



P00 – Langage C++

L'héritage

(parties 1 et 2/4)

1^{ère} année ingénieur informatique

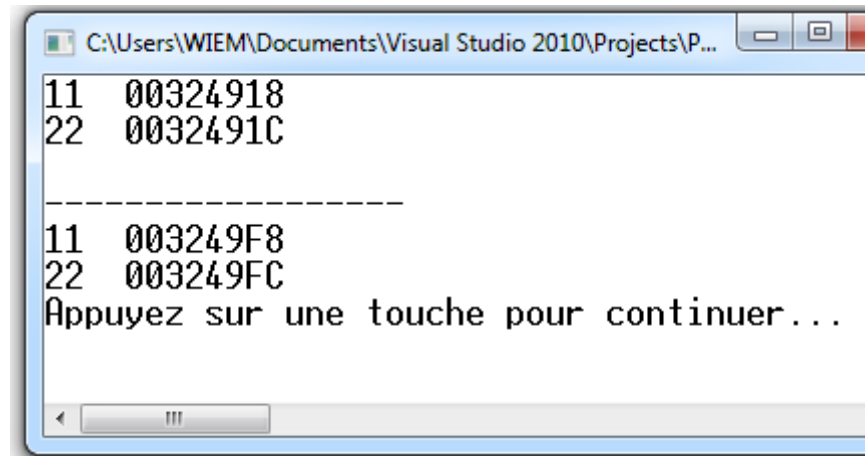
Mme Wiem Yaiche Elleuch

2019 - 2020

Remarque: vector<type de base>

```
void main()
{
    vector<int> a(2);
    // le tableau contient 2 éléments
    // initialisés à 0
    a[0]=11;
    a[1]=22;
    for(int i=0; i<a.size(); i++)
        cout<<a[i]<<" " <<&a[i]<<endl;
    cout<<"\n-----"<<endl;
    vector<int> b(a);
    for(int i=0; i<b.size(); i++)
        cout<<b[i]<<" " <<&b[i]<<endl;

    system("PAUSE");
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\P...
11 00324918
22 0032491C
-----
11 003249F8
22 003249FC
Appuyez sur une touche pour continuer...
```

- L'objet b (instance de vector) est une copie de l'objet a
- Les objets a et b possèdent chacun son propre tableau dynamique → en effet, les adresses des éléments sont différentes

Remarque

- Lorsqu'une classe contient un attribut vector **d'un type de base** (int, float, etc), il n'est pas nécessaire de rajouter un constructeur de recopie.
- Exemple

```
class facture
{
    int cf;
    date dateFacture;
    vector<float> articles;
    float totalFacture;
public:
    //methodes
```



Ne pas rajouter un
constructeur de recopie
➔ Un constructeur de
recopie par défaut sera créé

3 techniques de l'orienté objet

1. Encapsulation

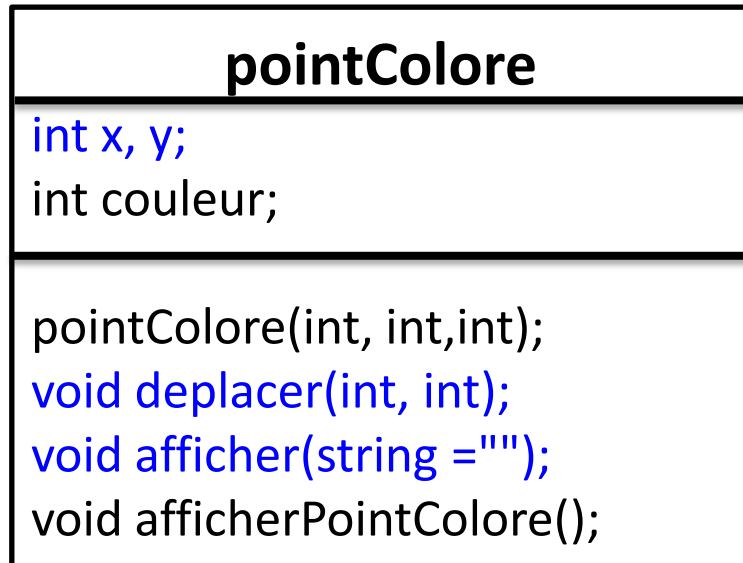
2. Héritage (classes)

3. Polymorphisme (méthodes)

L'héritage

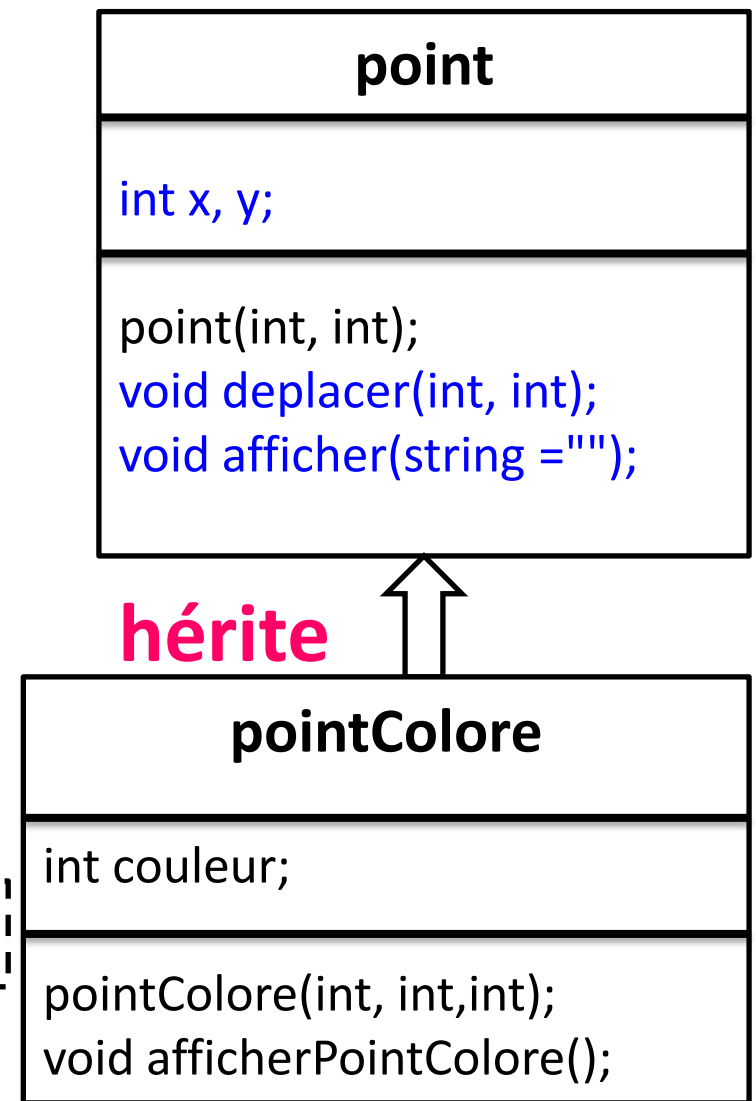
Définir une nouvelle classe à partir d'une **classe existante**, en ajoutant de **nouvelles données** et de **nouvelles méthodes**.

Avantage: réutilisation



Objet de type point

11	11
x	y

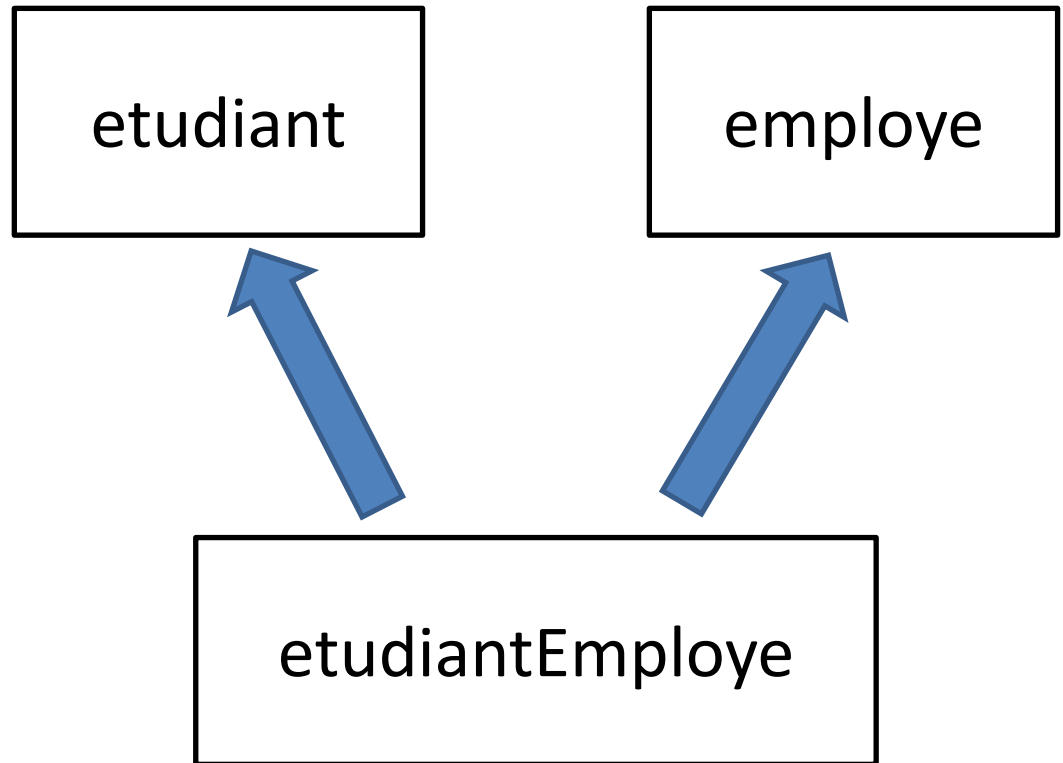


Objet de type pointCouleur

11	11	99
x	y	couleur

L'héritage multiple

La classe
etudiantEmploye hérite
simultanément les
données et les méthodes
des classes etudiant et
employe



jargon

- B est un A
- B is a A

- B est une descendante de A
- A est une ascendante de B

- B spécialise A
- A généralise B

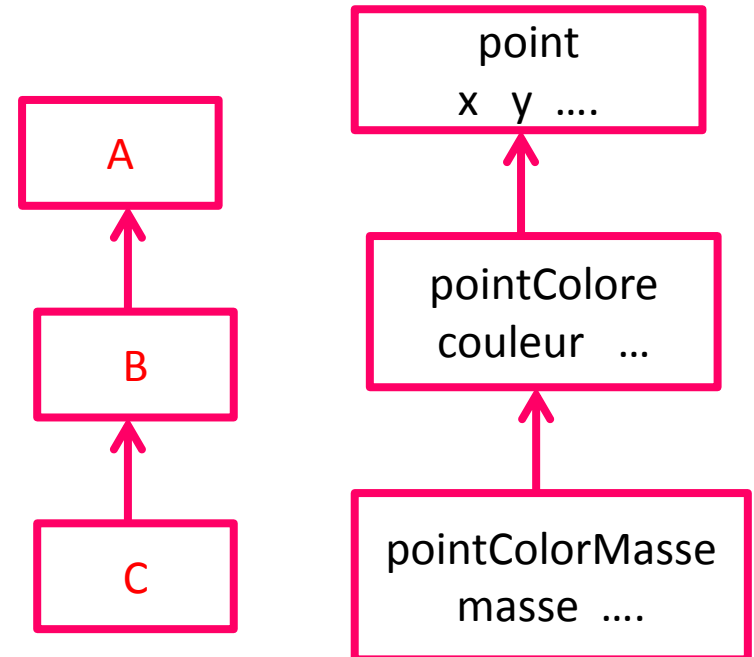
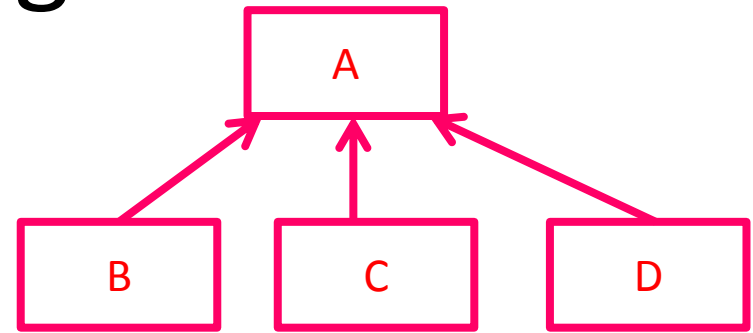
Classe A
Exemple: point
Classe de base
Classe mère
Super classe



Classe B
Exemple: pointCouleur
Classe dérivée
Classe fille
Sous classe

héritage

- **Plusieurs classes** pourront être **dérivées** d'une même classe de base.
 - l'héritage n'est pas limité à un seul **niveau**: une classe dérivée peut devenir à son tour une classe de base pour une autre classe.
- B est une **descendante directe** de A
 - A est une **ascendante directe** de B
 - C est une **descendante** de A
 - A est une **ascendante** de C



Point
int x; int y;
point(int, int); void deplacer(int, int); void afficher(string = "");



pointCouleur
int couleur;
pointCouleur(int, int, int); void afficherPointCouleur();



pointCouleurMasse
int masse;
pointCouleurMasse(int, int, int, int); void afficherPointCouleurMasse();

Héritage à 3 niveaux

Objet de type point

11	11
x	y

Objet de type pointCouleur

11	11	99
x	y	couleur

Objet de type pointCouleurMasse

11	11	99	88
x	y	couleur	masse

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

```
pointColore.h* x pointColore.cpp point.h point.cpp main.cpp
(Global Scope)
#pragma once
#include "point.h"
class pointColore:public point
{
    int couleur;
public:
    pointColore(int =2,int =3,int =4);
    void afficher_pointColore();
    ~pointColore(void);
};
```

class pointcol: **public** point
→ spécifie que *pointcol* est une classe dérivée de la classe de base *point*.

```
pointColore::pointColore(int abs, int ord, int couleur): point(abs,ord)
{
    this->couleur=couleur;
    cout<<"\n +++ appel constr point Colore +++ "<<this<<endl;
}
```

```
pointColore.h* pointColore.cpp* x point.h point.cpp main.cpp
pointColore afficher_pointColore()
#include "pointColore.h"
pointColore::pointColore(int abs, int ord, int coul):point(abs,ord)
{
    // appel du constructeur de la classe point
    cout<<"\n appel du constr de pointColore "<<this<<endl;
    couleur=coul;
}
```

```
pointColore::pointColore(int abs, int ord, int coul):point(abs,ord), couleur(coul)
{
    // appel du constructeur de la classe point
    // initialisation de l'attribut couleur dans l'entête du constructeur
    cout<<"\n appel du constr de pointColore "<<this<<endl;
}
```

```

void pointColore::afficher_pointColore()
{
    afficher();
    // appel de la méthode afficher de la classe point
    // ATTENTION cout<<x<<" "<<y; ==> ERREUR
    // car x et y sont privés dans la classe point
    cout<<"\n la couleur est "<<couleur<<endl;
}

```

```

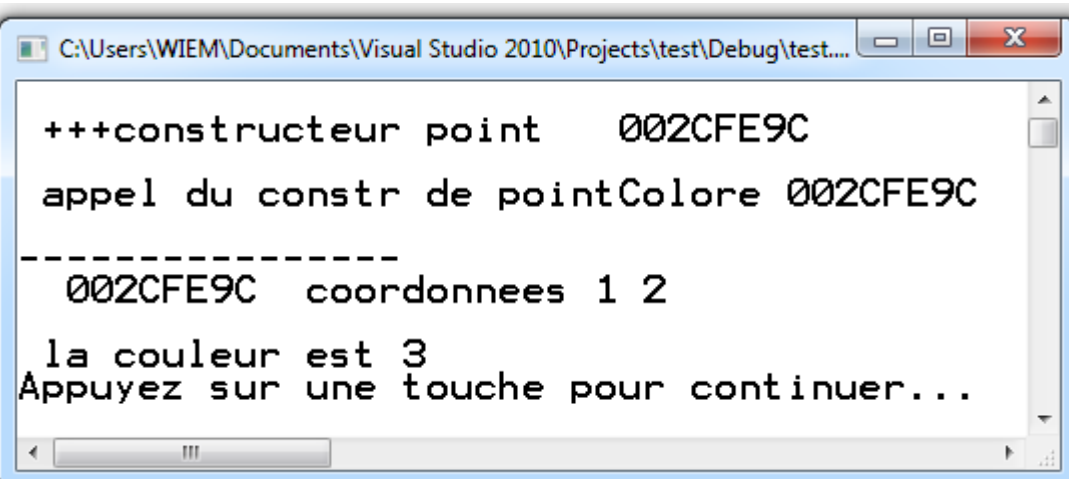
pointColore::~~pointColore(void)
{
    cout<<"\n appel du destr de pointColore "<<this<<endl;
}

```

```

#include"pointColore.h"
void main()
{
    pointColore pc(1,2,3);
    cout<<"\n-----"<<endl;
    pc.afficher_pointColore();
    system("PAUSE");
}

```



```

C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test....
+++constructeur point 002CFE9C
appel du constr de pointColore 002CFE9C
-----
002CFE9C coordonnees 1 2
la couleur est 3
Appuyez sur une touche pour continuer...

```

On peut surcharger/surdéfinir le constructeur

intColoreMasse.h pointColoreMasse.cpp pointColore.h × pointColore.cpp point.h

(Global Scope)

```
#pragma once
#include "point.h"
class pointColore : public point
{
protected:
    int couleur;
public:
    pointColore(int =88, int =88, int =88); {
    pointColore(point,int=88);
    ~pointColore(void);
    void afficher(string="");
};
```

```
void main()
{
    point pt(55,55);
    pointColore a(pt, 66);
    a.afficher();
    system("PAUSE");
}
```

```
void main()
{
    pointColore a(point(55,55), 66);
    a.afficher();
    system("PAUSE");
}
```

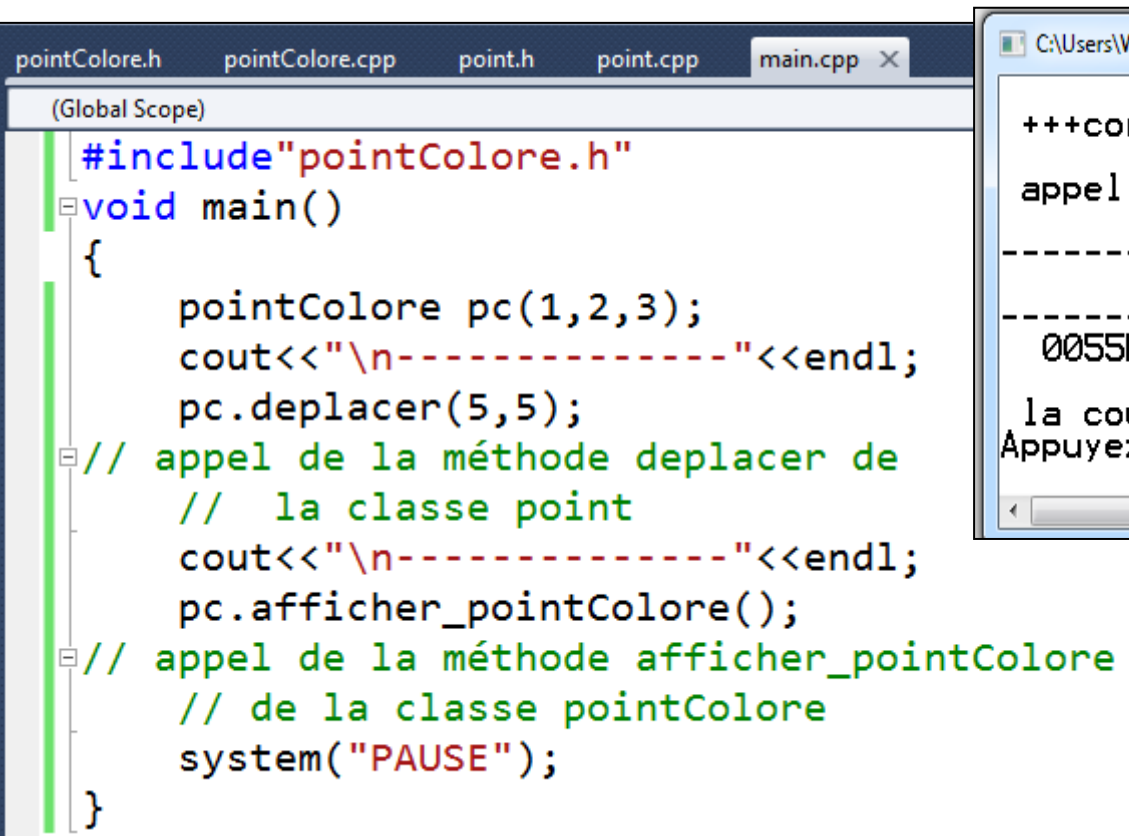
pointColoreMasse.h pointColoreMasse.cpp pointColore.h pointColore.cpp × point.h point.cpp test.cpp

(Global Scope)

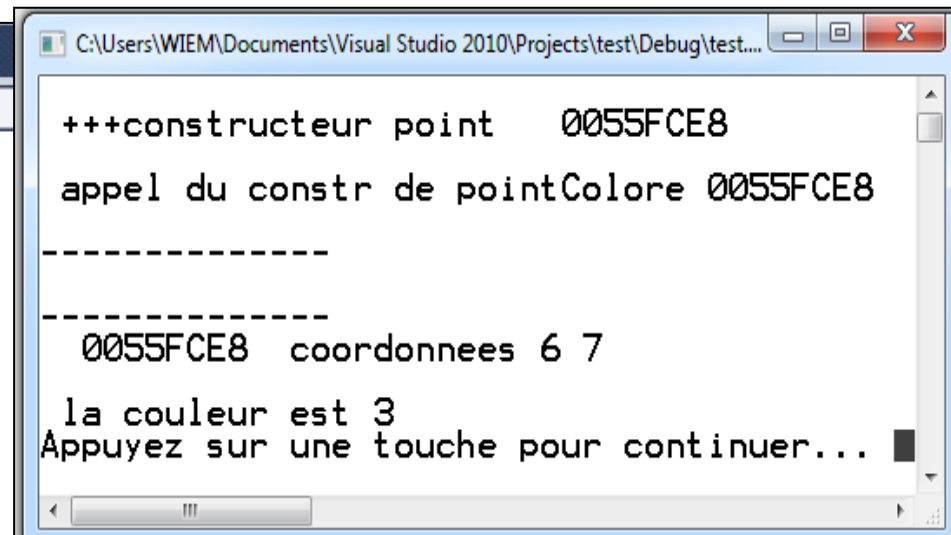
```
#include "pointColore.h"

pointColore::pointColore(point pt, int c): point(pt), couleur(c)
{
    cout<<"\n +++ appel du constr pointColore +++"<<this<<endl;
}
```

Appel d'une méthode de la classe pointColore et d'une méthode de la classe de base point



```
pointColore.h  pointColore.cpp  point.h  point.cpp  main.cpp X
(Global Scope)
#include "pointColore.h"
void main()
{
    pointColore pc(1,2,3);
    cout<<"\n-----"<<endl;
    pc.deplacer(5,5);
    // appel de la méthode deplacer de
    // la classe point
    cout<<"\n-----"<<endl;
    pc.afficher_pointColore();
    // appel de la méthode afficher_pointColore
    // de la classe pointColore
    system("PAUSE");
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test....
+++constructeur point 0055FCE8
appel du constr de pointColore 0055FCE8
-----
0055FCE8 coordonnees 6 7
la couleur est 3
Appuyez sur une touche pour continuer...
```

Un objet de type pointColore ne peut pas appeler les membres privés (données ou méthodes) de la classe point.

pointColoreMasse.h* x pointColoreMasse.cpp pointColore.h pointColore.cpp point.h point.cpp test

pointColoreMasse

```
#pragma once
#include "pointcolore.h"
class pointColoreMasse : public pointColore
{
protected:
    int masse;
public:
    pointColoreMasse(int=77, int=77, int=77, int=77);
    pointColoreMasse(pointColore, int=77);
    pointColoreMasse(point, int=77, int=77);
    ~pointColoreMasse(void);
    void afficher(string="");
};
```

pointColoreMasse.h* pointColoreMasse.cpp x pointColore.h pointColore.cpp point.h point.cpp test.cpp

pointColoreMasse

afficher(string msg)

```
#include "pointColoreMasse.h"
pointColoreMasse::pointColoreMasse(point pt, int c, int m): pointColore(pt,c), masse(m)
{
    cout<<"\n +++ appel du constr pointColoreMasse +++"<<this<<endl;
}

pointColoreMasse::pointColoreMasse( pointColore pt, int m): pointColore(pt), masse(m)
{
    cout<<"\n +++ appel du constr pointColoreMasse +++"<<this<<endl;
}

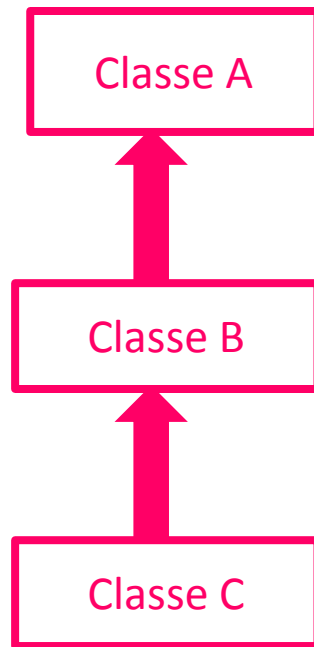
pointColoreMasse::pointColoreMasse(int x,int y, int c, int m):pointColore(x,y,c), masse(m)
{
    cout<<"\n +++ appel du constr pointColoreMasse +++"<<this<<endl;
}
```

```
pointColoreMasse::~~pointColoreMasse(void)
{
    cout<<"\n --- appel du destr pointColoreMasse ---"<<this<<endl;
}

void pointColoreMasse::afficher(string msg)
{
    cout<<msg<<endl;
    pointColore::afficher();
    cout<<"\n masse " <<masse<<endl;
}

void main()
{
    pointColoreMasse a(11,22,33,44);
    // appel du 1er constructeur
    a.afficher();
    cout<<"\n-----"<<endl;
    pointColoreMasse b( pointColore(11,22,33), 44);
    // appel du 2ème constructeur
    b.afficher();
    cout<<"\n-----"<<endl;
    pointColoreMasse c( point (11,22), 33 , 44);
    // appel du 3ème constructeur
    c.afficher();
    system("PAUSE");
}
```


Dans le cas de l'héritage



```
// A.h

#pragma once
#include <iostream>
using namespace std ;
#include <vector>
class A
{
    .....
};
```

```
// B.h
#include "A.h"
class B: public A
{
    .....
};
```

```
// C.h
#include "B.h"
class C: public B
{
    .....
};
```

Inclure tous les fichiers, les classes, etc dans la classe de base

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

Attributs protected dans la classe de base

```
class point
{
protected:
    int x;
    int y;
public:
    point(int =99,int =88);
    void deplacer(int,int);
    void afficher(string = "");
    bool coincide (point);
    point symetrique();
    void setX(int abs){x=abs;}
    void setY(int ord){y=ord;}
    int getX(){return x;}
    int getY(){return y;}
    ~point();
    void saisir_point();
};
```

```
void pointColore::afficher_pointColore()
{
    // afficher();
    cout<<x<<" "<<y<<" ";
    // x et y sont protected dans la classe point
    cout<<"\n la couleur est "<<couleur<<endl;
}
```

Les méthodes de la classe dérivée
ont accès aux membres
protégés de la classe de base

```
void main()
{
    point a(2,3);
    cout<<"\n-----"<<endl;
    cout<<a.x<<" "<<a.y<<endl;
    // les membres protégés d'une classe
    // restent inaccessibles à partir de main
    system("PAUSE");
}
```

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

REdéfinition

- Une fonction de la classe de base est redéfinie si elle n'est plus adaptée pour la classe dérivée ➔ **redéfinir pour enrichir(compléter) ou modifier ou annuler.**

Exemples:

- La fonction afficher de la classe point n'est pas satisfaisante si elle est appelée par un objet pointCouleur (n'affiche pas la couleur) => redéfinir afficher dans la classe pointCouleur
- La fonction déplacer de la classe point est satisfaisante si elle est appelée par un objet de type pointCouleur => ne pas redéfinir déplacer dans la classe pointCouleur.

Remarque: : ne pas confondre REdéfinition et SURdéfinition.

- Dans les deux cas, il s'agit d'avoir plusieurs fonctions avec **le même nom**.

```
class point
{
protected:
    int x;
    int y;
public:
    // Surdéfinition du constructeur
    point();
    point(int);
    point(int,int);
    // Surdéfinition de la méthode afficher
    void afficher();
    void afficher(string);
};
```

**Surdéfinition
(surcharge)**

```
class point
{
protected:
    int x;
    int y;
public:
    point(int =99,int =88);
    void deplacer(int,int);
    void afficher(string = "");
    ~point();
};
```

Redéfinition

```
class pointCouleur:public point
{
protected:
    int couleur;
public:
    pointCouleur(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointCouleur(void);
};
```

Surdéfinition vs Redéfinition

- **La signature d'une méthode:** Nom de la méthode + Arguments de la méthode (Le type de retour n'intervient pas)

```
void afficher(string = "");
```

```
void déplacer(int,int);
```

- **Surdéfinition/surcharge:** changement de la signature de la méthode
- **Redéfinition:** la méthode redéfinie dans la classe dérivée garde la même signature

Redéfinition de la méthode afficher dans la classe pointCouleur

```
void point::afficher(string msg)
{
    cout<<msg<<"  ";
    cout<<this<<"  coordonnees "<<x<<" "<<y<<endl;
}
```

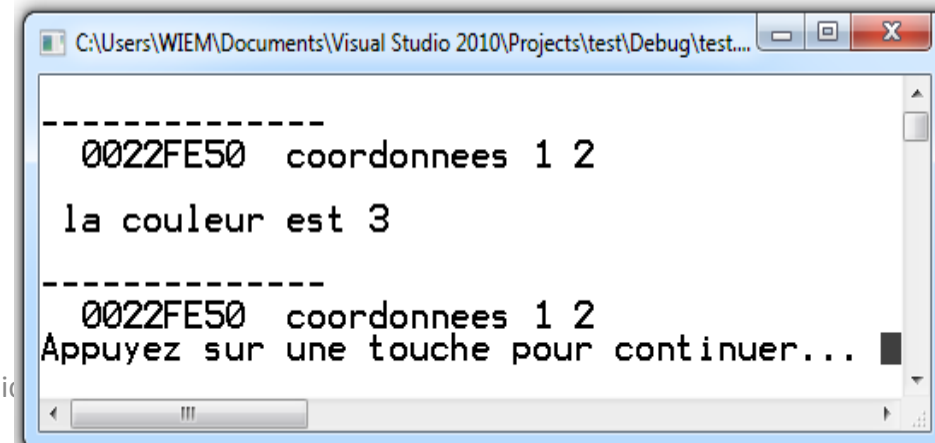
```
void pointCouleur::afficher(string msg)
{
    cout<<msg;
    // x et y sont protected dans la classe point
    cout<<x<<" "<<y<<endl;
    cout<<"\n la couleur est "<<couleur<<endl;
}
```

```
void pointCouleur::afficher(string msg)
{
    cout<<msg;
    point::afficher();
    // appel de la méthode afficher de la classe point
    cout<<"\n la couleur est "<<couleur<<endl;
}
```

Une fonction membre redéfinie peut appeler la même fonction de la classe de base

point::afficher();

```
void main()
{
    pointCouleur a(1,2,3);
    cout<<"\n-----"<<endl;
    a.afficher();
    // ==> appel de afficher de pointCouleur
    cout<<"\n-----"<<endl;
    a.point::afficher();
    // ==> appel de afficher de point
    system("PAUSE");
}
```



```
-----
0022FE50 coordonnees 1 2
la couleur est 3
-----
0022FE50 coordonnees 1 2
Appuyez sur une touche pour continuer...
```


Redéfinition de la méthode afficher dans la classe pointColoreMasse

```
void pointColoreMasse::afficher(string msg)
{
    cout<<msg<<endl;
    //cout<<x<<" "<<y<<" "<<couleur<<endl;
    pointColore::afficher();
    cout<<"\n la masse est "<<masse<<endl;
}
```

Redéfinition de la méthode afficher dans la classe pointColoreMasse pour l'annuler

```
void pointColoreMasse::afficher(string msg)
{
    // afficher est redéfinie pour être annulée
    // corps vide
}
```

Redéfinition des membres **données** d'une classe dérivée

le membre x défini dans
pointColore **s'ajoute** au
membre x hérité de point ;
il ne le remplace pas

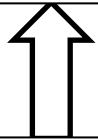
```
classe point  
{  
  int x;  
  ....  
};
```

```
classe pointColore: public point  
{  
  float x;  
  ....  
};
```

```
pointColore a;  
a.x; //x de pointColore  
a.point::x; //x de point
```

Résumé: Redéfinition de la méthode afficher

```
class point
{
protected:
    int x;
    int y;
public:
    point(int =99,int =88);
    void deplacer(int,int);
    void afficher(string = "");
    ~point();
};
```



hérite

```
class pointCouleur:public point
{
protected:
    int couleur;
public:
    pointCouleur(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointCouleur(void);
};
```



hérite

```
class pointCouleurMasse:public pointCouleur
{
    int masse;
public:
    pointCouleurMasse(int =2,int =3,int =4,int =5);
    void afficher(string = "");
    ~pointCouleurMasse(void);
};
```

Objet de type point

11	11
x	y

```
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<this<<" coordonnees "<<x<<" "<<y<<endl;
}
```

Objet de type pointCouleur

22	22	22
x	y	couleur

```
void pointCouleur::afficher(string msg)
{
    cout<<msg;
    point::afficher();
    cout<<"\n la couleur est "<<couleur<<endl;
}
```

Objet de type pointCouleurMasse

33	33	33	33
x	y	couleur	masse

```
void pointCouleurMasse::afficher(string msg)
{
    cout<<msg<<endl;
    pointCouleur::afficher();
    cout<<"\n la masse est "<<masse<<endl;
}
```

Remarque

- Soit la classe date (jour, mois annee, etc)
- Soit la classe facture (code, dateFacture, etc)
- ➔ c'est une erreur de déclarer facture une sous classe de date (en effet, une facture n'est pas une date).

```
class facture: public date| ERREUR
```

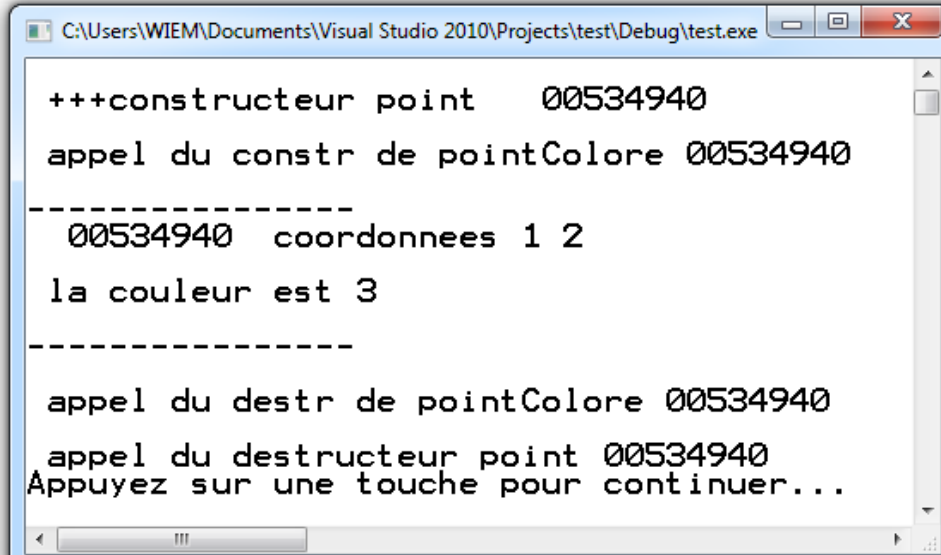
plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

Ordre d'appel des constructeurs et des destructeurs

- Constructeurs: point, pointCouleur
- Destructeurs: pointCouleur, point

```
void main()
{
    pointCouleur *q=new pointCouleur(1,2,3);
    cout<<"\n-----"<<endl;
    q->afficher_pointCouleur();
    cout<<"\n-----"<<endl;
    delete q;
    system("PAUSE");
}
```



```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test.exe

+++constructeur point    00534940
appel du constr de pointCouleur 00534940

-----
00534940  coordonnees 1 2
la couleur est 3
-----

appel du destr de pointCouleur 00534940
appel du destructeur point 00534940
Appuyez sur une touche pour continuer...
```

Exemple 2:

// création d'un objet pointCouleurMasse

Ordre d'appel des constructeurs:

point
pointCouleur
pointCouleurMasse

// destruction d'un objet pointCouleurMasse

Ordre d'appel des destructeurs:

pointCouleurMasse
pointCouleur
point

La hiérarchisation des appels

- Ces règles se généralisent au cas des classes dérivées, en tenant compte de l'aspect hiérarchique qu'elles introduisent.

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

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

```
// création et suppression
d'un objet de type B
A
B
~B
~A
```

- Pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au constructeur de A, puis le compléter par ce qui est spécifique à B et faire appel au constructeur de B.
- La même démarche s'applique aux destructeurs: lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A (les destructeurs sont appelés dans **l'ordre inverse** de l'appel des constructeurs).

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

Problème: Tester l'accès à un membre (attribut ou méthode) à partir de 3 endroits différents

```
class point
{
private:
    int x;
    void fctX();
protected:
    int y;
    void fctY();
public:
    int z;
    void fctZ();
    point(int =99,int =88);
    void afficher(string = "");
};
```

L'accès ou le non accès à un membre dépend de son statut (public, protected, private).

1. À partir d'une méthode de la même classe

```
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<x<<endl;
    fctX();
}
```

2. À partir d'une méthode de la sous classe

```
void pointColore::afficher(string msg)
{
    cout<<x<<endl;
    fctX();
    //...
}
```

3. À partir d'entités externes (exemple: main)

```
void main()
{
    point a;
    cout<<a.x<<endl;
    a.fctX();
    system("PAUSE");
}
```

Résumé: accès aux membres d'une classe

	Accès par une méthode de la même classe	Accès par une méthode de la classe dérivée	Accès par une entité externe (exemple: main)
Membre privé	?	?	?
Membre protégé	?	?	?
Membre public	?	?	?

Cas 1: membre (attribut ou méthode) privé

```
class point
{
private:
    int x;
    void fctX();
public:
    point(int =99,int =88);
    void afficher(string = "");
};
```

```
class pointColore:public point
{
```

```
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<x<<endl;
    fctX();
}
```

OK

```
void pointColore::afficher(string msg)
{
    cout<<x<<endl;
    fctX();
    //...
}
```

ERREUR

```
void main()
{
    point a;
    cout<<a.x<<endl;
    a.fctX();
    system("PAUSE");
}
```

ERREUR

membre privé (attribut x, méthode fctX):

- ➔ Accessible uniquement par les méthodes de la classe
- ➔ Inaccessible par les méthodes des sous classes
- ➔ Inaccessible par main

Résumé: accès aux membres d'une classe

	Accès par une méthode de la même classe	Accès par une méthode de la classe dérivée	Accès par une entité externe (exemple: main)
Membre privé	OUI	NON	NON
Membre protégé	?	?	?
Membre public	?	?	?

Cas 2: membre (attribut ou méthode) protégé

```
class point
{
protected:
    int x;
    void fctX();
public:
    point(int =99,int =88);
    void afficher(string = "");
};
```

```
class pointColore:public point
{
```

```
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<x<<endl;
    fctX();
}
```

OK

```
void pointColore::afficher(string msg)
{
    cout<<x<<endl;
    fctX();
    //...
}
```

OK

```
void main()
{
    point a;
    cout<<a.x<<endl;
    a.fctX();
    system("PAUSE");
}
```

ERREUR

Membre protégé (attribut x, méthode fctX):

- ➔ Accessible par les méthodes de la classe
- ➔ Accessible par les méthodes des sous classes
- ➔ Inaccessible par main

Résumé: accès aux membres d'une classe

	Accès par une méthode de la même classe	Accès par une méthode de la classe dérivée	Accès par une entité externe (exemple: main)
Membre privé	OUI	NON	NON
Membre protégé	OUI	OUI	NON
Membre public	?	?	?

Cas 3: membre (attribut ou méthode) public

```
class point
{
public:
    int x;
    void fctX();
public:
    point(int =99,int =88);
    void afficher(string = "");
};
```

```
class pointColore:public point
{
```

```
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<x<<endl;
    fctX();
}
```

OK

```
void pointColore::afficher(string msg)
{
    cout<<x<<endl;
    fctX();
    //...
}
```

OK

```
void main()
{
    point a;
    cout<<a.x<<endl;
    a.fctX();
    system("PAUSE");
}
```

OK

Membre public (attribut x, méthode fctX):

- ➔ Accessible par les méthodes de la classe
- ➔ Accessible par les méthodes des sous classes
- ➔ Accessible par main

Résumé: accès aux membres d'une classe

	Accès par une méthode de la même classe	Accès par une méthode de la classe dérivée	Accès par une entité externe (exemple: main)
Membre privé	OUI	NON	NON
Membre protégé	OUI	OUI	NON
Membre public	OUI	OUI	OUI

Modes de dérivation

```
class point
{
private:
    int x;
    void fctX();
public:
    point(int =99,int =88);
    void afficher(string = "");
};
```

```
class pointColore:public point
{
protected:
    int couleur;
public:
    pointColore(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointColore(void);
};
```

Dérivation
publique

```
class pointColore:protected point
{
protected:
    int couleur;
public:
    pointColore(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointColore(void);
};
```

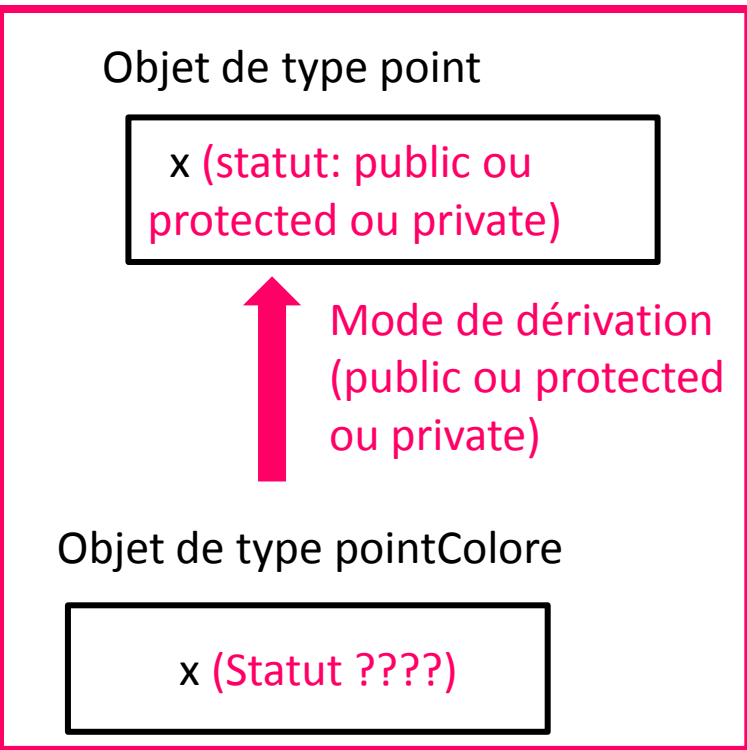
Dérivation
protégée

```
class pointColore:private point
{
protected:
    int couleur;
public:
    pointColore(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointColore(void);
};
```

Dérivation
privée

Selon le mode de dérivation,
le membre peut changer de
statut dans la classe dérivée

Problème



Exemple :

Objet de type point

x (public)

Mode de dérivation: private

Objet de type pointColore

x (private)

```
void main()
{
    point a;
    pointColore b;

    cout<<a.x<<endl; // OK
    cout<<b.x<<endl; // ERREUR
    system("PAUSE");
}
```

Cas 1: dérivation publique

Le statut du membre ne change pas dans la classe dérivée

Objet de type point

x (public)



Objet de type pointColore

x (public)

Cas 2: dérivation protégée

public → protected
protected → protected
private → private

Objet de type point

x (public)



Objet de type pointColore

x (protected)

Cas 3: dérivation privée

public → private
protected → private
private → private

Objet de type point

x (public ou protected)



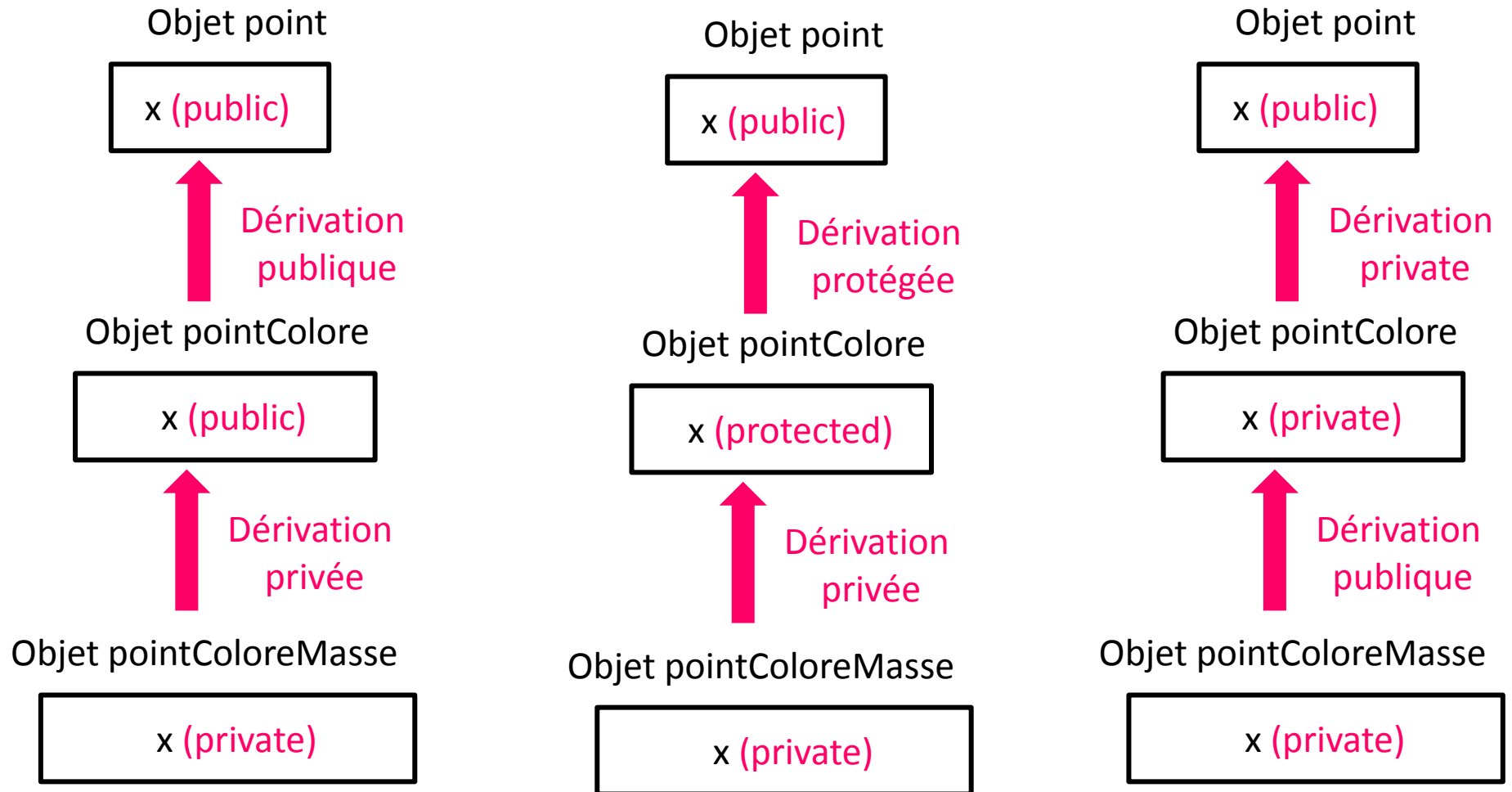
Objet de type pointColore

x (private)

Résumé: nouveaux statuts des membres de la classe de base dans la classe dérivée selon le type de dérivation

		Statut des membres de la classe de base		
		public	protected	private
Mode de dérivation	Public	public	protected	Private
	Protected	protected	protected	private
	Private	private	private	private

Exemples



plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

Conversion de types

```
void main()
{
    point a(11,11);
    pointColore b(22,22,22);
    pointColoreMasse c(33,33,33,33);
    // conversion d'un type dérivé en un type de base
    a=b; //OK
    b=c; //OK

    // conversion d'un type de base en un type dérivé
    b=a; //ERREUR
    c=b; //ERREUR
    system("PAUSE");
}
```

- `a=b;` **est légale**. Elle entraîne une **conversion** de `b` dans le type `point` et l'affectation du résultat à `a`. ➔ Cette conversion revient à ne conserver de `b` que ce qui est du type `point`, elle n'entraîne pas la création d'un nouvel objet.
- `b=a;` **//rejetée**

Conversion de pointeurs

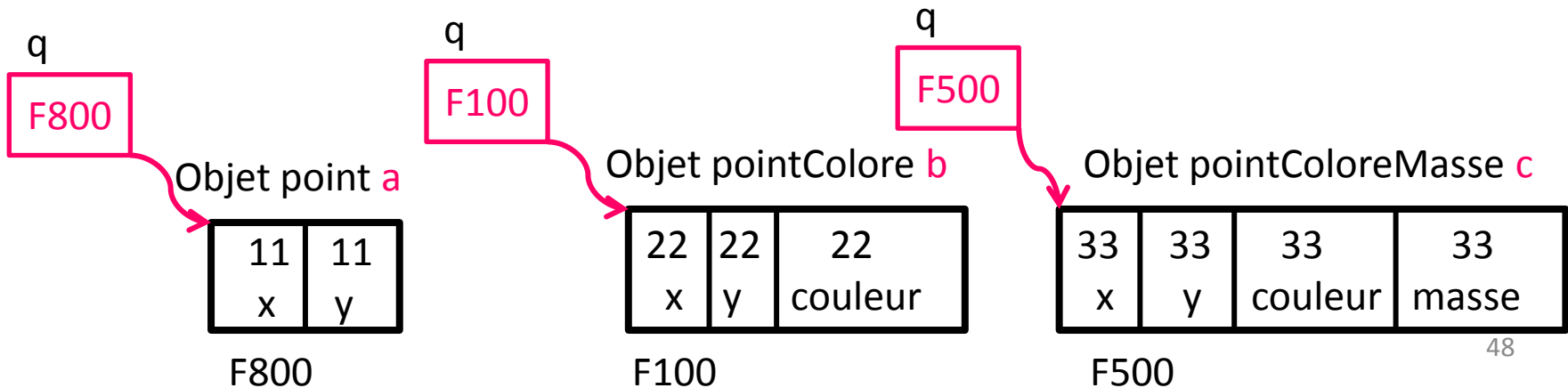
```
void main()
{
    point a(11,11);
    pointColore b(22,22,22);
    pointColoreMasse c(33,33,33,33);

    point *q;
    q=&a;
    q=&b;
    q=&c;

    system("PAUSE");
}
```

Le pointeur q (de type **point***),
peut pointer sur un point ou
sur un pointColore ou sur un
pointColoreMasse

q peut pointer sur un objet de la
classe point et sur tous les
objets descendants



Conversion de pointeurs

```
void main()
{
    point a(11,11);
    pointCouleur b(22,22,22);
    pointCouleurMasse c(33,33,33,33);

    pointCouleurMasse *q;
    q=&a; // ERREUR
    q=&b; // ERREUR
    q=&c; // OK

    system("PAUSE");
}
```

Le pointeur q de type
(**pointCouleurMasse***) ne
peut pas pointer sur les
objets ascendants.

affectation de pointeurs

```
void main()
{
    point a(11,11);
    pointColore b(22,22,22);
    pointColoreMasse c(33,33,33,33);

    point *pa=&a;
    pointColore *pb=&b;

    pa=pb; // conversion de pointColore* en point*

    pb=pa; // ERREUR

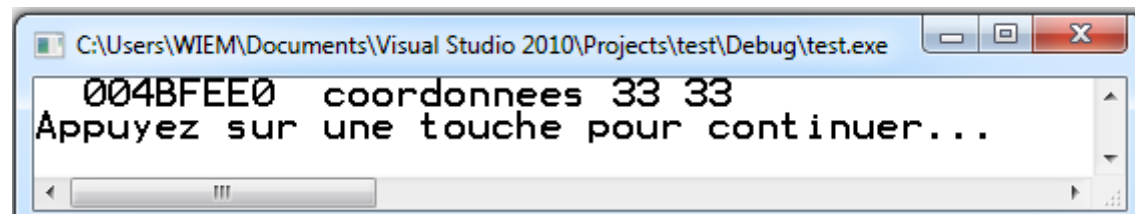
    system("PAUSE");
}
```

Problème: Typage statique des objets

```
void main()
{
    point a(11,11);
    pointColore b(22,22,22);
    pointColoreMasse c(33,33,33,33);

    point *q=&c;
    // q de type point*
    // l'objet pointé de type pointColoreMasse
    q->afficher();
    // appel de afficher de la classe point
    // ==> c'est le type du pointeur q qui est pris en compte
    // et non pas le type de l'objet pointé
    system("PAUSE");
}
```

Appel de afficher de la
classe point →
Affichage de x et y
seulement



**la méthode appelée dépend du type du pointeur q
et non pas du type de l'objet pointé.**

Solution: Typage dynamique

- C++ permet d'effectuer l'identification d'un objet au **moment de l'exécution** (et non pas à la compilation)
- Cela nécessitera l'emploi de **fonctions virtuelles**.
- Lorsqu'une fonction est redéfinie dans une classe dérivée, **elle doit être virtuelle dans la classe de base**.
- Lorsqu'une classe comporte une fonction virtuelle, elle doit rendre son **destructeur virtuel**.
- une **fonction virtuelle** est une fonction définie dans une classe et qui est destinée à être redéfinie dans les classes dérivées.

Déclarer la méthode afficher virtuelle

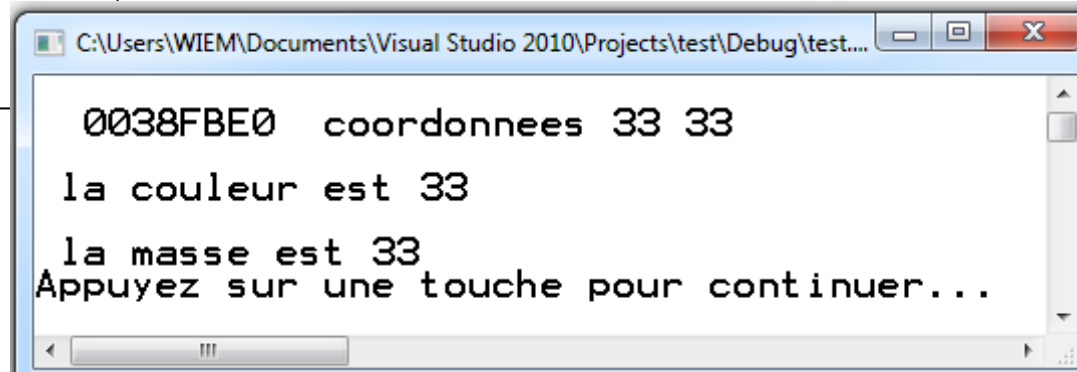
```
class point
{
protected:
    int x;
    int y;
public:
    point(int =99,int =88);
    void deplacer(int,int);
    virtual void afficher(string = "");
    virtual ~point();
};
```

```
void main()
{
    pointColoreMasse c(33,33,33,33);
    point *q=&c;

    q->afficher();

    system("PAUSE");
}
```

Indiquer que la méthode afficher est virtuelle (virtual), uniquement dans la classe point (classe de base).



```
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test....
0038FBE0 coordonnees 33 33
la couleur est 33
la masse est 33
Appuyez sur une touche pour continuer...
```

La méthode appelée dépend du type de l'objet pointé et non plus du type du pointeur.


Les méthodes virtuelles

- Lorsqu'une méthode virtuelle est appelée, la méthode à exécuter est choisie en fonction du **type** de l'objet.
- L'appel n'est donc résolu qu'à l'exécution, le type de l'objet ne peut pas être connu à la compilation.
- Le mot clé virtual, placé devant le prototype de la fonction, indique au compilateur que la fonction est **redéfinie** dans une classe dérivée.


Résumé: Redéfinition de méthode

- La méthode doit avoir la même signature dans les classes dérivées.
- les diverses redéfinitions d'une méthode peuvent être vues comme des versions de plus en plus améliorées, adaptées, etc.
- Lorsqu'une fonction est redéfinie dans une classe dérivée, **elle doit être virtuelle dans la classe de base.**
- Lorsqu'une classe comporte une fonction virtuelle, elle doit rendre son **destructeur virtuel.**

```
class point
{
protected:
    int x;
    int y;
public:
    point(int =99,int =88);
    void déplacer(int,int);
    virtual void afficher(string = "");
    virtual ~point();
};
```



```
class pointColore:public point
{
protected:
    int couleur;
public:
    pointColore(int =2,int =3,int =4);
    void afficher(string = "");
    ~pointColore(void);
};
```



```
class pointColoreMasse:public pointColore
{
    int masse;
public:
    pointColoreMasse(int =2,int =3,int =4,int =5);
    void afficher(string = "");
    ~pointColoreMasse(void);
};
```

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

3 techniques de l'orienté objet

1. Encapsulation

2. Héritage (classes)

3. Polymorphisme (méthodes)

Polymorphisme

- Le nom de polymorphisme vient du grec et signifie qui peut prendre plusieurs formes.
- une **méthode polymorphe** est une méthode qui a plusieurs formes.
- l'héritage concerne les classes; le polymorphisme concerne les méthodes
- Il existe 3 types de polymorphismes
 - **Polymorphisme statique (ad hoc):** **surdéfinition**/surcharge de fonctions
 - **Polymorphisme dynamique (d'héritage):** **redéfinition** de fonctions
 - Il est mis en œuvre à l'aide du mécanisme des méthodes virtuelles.
 - **Polymorphisme paramétrique (➔ chapitre template).**
- Une classe possédant des fonctions virtuelles est dite **classe polymorphe**.

Exemples de polymorphisme

```
class courbe
{
    vector<point*> tab;
public:
    courbe();
    void afficher(string = "");
    courbe(const courbe&);
    ~courbe(void);
    int taille();
    void ajouter (point, int =0);
    void ajouter (pointColore, int =0);
    void ajouter (pointColoreMasse, int =0);
    // surdéfinition de la méthode ajouter
    // ==> ajouter est une méthode polymorphe
    void supprimer(int =0);
};
```

Polymorphisme statique ad hoc:

surdéfinition

→ **ajouter** est une méthode
polymorphe

```
class point
{
protected:
    int x;
    int y;
    static int nb_objets;
public:
    point(int abs=99,int ord=88);
    virtual void afficher(string = "");
    // méthode afficher est une méthode virtuelle
    // Redéfinition de afficher dans les classes dérivées
    // afficher est une méthode polymorphe
    virtual ~point();
};
```

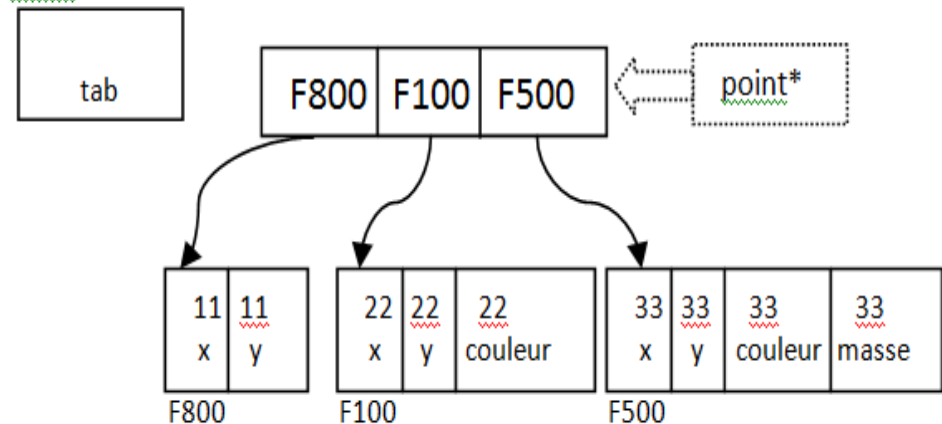
Polymorphisme dynamique d'héritage:
redéfinition

→ **afficher** est une méthode
polymorphe

Exercice d'application (voir corrigé)

```
class courbe
{
    vector<point*> tab;
public:
    courbe();
    void afficher(string = "");
    courbe(const courbe&);
    ~courbe(void);
    int taille();
    void ajouter (point, int =0);
    void ajouter (pointCouleur, int =0);
    void ajouter (pointCouleurMasse, int =0);
    void supprimer(int =0);
    void ajouter(point*, int =0);
};
```

objet courbe



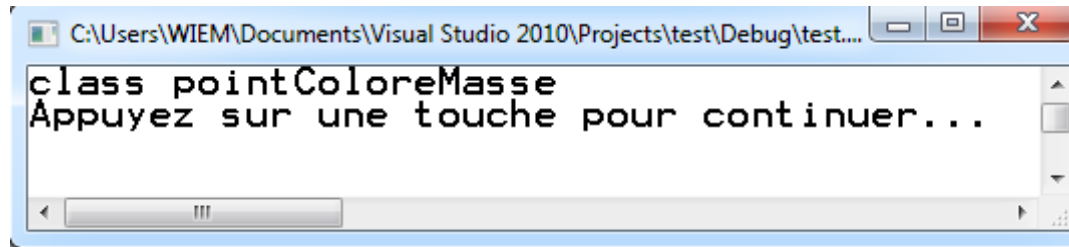
L'opérateur typeid

- il est possible, lors de l'exécution, de connaître le type d'un objet désigné par un pointeur (**identification des types à l'exécution**).
- il existe un **opérateur** à un opérande nommé **typeid** fournissant en résultat un **objet** de type prédéfini **typeinfo**.
- Cette classe contient la fonction membre **name()**, laquelle fournit une chaîne de caractères représentant le nom du type.

Exemples 1/2

Type d'un objet

```
#include<typeinfo>
void main()
{
    pointColoreMasse a;
    cout<<typeid(a).name()<<endl;
    system("PAUSE");
}
```

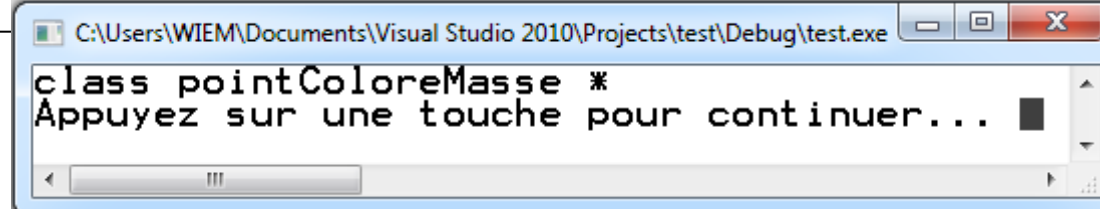


C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test.exe

```
class pointColoreMasse
Appuyez sur une touche pour continuer...
```

Type d'un pointeur

```
#include<typeinfo>
void main()
{
    pointColoreMasse *q;
    cout<<typeid(q).name()<<endl;
    system("PAUSE");
}
```



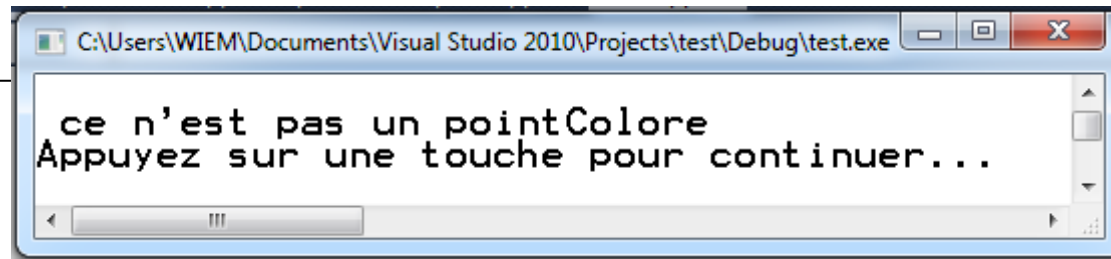
C:\Users\WIEM\Documents\Visual Studio 2010\Projects\test\Debug\test.exe

```
class pointColoreMasse *
Appuyez sur une touche pour continuer...
```

Exemples 2/2

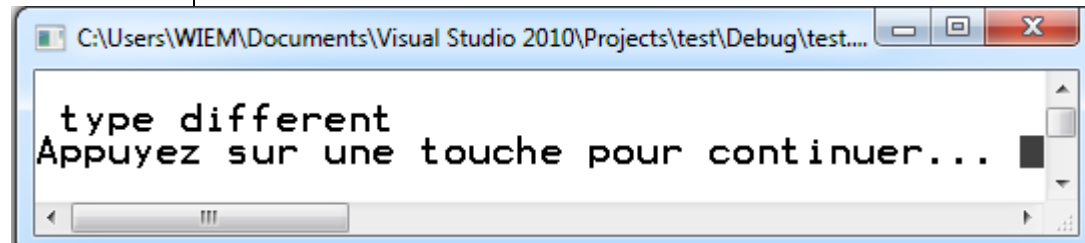
Identifier le type d'un objet

```
#include<typeinfo>
void main()
{
    pointColoreMasse a;
    if(typeid(a)== typeid(pointColore))
        cout<<"\n c'est un pointColore "<<endl;
    else cout<<"\n ce n'est pas un pointColore "<<endl;
    system("PAUSE");
}
```



Comparaison des types de deux objets

```
#include<typeinfo>
void main()
{
    pointColoreMasse a;
    point b;
    if (typeid(a)== typeid(b))
        cout<<"\n meme type "<<endl;
    else cout<<"\n type different"<<endl;
    system("PAUSE");
}
```



Opérateurs de transtypage

- Ce sont des opérateurs de **changement de type** (transtypage) (opérateur de conversion).
- Ils sont utiles dans le contexte du polymorphisme **pour convertir un objet d'une classe de base en une classe dérivée.**
- C++ propose 4 opérateurs de transtypage:
 - `static_cast`
 - `dynamic_cast`
 - `const_cast`
 - `reinterpret_cast`
- `static_cast<nouveauType> (Expression)`: modifie le type d'une expression en se basant sur le nouveau type demandé

L'opérateur: static_cast

`static_cast <nomType>(expr)`

- Cet opérateur est utilisé pour effectuer des conversions qui sont résolues à la compilation.
- Son utilisation :
 - changement de type d'un pointeur d'une classe de base en un pointeur d'une classe dérivée (point* ➔ pointColore*).
 - Changement de type d'un objet d'un type de base en un type dérivé (point ➔ pointColore)

Exemple

```
▣ courbe::courbe(const courbe &w)
{
    point *q;
    for(int i=0; i<w.tab.size(); i++)
    {
        if (typeid(*w.tab[i])==typeid(point))
            q=new point(*w.tab[i]);

        else if(typeid(*w.tab[i])==typeid(pointColore))
            q=new pointColore( static_cast<const pointColore*>(*w.tab[i]));

        else if(typeid(*w.tab[i])==typeid(pointColoreMasse))
            q=new pointColoreMasse( static_cast<const pointColoreMasse*>(*w.tab[i]));

        tab.push_back(q);
    }
}
```

Résumé: Les trois grands principes de la POO

- **Encapsulation** Permet de regrouper les données et les fonctions au sein d'une Classe et indique les droits d'accès (private, public, protected) à ces membres de cette classe.
 - Encapsulation des données: Protéger les données en les déclarant privés.
 - Accéder à ces données privées par l'intermédiaire de méthodes.
- **Héritage:** permet la création d'une classe à partir d'une **classe existante**
 - la classe dérivée contient les attributs et les méthodes de la classe de base en rajoutant d'autres.
 - permet ainsi la réutilisation de code
 - plusieurs types d'héritage : public, protégé ou privé.
- **Polymorphisme (d'héritage):** permettre de **redéfinir** dans une classe dérivée les méthodes dont elle hérite de la classe de base
 - une même méthode possède alors plusieurs formes.
 - Deux objets réagissent différemment au **même** appel de méthode
 - Le polymorphisme permet de modifier le comportement d'une classe

plan

1. La notion d'héritage
2. Utilisation des membres de la classe de base dans une classe dérivée
3. Redéfinition des membres d'une classe dérivée
4. Appel des constructeurs et des destructeurs
5. Contrôle des accès
6. Compatibilité entre classe de base et classe dérivée
7. Le polymorphisme
8. Le constructeur de copie et l'héritage
9. Autre situation de méthode virtuelle
10. Les fonctions virtuelles pures pour la création de classes abstraites

Le constructeur de copie et l'héritage

```
class etudiant
{
protected:
    int nb_notes;
    int *notes;
public:
    ...
    etudiant(const etudiant &);
};
```

```
class etudiant_salarie: public etudiant
{
    int nb_mois;
    int *mois;
public:
    ...
    etudiant_salarie(const etudiant_salarie &);
};
```

```
//etudiant a;
// etudiant b=a;
etudiant::etudiant(const etudiant &w)
{
    cout<<"+++ constructeur de copie etudiant +++"<<endl;
    nb_notes=w.nb_notes;
    notes=new int [nb_notes];
    for(unsigned int i=0; i<nb_notes; i++)
        notes[i]=w.notes[i];
}
```

Le constructeur de recopie et l'héritage

```
etudiant_salarie::etudiant_salarie(const etudiant_salarie &w):etudiant(w)
{
    // la copie de nb_notes et les notes se fait par l'appel: etudiant(w)
    cout<<"\n +++ constructeur de recopie etudiant salarie +++"<<endl;
    nb_mois=w.nb_mois;
    mois= new int[nb_mois];
    for(unsigned int i=0; i<nb_mois; i++)
        mois[i]=w.mois[i];
}
```

D'une manière générale:

```
// B est une sous classe de A
B (const B & w) : A (w)
// w de type B, est converti dans le type A pour être transmis au
constructeur de recopie de A
{
    //recopie de la partie de w spécifique à B (non héritée de A)
}
```