

Programmation Orientée Objet (C++) : Héritage multiple

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Vidéos et transparents

<https://www.coursera.org/learn/programmation-orientee-objet-cpp/home/week/6>

 Semaine 6



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 *hérite plusieurs fois d'une même super-classe*, il faut déclarer les dérivations concernées comme **virtuelles**

```
NomSousClasse: [public] virtual NomSuperClasseVirtuelle
```

Constructeur :

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

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

Concepts fondamentaux

- ▶ Buts, syntaxe (aucune difficulté)
- ▶ Ordre d'appel des constructeurs/destructeurs
 - ☞ ordre de déclaration *d'héritage*
- ▶ Sens (= sémantique) de l'héritage multiple ?
 - ▶ diagramme en losange
 - ▶ héritage et classes *virtuel(les)*
 - ▶ appel du constructeur de la classe virtuelle

Etude de cas (simples)

Que faut-il corriger pour que le code suivant compile :

```
class A { public: A(int x) :  
           a(x) {}  
         private: int a; };  
  
class B : public virtual A    { public: B() : A(0) {} }  
  
class C : public virtual A    { public: C() : A(1) {} }  
  
class D : public B, public C {  
  
int main()  
{  
    D d1;  
    return 0;  
}
```

TROIS solutions

Il y a *trois* possibilités :

- ajouter un constructeur par défaut à D avec appel explicite au constructeur de A :

```
class A { public: A(int x) : a(x) {}  
         private: int a;          };
```

```
class B : public virtual A    { public: B() : A(0){} };
```

```
class C : public virtual A    { public: C() : A(1){} };
```

```
class D : public B, public C { public: D() : A(42){} };
```

```
int main()  
{  
    D d1;  
    return 0;  
}
```

TROIS solutions

Il y a *trois* possibilités :

- ajouter un constructeur par défaut à A :

```
class A { public: A(int x = 42) : a(x) {}  
        private: int a; }
```

```
class B : public virtual A { public: B() : A(0) {} };
```

```
class C : public virtual A { public: C() : A(1) {} };
```

```
class D : public B, public C { }
```

```
int main()  
{  
    D d1;  
    return 0;  
}
```

TROIS solutions

Il y a *trois* possibilités :

- supprimer les héritages virtuels

```
class A { public: A(int x) : a(x) {}  
        private: int a;};
```

```
class B : public A { public: B() : A(0) {} };
```

```
class C : public A { public: C() : A(1) {} };
```

```
class D : public B, public C {
```

```
int main()  
{  
    D d1;  
    return 0;  
}
```


Cas numéro 2

Le code suivant compile-t-il ? ➡ **OUI!** ...et sans les
`virtual` ? non ! (ambiguïté)

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 3

Le code suivant compile-t-il ?

 **NON !**

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles :

► supprimer une des ambiguïtés :

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles :

► désambiguïser l'appel :

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.A::f(); // ou B:: ou C::
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles :

► désambigüiser à l'aide de `using` :

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ public: using A::f; }; // ou B::f ou C::f

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles :

► désambiguïser en redéfinissant :

```
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ public: void f() const { cout << "D "; } };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 4

Le code suivant compile-t-il ?

 **NON!**

```
class A
{ public: virtual void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 4 : DEUX solutions

Il n'y a ici que *deux* corrections possibles :
les solutions

- 2 (désambigüiser l'appel) :

```
d1.A::f(); // ni B:: ni C::
```

- et 3 (utiliser `using`) :

```
class D : public B, public C  
{ public: using A::f; }; // ni B::f ni C::f
```

NE fonctionnent **PAS** :

```
no unique final override for  
'virtual void A::f() const' in 'D'
```

De la norme C++ :

« *In a derived class, if a virtual member function of a base class subobject has more than one final override the program is ill-formed.* »

Cas numéro 4 : DEUX solutions

Ce qui ne nous laisse que *deux* corrections possibles :

- ▶ supprimer l'ambiguïté (ce qui n'est souvent pas possible) ;
- ▶ redéfinir la méthode :

```
class A
{ public: virtual void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const override { cout << "B "; } };

class C : public virtual A
{ public: void f() const override { cout << "C "; } };

class D : public B, public C
{ public: void f() const override { cout << "D "; } };
```

Dernière question

Quelle différence entre les cas 3 et 4 ?

Que change le `virtual` ?

Donnez un exemple *illustratif*.

```
D un_d;  
A* ptr(&un_d);  
ptr->f();
```

Pour préparer le prochain cours

- ▶ Vidéos et quiz du MOOC semaine 7 :
 - ▶ Etude de cas : présentation et modélisation du problème [12 :16]
 - ▶ Etude de cas : affichage polymorphique [8 :58]
 - ▶ Etude de cas : surcharge d'opérateur et première version [13 :55]
 - ▶ Etude de cas : modélisation des mécanismes [14 :46]
 - ▶ Etude de cas : copie profonde [14 :36]

- ▶ Le prochain cours :
 - ▶ de 14h15 à 15h (compléments)