Chapitre 12
L'héritage en C++

1- Généralités

✓**L'héritage est une technique rationnelle permettant de réutiliser et de spécialiser les classes existantes.**

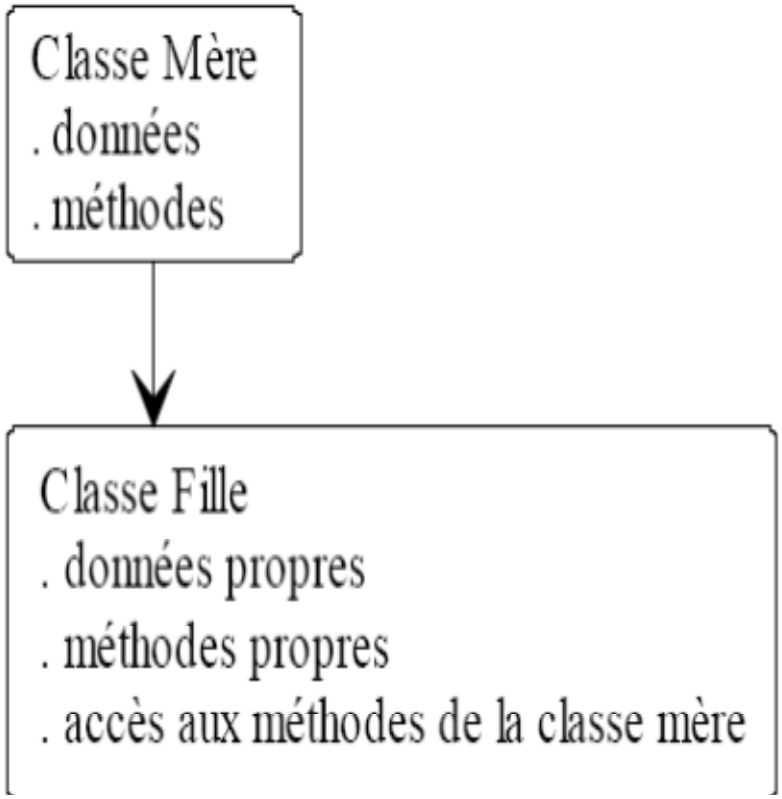
✓**L'héritage offre la possibilité de créer une classe à partir d'une autre.**

✓**Cela signifie que la nouvelle classe bénéficie des attributs (données membres) et des comportements (fonctions membres) de la classe dont elle dérive.**

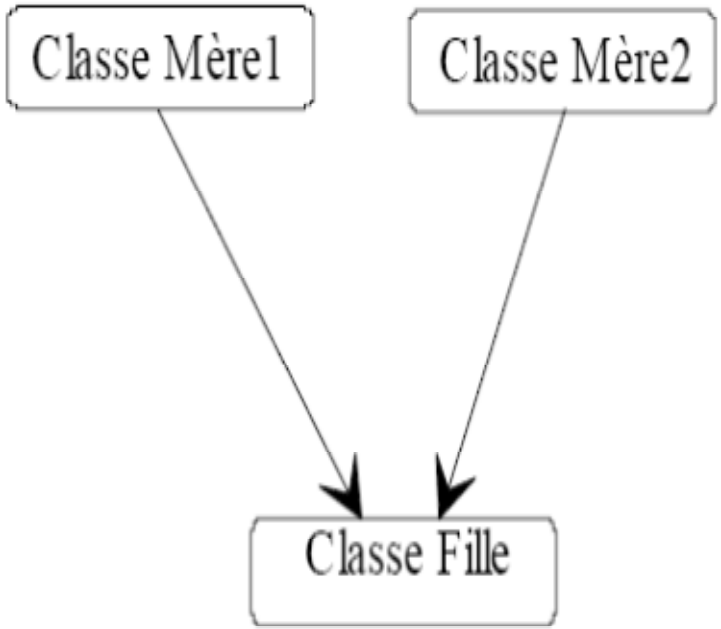
✓**Le concept d'héritage est aussi connu sous le nom de dérivation des classes.**

✓ **La classe dont dérive une classe se nomme la classe de base, la superclasse ou la classe mère… La classe dérivée se nomme la classe fille ou la sous-classe.**

Héritage simple:



Héritage multiple:



2- Syntaxe générale :

NomClasseDérivée : [public|private] ClasseBase1 {, [public|private] ClasseBasei} i=1..n

Exemples :

Déclaration	Commentaire
class Cercle : public ObjetGraphique	Héritage simple : Cercle dérive d'ObjetGraphique en mode public
class TexteGraphique : public ObjetGraphique, public Chaine	Héritage multiple, (double en fait) : TexteGraphique dérive d'ObjetGraphique et de Chaine et ce, à chaque fois en public.
class vecteur3 : private vecteur	vecteur3 hérite uniquement de la classe vecteur mais en private.

vecteur.h

```
class vecteur // classe mère
{
private:
float x,y;
public:
void initialise(float,float);
void homothetie(float);
void affiche();
};
```

vecteur.cpp

```
#include <iostream>
#include <vecteur.h>
using namespace std;
void vecteur::initialise(float abs    ,float ord    )
{
x=abs; y=ord;
}
void vecteur:: homothetie(float val)
{
x = x*val; y = y*val;
}
void vecteur::affiche()
{
cout<<"x = "<<x<<" y = "<<y<<"\n";
}
```

vecteur3.h

```
#include <vecteur.h>
// classe fille
class vecteur3:public vecteur
{
private :
float z;
public:
void initialise3(float,float,float);
void homothetie3(float);
void hauteur(float ha)
{z = ha;}
void affiche3();
};
```

vecteur3.cpp

```
#include <iostream>
#include <vecteur.h>
#include <vecteur3.h>
using namespace std;
void vecteur3::initialise3(float abs   ,float ord   ,float haut   )
{ // appel de la fonction membre de la classe vecteur
   initialise(abs,ord); z = haut;
}
void vecteur3:: homothetie3(float val)
{ // appel de la fonction membre de la classe vecteur
  homothetie(val); z = z*val;
}
void vecteur3::affiche3()
{ // appel de la fonction membre de la classe vecteur
   affiche();
   cout<<"z = "<<z<<"\n";
}
```

main.cpp

```
#include <iostream>
#include <vecteur.h>
#include <vecteur3.h>
using namespace std;
int main()
{
vecteur3 v, w;
v.initialise3(5,4,3);
v.affiche3(); // fonctions de la fille
w.initialise(8,2);
w.hauteur(7);
w.affiche(); // fonctions de la mère
cout<<"*****************************\n";
w.affiche3();
w.homothetie3(6);
w.affiche3(); //fonctions de la fille
return 0 ;
}
```

Remarque :

L’amitié n’est pas transmissible: une fonction amie de la classe mère ne sera amie que de la classe fille que si elle a été déclarée amie dans la classe fille.

3- Le rôle du modificateur d'accès protected

✓**Un attribut** protected **n'est pas accessible à l'extérieur de la classe mais l'est aux classes dérivées.**

✓**Ce modificateur est donc intermédiaire entre** private **et** public**.**

✓**A l'instar de** private**, il respecte le principe d'encapsulation tout en favorisant la transmission des attributs entre classe mère et classe fille.**

✓**Le tableau suivant récapitule les accès fournis par ces trois modificateurs:**

Modificateur d'accès	Visibilité dans les classes filles	Visibilité depuis l'extérieur
Private	Non	Non
Protected	Oui	Non
Public	Oui	Oui

✓**Reprenons maintenant la définition classique de l'héritage :**

La classe dérivée est une forme spécialisée de sa classe mère.

✓**Exprimé ainsi, on peut déduire :**

. *La classe mère transmet à sa classe fille tous ces membres, attributs et méthodes.*

✓**En effet, si un objet de la classe dérivée est une forme spécialisée d'un objet de la classe mère, il est logique qu'il puisse utiliser directement les attributs de la classe mère, donc, de ce point de vue, les attributs doivent être placés en** protected **et l'héritage en mode** public**.**

Exemple:

```
class vecteur // classe mère
{
protected:
float x,y;
public:
vecteur(float,float); // constructeur
vecteur(vecteur &); // constructeur par recopie
void affiche();
~vecteur(); // destructeur
};

void vecteur3::affiche()
{
cout<< "x = "<<x<<" y= "<<y<< " z = "<<z<<"\n";
}
```

- ✓ **La fonction affiche de la classe vecteur3 a accès aux données x et y de la classe vecteur.**
- ✓ **Cette possibilité viole le principe d'encapsulation des données, on l'utilise pour simplifier le code généré.**

4- Spécification de l'héritage

✓**Lorsqu'on dérive une classe, il faut indiquer entre l'opérateur : et le nom de la classe de base, un des mots clés public, private ou protected.**

✓**Dans la pratique, on utilise presque exclusivement la spécification public.**

✓**C++ impose deux règles d'utilisation des étiquettes dans le cas de l'héritage :**

- ➢**Si l'on omet d'employer un de ces trois mots clés, le compilateur utilisera comme valeur par défaut l'étiquette private.**
- ➢**Le choix de l'étiquette ne change rien pour la classe elle-même; la spécification de l'héritage n'intervient que pour les classes dérivées.**

✓**Le choix de spécification d'héritage permet de spécifier comment se propagera l'héritage (en terme d'accès) dans une hiérarchie de classes.**

✓**Cela signifie que l'on peut préciser pour une classe, la politique d'accès de ses propres classes dérivées.**

- ✓**Si B dérive de A avec le mot clé :**
  - ➢***public*** **: La protection des membres de la classe A reste inchangé au niveau de la classe B.**
  - ➢***protected*** **: Les membres *public* et *protected* de A sont considérés comme *protected* dans la classe B. Dans ce cas, les classes dérivées de B ont toujours accès aux membres de A.**
  - ➢***private*** **: Les membres *public* et *protected* de A sont considérés comme *private* dans la classe B. Dans ce cas les classes dérivées de B n'ont plus accès aux membres de A.**

Exemple:

```
class A
{
     public :
          int i ;
protected :
          int j ;
private :
          int k ;
} ;

class B : public A
{    // accès public à i
     // accès protected à j
     // pas d'accès à k (private dans A)
} ;

class C : protected A
{    // accès protected à i (augmentation du niveau de sécurité)
     // accès protected à j
     // pas d'accès à k (private dans A)
} ;

class D : private A
{    // accès private à i (augmentation du niveau de sécurité)
     // accès private à j (augmentation du niveau de sécurité)
     // pas d'accès à k (private dans A)
} ;

class E : public D
{    // pas d'accès à i (private dans D)
     // pas d'accès à j (private dans D)
     // toujours pas d'accès à k
} ;
```

5- Construction et destruction des objets des classes dérivées

✓**Le constructeur d'une classe dérivée doit obligatoirement appeler un constructeur de sa classe de base.**

✓**Lorsqu'on utilise des constructeurs par défaut, le compilateur gère automatiquement l'enchaînement de l'appel des constructeurs, ce qui simplifie la tâche.**

✓**En revanche, et dans le cas général où les constructeurs possèdent des arguments, la mise en place d'une relation d'héritage nécessite de fournir des arguments aux constructeurs de la classe de base par un appel explicite à ce constructeur dans la classe dérivée.**

✓**La destruction des objets liés par une relation d'héritage passe par un enchaînement de l'appel aux classes impliquées.**

✓**Ce processus est simple à mettre en œuvre dans la mesure où une classe ne peut disposer que d'un seul destructeur, ce que le compilateur gère lui-même.**

✓**Chaque fois que l'on crée un objet lié par une relation d'héritage, il y a appel au constructeur de la classe de base, et ainsi de suite… En respectant ce processus, les objets de bases sont créés avant les objets dérivés.**

✓**Pour les destructeurs, c'est l'ordre inverse qui est employé, ce qui signifie que le destructeur d'une classe dérivée est appelé avant celui de sa classe de base.**

```
class A
{
      ...
    public :
        A( ..... ) ; // constructeur
        ~A ( ) ; // destructeur
        .......
};

class B : public A
{
      ...
  public :
     B ( ..... ) ; // constructeur
     ~B( ) ; // destructeur
       ........
};
```

- ✓ **Si on déclare un objet B, seront exécutés**
  - ➢ **Le constructeur de A, puis le constructeur de B,**
  - ➢ **Le destructeur de B, puis le destructeur de A.**
- ✓ **L'exemple suivant illustre ce dernier propos et utilise des constructeurs sans arguments, dont on rappelle qu'ils sont les constructeurs par défaut !**

```
using namespace std;
class vecteur // classe mère
{ protected:
      float x,y;
  public:
      vecteur(); // constructeur
      void affiche();
      ~vecteur(); // destructeur
};
vecteur::vecteur()
      {  x=1; y=2; cout<<"Constructeur de la classe mère\n";}
void vecteur::affiche()
      { cout<<"x = "<<x<<" y = "<<y<<"\n"; }
vecteur::~vecteur()
      { cout<<"Destructeur de la classe mère\n";}

class vecteur3 : public vecteur // classe fille
{ private :
      float z;
  public:
    vecteur3(); // Constructeur
    void affiche();
    ~vecteur3();
};
vecteur3::vecteur3()
      { z = 3; cout<<"Constructeur de la classe fille\n"; }
void vecteur3::affiche()
      { vecteur::affiche(); cout<<"z = "<<z<<"\n"; }
vecteur3::~vecteur3()
      { cout<<"Destructeur de la classe fille\n"; }

void main()
   { vecteur3 v;
     v.affiche(); }
```

✓**Lorsque il faut passer des paramètres aux constructeurs, on a la possibilité de spécifier au compilateur vers lequel des 2 constructeurs, les paramètres sont destinés :**

```cpp
#include <iostream>
using namespace std;
// Héritage simple
class vecteur // classe mère
{ protected:
    float x,y;
  public:
    vecteur(float,float); // constructeur
    void affiche();
    ~vecteur(); // destructeur
};
vecteur::vecteur(float abs=1, float ord=2)
        { x=abs;y=ord; cout<<"Constructeur de la classe mère\n";}
void vecteur::affiche()
        { cout<<"x = "<<x<<" y = "<<y<<"\n";}
vecteur::~vecteur()
        { cout<<"Destructeur de la classe mère\n"; }

class vecteur3:public vecteur // classe fille
{ private:
     float z;
  public:
     vecteur3(float, float, float); // Constructeur
     void affiche();
     ~vecteur3();
};
vecteur3::vecteur3(float abs=3, float ord=4, float haut=5):vecteur(abs,ord)
{// les 2 1ers paramètres sont pour le constructeur de la classe mère
   z = haut; cout<<"Constructeur fille\n";
   }
void vecteur3::affiche()
    { vecteur::affiche();
      cout<<"z = "<<z<<"\n"; }
vecteur3::~vecteur3()
{ cout<<"Destructeur de la classe fille\n"; }
```

```
int main()
{ vecteur u;
vecteur3 v, w(7,8,9);
u.affiche();
v.affiche();
w.affiche();
return 0;
}
```

Cas du constructeur par recopie

Rappel :

✓ **Le constructeur par recopie est appelé dans 2 cas :**

- **Initialisation d'un objet par un objet de même type :**

```
vecteur a (3,2) ;
vecteur b = a ;
```

- **Lorsqu'on passe à une fonction un objet par valeur :**

```
vecteur a, b ;
bool x;
 x = a.compare(b) ;
```

✓ **Dans le cas de l'héritage, on doit définir un constructeur par recopie pour la classe fille, qui appelle le constructeur par recopie de la classe mère.**

```
// classe mère
class vecteur
{ protected:
    float x,y;
  public:
     vecteur(float,float); // constructeur
     vecteur(vecteur &); // constructeur par recopie
     void affiche();
     ~vecteur(); // destructeur
};
vecteur::vecteur(float abs=1, float ord=2)
    { x=abs; y=ord; cout<<"Constructeur de la classe mère\n"; }
vecteur::vecteur(vecteur &v)
    { x=v.x; y=v.y; cout<<"Constructeur par recopie de la classe mère\n";}
void vecteur::affiche()
{  cout<<"x = "<<x<<" y = "<<y<<"\n"; }
vecteur::~vecteur()
{ cout<<"Destructeur de la classe mère\n"; }

class vecteur3:public vecteur // classe fille
{ private:
      float z;
  public:
      vecteur3(float, float, float); // Constructeur
      vecteur3(vecteur3 &); // Constructeur par recopie
      void affiche();
      ~vecteur3();
};
vecteur3::vecteur3(float abs=3, float ord=4, float haut=5) :vecteur(abs,ord)
   { z = haut; cout<<"Constructeur de la classe fille\n"; }
vecteur3::vecteur3(vecteur3 &v) : vecteur(v)// appel du constructeur par recopie de la classe vecteur
   { z = v.z; cout<<"Constructeur par recopie de la classe fille\n"; }
void vecteur3::affiche()
    { vecteur::affiche(); cout<<"z = "<<z<<"\n"; }
vecteur3::~vecteur3()
    { cout<<"Destructeur de la classe fille\n"; }
```

```
int main()
{
vecteur3 v(5,6,7);
vecteur3 w = v;
 v.affiche();
 w.affiche();
 return 0;
}
```

6- Conversions de type entre classes mère et fille

Règle :

✓**La conversion d'un objet de la classe fille en un objet de la classe mère est implicite. La conversion d'un objet de la classe mère en un objet de la classe fille est interdite.**

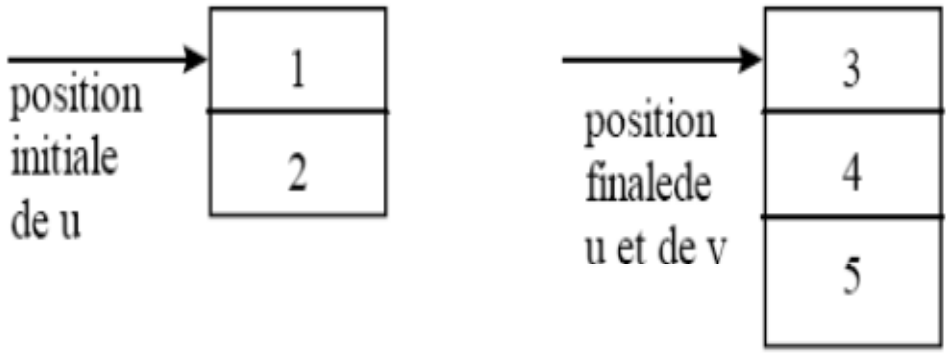
✓**Autrement dit :**

```
vecteur u ;
vecteur3 v;
u = v ;// est autorisée : conversion fictive de v en un vecteur
v = u; // est interdite
```

Exemple:

```
vecteur *u ;
// réservation de mémoire, le constructeur de vecteur est exécuté
u = new vecteur ;
vecteur3 *v ;
// réservation de mémoire, le constructeur de vecteur3 est exécuté
v = new vecteur3 ;
u = v ; // autorisé, u vient pointer sur la même adresse que v
v = u; // interdit
delete u ;
delete v ;
```

On obtient la configuration mémoire suivante :



Remarques :

- **Tous les membres d'une classe, et notamment les méthodes, sont hérités par la classe dérivée. Cependant, il y a quelques exceptions. D'abord les constructeurs et destructeurs ne sont pas hérités, ils ont leurs propres règles.**

- **Les opérateurs sont hérités normalement, comme d'autres fonctions. Cependant, aucune opération n'est réalisée sur les nouveaux membres, c'est pourquoi il est généralement préférable de redéfinir ces opérateurs.**

- **Enfin l'opérateur d'affectation est un cas particulier, car il n'est pas à proprement parler hérité non plus. Lorsqu'il n'est pas redéfini explicitement dans une classe dérivée, il recopie membre à membre les nouveaux membres de cette classe dérivée, et appelle l'opérateur d'affectation de la classe de base pour la copie de la partie héritée. Lorsqu'on le redéfinit pour la classe dérivée, l'opérateur pour la classe de base n'est pas appelé, il faut donc le faire explicitement.**

7- Héritage multiple, constructeurs et destructeurs

✓**Pour les destructeurs, il n'y a toujours aucun problème. Quand aux constructeurs, le problème se pose uniquement dans le cas où l'on utilise des constructeurs avec arguments.**

✓**On rappellera seulement que si l'on ne crée aucun constructeur, le C++ crée le constructeur sans argument par défaut et qu'il l'appelle automatiquement si aucun appel n'est explicité dans le constructeur d'une classe dérivée.**

✓**On donne maintenant un exemple d'implémentation de l'héritage en utilisant des constructeurs avec arguments, ce qui doit suffire à éclairer la syntaxe de l'héritage multiple.**

```
class A
     {    protected :
               int a ;
          public :
               A(int aa) ;
} ;
class B
{          protected :
               int b ;
          public :
               B(int bb) ;
     } ;
class C : public A, public B
     {    protected :
               int c ;
          public :
               C(int aa, int bb, int cc) ;
      } ;

A::A(int aa)
    {   a = aa ;
        cout << "Constructeur A \n";
    }

B::B(int bb)
{    b = bb ;
     cout << "Constructeur B \n"
}

C::C(int aa, int bb, int cc):A(aa), B(bb)
{    c = cc ;
     cout << "Constructeur C \n"
}
```

Le programme suivant :

```
int main()
{
    C objetC(1,2,3) ;
    return 0 ;
}
```

affiche le résultat suivant :

Constructeur A
Constructeur B
Constructeur C

8- Les classes virtuelles

✓Les classes virtuelles sont généralement utilisées dans le cas de l'héritage multiple afin de permettre à plusieurs classes de partager le même objet de base sans conflit !

✓En fait, elles n'ont qu'un seul objectif : compenser un mauvais diagramme de classe.

✓Considérons les déclarations de classes suivantes :

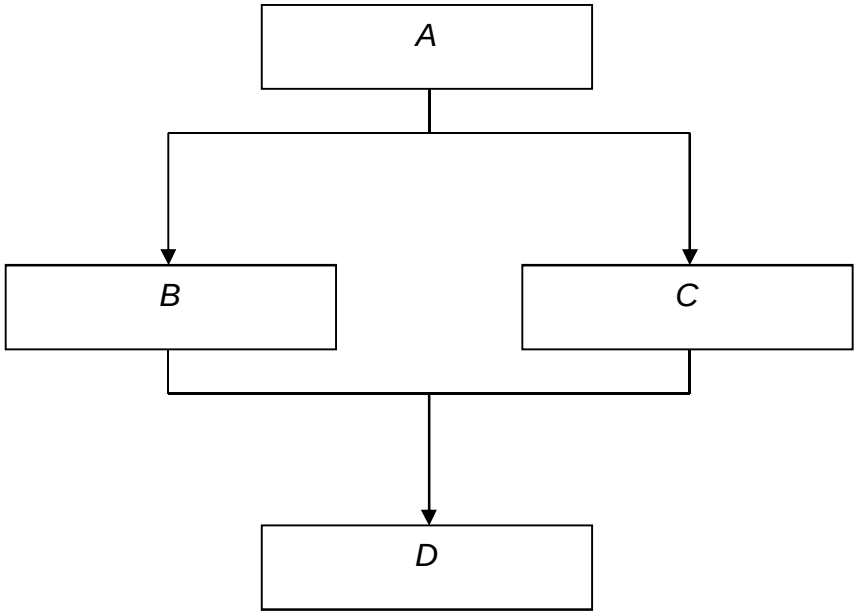
```
class A
{
} ;

class B : public A
{
} ;

class C : public A
{
} ;

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

On donne le diagramme de classes correspondant :



✓**En tenant compte de la hiérarchie des classes, si l'on souhaite créer un objet de type D, le C++ va créer les objets suivant :**

B, A (car B dérive de A), C, A (car C dérive de A) et enfin D.

✓**Cette allocation respecte le diagramme de classes mais risque de provoquer des erreurs si à partir de l'objet de type D, on souhaite accéder aux membres de la classe A.**

✓**Dans ce cas, le compilateur générera un message d'erreur du type *Ambiguous member* dans la mesure où il ne saura pas quel objet choisir !**

✓**Pour résoudre ce problème, le C++ propose la possibilité de faire de la classe A une classe virtuelle. De cette manière, il n'existera qu'un seul objet de type A pour chaque objet de type D.**

✓**L'exemple suivant met en œuvre la syntaxe à respecter pour définir une classe virtuelle (mot clé *virtual* devant la spécification d'héritage relatif à la classe A).**

```
class A
    {   public :
            int a ;
            A(int ) ;
    } ;
class B : virtual public A
    {   public :
            int b ;
            B(int , int ) ;
    } ;
class C : virtual public A
    {   public :
            int c ;
            B(int , int ) ;
    } ;

class D : public B, public C
    {   public :
            int c ;
            C(int , int , int , int ) ;
      } ;

A::A(int aa)
    { a = aa ; cout << "Constructeur A \n"; }
B::B(int aa, int bb) :A(aa)
   { b = bb ; cout << "Constructeur B \n"; }
C::C(int aa, int cc):A(aa)
  { c = cc ; cout << "Constructeur C \n"; }
D::D(int aa, int bb, int cc, int dd):A(aa), B(aa, bb), C(aa, cc)
  { d = dd ; cout << "Constructeur D \n" }
```

Le programme suivant :

```
int main()
{
    D objetD(1,2,3,4) ;
    return 0 ;
}
```

affiche le résultat suivant :

Constructeur A
Constructeur B
Constructeur C
Constructeur D

✓**En plus d'appeler le constructeur de la classe B et C, le constructeur de la classe D appelle aussi celui de la classe A !**

✓**C'est grâce à ce principe qu'il n'y a plus qu'un seul objet de type A créé pour un objet de type D.**

✓**Les classes virtuelles offrent peu d'intérêt dans la mesure où elles ne servent généralement qu'à compenser la déficience d'une hiérarchie de classes. Le meilleur choix consiste simplement à éviter ce style d'architecture !**