Université Ibn Tofail
جامعة ابن طفيل
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵉⴱⵏ ⵟⴼⴰⵢⵍ
Ibn Tofail University

POO sous C++
Héritage

Med. AMNAI

Filière SMI - S5

Département d'Informatique

14 décembre 2020

Plan

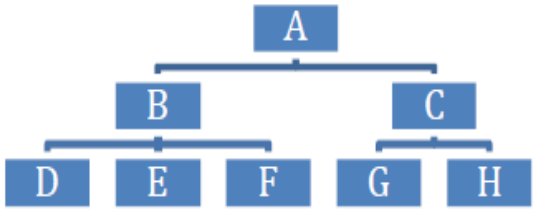
Plan

Plan

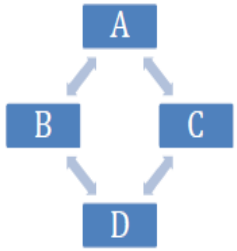
Plan

L'héritage en général

D'une façon générale, l'héritage peut être représenté par les arbres suivants :



Héritage simple



Héritage complexe

Classe de base 'point' définie dans le fichier 'point.h'

```
class point{
    int x, y;
    public:
        void initialise(int,int);
        void deplace(int,int);
        void affiche();
    };
    void point::initialise(int a,int b){x=a; y=b;}
    void point::deplace(int a,int b){x=x+a; y=y+b;}
    void point::affiche(){cout<<"Point : "<<x<<" - "<<y<<endl;}
```

Classe 'point_colore' derivée de la classe 'point'

- La déclaration '**class point_colore : public point** spécifie que la classe '**point_colore**' est **dérivée** de la classe de base '**point**'.

- Le mot clé '**public**' signifie que les membres publics de la class de base seront des membres publics de la classe dérivée.

```
//La classe 'point_colore' derivée de la class 'point'

#include <point.h>

Class point_colore : public point{
    int couleur;
    public:
        void colorer(int c) {
            couleur=c;
        }
};
```

Exemple

```
main()
{
    point_colore p;
    p.initialise(12,27);
    p.colorer(7); p.affiche();
    p.deplace(1,2); p.affiche();
}
```

```
Point : 12 - 27
Point : 13 - 29
```

Utilisation dans une classe dérivée

- Après avoir fait appel à la méthode '**colorer**', la méthode '**affiche**' ne donne aucune information sur la couleur d'un point.
- Créer une fonction '**affiche_c**' membre de '**point_colore**' pour afficher les coordonnées **x**, **y** et la couleur **c**.

```
void affiche_c(){
    cout<<"Point : "<<x<<" - "<<y;
    cout<<"En couleur : "<<couleur<<endl;
}
```

Or, une classe dérivée n'a pas accès aux membres privés de la classe de base. La solution est :

```
void affiche_c(){
    affiche(); cout<<" en couleur : "<<couleur<<endl;
}
```

Utilisation dans une classe dérivée (suite)

```
class point{ //usemembre.cpp
    int x, y;
    public:
        void initialise(int a,int b){x=a;y=b;}
        void deplace(int a,int b){x=x+a; y=y+b;}
        void affiche(){cout<<"Point : "<<x<<" - "<<y;}
} ;

class point_colore:public point{
    int couleur;
    public:
        void initialise_c(int c){
            couleur=c;
        }

        void colorer(int c){
            couleur=c;
        }
        void affiche_c(){
            affiche();
            cout<<" En couleur : "<<couleur<<endl;
        }
};

main()
{
    point_colore p;
    p.initialise(12,9); p.colorer(7);
    p.affiche(); cout<<endl; p.affiche_c();
    p.deplace(1,2);p.affiche(); cout<<endl; p.affiche_c();
}
```

```
Point : 12 - 9
Point : 12 - 9 En couleur : 7
Point : 13 - 11
Point : 13 - 11 En couleur : 7
```

Redéfinition des fonctions membres

- Les méthodes '**affiche**' de la classe '**point**' et '**affiche_c**' de la classe '**point_colore**' font un travail analogue. De même pour les fonctions '**initialise**' et '**initialise_c**'.

- Il est possible de **redéfinir** la fonction '**affiche**' ou la fonction '**inialise**' pour la classe **dérivée**.

Exemple

```
// redif.cpp
class point{
    int x, y;
    public:
        void initialise(int,int);
        void deplace(int,int);
        void affiche();
    };
    void point::initialise(int a,int b){x=a; y=b;}
    void point::deplace(int a,int b){x=x+a; y=y+b;}
    void point::affiche(){cout<<"Point : "<<x<<" - "<<y;}

class point_colore:public point{
    int couleur;
    public:
        void colorer(int c){couleur=c;}
        void affiche(){
            point::affiche();
            cout<<" en couleur : "<<couleur<<endl;
        }
        void initialise(int a,int b,int c){
            point::initialise(a,b); couleur=c;
        }
};

main(){
    point_colore p;
    p.initialise(12,27,9);
    p.affiche();
    p.deplace(10,20); p.affiche();
    p.colorer(7); p.affiche();
}
```

```
Point : 12 - 27 en couleur : 9
Point : 22 - 47 en couleur : 9
Point : 22 - 47 en couleur : 7
```

Appel des constructeurs et des destructeurs

- Pour **créer un objet** de la classe **dérivée 'point_colore'**, il faut tout d'abord créer un objet de la classe de **base 'point'**,
- Càd , **faire appel au constructeur** de la classe de base '**point**', le compléter par ce qui est spécifique a la classe '**point_colore**'
- Ensuite **Faire appel au constructeur** de la classe '**point_colore**'.

```
class point{
    int x,y;
    public:
        point(int,int);
};

class point_colore:public point{
    int couleur;
    public:
        point_colore(int,int,int);
};
```

Appel des constructeurs et des destructeurs (suite)

Si on souhaite que le constructeur de la classe '**point_colore**' retransmette au constructeur de la classe '**point**' les premières informations reçues, on écrira :

- **point_colore(int a, int b, int c) :point(a, b) ;**
- Ainsi, la déclaration : '**point_colore p(2,4,5) ;**' entraînera :
  - l'appel du constructeur '**point**' qui recevra les valeurs **2** et **4**.
  - l'appel du constructeur '**point_colore**' qui recevra les valeurs **2**, **4** et **5**.

Il est toujours possible de mentionner les valeurs par défaut :

- **point_colore(int a=0, int b=2, int c=5) :point(a,b) ;**
- Donc la déclaration '**point_colore p(17) ;**' entraîne :
  - appel du constructeur '**point**' avec les valeurs **17** et **2**.
  - appel du constructeur '**point_colore**' avec les valeurs **17**, **2** et **5**.

Exercice

Reprendre le programme précédent en ajoutant les constructeurs et les destructeurs correspondants, tout en affichant les moments de construction et de destruction des objets. Prévoir aussi des objets dynamiques.

Remarques

- Si la classe de *base ne possède pas de constructeur, aucun problème* particulier ne se pose. De même si elle ne possède pas de destructeur.
- En revanche, si la classe *dérivée ne possède pas de constructeur*, alors que la classe de *base en comporte*, le **problème** sera posé lors de la **transmission** des informations attendues par le constructeur de la classe de base.
- La seule *situation acceptable* est celle où la classe de *base dispose d'un constructeur sans arguments*.
- Lors de la transmission d'information au constructeur de la classe de base, on peut utiliser des expressions ou des arguments.

```
point_colore(int a=5, int b=5, int c=4) :point(a*2, b*5);
```

Héritage privé

Pour que l'utilisateur d'une classe dérivée **n'ait pas accès aux membres publics de sa classe de base**, il suffit de remplacer le mot '**public**' par '**private**' dans sa déclaration.

```
class point{
    int x, y;
    public:
        void initialise(int,int) {x=a; y=b;}
        void deplace(int,int) {x=x+a; y=y+b;}
        void affiche(){ ..... }
};

class point_colore : private point{
    int couleur;
    public:
        void colorer(int c){
            couleur=c;
        }
};
```

Si on a : **point_colore O(12,4,6) ;** les appels suivants sont rejetés :

```
O.affiche(); ou O.point::affiche();
O.deplace(1,5); ou O.point::depace(1,5);
```

Remarques

- Cette technique de fermeture d'accès à la classe de base ne sera employée que dans des cas précis,
- lorsque par exemple **toutes les fonctions** utiles de la classe de **base sont redéfinies dans** la classe **dérivée**, et qu'il n'y a aucune raison de laisser l'utilisateur accéder aux anciennes.

Membres protégés d'une classe

- Les membres protégés restent inaccessibles à l'utilisateur (comme les membres privés).
- Mais ils seront accessibles aux membres d'une éventuelle classe dérivée, tout en restant inaccessibles aux utilisateurs de cette classe.

Exemple

```
class point{
    protected:
    int x,y;
    public:
        void initialise(int,int);
        void deplace(int,int);
        void affiche();
};
```

On peut donc déclarer dans la classe '**point_colore**' dérivée de la classe '**point**' une fonction '**affiche**' qui accède aux membres protégés **x** et **y**.

```
class point_colore:public point{
    int couleur;
    public:
        void affiche(){
            cout<<"Point coloré " <<x<<" - " <<y<<endl;
            cout<<"en couleur : "<<couleur<<endl;
        }
};
```

RQ : Lorsqu'une classe dérivée possède une classe **amie**, cette dernière dispose des **mêmes droits d'accès** que les fonctions **membres de la class dérivée**.

Conversion d'un objet dérivé dans un objet d'un type de base

Si on a :

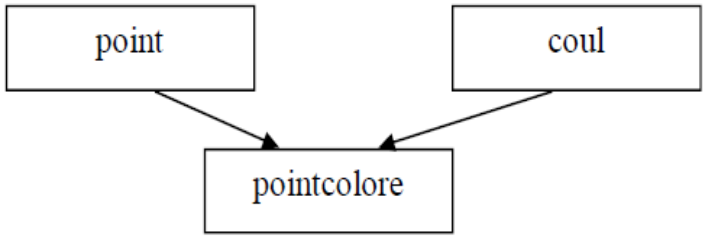
- **point o1 ;**
- **point_colore o2 ;**

Alors :

- l'affectation '**o1=o2 ;** est **juste**.
- l'affectation '**o2=o1 ;**' est **rejetée** (si **o2** a des arguments définis par défaut, on n'aura pas de problème).

Mise en oeuvre de l'héritage multiple

La classe **pointcolore** hérite des deux (multiple) classes '**point**' et '**coul**'.



Mise en oeuvre (Exemple)

```
//hrtmult.cpp
class point{
    int x,y;
    public:
    point(int a,int b){
        x=a ;y=b ;
        cout<<"Const. de point ";
    }
    ~point(){cout<<"Destructeur de point ";}
    void affiche(){cout<<"point "<<x<<" - "<<y;}
};
//-----------------------------------------------------
class coul{
    int couleur ;
    public :
        coul(int c){
            couleur=c ;
            cout<<"Const. de coul ";
        }
        ~coul(){cout<<"Destruction de coul ";}
        void affiche(){cout<<" Couleur : "<<couleur<<endl;}
};
//-----------------------------------------------------
class point_colore:public point,public coul{
    public:
        point_colore(int a,int b,int c):point(a,b),coul(c){
            cout<<"const. de pointcolore\n";
        }
        ~point_colore(){
            cout<<"Destruction de pointcolore\n";
        }
        void affiche(){
            point::affiche();
            coul::affiche();
        }
};
```

Mise en oeuvre (Exemple)

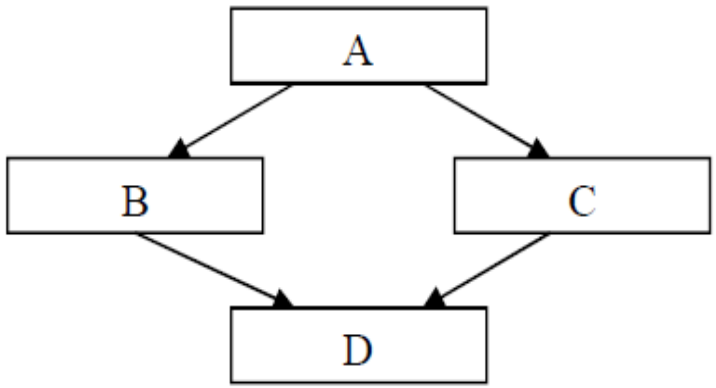
```
class point_colore:public point,public coul{
    public:
        point_colore(int a,int b,int c):point(a,b),coul(c){
            cout<<"const. de pointcolore\n";
        }
        ~point_colore(){
            cout<<"Destruction de pointcolore\n";
        }
        void affiche(){
            point::affiche();
            coul::affiche();
        }
};


//-----------------------------
main(){
    point_colore pt(100,200,3);
    pt.affiche();
    pt.point::affiche();
    pt.coul::affiche();
}
```

```
Const. de point Const. de coul const. de pointcolore
point 100 - 200 Couleur : 3
point 100 - 200 Couleur : 3
Destruction de pointcolore
Destruction de coul Destructeur de point
```

Classes virtuelles

```
class A{
    int x,y;
    ...
};
class B:public A{
    ...
};
class C:public A{
    ...
};
class D:public B, public C{
    ...
};
```



Impliquent que la classe '**D**' hérite deux fois de la classe '**A**', donc les membres de '**A**' vont apparaître deux fois dans '**D**'.

- les **fonctions** membres **ne sont pas** réellement dupliquées.
- les **données** membres seront effectivement **dupliquées**.

Classes virtuelles (suite)

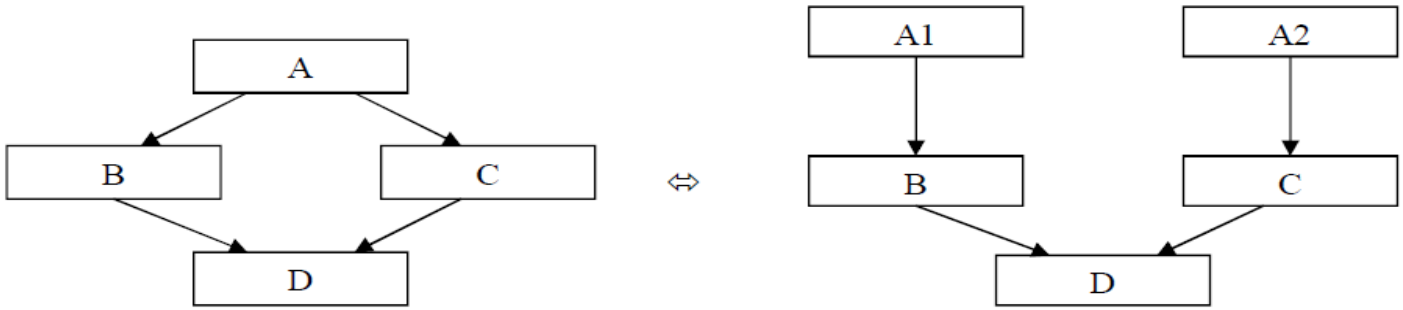
- si on veut laisser cette duplication, on utilise :
  A : :B : :x <> A : :C : :x ou B : :x <> C : :x
- si on **ne veut pas cette duplication**, on précisera la classe '**A**' comme **classe virtuelle** dans les déclarations des classes '**B**' et '**C**'.

```
class A{
    int x,y;
    ...
};
class B:public virtual A{
    ...
};
class C:public virtual A{
    ...
};
class D:public B, public C{
    ...
};
```

Le mot-clé '**virtual**' peut être placé **avant** ou **après** le mot-clé '**public**'.

Appel des constructeurs et des destructeurs

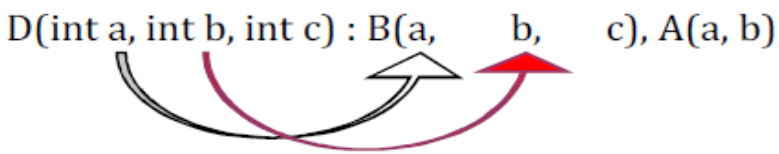
- Si '**A**' **n'est pas déclarée** '**virtual**' dans les classes '**B**' et '**C**', les constructeurs seront appelés dans l'ordre : '**A1**', '**B**', '**A2**', '**C**' et '**D**'.
- Si '**A**' a été déclarée '**virtual**' dans '**B**' et '**C**', on ne construira qu'un seul objet de type de '**A**' (et non pas deux objets).



Quels arguments faut-il transmettre ?

Quels arguments faut-il transmettre alors au **constructeur** ? Ceux prévus par '**B**' ou ceux prévus par '**C**' ?

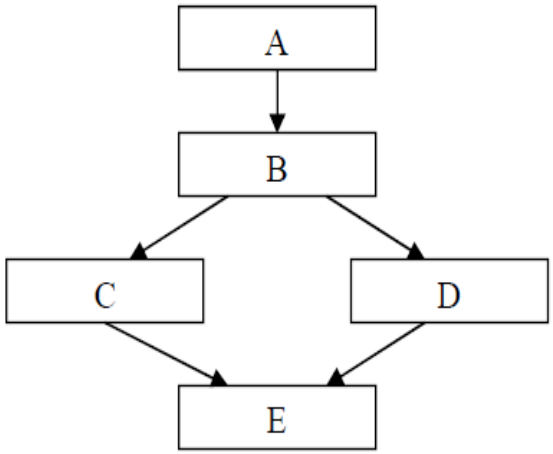
- Le choix des arguments à transmettre au constructeur de '**A**' se fait, non plus dans '**B**' ou '**C**', mais dans '**D**'.
- Uniquement dans ce cas, C++ autorise à spécifier dans le constructeur de '**D**', des informations destinées à '**A**'.



Bien entendu, il sera inutile de préciser des informations pour '**A**' au niveau des constructeurs '**B**' et '**C**'.

Ordre d'appels des constructeurs

En ce qui concerne l'**ordre** des appels, le constructeur d'une classe virtuelle **est toujours appelé avant les autres**. Dans notre cas, on a l'ordre **'A', 'B', 'C' et 'D'**.



L'ordre des appels des constructeurs est : **B, A, C, D et E**.