



P00 – Langage C++ L'héritage (parties 3 et 4/4)

1^{ère} année ingénieur informatique

Mme Wiem Yaiche Elleuch

2019 - 2020

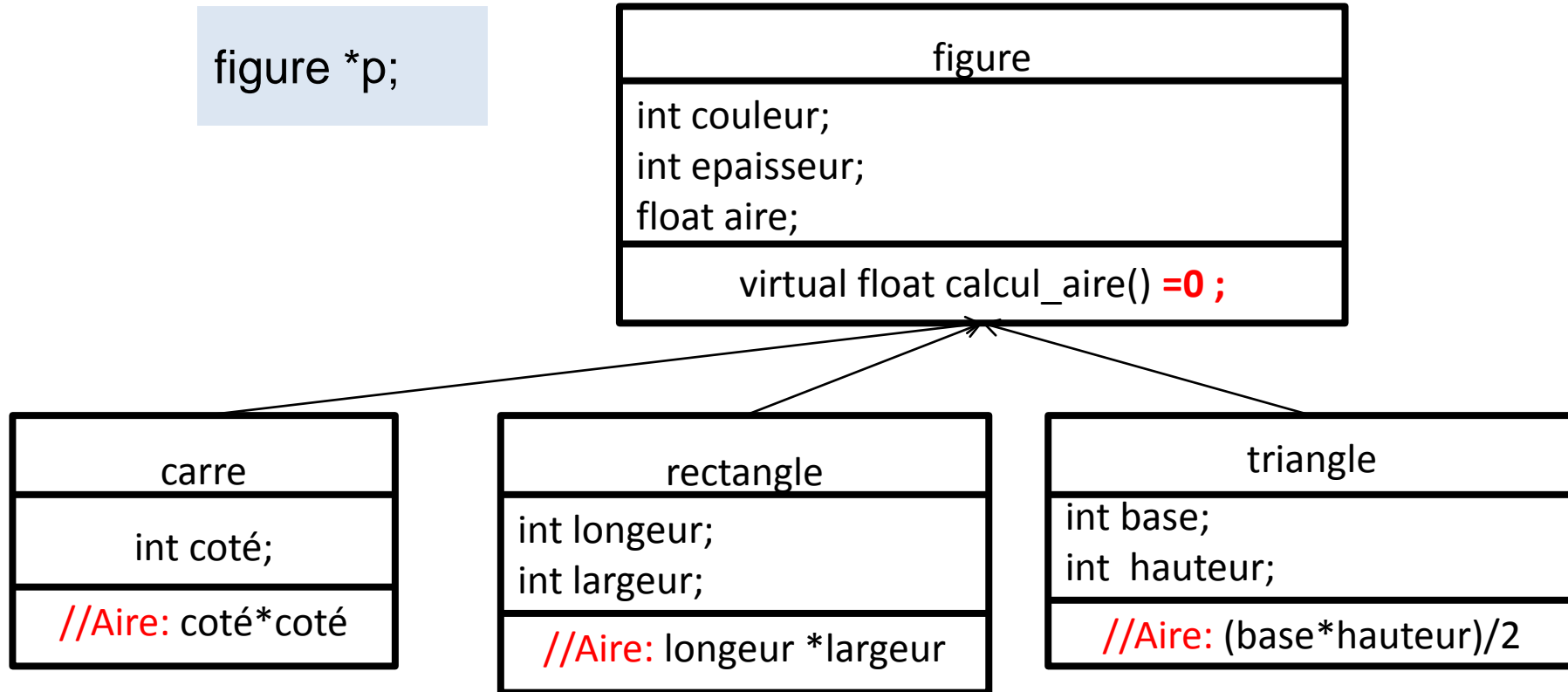
plan

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 constructeur de copie et l'héritage
8. Autre situation de méthode virtuelle
9. Les fonctions virtuelles pures pour la création de classes abstraites

Exemple de classe abstraite

méthode virtuelle **pure**

figure *p;



Méthode virtuelle pure: sa définition est **nulle (0)**

Classe abstraite:

- contient au moins une méthode virtuelle pure
- classe **non instanciable** (impossible de créer des objets de cette classe)

classe abstraite

méthode virtuelle **pure**

- les "**fonctions virtuelles pures**" sont des fonctions virtuelles dont la définition est **nulle** (0)

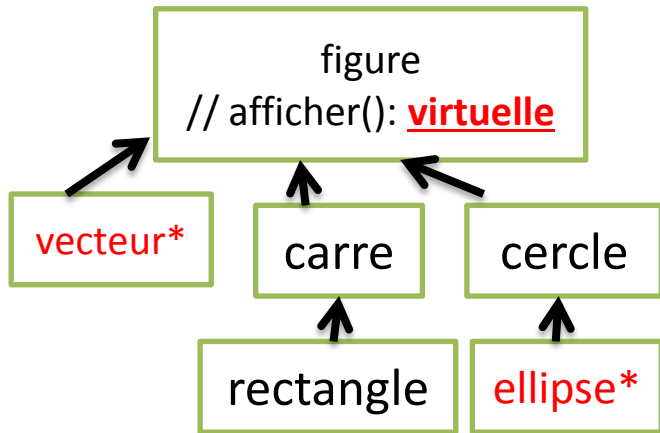
```
virtual void fct (.....) = 0 ;
```
- Généralement, il est difficile de leur donner une définition (implémentation).
- Une classe comportant **au moins** une **fonction virtuelle pure** est considérée comme **abstraite** et il n'est plus possible de **déclarer des objets de son type**.
- **But des classes abstraites**: Les classes abstraites sont destinées non pas à instancier des objets, mais simplement à donner naissance à d'autres classes par héritage.
- Les classes abstraites sont placées à des **niveaux élevés** de la hiérarchie (d'héritage). Elles servent de **base** à d'autres classes dérivées.

classe abstraite

méthode virtuelle **pure**

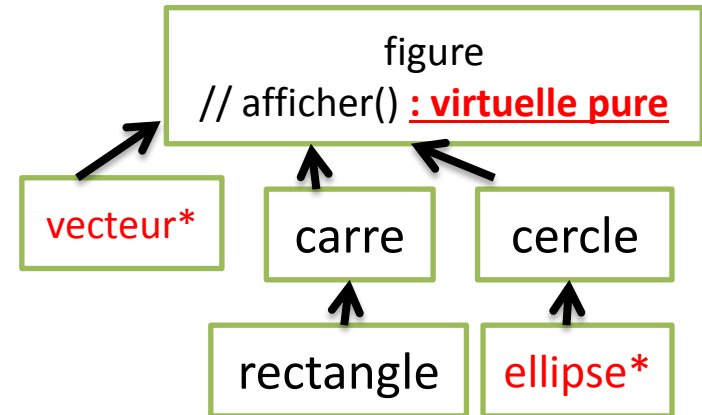
- La classe abstraite **doit** avoir des classes dérivées (qui, elles, auront des objets).
- Il est possible de déclarer un pointeur sur la classe abstraite.
- Une fonction déclarée virtuelle pure dans une classe de base doit **obligatoirement** être redéfinie ultérieurement dans une classe dérivée .
- C'est une **contrainte** pour les classes dérivées, qui seront **obligées** d'implémenter la méthode virtuelle pure.
- Une classe dérivée d'une classe abstraite est également abstraite si elle ne fournit pas de définition aux méthodes virtuelles pures de sa classe de base.
- Exemple1: un véhicule est soit un avion, soit un bateau, soit une voiture: ➔ créer classe véhicule abstraite
- Un être humain est soit un homme ou une femme
- Un mammifère est un chat ou un chien etc
- Un compte est soit un compte courant ou un compte épargne
- Un point peut aussi être un pointCouleur, ou un pointMasse

Exemple: 2 scénarios



Scénario 1: la méthode afficher est déclarée **virtuelle** dans la classe figure. Afficher est redéfinie uniquement dans les classes vecteur et ellipse

➔ On peut créer des objets figure, vecteur, carre, cercle, rectangle et ellipse.



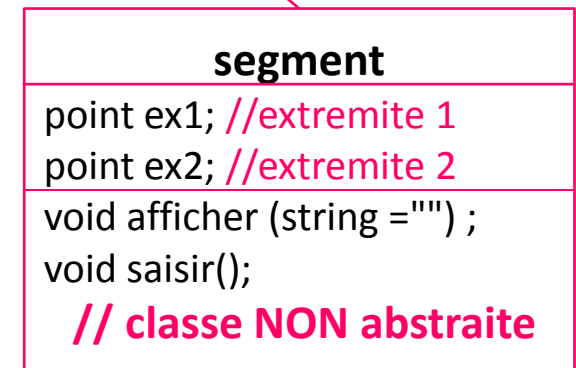
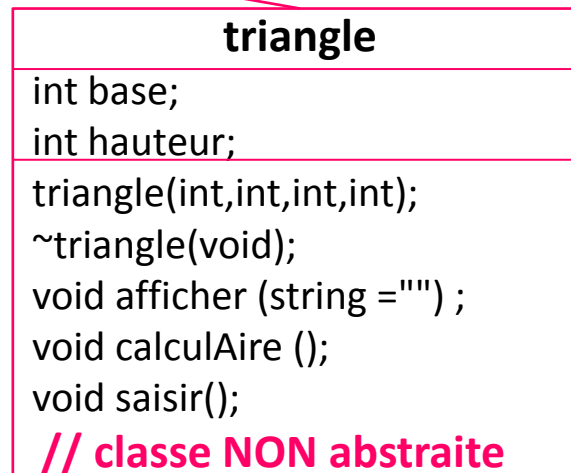
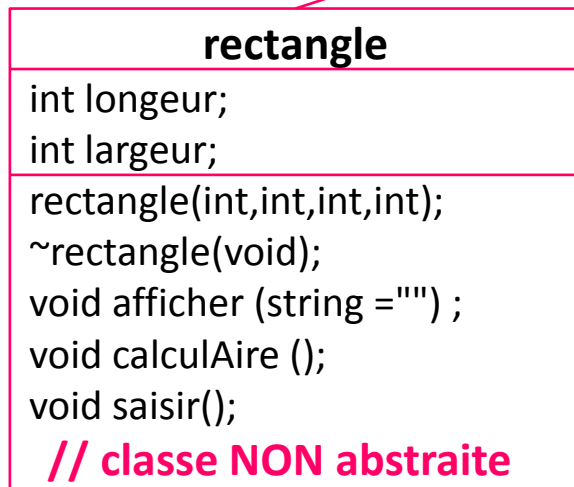
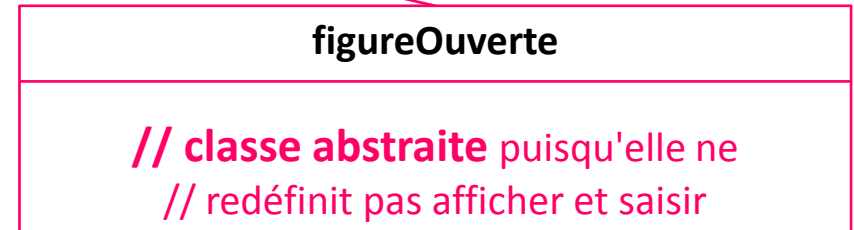
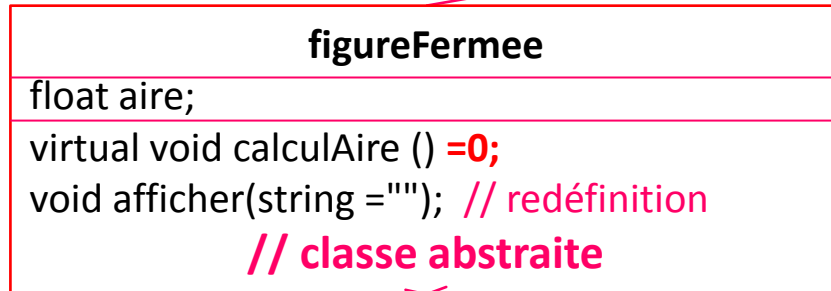
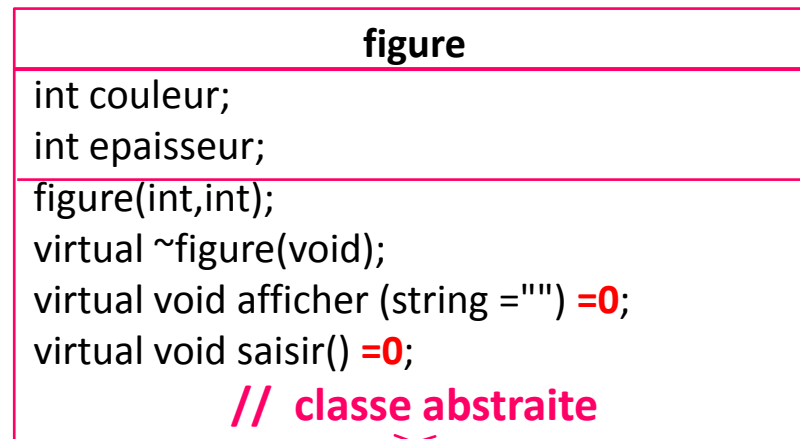
Scénario 2: la méthode afficher est déclarée **virtuelle pure** dans la classe figure. Afficher est redéfinie uniquement dans les classes vecteur et ellipse

➔ Les classes figure, carre, cercle rectangle sont abstraites donc non instanciables

➔ Il est possible de créer uniquement des objets vecteur et ellipse

➔ les classes carre et rectangle "ne servent à rien"

Exemple (voir corrigé)



```
class figure
{
protected:
    int couleur;
    int epaisseur;
public:
    figure(int =11, int =11);
    virtual ~figure(void);
    virtual void afficher(string = "")=0;
    virtual void saisir() =0;
};
```

```
class figureFermee:public figure
{
protected:
    float aire;
public:
    figureFermee(int =22, int =22);
    virtual ~figureFermee(void);
    virtual void calculAire() =0;
    void afficher(string = "");
};
```

```
class segment : public figureOuverte
{
protected:
    point ex1; // objet membre
    point ex2; // objet membre
public:
    segment(int =33,int =33, int =33, int =33, int =33, int =33);
    segment (int , int ,point,point);
    ~segment(void);
    void afficher(string = "");
    void saisir();
};
```

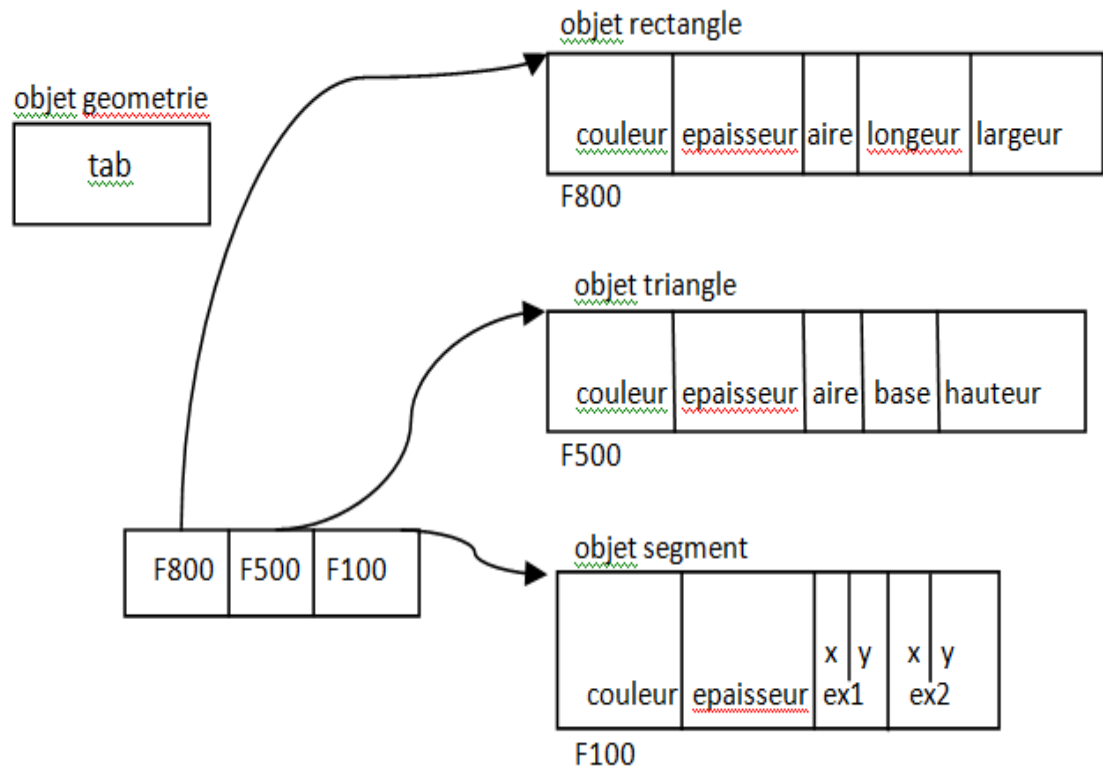
```
class figureOuverte:public figure
{
public:
    figureOuverte(int =66, int =66);
    ~figureOuverte(void);
};
```

```
class rectangle: public figureFermee
{
protected:
    int longueur;
    int largeur;
public:
    rectangle(int =33, int =33, int =33, int =33);
    ~rectangle(void);
    void afficher(string = "");
    void calculAire();
    void saisir();
};
```

```
class triangle:public figureFermee
{
    int base;
    int hauteur;
public:
    triangle(int =55, int =55, int =55, int =55);
    ~triangle(void);
    void afficher(string = "");
    void calculAire();
    void saisir();
};
```


(Voir corrigé)

```
main.cpp  courbe.cpp  geometrie.h X geometrie.cpp  point.h  segment.h
(Global Scope)
class geometrie
{
    vector<figure*> tab;
public:
    geometrie(void);
    ~geometrie(void);
    geometrie(const geometrie&);
    void afficher(string = "");
    void ajouter(rectangle, int =0);
    void ajouter(triangle, int =0);
    void ajouter(segment, int =0);
    void supprimer(int =0);
    int taille();
    void ajouter(figure*, int i=0);
};
```



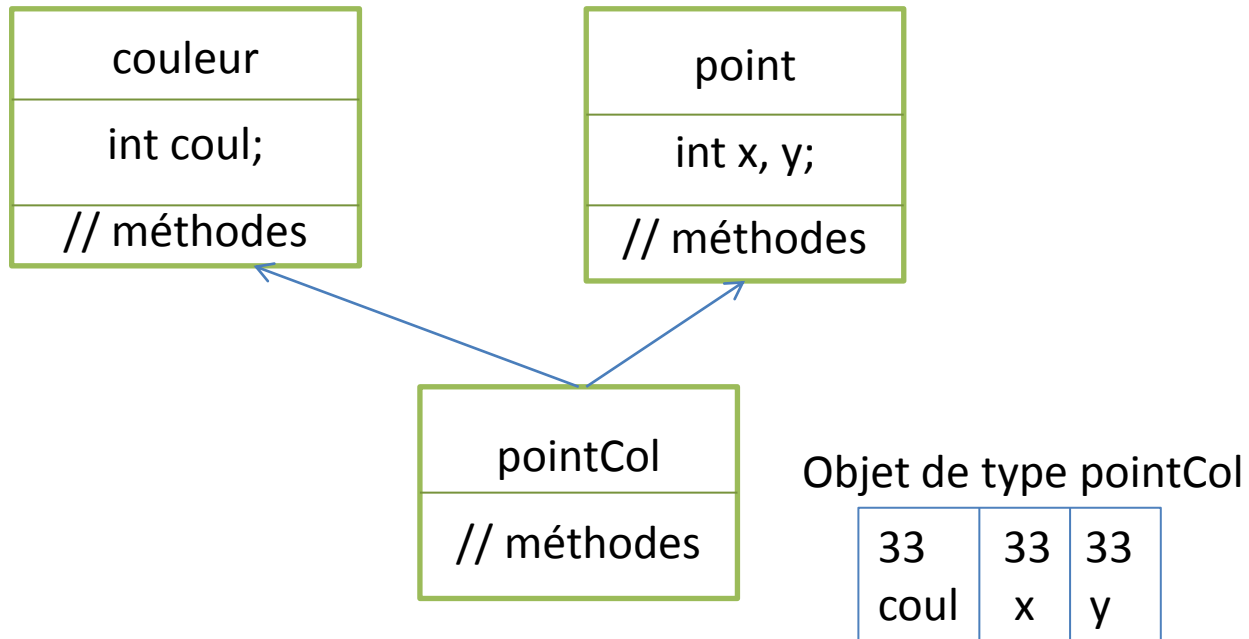
L'héritage multiple (partie 4/4)

plan

1. Mise en œuvre de l'héritage multiple
2. Pour régler les éventuels conflits: les classes virtuelles
3. Appels des constructeurs et des destructeurs: cas des classes virtuelles

Exemple

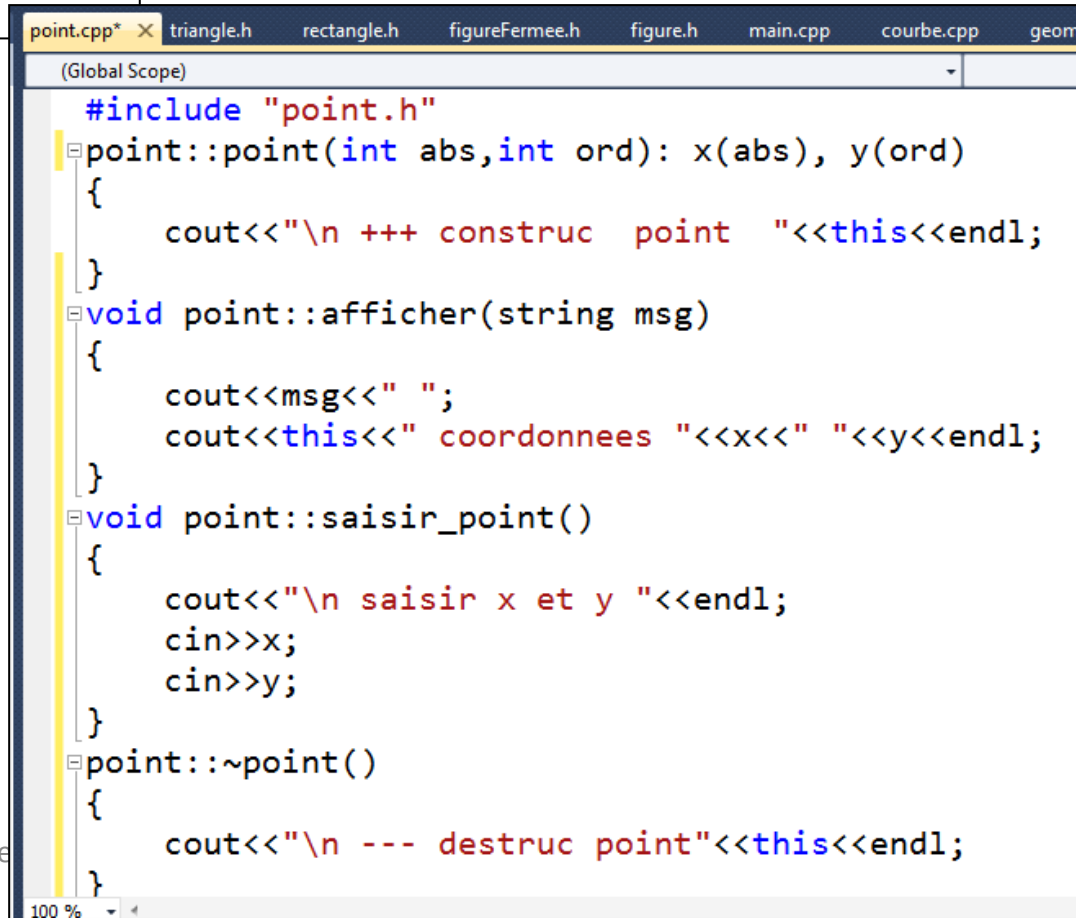
- la plupart des concepts qui concernent l'héritage simple s'étendent à l'héritage multiple.
- Soit la classe pointCol qui hérite de deux autres classes nommées *point* et *couleur*. (voir corrigé)



```

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

```



```

point.cpp* x triangle.h rectangle.h figureFermee.h figure.h main.cpp courbe.cpp geom
(Global Scope)
#include "point.h"
point::point(int abs,int ord): x(abs), y(ord)
{
    cout<<"\n +++ construc point "<<this<<endl;
}
void point::afficher(string msg)
{
    cout<<msg<<" ";
    cout<<this<<" coordonnees "<<x<<" "<<y<<endl;
}
void point::saisir_point()
{
    cout<<"\n saisir x et y "<<endl;
    cin>>x;
    cin>>y;
}
point::~~point()
{
    cout<<"\n --- destruc point"<<this<<endl;
}

```

```
class couleur
{
protected:
    int coul;
public:
    couleur(int =11);
    virtual ~couleur(void);
    virtual void afficher(string = "");
};
```

```
couleur.cpp* x couleur.h* point.cpp* triangle.h rectangle.h figureFermee.h figure.h
(Global Scope)
#include "couleur.h"
couleur::couleur(int c): coul(c)
{
    cout<<"\n +++ constr couleur "<<this<<endl;
}
couleur::~~couleur(void)
{
    cout<<"\n --- desstr couleur "<<this<<endl;
}
void couleur::afficher(string msg)
{
    cout<<msg<<endl;
    cout<<"\n la couleur est "<<coul<<endl;
}
```

Ordre de l'appel des constructeurs:

1. couleur
2. point
3. pointCol

Ordre de l'appel des destructeurs (ordre inverse):

1. pointCol
2. Point
3. couleur

```
pointCol.cpp* pointCol.h* couleur.cpp* couleur.h* point.cpp* triangle.h rectan
(Global Scope)
#pragma once
#include "point.h"
#include "couleur.h"
class pointCol: public couleur, public point
{
public:
    pointCol(int =33, int =33, int =33);
    ~pointCol(void);
    virtual void afficher(string = "");
};
```

```
pointCol.cpp* pointCol.h couleur.cpp* couleur.h* point.cpp* triangle.h rectangle.h
pointCol
#include "pointCol.h"
pointCol::pointCol(int abs, int ord, int c):
    point(abs,ord), couleur(c)
{
    // appel du constructeur de la classe point
    // appel du constructeur de la classe couleur
    cout<<"\n +++ constr pointCol "<<this<<endl;
}
pointCol::~~pointCol(void)
{
    cout<<"\n --- destr pointCol "<<this<<endl;
}
void pointCol::afficher(string msg)
{
    cout<<msg<<endl;
    point::afficher();
    couleur::afficher();
}
```

Remarque: cas des membres données

- La même démarche s'appliquerait à des membres données

```
class A
{ ....
protected:
    int x;
    ...
};
```

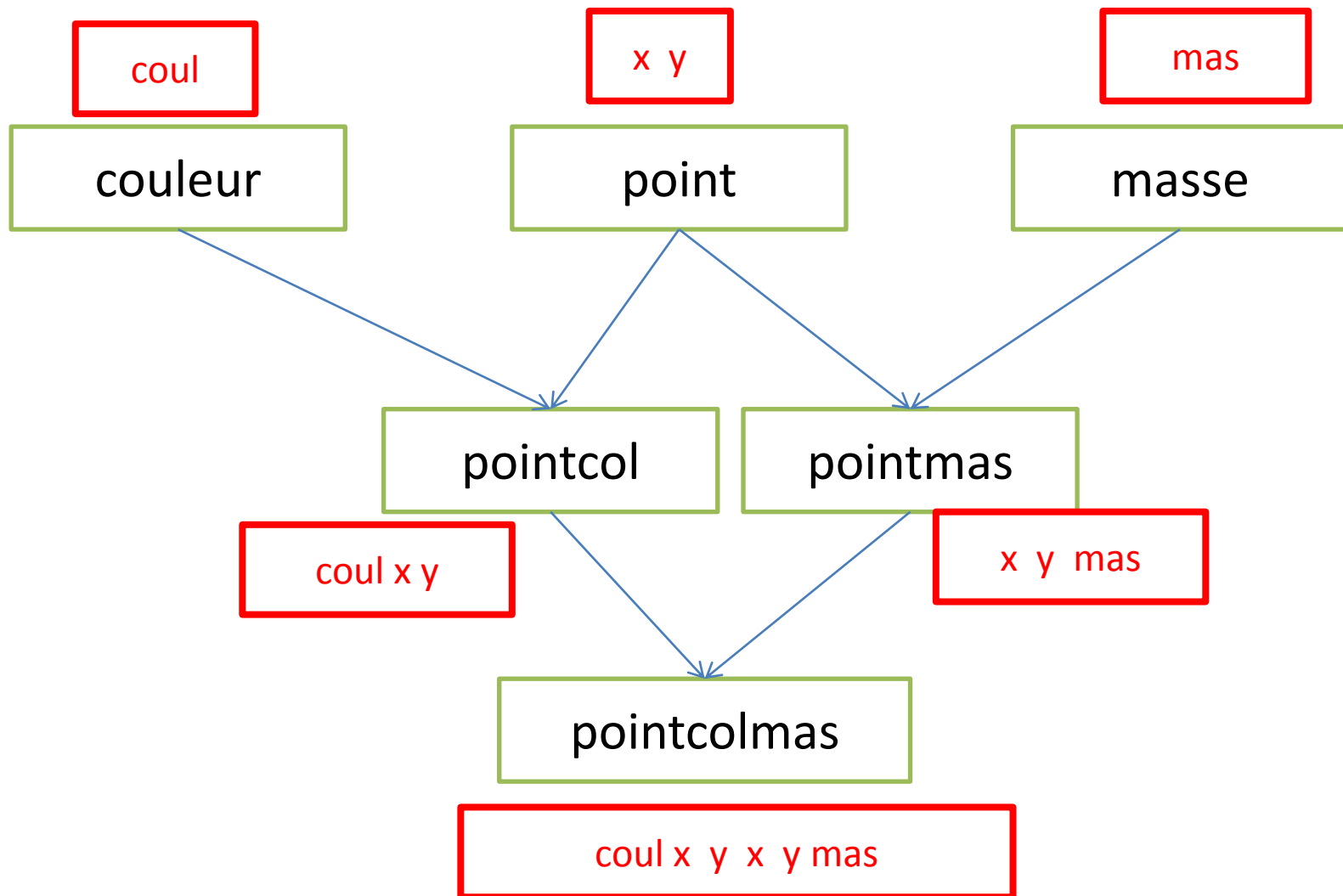
```
class B
{ ...
protected:
    int x;
    ....
};
```

```
class C: public A, public B
{ .....
    cout<< A::x;
    cout<<B::x;
};
```

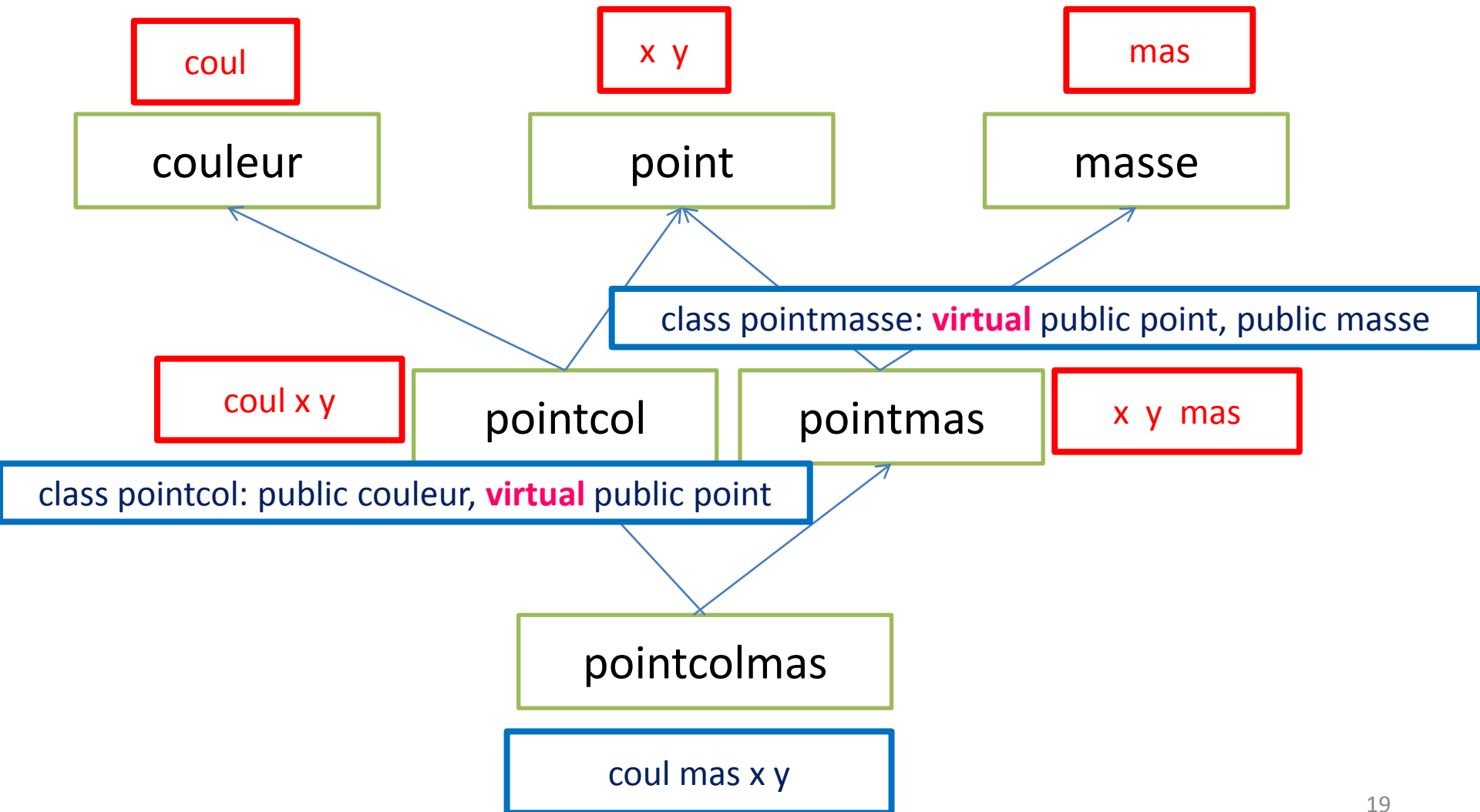

Plan

1. Mise en œuvre de l'héritage multiple
- 2. Pour régler les éventuels conflits: les classes virtuelles**
3. Appels des constructeurs et des destructeurs: cas des classes virtuelles

Problème: duplication des données de la classe point



Solution: déclarer la classe point virtuelle (voir corrigé)



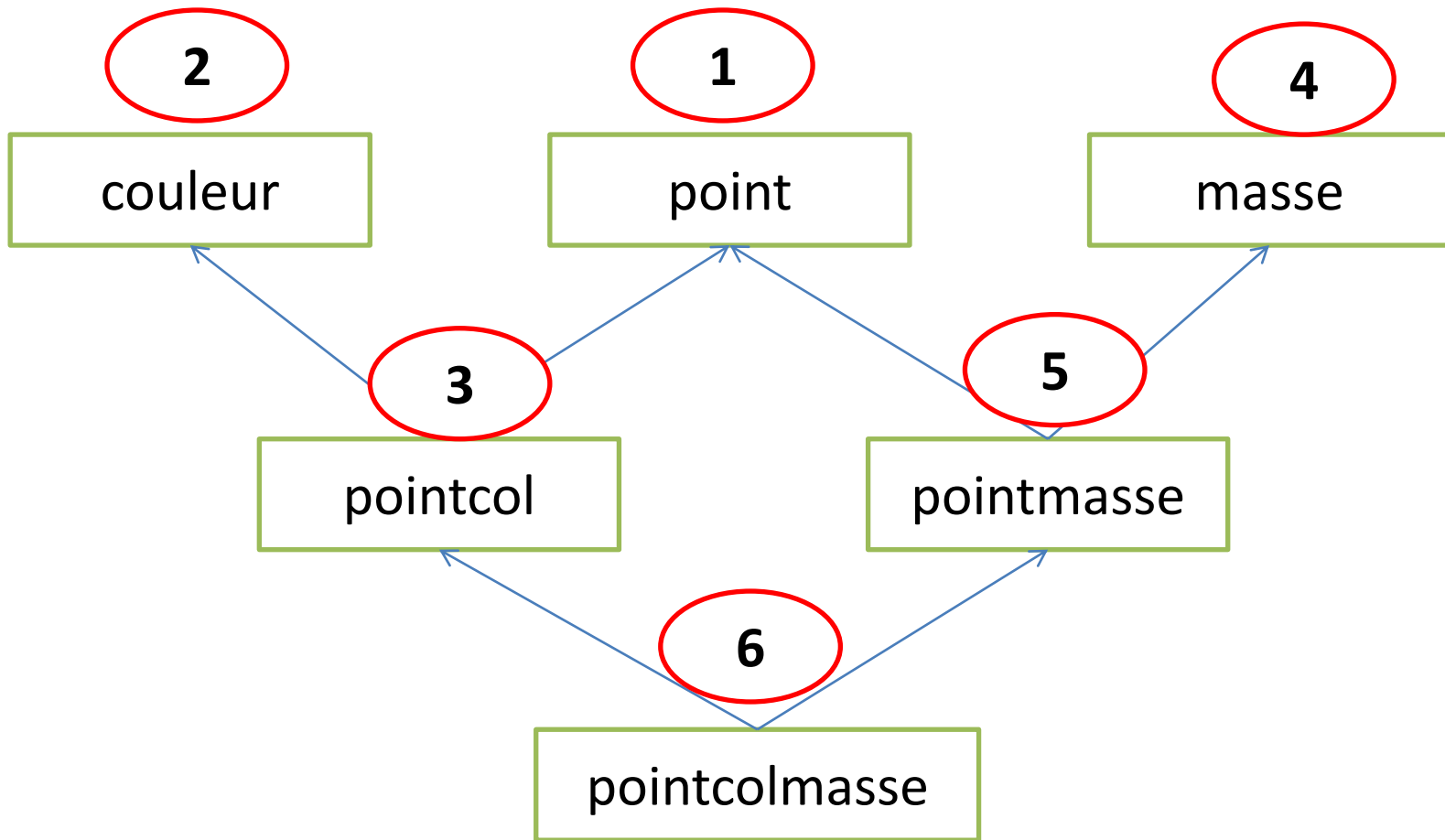
Classe virtuelle

- En général, on ne souhaitera pas cette duplication des données.
- En fait, il est possible de n'incorporer qu'**une seule fois** les membres de point (x et y) dans la classe pointcolmasse.
- Pour cela, il faut préciser, dans les déclarations des **classes pointcol et pointmasse (pas celle de pointcolmasse)** que la classe point est "**virtuelle**" (mot clé *virtual*).
- Les *méthodes virtuelles* n'ont strictement rien à voir avec les classes virtuelles, bien qu'elles utilisent le même mot clé *virtual*. Ce mot clé est utilisé ici dans un contexte différent.
- Le mot *virtual* peut être placé indifféremment avant ou après le mot *public*.

plan

1. Mise en œuvre de l'héritage multiple
2. Pour régler les éventuels conflits: les classes virtuelles
- 3. Appels des constructeurs et des destructeurs: cas des classes virtuelles**

Ordre d'appel des constructeurs dans le cas d'une classe virtuelle



Ordre d'appel des destructeurs dans le cas d'une classe virtuelle

