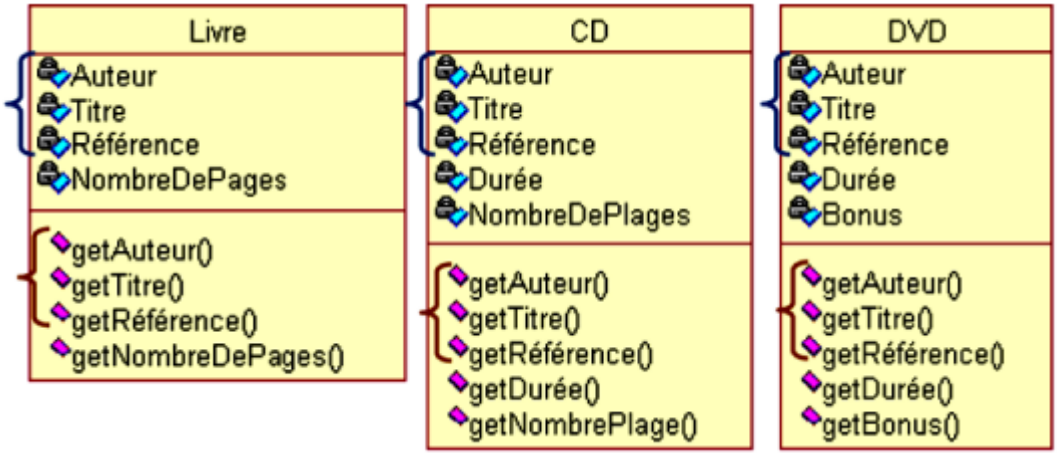
Chapitre 4. Héritage et polymorphisme

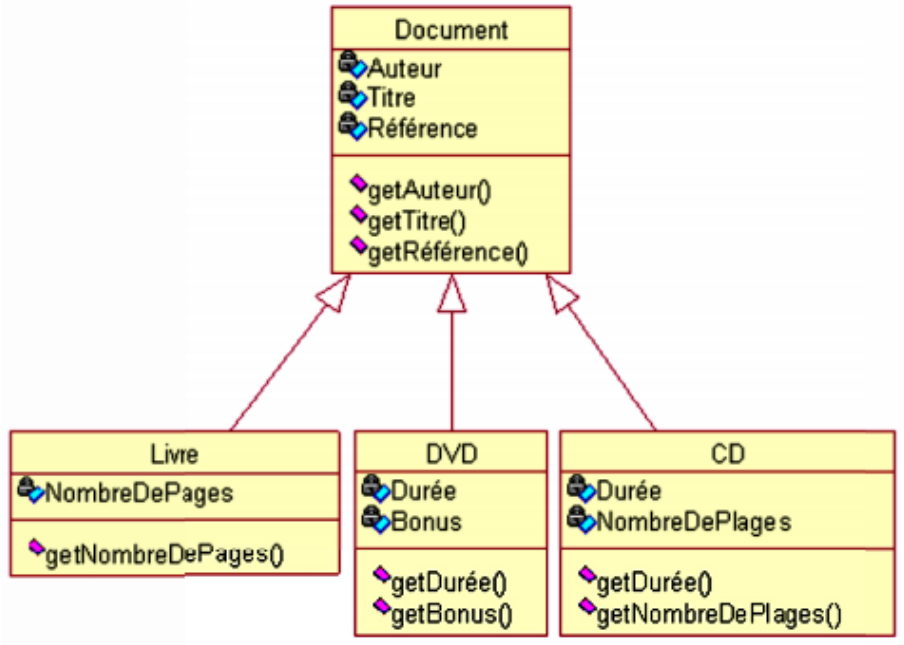
1. HERITAGE EN C++

1.1. Introduction

Imaginons que nous devions fabriquer un logiciel qui permet de gérer une bibliothèque. Cette bibliothèque comporte plusieurs types de documents ; des CDs, ou des DVDs. Une première étude nous amène à mettre en œuvre les classes suivantes :

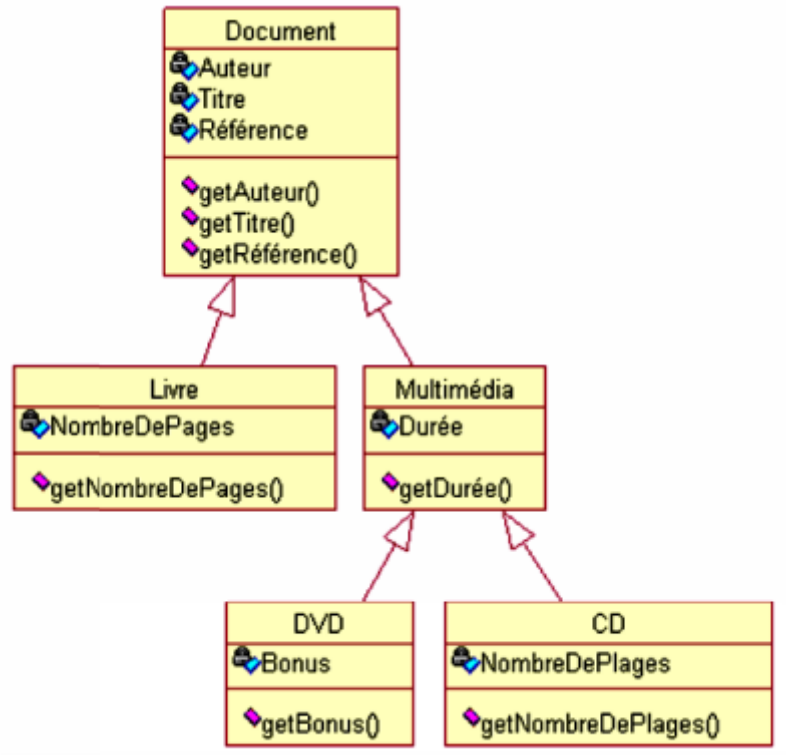


Dans les trois types de documents, un certain nombre de caractéristiques se retrouvent systématiquement et afin d'éviter la répétition des éléments constituant chacune des classes, il est préférable de factoriser toutes ces caractéristiques communes pour en faire une nouvelle classe plus généraliste.
Quel que soit le type de document, il comporte au moins un titre, un auteur, etc. Le nom de cette nouvelle classe générale s'appelle Document.



La généralisation se représente par une flèche qui part de la classe fille vers la classe mère. Par exemple, Un Livre possède, certes un nombre de page, mais en suivant la flèche indiquée par la relation de généralisation, elle comporte également un nom d'auteur, un titre, une référence. En fait, la classe Livre hérite de tout ce que possède la classe Document, les attributs comme les méthodes.

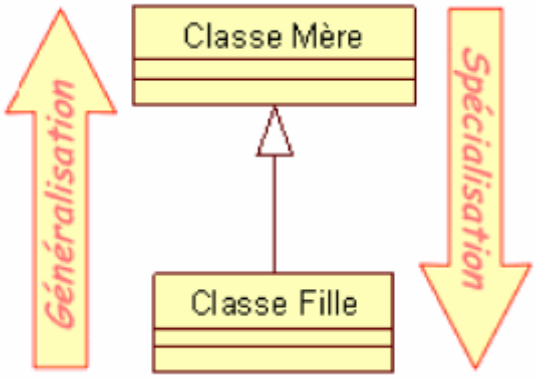
Dans cet exemple, nous avons un seul niveau d'héritage, mais il est bien entendu possible d'avoir une hiérarchie beaucoup plus développée. D'ailleurs, si nous regardons de plus près, nous remarquons que nous pouvons appliquer une nouvelle fois la généralisation en factorisant la durée du support CD et du support DVD. En fait, il s'agit dans les deux cas d'un support commun appelé Multimédia.



1.2. Définitions

L'héritage est une technique permettant de construire une classe à partir d'une ou de plusieurs autres classes dites : classe mère ou superclasse ou classe de base.

- La classe dérivée est appelée: Classe fille ou sous-classe.



- Les sous-classes héritent des caractéristiques de leurs classes parentes.
- Les attributs et les méthodes déclarées dans la classe mère sont accessibles dans les classes fils comme s'ils avaient été déclarés localement.
    - Créer facilement de nouvelles classes à partir de classes existantes (réutilisation du code)

1.3. Héritage simple

La classe dérivée hérite les attributs et les méthodes d'une seule classe mère.

Syntaxe:

```
class ClasseMere
{   //...
};
class ClasseDerivee: <mode> ClasseMere
{   //...
};
```

Remarque : mode: optionnel permet d'indiquer la nature de l'héritage: private, protected ou public. Si aucun mode n'est indiqué alors l'héritage est privé.

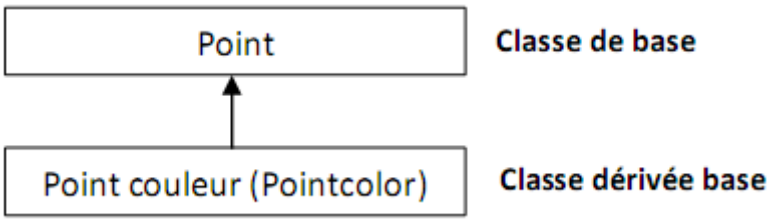
Exemple :

```
#include <iostream>
using namespace std;

class Point
{   int x;
    int y;
 public:
    void initialise (int , int) ;
    void afficher() ;
};
void Point::initialise (int a, int b)
{
    x=a;
    y=b;
}

void  Point::afficher ()
{
    cout<<"le point est en position de"<<x<<"et"<<y<<endl;
}

class Pointcolor: public Point
{   int couleur;
  public:
    void setcolor(int c)
    {  couleur=c ;  }
};
int  main()
{   Pointcolor p ;
   p.initialise (10,20) ;
   p.setcolor(5) ;
   p.afficher() ;   // (10, 20)
}
```



1.3.1. Utilisation des membres de la classe de base dans une classe dérivée

L'exemple précédent, la classe Pointcolor telle que nous l'avons définie présente des lacunes. Par exemple, lorsque nous appelons affiche pour un objet de type pointcolor nous n'obtenons aucune information sur sa couleur. Donc, en écrire une nouvelle fonction Affichercolor membre public de la classe Pointcolor aurait un accès aux membres privés de point ce qui serait contraire au principe d'encapsulation. En revanche, rien n'empêche à une classe dérivée d'accéder à n'importe quel membre public de sa classe de base.

```
#include <iostream>
using namespace std;

class Point
{   int x;
    int y;
 public:
    void initialise (int , int) ;
    void afficher() ;
};
void Point::initialise (int a, int b)
{
    x=a;
    y=b;
}
void  Point::afficher ()
{
    cout<<"le point est en position de"<<x<<"et"<<y<<endl;
}

class Pointcolor: public Point
{   int couleur;
  public:
    void setcolor(int c)
    {  couleur=c ;  }

    void Affichercolor()
{     afficher() ;
    cout << "couleur="<< couleur<<endl ;
}

};
int  main()
{   Pointcolor p ;
   p.initialise (10,20) ;
   p.setcolor(5) ;
   p.afficher() ;   // (10, 20)
   p.Affichercolor();
}
```

D'une manière analogue, nous pouvons définir dans pointcolor une fonction d'initialisation comme initialisercolor :

```
void initialisercolor(int a, int b, int c)
{   initialise(a,b) ;
   couleur=c ;
}
```

1.3.2. Contrôle d'accès pendant l'héritage

Statut des membres de la classe dérivée en fonction du statut des membres de la classe de base et du mode de dérivation:

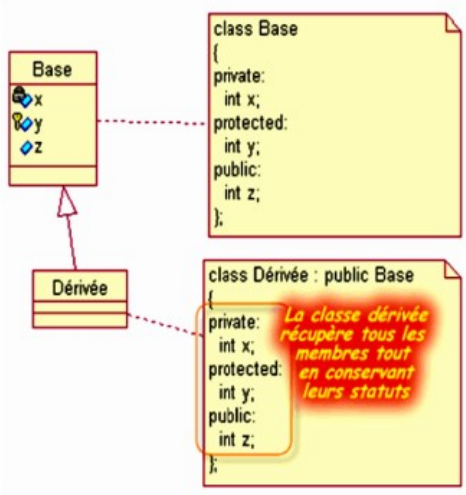
Statut des membres de base				
Mode de dérivation		Public	Protected	Private
	Public	Public	Protected	Inaccessible
	Protected	Protected	Protected	Inaccessible
	Private	Private	Private	Inaccessible

Lorsqu'une classe dérivée possède des fonctions amies, ces derniers disposent exactement des mêmes autorisations d'accès que les fonctions membres de la classe dérivée. En particulier, les fonctions amies d'une classe dérivée auront bien accès aux membres déclarés protégés dans sa classe de base.

En revanche, les déclarations d'amitié ne s'héritent pas. Ainsi, si f a été déclaré amie d'une classe A et si B dérive de A, f n'est pas automatiquement amie de B.

a. Dérivation publique

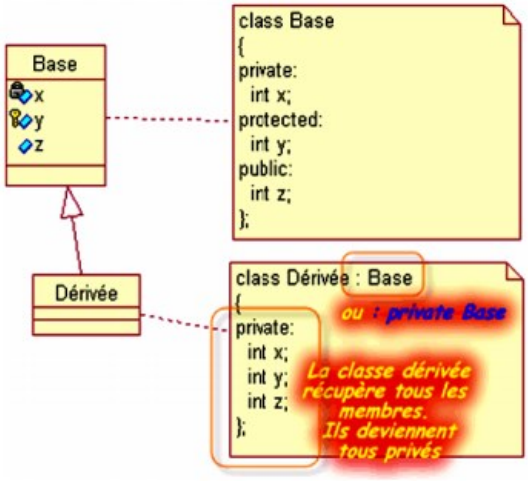
- Les membres publics de la classe de base sont accessibles « à tout le monde », c'est à la fois aux méthodes, aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.
- Les membres protégés de la classe de base sont accessibles aux méthodes et aux fonctions amies de la classe dérivée, mais pas aux utilisateurs de cette classe dérivée.
- Les membres privés de la classe de base sont inaccessibles à la fois aux méthodes ou aux fonctions amies de la classe dérivée et aux utilisateurs de la classe dérivée.



b. Dérivation privée

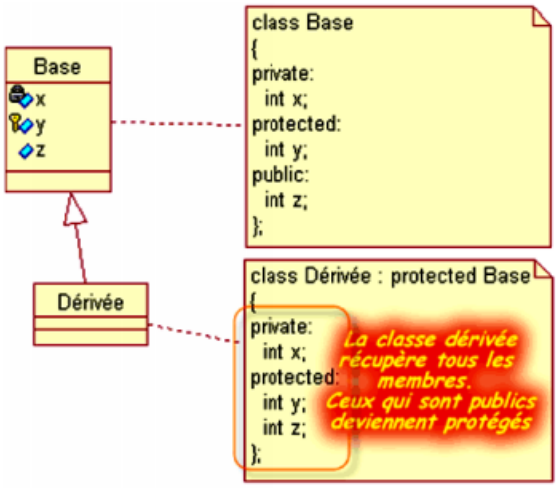
Les membres hérités publics et protégés d'une classe de base privée deviennent des membres privés de la classe dérivée.

Cette technique permet d'interdire, aux utilisateurs d'une classe dérivée, l'accès aux membres publics de sa classe de base. Cela sous-entend que seules les méthodes de la classe dérivée devront être utilisées, sinon il faudra redéfinir les méthodes de la classe de base. Pour les dérivations à partir de la classe dérivée, l'accès à la classe de base devient alors totalement inaccessible, les petits enfants n'ont donc pas accès aux membres de leurs grands-parents.



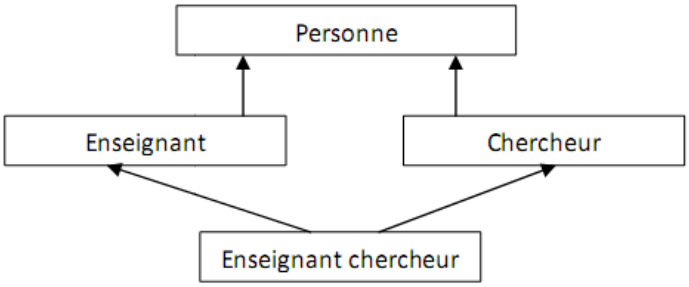
c. Dérivation protégée

Les membres hérités publics et protégés d'une classe de base protégée deviennent des membres protégés de la classe dérivée. Nous retrouvons le même principe que pour une dérivation privée, la seule différence concerne les enfants éventuels de la classe dérivée puisque dans ce cas-là, les petits enfants peuvent atteindre des membres protégés.



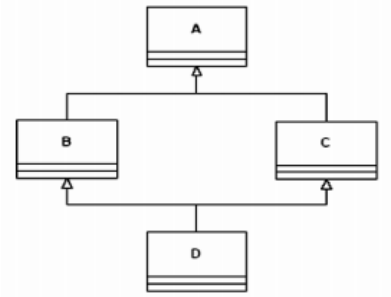
1.4. Héritage multiple

La classe dérivée hérite les attributs et les méthodes à partir de plusieurs classes mères.



L'héritage multiple est possible en C++, par contre le Java ne supporte pas l'héritage multiple. Parmi les problèmes de l'héritage multiple est possible en C++, on trouve:

- Si les deux classes mères ont des attributs ou des méthodes de même nom (collision des noms lors de la propagation).
- La classe D hérite deux fois les attributs de A, une fois à travers la classe B et la fois à travers C.



Syntaxe

```
Class  Classe_mere1
{   //...
};
class Classe_mere2
{    //...
};
class Classe_derivee: <mode> Classe
{    //...
};
```

1.5. Héritage et constructeur

Si la classe mère contient un constructeur dont l'appel est obligatoire alors le constructeur de la classe dérivée doit obligatoirement appeler celui de la classe mère. Il faut spécifier le nom et les paramètres du constructeur mère après le prototype constructeur fils.

Exemple

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

class Rectanglecolor: public Rectangle
{     int couleur;
   public:
     Rectanglecolor(int a, int b, int c): Rectangle(a, b)
     { couleur=c; }
};
```

On peut mentionner des arguments par défaut dans Rectanglecolor

```
Rectanglecolor(int a=0, int b=0, int c=1): Rectangle(a, b)
```

Dans ce cas, la déclaration : `Rectanglecolor(5);` entrainera l'appel de Recatngle avec les arguments 5 et 0 et l'appel de Rectanglecolor avec les arguments 5, 0 et 1

1.6. Hiérarchisation des appels

Soit les classes suivantes:

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

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. ce mécanisme est pris en charge par C++ : il n'y aura pas à prévoir dans le constructeur de B l'appel du constructeur de A.

La même application s'applique aux destructeur : 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 destructeurs).

Exemple

```
//Hiérarchisation des appels
//=============================================================================
#include <iostream>
using namespace std;

class Point
{
    int x;
    int y;

    public:
    Point(int abs=0, int ord=0)
     {  cout <<"++constr. point: "<<abs<<"," <<ord<<endl ;
        x=abs ; y=ord ;
     }
    ~Point()
    { cout <<"-- desctr. point: "<<x<<","<<y<<endl ; }
};

class Pointcolor: public Point
{     int couleur;
   public:
      Pointcolor(int , int , int) ;
      ~Pointcolor ()
      { cout<<"-- destr. pointcol -couleur: "<<couleur<<endl ; }
};
Pointcolor::Pointcolor(int abs=0 ,int ord=0, int c=1):Point(abs, ord)
{    couleur=c;
     cout <<"++constr. pointcol: "<<abs<<"," <<ord<<","<<couleur;
}
```

```
class Pointcolor: public Point
{     int couleur;
   public:
      Pointcolor(int , int , int) ;
      ~Pointcolor ()
      { cout<<"-- destr. pointcol -couleur: "<<couleur<<endl ; }
};
Pointcolor::Pointcolor(int abs=0 ,int ord=0, int c=1):Point(abs, ord)
{    couleur=c;
     cout <<"++constr. pointcol: "<<abs<<"," <<ord<<","<<couleur;
}

int main()
{      Pointcolor a(10,15,3) ;
       Pointcolor b(2,3) ;
       Pointcolor c(12) ;
       Pointcolor *adr;
       adr = new Pointcolor(12,25);
       delete adr ;
}
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AUTO_ELN_ELT...
++constr. point: 10,15
++constr. pointcol: 10,15,3++constr. point: 2,3
++constr. pointcol: 2,3,1++constr. point: 12,0
++constr. pointcol: 12,0,1++constr. point: 12,25
++constr. pointcol: 12,25,1-- destr. pointcol -couleur: 1
-- desctr. point: 12,25
-- destr. pointcol -couleur: 1
-- desctr. point: 12,0
-- destr. pointcol -couleur: 1
-- desctr. point: 2,3
-- destr. pointcol -couleur: 3
-- desctr. point: 10,15
```

1.7. Compatibilité entre classe de base et classe dérivée

Dans le cadre de l'héritage, il sera possible de passer d'une classe dérivée vers une classe de base. Par contre l'inverse ne sera pas possible.

```
#include <iostream>
using namespace std;
class Forme
{    int x;
     int y;
  public:
     Forme(int x, int y)
     { this->x=x ; this->y=y ;}

    void deplacer(int dx, int dy)
     { x+=dx;y+=dy ; }
};

class Cercle: public Forme
{     int rayon;
   public:
     Cercle(int x, int y, int r):Forme(x, y)
      { rayon=r ; }

     void agrandir(int a)
     { rayon+=a; }
};

int main()
{       Forme f1(2,3);   //Création de l'objet f1.
        Cercle c1(15, -1, 50) ; // Création de l'objet c1
        c1.deplacer(3, 4) ; //Possibilité de déplacer le cercle c1 puisque la méthode a été héritée.
        c1.agrandir(10) ; //Agrandissement du cercle c1 grâce à la méthode agrandir  agrandir définie par la classe Cercle
        f1=c1 ; // le cercle  c1  est aussi une forme
        f1.deplacer(5, -8) ; // la méthode associée a été définie dans la classe Forme
        c1=f1 ; // f1 ne dispose pas de rayon ( une erreur de compilation)
}
```

2. Polymorphisme

2.1. Introduction

Le terme polymorphisme vient du grec, est composé de deux mots : **poly** signifie plusieurs et **morphisme** signifie forme. Le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes. Pour bien comprendre la puissance du polymorphisme, nous étudions l'exemple suivant

```
#include <iostream>
using namespace std;

class Point
{
    int a, b;
    public:
     void affiche();
};

 void  Point::affiche()
       {
           cout<<"la classe mere"<<endl;
       }
```

```
class Pointfille:public Point
{
    public:
        void affiche();

};
void Pointfille::affiche()
    {
        cout<<"la class fille"<<endl;
    }
```

```
int main()
{
    Point p;
    Pointfille pf;
    Point *ap;

    ap=&p; ap->affiche();// afficher "la classe mere
    ap=&pf; ap->affiche();// afficher la classe mere
    return 0;
}
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AUT...
la classe mere
la classe mere

--------------------------------
Process exited after 0.1557 seconds with return value
Appuyez sur une touche pour continuer...
```

Le choix de la fonction affiche est conditionné par le type de **ap** connu lors de la compilation. Puisque **ap** est de type **Point**, alors il utilise la méthode **Point :: affiche().**
Pour résoudre ce problème il faut utiliser la fonction **virtuelle.**

2.2. Fonction virtuelle

```
#include <iostream>
using namespace std;

class Point
{
    int a, b;
    public:
    virtual void affiche();
};

 void  Point::affiche()
       {
           cout<<"la classe mere"<<endl;
       }
```

```
class Pointfille:public Point
{
    public:
        void affiche();

};
void Pointfille::affiche()
    {
        cout<<"la class fille"<<endl;
    }
```

```
int main()
{
    Point p;
    Pointfille pf;
    Point *ap;

    ap=&p; ap->affiche();// afficher "la classe mere
    ap=&pf; ap->affiche();// afficher la classe fille

    return 0;
}
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AUTO_ELN_EL...
la classe mere
la class fille

--------------------------------
Process exited after 0.4581 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Le choix de la fonction est maintenant conditionné par le type exact de l'objet pointé par **ap** connu au moment de l'exécution puisque p nous emmène sur un **Pointfill**e alors on utilise **Pointfille ::affiche()**

2.3. Définition de Polymorphisme

Mécanisme qui consiste à définir des fonctions de même nom dans les classes de bases et les classes dérivées et qui répondent différemment à un même appel. Les classes dérivées héritent tous les membres publics et protégés de la classe mère.

- Déclarer **virtual** la fonction concernée dans la classe la plus générale de la hiérarchie d'héritage.
- Toutes les classes dérivées qui apportent une nouvelle version de **affiche()** utiliseront leur fonction à elles.
- Pour appeler la fonction **affiche()** de **Point** en spécifiant **p->Point ::affiche()**

2.4. Destructeur virtuel

Exemple : nous étudions l'exemple suivant

```
// Destructeur virtual
///================================
#include <iostream>
using namespace std;


class Point
{   protected :
       int *p;

   public :
      Point()
       { p=new int[2] ;
         cout<<"Point()"<<endl ;
       }

~Point()
     { delete [] p ;
       cout<<"~Point()"<<endl ;
      }
} ;
```

```
class Pointfille : public Point
{      int *q;
    public :
       Pointfille ()
       { p=new int[20] ;
          cout<<"Pointfille ()"<<endl ;
       }
       ~Pointfille ()
      { delete [] p ;
        cout<<"~Pointfille ()"<<endl ;
      }
} ;
```

```
int main()
{
  for (int I =0; I<4;I++)
     {
     Point *pa = new Pointfille ();
        delete pa;
     }
}
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AUTO_ELN_ELT\electronique\...
Point()
Pointfille ()
~Point()
Point()
Pointfille ()
~Point()
Point()
Pointfille ()
~Point()
Point()
Pointfille ()
~Point()

--------------------------------
Process exited after 0.03923 seconds with return value 0
```

ILfaut faire appel au destructeur de **Pointfille** avant celui de **Point**, puisque **pa** référence un **Pointfille.** Dans ce cas on a un espace mémoire alloué non libéré : (fuite de mémoire).

Pour résoudre ce problème il faut introduire un : **destructeur virtuel.**

Donc la classe **Point** devient :

```
class Point
{   protected :
       int *p;

   public :
      Point()
       { p=new int[2] ;
         cout<<"Point()"<<endl ;
       }

   virtual ~Point()
       { delete [] p ;
         cout<<"~Point()"<<endl ;
       }
} ;
```

```
D:\doc univ DBKM\cours_Licence_Master\programme_AUTO_E...
Point()
Pointfille ()
~Pointfille ()
~Point()
Point()
Pointfille ()
~Pointfille ()
~Point()
Point()
Pointfille ()
~Pointfille ()
~Point()
Point()
Pointfille ()
~Pointfille ()
~Point()

--------------------------------
Process exited after 1.29 seconds with return value 0
Appuyez sur une touche pour continuer... _
```

2.5. Fonction virtuelle pure- – classe abstraite

On peut définir des classes destinées non pas à instancier des objets, mais simplement à donner naissance à d'autres classes par héritage (**Classes Abstraites**). Nous pouvons définir de telles classes, en déclarant des fonctions membres virtuelles dont on ne précise pas le contenu dans la classe de base.

Une fonction virtuelle pure se déclare en remplaçant le corps de la fonction par les symboles = 0.

Syntaxe :

```
class NomClasse
{
    // …
    virtual TypeRetout nomFonction (liste des arguments)=0;
   //…
}
```

- 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.
- Une fonction déclarée virtuelle pure dans une classe de base doit obligatoirement être redéfinie dans une classe dérivée ou déclarée à nouveau virtuelle pure ; dans ce dernier cas, la classe dérivée est aussi abstraite.

Exemple:

```
///================================
// Fonction virtuelle pure- - class
///================================

#include <iostream>

using namespace std;
class Figure
{   public:
       virtual float perimetre()=0;
       virtual char * getNom()=0;
};
```

```
class UnRectangle:public Figure
{       float longueur;
        float largeur;
        public :
        UnRectangle(float longueur, float largeur)
        {
            this->longueur=longueur;
            this->largeur=largeur;
        }

       float perimetre()
       {
       return 2*(longueur+largeur);
       }

       char* getNom()
       {
       return "RECTANGLE";
       }
};
```

```
class UnRectangle:public Figure
{       float longueur;
        float largeur;
        public :
        UnRectangle(float longueur, float largeur)
        {
            this->longueur=longueur;
            this->largeur=largeur;
        }

       float perimetre()
       {
       return 2*(longueur+largeur);
       }

       char* getNom()
       {
       return "RECTANGLE";
       }
};
```

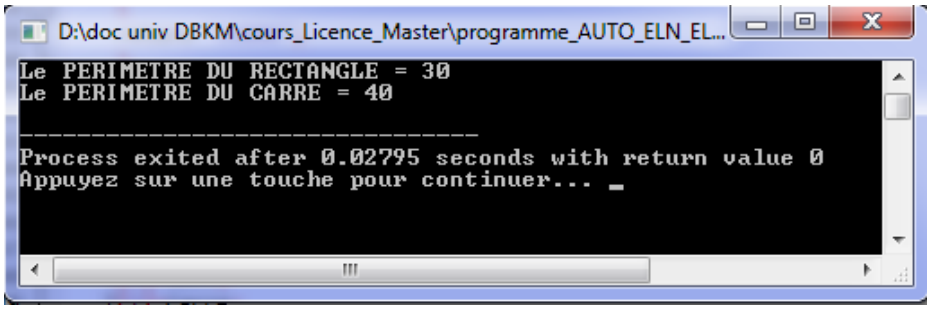
```
class UnCarre:public UnRectangle
{       float cote;

    public:
       UnCarre(float cote):UnRectangle(cote,cote)
       {
       this->cote=cote;
        }

      float perimetre()
      {
      return cote*4;
       }

     char* getNom()
     {
     return "CARRE";
     }
};
```

```
int main()
{
    UnRectangle *Rect=new UnRectangle(10,5);
    UnCarre *Carre=new UnCarre(10);
   Figure *T[2];
   T[0]=Rect;
   T[1]=Carre;
   for (int i=0;i<=1;i++)
   {
     cout<<"Le PERIMETRE DU "<<T[i]->getNom()<<" = "<< T[i]->perimetre()<<endl;
   }
}
```



- La classe figure est une abstraction. Un programme ne créera pas d'objet Figure.
- On remarque que la fonction **perimetre()** introduite au niveau de la classe Figure n'est pas un service rendu aux programmeurs mais une contrainte :
- Son rôle n'est pas de dire ce qu'est le périmètre d'une figure, mais d'obliger les futures classes dérivées de figure à le dire.