Chapitre VIII: Surcharge d'opérateurs

VIII.1 Introduction : surcharge d'opérateurs

C++ autorise la surdéfinition de fonctions, qu'il s'agisse de fonctions membres ou de fonctions indépendantes.

Exemple:

```
class ratio
{  private :
    int num, den ;
  public :
     ratio ( n=0, d=1) ;
     ratio(n) ;
} ;
```

Attribuer le même nom à des fonctions différentes, lors de l'appel, le compilateur fait le choix de la bonne fonction suivant le nombre et les types d'arguments.

C++ permet aussi la surdéfinition d'opérateurs [2]

Exemple :

a+b ;

Le symbole + peut designer suivant le type de a et b :

- L'addition de deux entiers
- Addition de deux réels
- Addition de deux doubles.
- + est interprété selon le contexte.

En C++ on peut surdéfinir n'importe quel opérateur existant (unaire ou binaire). Ce qui va nous permettre de créer par le biais de classes, des types avec des opérateurs parfaitement intégrés.

✍ On peut donner une signification à des expressions comme a+b, a-b, a*b et a/b pour la classe ratio

VIII.2 Mécanisme de surdéfinition [2]

Considérons la classe point :

```
class point
{  private :
    int x, y ;
  public :
    // partie publique
    ...
} ;
```

Supposons que nous souhaitons définir l'opérateur (+) afin de donner une signification à l'expression a+b, tels que, a et b sont de type point.

On considère que la somme de deux points est un point dont les coordonnées sont la somme de leurs coordonnées.

Pour ***surdéfinir*** (***surcharger***) cet opérateur en C++, il faut définir une fonction de nom :

Operatorxx où **xx** est le symbole de l'opérateur.

La fonction **operator+** doit disposer de deux arguments de types point et fournir une valeur de retour de même type.

Cette fonction peut être définie en tant que fonction membre de la classe (méthode) ou fonction indépendante (fonction amie).

VIII.3 Surcharge d'opérateurs par des fonctions amies

Le prototype de notre fonction sera : **point operator+ (point, point)**
Les deux opérandes correspondent aux deux opérandes de l'opérateur +
Le reste du travail est classique [2] :

- Déclaration d'amitié
- Définition de la fonction

Exemple :

```
class point
{  private :
    int x, y ;
  public :
    point (int abs =0, ord =0) ;
    friend point operator+ (point , point) ;
    void afficher( );
} ;
point operator+ (point a, point b)
{
point p;
p.x =a.x +b.x;
p.y = a.y + b.y;
return p;
}
void main()
{
 point a (1,2);
 point b (2,5);
 point c;
 c= a+b; c.afficher();
 c= a+b+c; c.afficher();
}
```

On a surchargé l'opérateur **+** pour des objets de type point en employant une fonction amie.

VIII.4 Surcharge d'opérateurs par des fonctions membres

Dans ce cas, le premier opérande sera l'objet ayant appelé la fonction membre

Exemple :

L'expression a+b sera interprétée par le compilateur comme a.operator+ (b) ;
Le prototype de notre fonction membre est alors: point operator+ (point);

Exemple:

```
class point
{  private:
    int abs, ord;
  public:
    point (int abs, int ord);
    point operator+ (point a);
    void affiche();
};
point point ::operator+ (point a);
{  point p;
  p.x = x + a.x;
  p.y = y + a.y;
  return (p) ;
}
void main()
{
  point a (1,2);
  point b (2,5);
  point c;
  c= a+b; c.afficher();
  c= a+b+c; c.afficher();
}
```

Remarque :

Dans ce cas: c = a+b; c = a.operator+ (b);

Dans le 1er cas : a= operator+(a,b);

VIII.5 Surcharge en général

On a vu un exemple de surdéfinition de l'opérateur + lorsqu'il reçoit deux opérandes de type point. Ces de deux façons :

- fonctions amies
- fonctions membres

Remarques :

- Il faut se limiter aux opérateurs existants. Le symbole qui suit le mot clé **operator** doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est pas permis de créer de nouveaux symboles.
- Certains opérateurs ne peuvent pas être redéfinis de tout (.)
- Il faut conserver la pluralité de l'opérateur. unaire ou binaire.

  Binaire : + - / * -> ( )

  Unaire : -- ++ new delete
- Il faut se placer dans un contexte de classe :

On ne peut surdéfinir un opérateur que s'il comporte au moins un argument de type classe → Une fonction membre.

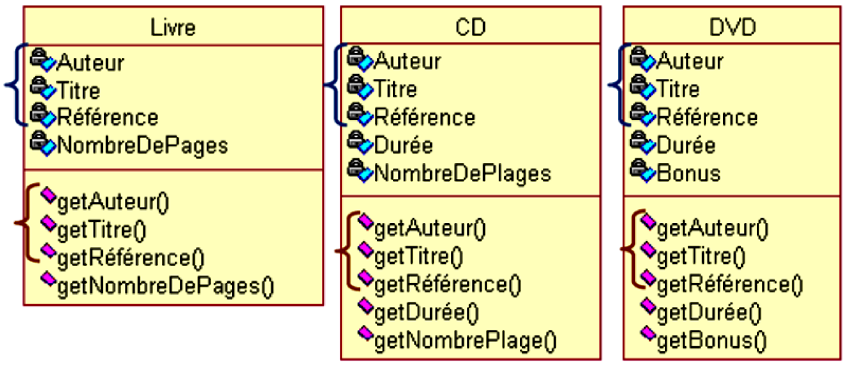
Une fonction indépendante ayant au moins un argument de type classe (Fonction amie).

Chapitre IX: HERITAGE EN C++

IX.1 Introduction

IX.1.1 Exemple [7]

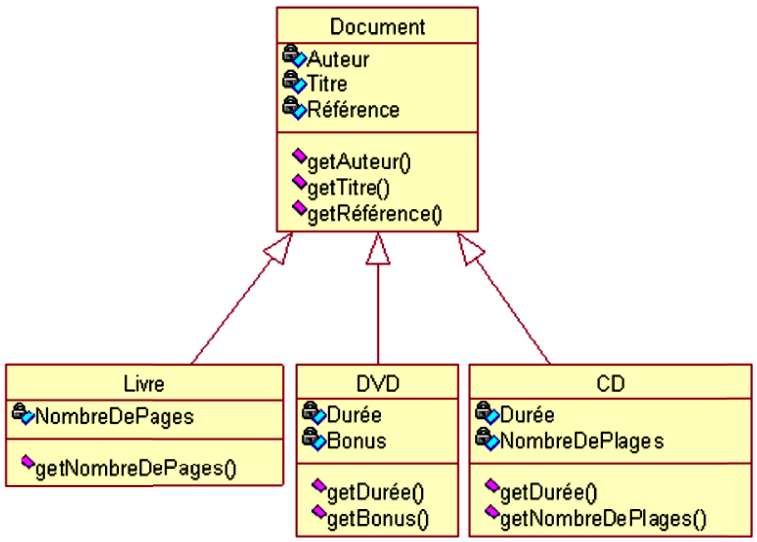
Imaginons que nous devions fabriquer un logiciel qui permet de gérer une bibliothèque. Cette bibliothèque comporte plusieurs types de documents; des livres, des CDs, ou des DVDs. Une première étude nous amène à mettre en œuvre les classes suivantes:



Nous remarquons que dans les trois types de documents, un certain nombre de caractéristiques se retrouvent systématiquement.

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.

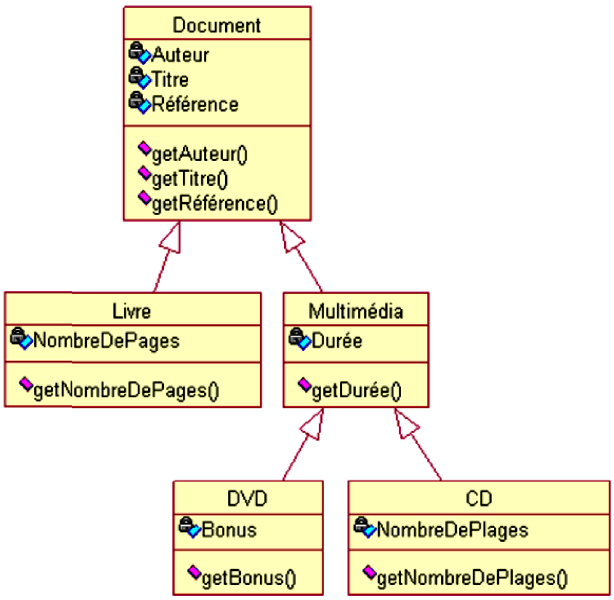
En effet, nous pouvons dire que, d'une façon générale, et quelque soit le type de document, il comporte au moins un titre, un auteur, etc. il semble aller de soi, que le nom de cette nouvelle classe générale s'appelle justement Document.



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

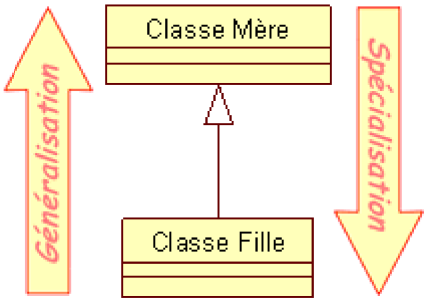
référence. En fait, la classe Livre hérite de tout ce que possède la classe Document, les attributs comme les méthodes.

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 prêt, 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.



IX.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 parents.
- 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)

IX.2 Héritage simple

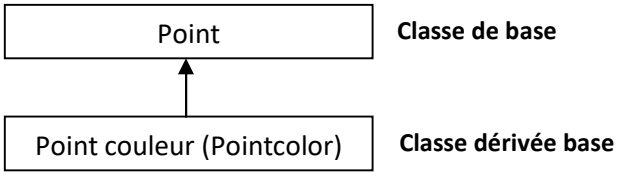
IX.2.1 Définition: *La classe dérivée hérite les attributs et les méthodes d'une seule classe mère.*

Syntaxe:

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

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

Exemple :




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

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

void main()
{   Pointcolor p ;
   p.initialiser (10,20) ;
   p.setcol(5) ;
   p.afficher() ;   // (10, 20)
}
```

IX.2.2 Utilisation des membres de la classe de base dans une classe dérivée

L'exemple précédent, destiné à montrer comment s'exprime l'héritage en C++, ne cherchait pas à explorer toutes les possibilités.

Or 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.

Une première façon d'améliorer cette situation consiste à écrire une nouvelle fonction membre public :

```
void Affichercolor()
{   cout << "("<<x<<","<<y<<")"<<endl ;
    cout << "couleur="<< couleur<<endl ;
}
```

Mais alors cela signifierait que la fonction *Affichercolor* membre 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. D'ou une définition possible d' *Affichercolor:*

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

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

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

IX.2.4 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 [7]:

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

Remarque :

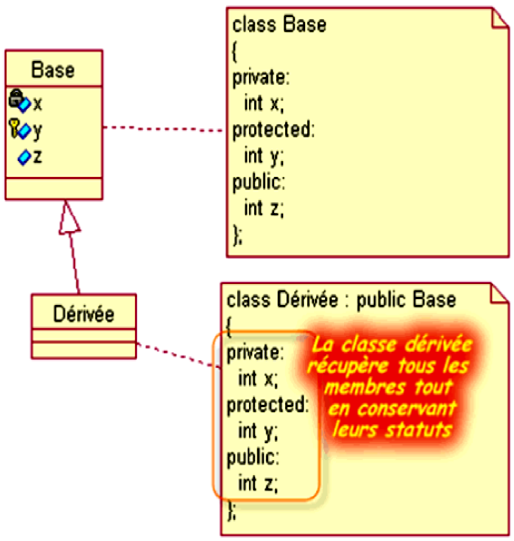
- Lorsqu'une classe dérivée possède des fonctions amies, ces derniers disposent exactement des mêmes autorisations d'accès que les fonctions membres de la classe dérivée. En particulier, les fonctions amies d'une classe dérivée auront bien accès aux membres déclarés protégés dans sa classe de base.
-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

Dérivation publique [14]**:**

1. Les membres *publics* de la classe de base sont accessibles « à tout le monde », c'est-à dire à la fois aux méthodes, aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.

2. 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.

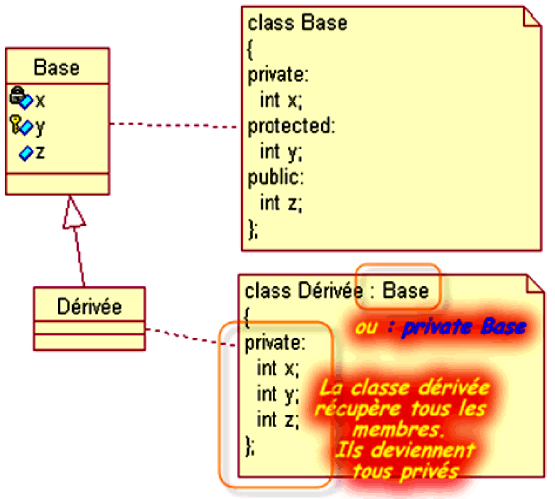
3. Les membres *privés* de la classe de base sont inaccessibles à la fois aux méthodes ou aux fonctions amies de la classe dérivée et aux utilisateurs de la classe dérivée.



Dérivation privée:

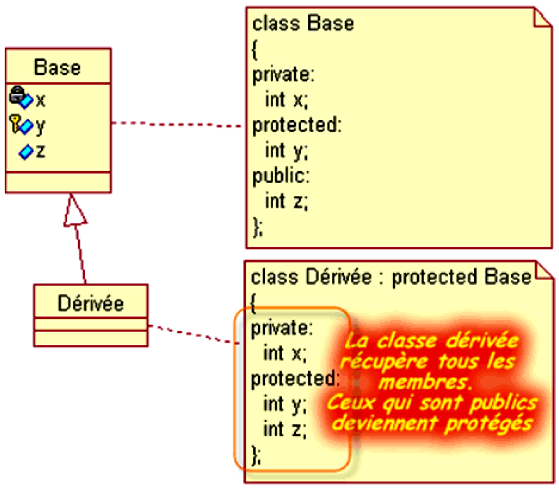
1. Les membres hérités publics et protégés d'une classe de base privée deviennent des membres privés de la classe dérivée.

2. 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 (voir plus loin). 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 leur grands parents.



Dérivation protégée:

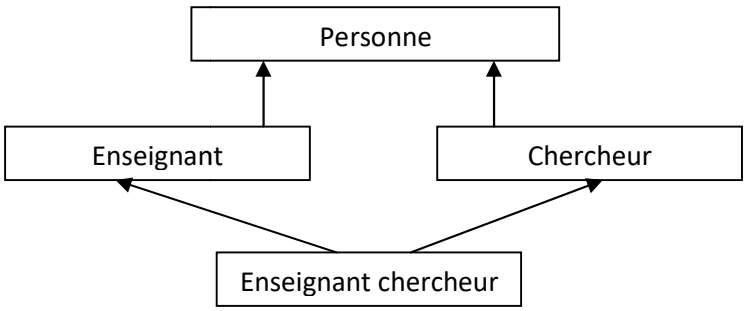
1. Les membres hérités publics et protégés d'une classe de base protégée deviennent des membres protégés de la classe dérivée.
Nous retrouvons le même principe que pour une dérivation privée, la seule différence concerne les enfants éventuels de la classe dérivée, puisque dans ce cas là, les petits enfants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



IX.3 Héritage multiple [7]

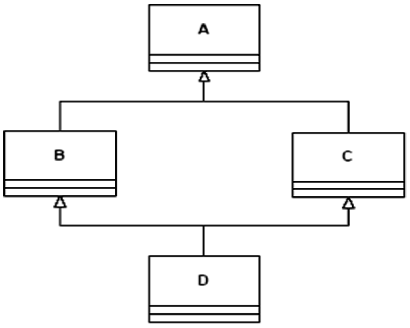
La classe dérivée hérite les attributs et les méthodes à partir de plusieurs classes mères.

Exemple:



Remarque :
L'héritage multiple est possible en C++ mais permet de poser quelques problèmes :

- 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 l'autre fois à travers C.



- Java ne supporte pas l'héritage multiple.

Syntaxe :

```
Class  Classe_mere1
{  //…
};
class Classe_mere2
{   //…
};
class Classe_derivee: <mode> Classe_mere1, <mode> Classe_mere2
{   //…
};
```

IX.4 Héritage et constructeur [14]

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

Exemple:

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

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

Il est possible de mentionner des arguments par défaut dans *Pointcolor*, par exemple :

```
Pointcolor(int a=0, int b=0, int c=1): Point(a, b)
```

Dans ces conditions, la déclaration :

```
Pointcolor  b(5) ;
```

Entrainera :

- l'appel de *Point* avec les arguments 5 et 0
- l'appel de *Pointcolor* avec les arguments 5, 0 et 1

✍ 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:

```
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;
}

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

Résultat de l'exécution :

```
++ 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

Process returned 0 (0x0)   execution time : 3.318 s
Press any key to continue.
```

Remarque :

Quelque soit les situations, nous disposons toujours des quatre mêmes phases pour la création de l'objet et toujours dans le même ordre.

1. *Allocation mémoire* nécessaire pour contenir tous les attributs que comporte l'objet.
2. *Appel du constructeur de la classe dérivée*. Ce constructeur est appelé mais pas encore exécuté. Appel du constructeur de la classe de base spécifié par la liste d'initialisation en récupérant les bons arguments pour les attributs de la classe de base.

3. *Appel et exécution du constructeur de la classe de base*. A moins que la classe de base soit elle-même une classe dérivée d'une autre classe de base, les instructions qui constituent le corps du constructeur sont exécutées.

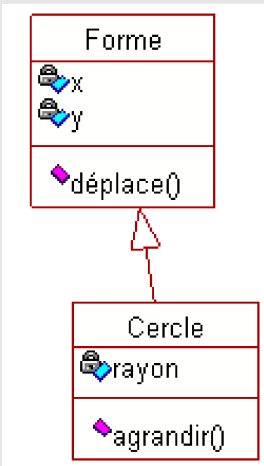
4. *Exécution du constructeur de la classe dérivée*. Puisque la partie générale est bien initialisée, nous pouvons nous occuper de la partie spécifique à la classe dérivée. Les instructions du corps du constructeur sont donc exécutées.

IX.5 Compatibilité entre classe de base et classe dérivée [14]

Nous pouvons toujours dire qu'un cercle est aussi une forme et qu'un élève est aussi une personne. La réciproque n'est pas vraie. Selon le besoin, nous savons établir des conversions implicites qui permettent de passer d'un type vers un autre. 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.

Exemple:

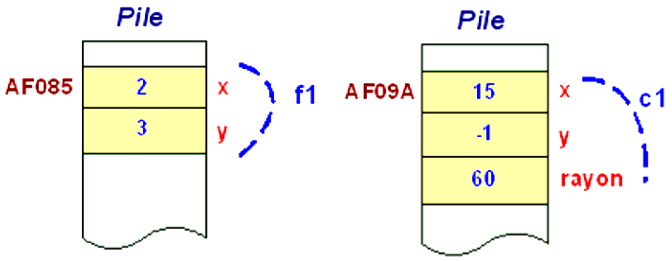
```
class Forme
{   int x;
    int y;
 public:
    Forme(int x, int y)
    { this->x=x ; this->y=y ;}
   void deplacer(int dx, int dy)
    { x+=dx;y+=dy ; }
};

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; }
};

void main()
{      Forme f1(2,3);   //1
       Cercle c1(15, -1, 50) ; //2
       c1.deplacer(3, 4) ; //3
       c1.agrandir(10) ; //4
       f1=c1 ; //5
       f1.deplacer(5, -8) ; //6
       c1=f1 ; //7
}
```



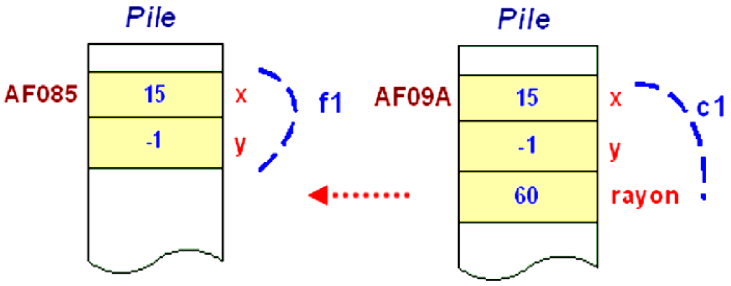
Commentaire:

1. Création de l'objet *f1*.
2. Création de l'objet *c1*.
3. Possibilité de déplacer le cercle *c1* puisque la méthode a été héritée.

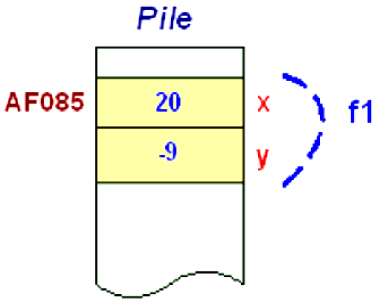
4. Agrandissement du cercle *c1* grâce à la méthode *agrandir* définie par la classe *Cercle*.



5. A droite et à gauche de l'opérateur d'affectation, les types sont différents. C'est toujours le type qui est à droite qui est transformé vers le type de gauche. Ici, le cercle *c1* est aussi une forme, donc la conversion implicite est lancée. Cette démarche paraît normale puisqu'à l'issue de cette opération, les attributs de l'objet *f1* sont parfaitement définis.



6. Dans ce contexte, il est également possible de changer de position puisque, de toute façon, la méthode associée a été définie dans la classe *Forme*.



7. Tentative d'affectation d'une forme dans un cercle. Nous obtenons également une ***erreur de compilation***. Il s'agit également d'un changement de type. Si ce casting était toléré, cela voudrait dire que nous autoriserions d'avoir des attributs avec des valeurs *aléatoires !*
Effectivement *f1* ne dispose pas de *rayon*, ainsi l'attribut *rayon* de *c2* se retrouverait sans aucune valeur bien précise. Cette démarche n'est pas tolérée par le compilateur et nous le comprenons.