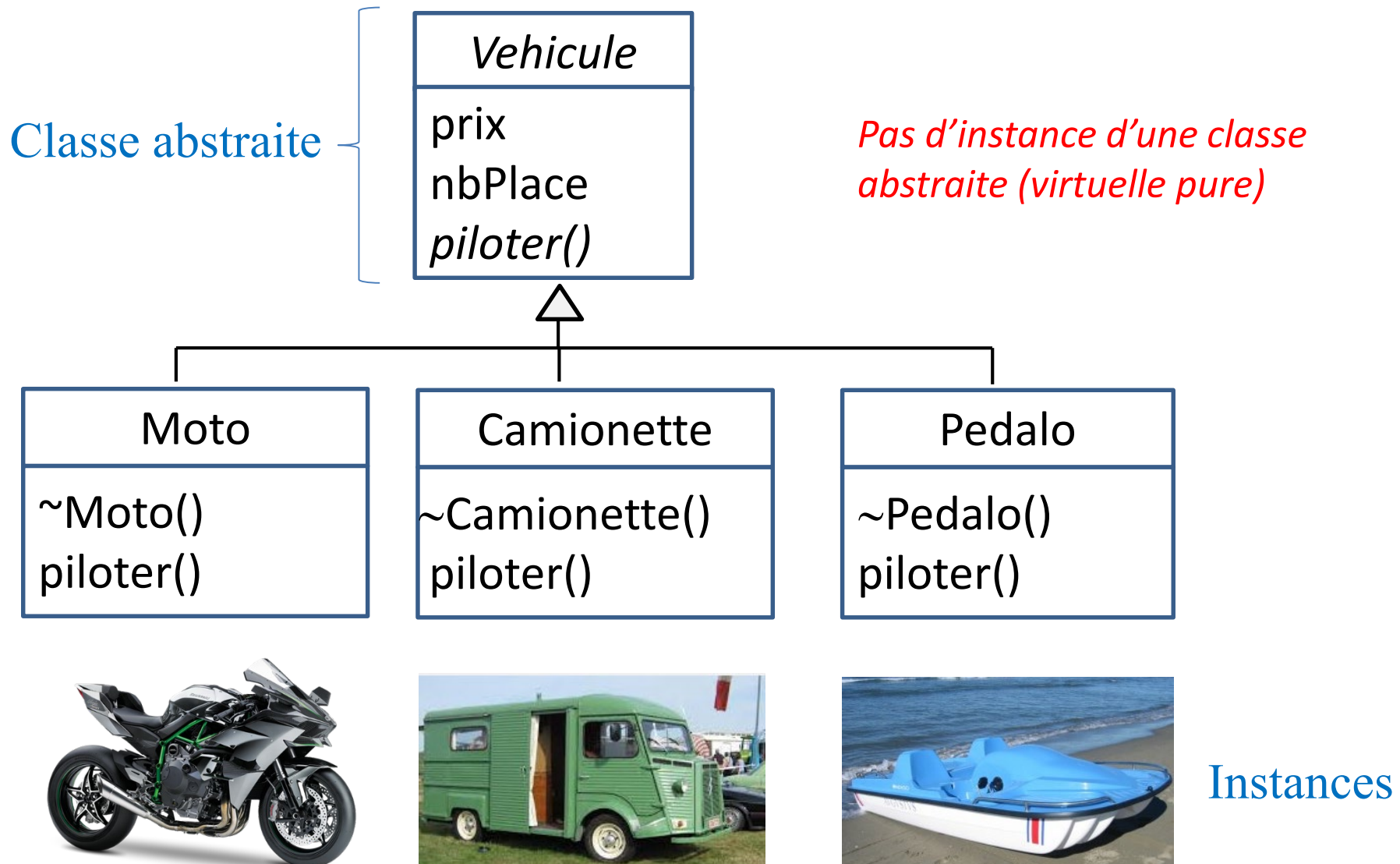
```
Point* ptr= new NamedPoint;
...
delete ptr;
```



4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
- 8. Classes abstraites**
9. Héritage multiple

4.8 Classe abstraite



4.8 Classes abstraites

- Parfois, une classe de base est utilisée pour regrouper les **caractéristiques communes** de plusieurs classes (`Vehicule`, `Animal`, ...)
- Certaines méthodes peuvent ne pas être implémentées** dans une classe de base, mais **doivent** l'être dans toutes les classes dérivées. Ces méthodes sont appelées **virtuelles pures**.

```
virtual void show() = 0;
```

- Une classe qui contient au moins une méthode virtuelle pure ne pourra pas être instanciée: **classes abstraites**
- Une classe abstraite **doit être dérivée** et ses méthodes virtuelles pures **redéfinies** dans les classes dérivées.

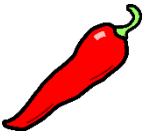


Série 4.3

4. Table des matières

1. Héritage: classe dérivée
2. Contrôle d'accès
3. Mode de dérivation
4. Hiérarchie de classes
5. Conversions de types
6. Type statique et Type dynamique
7. Fonctions virtuelles et Polymorphisme
8. Classes abstraites
- 9. Héritage multiple**

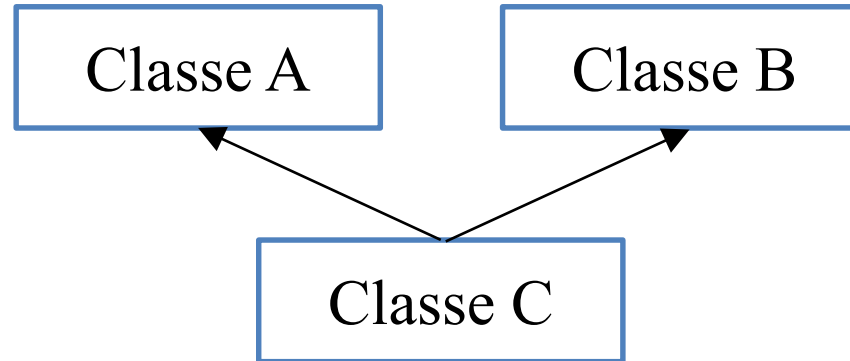
4.9 Héritage multiple



- En C++, une sous-classe peut hériter de **plusieurs super-classes** :



Héritage multiple.pdf

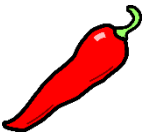


- Comme pour l'héritage simple, la sous-classe **hérite**, à partir des super-classes, de:
 - **tous** leurs attributs et méthodes (*sauf constr./ destructeurs*)
 - leur type
- Ordre d'appel des constructeurs



Série
4.4Q

4.9 Héritage multiple



```
class A{
    int a;
    void f()
        {cout << "A::f " << a;}
    A(int i):a(i){cout<<"Cons A";}
};
```

```
class B{
    int b;
    void f()
        {cout << "B::f " << b;}
    B(int i):b(i){cout<<"Cons B";}
};
```

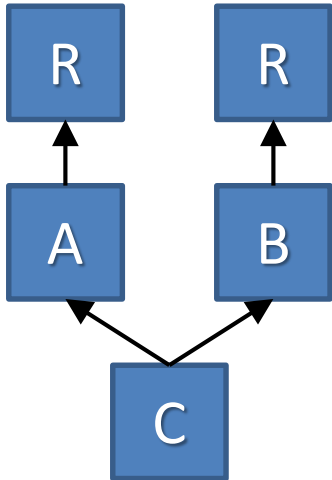
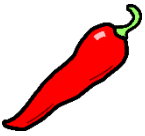
```
class C: public B, public A {
    int c;
    void f()
        {cout << "C::f " << c; }
    C(int i, int j, int k=100):
        A(i),B(j),c(k)
        {cout << "Cons C"; }
};
```

```
main{
    C c(10,20);
    c.A::f(); // f() de A
    c.B::f();
    c.f();
}
```

```
Constructeur B
Constructeur A
Constructeur C
A::f 10
B::f 20
C::f 100
```

- Définit l'ordre d'appel des constructeurs
- Appel des constructeurs

4.10 Héritage virtuel



Héritage multiple avec une superclasse commune .
Comment accéder aux membres de R ?

4.10 Héritage virtuel : Problématique (1)

```
class R{
    int r;
    R(int i):r(i){}
    void fr() ←
    {cout<<"V:fr r="<< r;}
};
```

```
class A : public R
{
    A(int i):R(i){}
};
```

```
class B : public R
{
    B(int i):R(i){}
};
```

```
class C : public A, public B {
    C(int i0, int i1):A(i0)B(i1){}
};
```

C c(10, 20);

Essai 1

```
int main()
{
    C c(10,20);
    c.fr();
}
```

Compilation:
**error request for
member fr is ambiguous**

Essai 2

```
int main()
{
    C c(10,20);
    c.R::fr();
}
```

Compilation:
**error 'R' is an
ambiguous base of C**

4.10 Héritage virtuel : Problématique (2)

```
class R{
    int r;
    R(int i):r(i){}
    void fr()
        {cout<<«R:fr r="<< r;}
};
```

```
class A : public R
{
    A(int i):R(i){}
};
```

```
class B : public R
{
    B(int i):R(i){}
};
```

```
class C : public A, public B {
    C(int i0, int i1):A(i0)B(i1){}
};
```

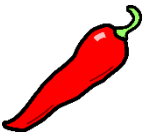
Même code que page précédente

Essai 3

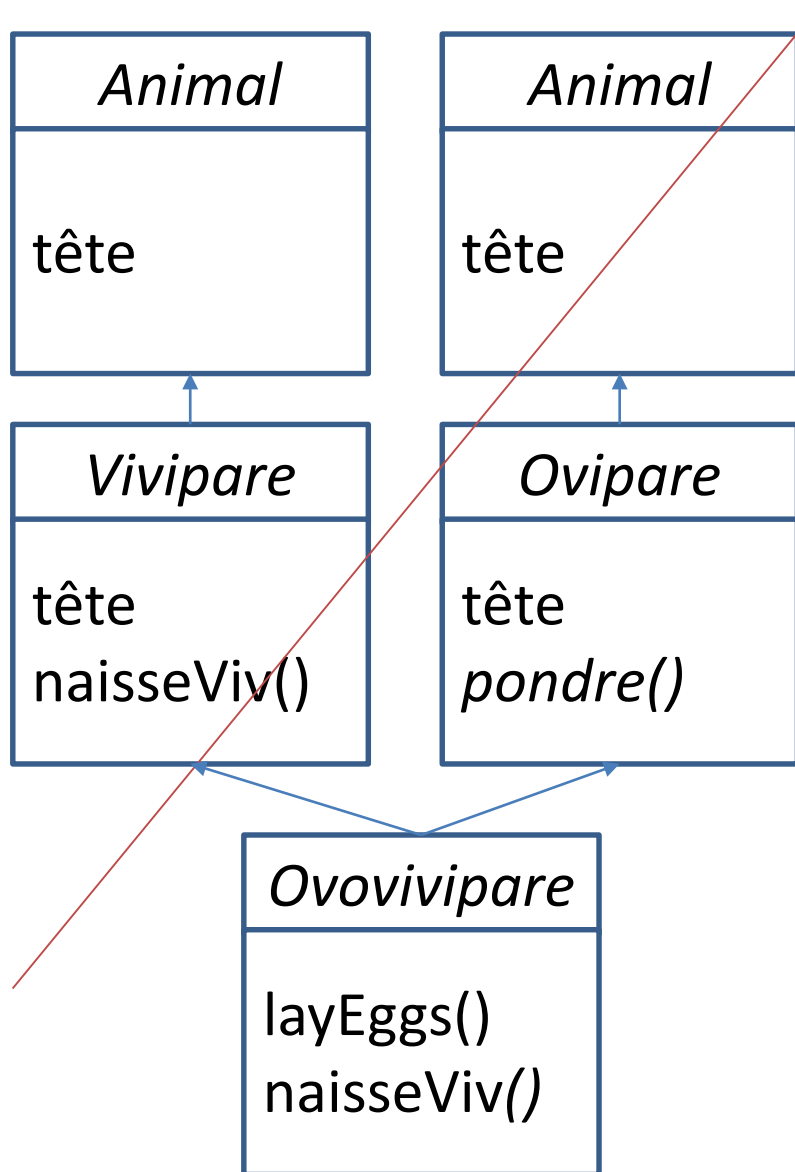
```
int main()
{
    C c(10,20);
    c.A::fr();
    c.B::fr();
}
```

```
Constructeur R 10
Constructeur A 10
Constructeur R 20
Constructeur B 20
R:fr r=10
R:fr r=20
```

Il y a deux instances de R → on doit passer par les classes dérivée pour y accéder

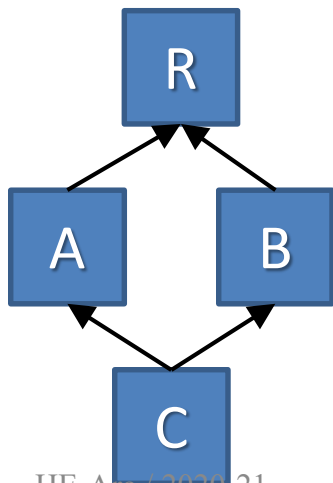
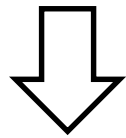
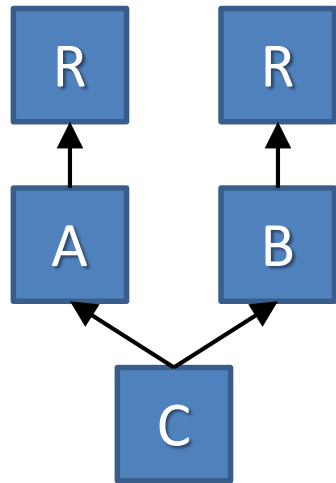
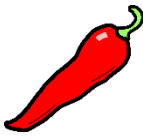


4.10 Héritage virtuel



Problème:
Une instance de la classe
Ovovivipare a 2 têtes!

4.10 Héritage virtuel: Solution



Solution:

Principe: On lève cette ambiguïté en n'utilisant qu'une seule instance de la classe R

Résolution par classes virtuelles

- A et B **héritent virtuellement** de la classe R => A et B vont se partager la classe racine R.
- Les classes virtuelles sont construites avant les classes non-virtuelles
- Les constructeurs des classes de bases sont invoqués dans l'ordre de la liste de la dérivation.

4.10 Héritage virtuel : Solution



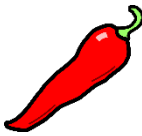
```
class R {  
    int r;  
    R(int i):r(i)  
        {cout << "Cons R";}  
    void fr()  
        {cout<< "R:fr()  r="<< r;}  
};
```

```
class A : virtual public R  
{  
    A(){cout<< "Cons A";}  
};
```

```
class B : virtual public R  
{  
    B(){cout<< "Cons B";}  
};
```

```
class C : public B, public A {  
    C(int i0, int i1): A(),B(),R(i0+i1)  
        {cout << "Cons C";}  
};
```

4.10 Héritage virtuel : Solution



```
int main()
{
    C c(10,20);
    c.fr();
    return 0;
}
```

```
Cons R
Cons B
Cons A
Cons C
R:fr() r = 30
```

Ordre d'appel des constructeurs

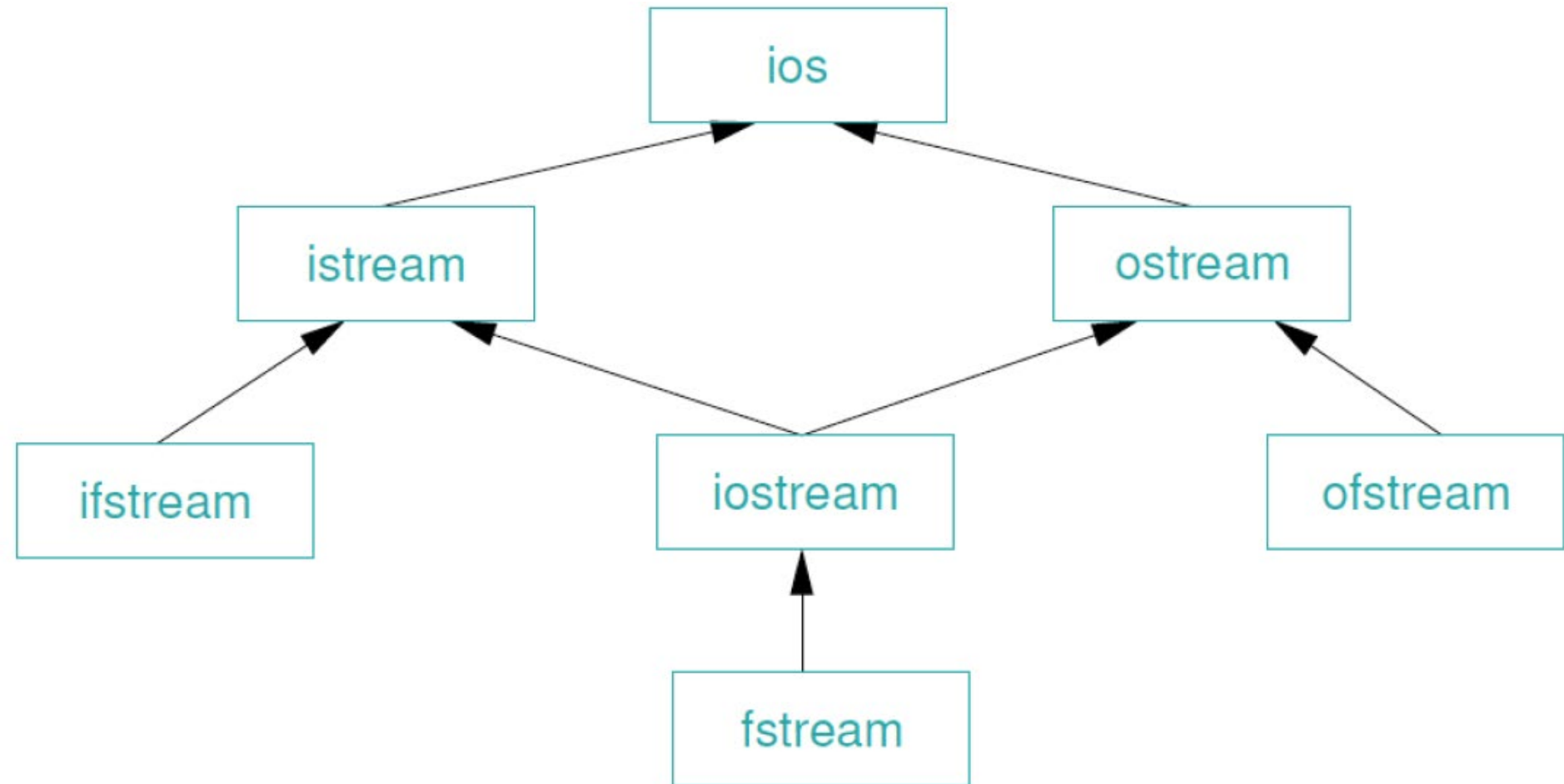
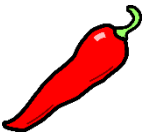
```
class C : public B, public A {
    C(int i0, int i1): A(),B(),R(i0+i1)
    {cout << "Cons C";}
};
```

Appel explicite au constructeur de R

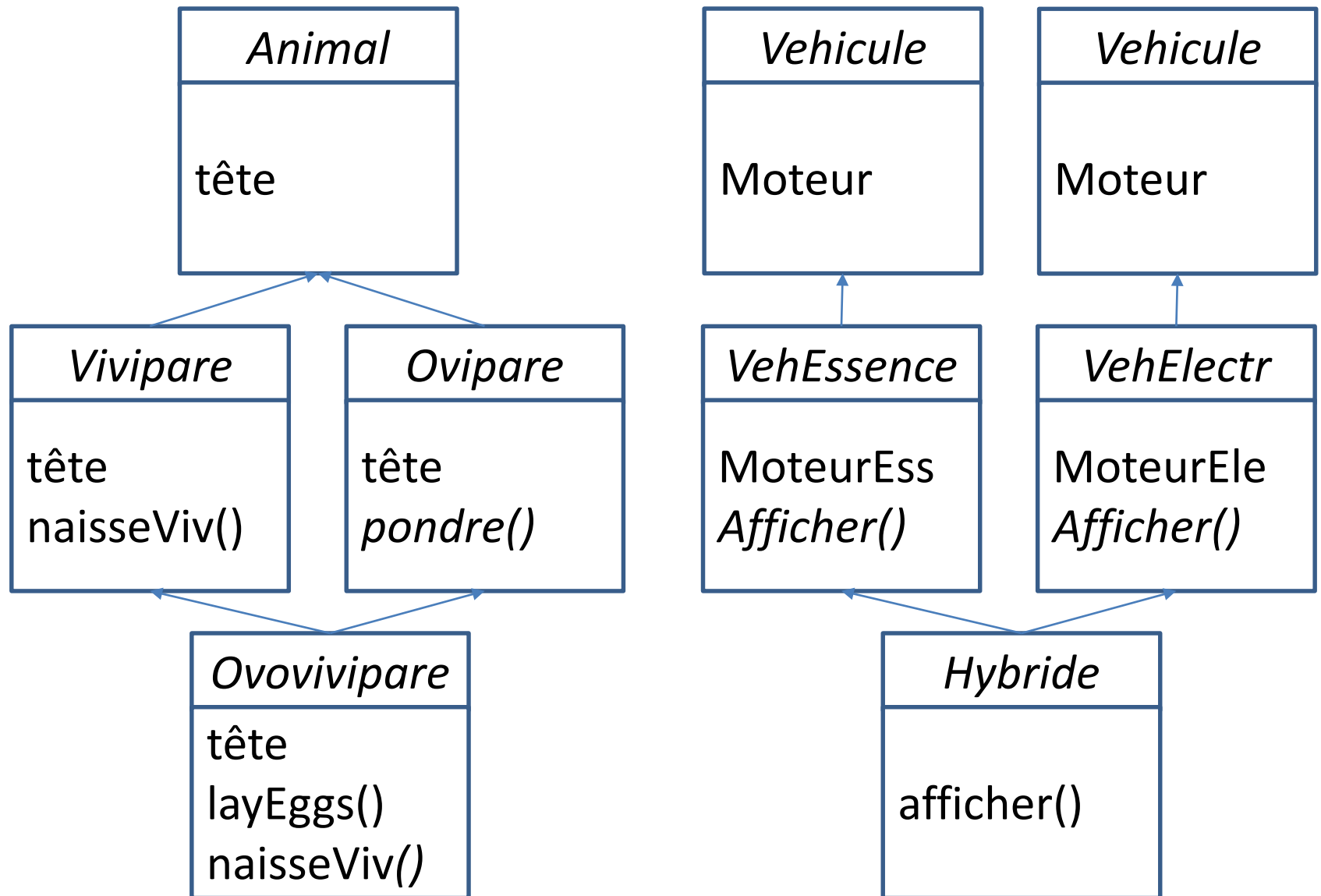
Les classes virtuelles sont
construites avant les classes non-
virtuelles

C'est la classe C qui appelle
le constructeur de R, plus A ni B

4.10 Héritage virtuel



4.11 Héritage multiple: Problématique 2



4.11 Résumé: Héritage

Spécifier un lien d'héritage:

```
class Sousclasse: [public] SuperClass { ... };
```

Droits d'accès

protected *accès autorisé au sein de la hiérarchie.*

Masquage

attributs et méthodes peuvent être **redéfinis** dans une sous-classe

Accès à un membre caché: `SuperClasse::membre`

Liste d'initialisation (constructeur de la superclasse)

```
class SousClasse : public SuperClasse
{
    SousClasse(liste de paramètres) : SuperClasse(Arguments),
                                         attribut1(val1), ..., attributN(valN)
    { /* code constructeur SousClasse */ }
};
```

4.11 Résumé: Polymorphisme

Résolution dynamique des liens : choix des méthodes à invoquer **lors de l'exécution du programme en fonction de la nature réelle des instances.**

2 ingrédients :

méthodes virtuelles et références/pointeurs

Méthode virtuelle :

virtual Type nom_methode(liste d'arguments) [const];

Méthode virtuelle *pure (abstraite)* :

virtual Type nom_methode(liste d'arguments) = 0;

Classe abstraite : contient *au moins une méthode abstraite*

Collection hétérogène : des pointeurs sur les instances doivent être manipulés, et non pas les instances directement.

4.11 Résumé: Héritage multiple

```
class nomSousClasse :    [public] nomSuperClasse1, ...
                        [public] nomSuperClasseN
```

Collision de noms d'attributs/méthodes : c'est la sous-classe qui hérite de ces attributs, méthodes qui doit définir *le sens de leur utilisation*.

Classe virtuelle

pour éviter qu'une sous-classe n'hérite plusieurs fois d'une même super-classe, il faut déclarer les dérivations concernées comme **virtuelles**:

NomSousClasse : [public] **virtual NomSuperClasseVirtuelle**

Constructeur

C'est la **classe la plus dérivée qui initialise la super-classe virtuelle**.

```
SousClasse(liste de parametres) :
    SuperClasse1(arguments1) ,
    ...
    SuperClasseN(argumentsN) ,
    attribut1(valeur1)
    ...
    attributK(valeurK)
{ // Autre code du constructeur }
```