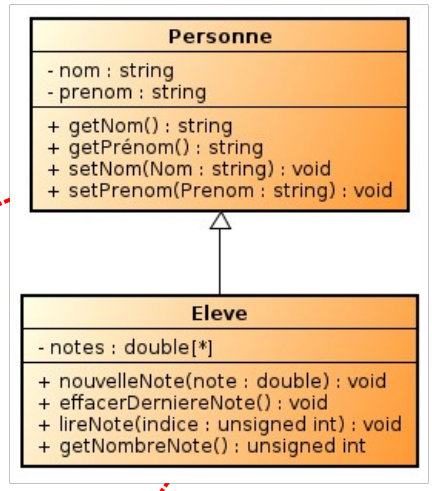
Dans le chapitre précédent, nous avons évoqués les principes généraux de l'héritage sans tenir compte d'un langage quelconque. Ce chapitre sera donc consacré à l'étude de l'héritage simple codé avec le langage C++.

## Déclaration de l'héritage simple (Classes sans constructeurs)

Le diagramme ci-contre nous montre un exemple simple d'héritage. Il n'est, certes, pas très sophistiqué, mais il va me servir d'appui pour bien spécifier les différentes écritures.




```cpp
#ifndef PersonneH
#define PersonneH
#include <string>
using namespace std;
//---------------------------------------------------------------------------
class Personne
{
  string nom, prenom;
public:
  string getNom() const;
  string getPrenom() const;
  void setNom(string nouveauNom);
  void setPrenom(string nouveauPrenom);
};
//---------------------------------------------------------------------------
inline string Personne::getNom() const          { return nom; }
inline string Personne::getPrenom() const       { return prenom; }
inline void Personne::setNom(string Nom)        { nom = Nom; }
inline void Personne::setPrenom(string Prenom) { prenom = Prenom; }
//---------------------------------------------------------------------------
#endif
```

La déclaration et la définition de la classe « *Elève* » est classique, c'est-à-dire, que l'écriture est comme une classe normale.

Seule, la signature de la classe est particulière, puisqu'il s'agit d'indiquer que la classe « *Elève* » hérite de la classe « *Personne* ».

Pour indiquer une dérivation, il faut utiliser l'opérateur « **:** » (déjà utilisé dans les listes d'initialisation, ici il s'agit d'une liste de dérivation) suivit de nom de la classe de base. Pour un héritage classique, il est nécessaire de faire une dérivation publique.

```cpp
#define EleveH
#include "Personne.h"
#include <vector>
using namespace std;
//---------------------------------------------------------------------------
class Eleve : public Personne
{
  vector<double> notes;
public:
  void nouvelleNote(double note);
  void effacerDerniereNote();
  double lireNote(unsigned int indice);
  unsigned int getNombreNotes();
};
//---------------------------------------------------------------------------
inline void Eleve::nouvelleNote(double note)  { notes.push_back(note); }
inline void Eleve::effacerDerniereNote()      { notes.pop_back(); }
inline unsigned int Eleve::getNombreNotes()   { return notes.size(); }
inline double Eleve::lireNote(unsigned int indice)   { return notes[indice]; }
//---------------------------------------------------------------------------
#endif
```

*La classe Eleve hérite de Personne*

*La définition des méthodes s'effectue de façon classique.*

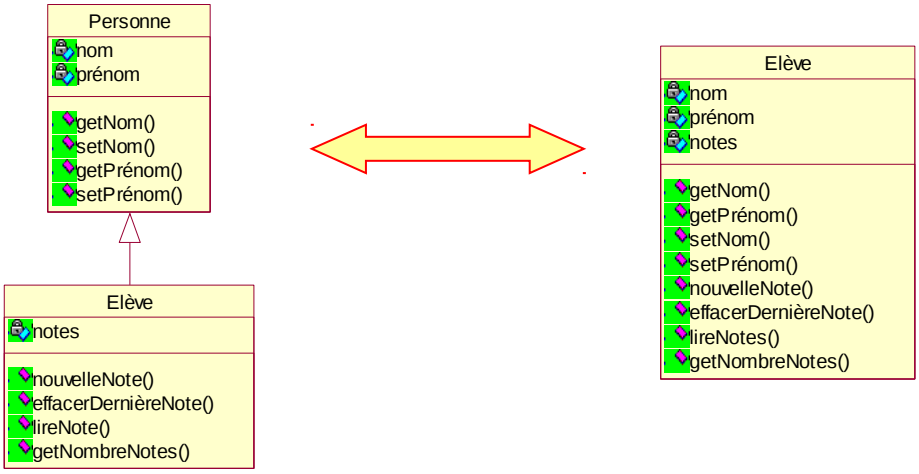
```cpp
  int choix; double note; string nom;
  Eleve eleve;
  do {
    cout << "---------------------------" << endl;
    cout << "Nom et prenom.............1" << endl;
    cout << "Ajouter une note..........2" << endl;
    cout << "Enlever une note..........3" << endl;
    cout << "liste des notes...........4" << endl;
    cout << "Quitter le programme......0" << endl;
    cout << "---------------------------" << endl;
    cout << "Votre choix : "; cin >> choix;
    cout << "---------------------------" << endl;
    switch (choix) {
      case 1 : cout << "Nom de l'élève : ";  cin >> nom;
               eleve.setNom(nom);
               cout << "Prénom de l'élève : ";  cin >> nom;
               eleve.setPrenom(nom); break;
      case 2 : cout << "Nouvelle note : ";  cin >> note;
               eleve.nouvelleNote(note); break;
      case 3 : eleve.effacerDerniereNote(); break;
      case 4 : cout << eleve.getNom() << ' ' << eleve.getPrenom() << endl;
               for (int i=0; i<eleve.getNombreNotes(); i++)
                 cout << eleve.lireNote(i) << ' '; cout << endl; break;
    }
  } while (choix);
```

*Déclaration de l'objet élève*

*L'élève est aussi une Personne, il est alors possible d'utiliser toutes les méthodes issues de l'héritage.*

## Utilisation de tous les membres de la classe dérivée

Lorsqu'une classe dérivée hérite d'une classe de base, elle s'approprie de tout le comportement de la classe de base. La classe dérivée récupère donc l'ensemble des attributs et des méthodes de la classe de base. C'est comme si nous avions une seule classe avec une fusion de tous les attributs et de toutes les méthodes.




## Contrôle des accès

Effectivement, l'enfant récupère tout ce que possède le parent. Mais attention, il existe quand même un petit problème d'accessibilité. Bien que « *Elève* » soit aussi une « *Personne* » avec un nom et un prénom, malgré tout, ces deux attributs ne sont pas accessibles directement puisqu'ils sont déclarés privés dans la classe de base.

***Les statuts des membres d'une classe***

1. ***privé*** : Le membre (généralement l'attribut) n'est accessible qu'aux méthodes de la classe (publiques ou privées). De l'extérieur, il n'est pas possible d'atteindre ce membre. Même sa descendance ne peut pas y accéder directement. Seule l'amitié donne des droits d'accès privilégiés pour atteindre les attributs privés.
2. ***public*** : le membre est cette fois-ci accessible non seulement aux méthodes de la classe et aux fonctions amies, mais également à l'utilisateur de la classe, les enfants compris.

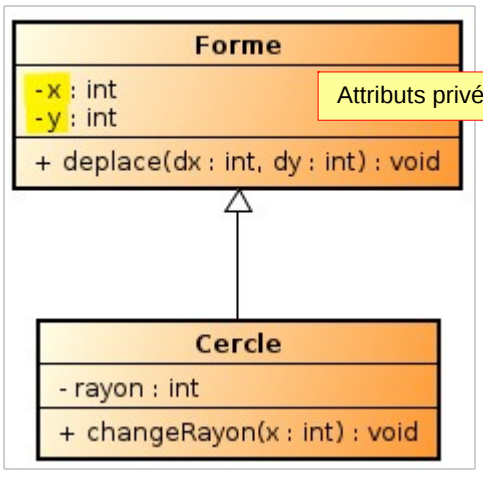
Ainsi, pour en revenir à notre scénario, un élève ne peut pas savoir le nom et le prénom directement. Heureusement, il existe des méthodes de lecture qui ont été mis en œuvre dans la classe de base pour connaître indirectement son propre nom – ***getNom()*** - et son propre prénom – ***getPrenom()***.

Il existe une règle de conception de hiérarchie qui stipule que la classe dérivée ne doit pas avoir besoin de connaître les détails de l'implémentation de la classe de base. Je pense qu'il est bon d'avoir en tête cette démarche. Elle correspond à un principe de sécurité maximale. Pour notre exemple, cela ne pose pas de problème puisqu'il existe des méthodes pour atteindre tous les attributs privés de la classe de base.

La classe de base ne possède pas toujours des méthodes de lecture pour atteindre les attributs. Il serait alors souhaitable d'être moins rigoureux et de permettre uniquement à la descendance (l'enfant ou l'enfant de l'enfant) l'accès à tous les attributs (ou éventuellement une partie) de la classe parente. Il existe un nouveau statut qui propose cette autorisation particulière. Il s'agit du statut ***protected***.

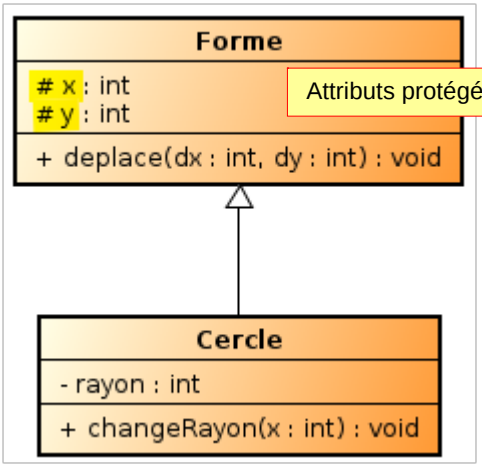
***Les statuts des membres d'une classe***

3. ***protected*** : Les membres protégés se présentent comme des membres privés pour l'utilisateur de la classe de base, mais ils sont comparables à des membres publics pour la classe dérivée et pour toute la descendance. Pour l'utilisateur des classes dérivées, les membres protégés continuent à être considérés comme des membres privés.




```
1  class Forme
2  {
3    int x, y;      Les attributs sont privés
4  public:
5    void deplace(int dx, int dy) { x+=dx; y+=dy; }
6  };
7  //-----------------------------------------------------------------------
8  class Cercle : public Forme
9  {
10   int rayon;
11 public:
12   Cercle(int x, int y, int r) { this->x = x; this->y = y; rayon = r;  }
13   void changeRayon(int r) { rayon = r; }
14 };
15 //-----------------------------------------------------------------------
16 int main()
17 {
18   Cercle c(2, 3, 15);
19   c.deplace(5, 5);     // action autorisée, fait partie de cercle par héritage
20   c.changeRayon(50);   // action autorisée
21   c.x = 12;            // action non autorisée, x est privé.
22   return 0;
23 }
```

actions non autorisées x et y sont privés




```
1  class Forme
2  {
3  protected:
4    int x, y;
5  public:
6    void deplace(int dx, int dy) { x+=dx; y+=dy; }
7  };
8  //-----------------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int x, int y, int r) { this->x = x; this->y = y; rayon = r;  }
14   void changeRayon(int r) { rayon = r; }
15 };
16 //-----------------------------------------------------------------
17 int main()
18 {
19   Cercle c(2, 3, 15);
20   c.deplace(5, 5);     // action autorisée, fait partie de cercle par héritage
21   c.changeRayon(50);   // action autorisée
22   c.x = 12;            // action non autorisée, x est protégé,
23   return 0;            // donc non accessible de l'extérieur
24 }
```

Les attributs sont protégés ils seront donc accessibles directement par les enfants

actions autorisées x et y sont protégés

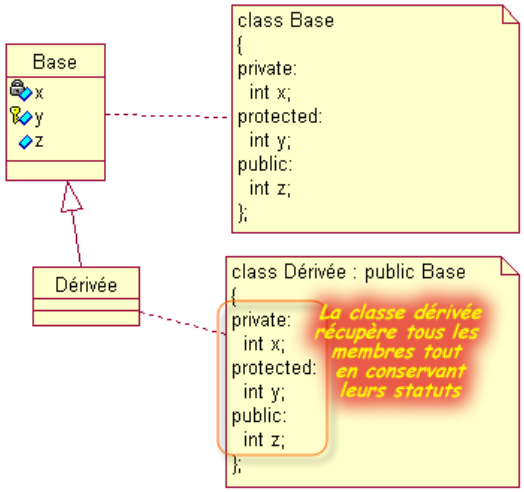
Nous pensons souvent que pour avoir le maximum de souplesse, il suffit de déclarer protégés tous les attributs de la classe de base. Ainsi les enfants dans toute la hiérarchie peuvent y accéder sans aucune restriction. Toutefois cela peut être dangereux puisque un développeur par héritage peut modifier le comportement prévu par la classe de base.

Il ne faut pas que les attributs de la classe de base soient systématiquement protégés. Je pense que par réflexe, il faut d'abord les considérer comme privés, parce qu'il n'est pas toujours nécessaire que les enfants puissent accéder directement à tout ce que font les parents.

## Dérivation publique, protégée et privée

### Dérivation publique :

Les membres hérités d'une classe de base publique conservent leurs niveaux d'accès à l'intérieur de la classe dérivée. C'est le comportement normal prévu, par exemple, lors de la conception d'une application en UML. En général dans une hiérarchie de dérivation publique, chaque classe ultérieurement dérivée a accès à la série combinée des membres protégés et publics des classes de base précédentes le long de cette branche particulière de la hiérarchie.

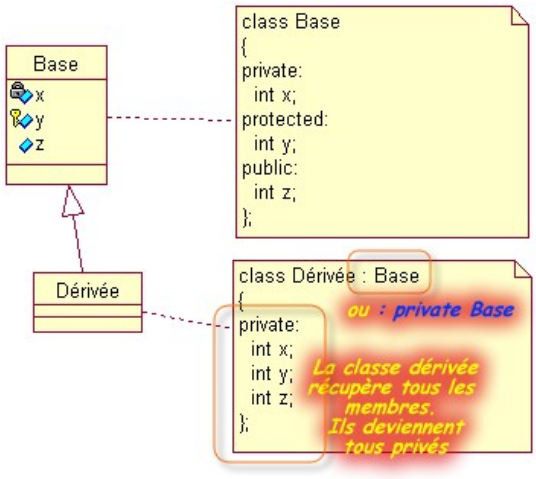



Rappelons :

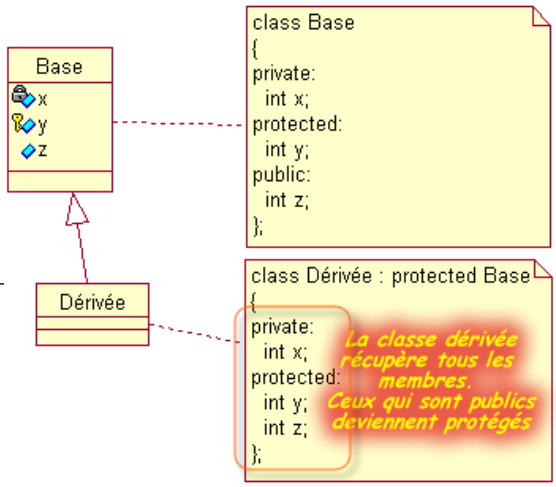
1. Les membres *publics* de la classe de base sont accessibles « à tout le monde », c'est-à-dire à la fois aux méthodes, aux fonctions amies de la classe dérivée ainsi qu'aux utilisateurs de la classe dérivée.
2. les membres *protégés* de la classe de base sont accessibles aux méthodes et aux fonctions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

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 utiliser, 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 grand parent.



**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.




## Constructions

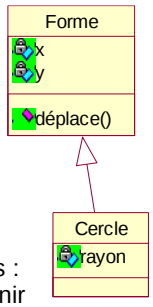
Pour une classe, à chaque fois que nous fabriquons un nouvel objet, le souci porte sur sa phase de création. En effet, il est toujours important que l'état de l'objet soit prédéterminé afin que la suite de ses comportements respecte au mieux les désirs du programmeur. Lorsque nous parlons de l'état de l'objet, il s'agit en fait de la valeur de chacun des attributs. C'est le (ou les) constructeur qui gère ce genre de problème au moment de la phase de création.

Vous pouvez bien entendu ne pas utiliser de constructeurs, c'est alors la performance en terme de rapidité qui est privilégiée. Dans ce cas précis, la valeur des attributs est alors totalement aléatoire ce qui correspond également à un état aléatoire de l'objet. Il est toutefois souvent préférable qu'un objet soit dans un état bien précis dès sa naissance.

Dans le mécanisme d'héritage, les enfants devront également s'occuper de cette phase de création. Ils héritent de leur parent, et récupèrent donc tout leur comportement, phase de création comprise.

L'enfant doit uniquement se préoccuper de ce qui fait sa spécificité par rapport au parent. Le parent intègre déjà toute la partie générale. Pour la phase de construction c'est la même chose, l'enfant doit gérer ses propres attributs alors que le parent s'est déjà occupé des siens. Les constructeurs du parent ont déjà été mis au point (la classe parente est normalement autonome et elle est capable de se créer toute seule).

Nous voyons là l'intérêt de l'héritage. Nous ne nous occupons que d'une petite partie à la fois. Par contre, si le parent possède plusieurs constructeurs, ce qui suppose qu'il possède plusieurs possibilités de création, il faudra que l'enfant en tienne compte ou du moins choisisse le constructeur qui convient le mieux à sa propre création.




Nous retrouvons là la même problématique que nous avons déjà rencontré lors de l'étude de la composition, c'est-à-dire, lorsqu'un objet est intégré dans une classe. Nous avons bien souligné que cet objet doit être construit pour pouvoir l'utiliser de façon correcte ultérieurement. Par chance, nous utilisons la même syntaxe, c'est-à-dire les listes d'initialisation.

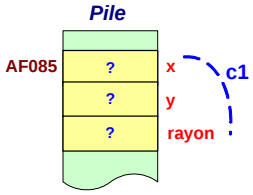
**La classe de base et la classe dérivée possèdent un constructeur par défaut :**

```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x=0, int y=0)  { this->x = x; this->y = y; }
6    void deplace(int dx, int dy);
7  };
8  //-----------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int rayon=0) { this->rayon = rayon;  }
14 };
15 //-----------------------------------------------------
16 int main()
17 {
18   Cercle c1;
19   return 0;
20 }
```

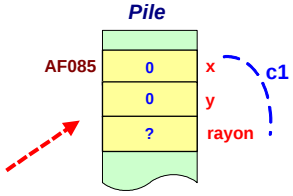
Appel du constructeur relatif à la classe Forme
Appel du constructeur relatif à la classe Cercle
Un nouvel objet demande à être créé

La création d'un objet se déroule en quatre phases :

1. *Allocation mémoire* nécessaire pour contenir l'ensemble des attributs que comporte l'objet. Puisqu'il s'agit d'un héritage, l'objet alloue l'espace mémoire pour ses propres attributs ainsi que l'espace mémoire nécessaire aux attributs délivrés par l'héritage. Sinon l'objet ne serait pas complet.

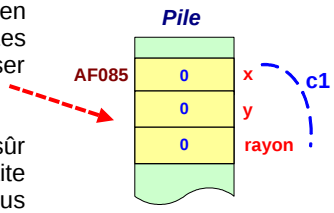



2. *Appel du constructeur de la classe dérivée*. Ce constructeur est appelé mais pas encore exécuté. En effet, le constructeur de la classe dérivée ne s'occupe uniquement de ce qui fait la spécificité de la classe, c'est-à-dire, initialiser ses propres attributs. Avant d'effectuer cela, il faut être sûr que toute la structure générale soit elle-même bien initialisée. C'est la classe de base qui s'occupe justement de la structure de base et qui gère l'initialisation de ces propres attributs. Donc, avant l'exécution du constructeur de la classe dérivée, c'est le constructeur de la classe de base qui est appelé.




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. Au minimum, ces instructions consistent à donner une valeur correcte aux attributs afin que l'objet par la suite n'ait pas de comportement aléatoire. (Si cette classe de base est également une classe dérivée, le système appelle d'abord le constructeur de sa classe de base avant l'exécution du constructeur).

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. Il s'agit également d'initialiser les attributs relatifs à la classe dérivée.




La création d'un objet passe systématiquement par ces quatre phases. Ainsi, nous sommes sûr que l'objet est correctement initialisé. Chaque phase joue son rôle et s'occupe que d'une petite partie, ce qui rend la lecture plus facile. La maintenance s'en trouvera également d'autant plus simplifiée.

Lorsque vous ne disposez d'aucun constructeur, ces quatre phases sont quand même accomplies. Effectivement, les constructeurs par défaut existent. Par contre, ils ne disposent d'aucunes instructions par défaut. En fait, ils ne font rien. Du coup, la valeur de chacun des attributs est généralement aléatoire, sauf dans le cas d'un objet déclaré en tant que variable globale.

### La classe de base dispose d'un constructeur par défaut et la classe dérivée d'un constructeur avec un paramètre :

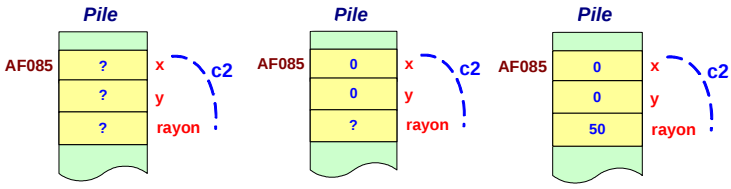
```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x=0, int y=0)   { this->x = x; this->y = y; }
6    void deplace(int dx, int dy);
7  };
8  //---------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int rayon) { this->rayon = rayon;  }
14 };
15 //---------------------------------------------------------
16 int main()
17 {
18   Cercle c2(50);
19   return 0;
20 }
```

*Appel du constructeur relatif à la classe Forme*

*Appel du constructeur relatif à la classe Cercle*

*Un nouvel objet demande à être créé*

Par rapport au scénario précédent, rien ne change vraiment, les quatre phases sont appelées dans le même ordre.




### La classe de base dispose d'un constructeur avec paramètres :

Lorsque nous disposons d'un constructeur par défaut, l'appel se fait implicitement, c'est-à-dire automatiquement. Lorsque nous avons un constructeur avec arguments, cette fois-ci, il est nécessaire de faire un appel explicite afin d'envoyer les bons arguments au constructeur pour que l'initialisation des attributs correspondent à l'objet désiré.

```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x, int y)   { this->x = x; this->y = y; }
6    void deplace(int dx, int dy);
7  };
8  //---------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int x, int y, int rayon) : Forme(x, y) { this->rayon = rayon;  }
14 };
15 //---------------------------------------------------------
16 int main()
17 {
18   Cercle c3(12, -5, 50);
19   return 0;
20 }
```

*Appel et exécution du constructeur de la classe de base*

*Exécution du corps du constructeur de la classe dérivée*

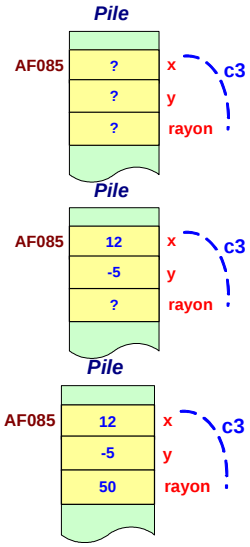
*Appel du constructeur de la classe dérivée*

*Liste d'initialisation*

*Un nouvel objet est créé avec passage de trois arguments*

Du coup, pour la classe dérivée, il est nécessaire de disposer au moins d'un constructeur qui fasse un appel explicite au constructeur de la classe de base en propageant les bons arguments nécessaires aux attributs généraux relatifs à la classe de base. Nous avons déjà rencontré ce genre de situation. La solution consiste à utiliser une liste d'initialisation, dont la syntaxe d'ailleurs se rapproche de l'écriture de l'héritage, notamment avec l'utilisation de l'opérateur « **:** ».

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

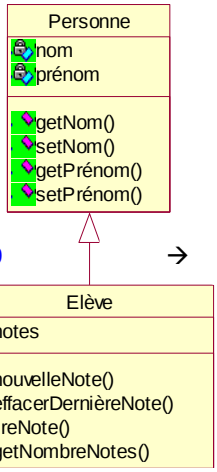



1. *Allocation mémoire* nécessaire pour contenir tous les attributs que comporte l'objet.
2. *Appel du constructeur de la classe dérivée*. Ce constructeur est appelé mais pas encore exécuté. Appel du constructeur de la classe de base spécifié par la liste d'initialisation en récupérant les bons arguments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

Pour le concepteur de la classe dérivée, il n'est pas nécessaire de se pencher de nouveau sur les attributs de la classe de base, il devient un simple utilisateur. Il se préoccupe seulement des attributs de la classe dérivée. D'ailleurs, si vous remarquez bien, il est impossible d'atteindre directement les attributs de la classe de base puisqu'ils sont privés.

## Compatibilité entre classe de base et classe dérivée

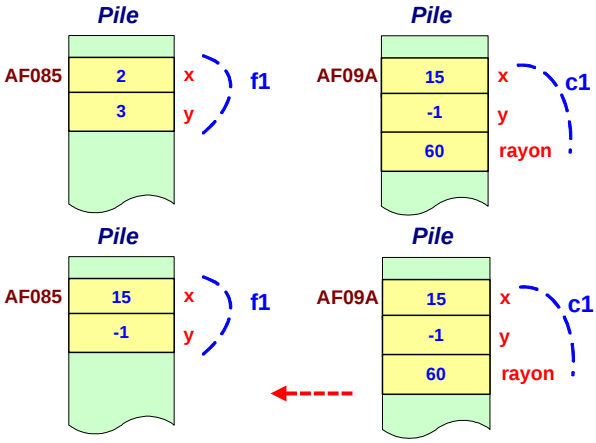
Nous pouvons toujours dire qu'un cercle est aussi une forme et qu'un élève est aussi une personne. La réciproque n'est pas vraie. Selon le besoin, nous savons établir des conversions implicites qui permettent de passer d'un type vers un autre. Dans le cadre de l'héritage, il sera possible de passer d'une classe dérivée vers une classe de base. Par contre l'inverse ne sera pas possible.



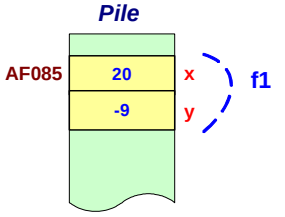

```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x, int y)              { this->x = x; this->y = y; }
6    void deplace(int dx, int dy)     { x+=dx; y+=dy; }
7  };
8  //-----------------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int x, int y, int rayon) : Forme(x, y) { this->rayon = rayon;  }
14   void agrandir(int dr) { rayon+=dr; }
15 };
```

```
17 int main()
18 {
19   Forme f1(2, 3);
20   Cercle c1(12, -5, 50);
21   c1.deplace(3, 4);
22   c1.agrandir(10);
23   f1 = c1;
24   f1.deplace(5, -8);
25   f1.agrandir(10);
26   Cercle c2(f1);
27   return 0;
28 }
```

Création de l'objet « *f1* ».
20 → Création de l'objet « *c1* ».
21 → Possibilité de déplacer le cercle « *c1* » puisque la méthode a été héritée.
22 → Agrandissement du cercle « *c1* » grâce à la méthode « *agrandir* » définie par la classe « *Cercle* ».

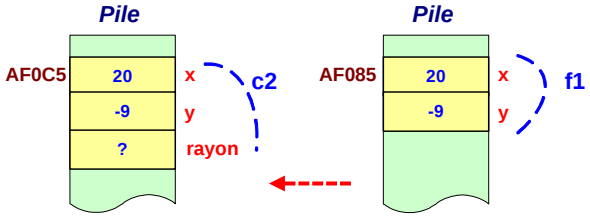



23 → Â droite et à gauche de l'opérateur d'affectation, les types sont différents. C'est toujours le type qui est à droite qui est transformé vers le type de gauche. Ici, le cercle « *c1* » est aussi une forme, donc la conversion implicite est lancée. Cette démarche paraît normale puisqu'à l'issue de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




24 → 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* ».
25 → Par contre, à partir d'une forme, il est impossible de demander un agrandissement puisque cette démarche n'existe que d'après la classe « *Cercle* ». Une erreur de compilation est alors déclenchée.

26 → Tentative de création d'un cercle à partir d'une forme. 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. Par ailleurs, dans le chapitre précédent, nous avons découvert qu'au moment de la création d'un objet d'une classe dérivée, le constructeur de cette classe dérivée faisait appel à un constructeur de la classe de base. En aucun moment nous avons l'inverse, c'est-à-dire un constructeur de la classe de base qui fait appel à un constructeur de la classe dérivée. Cela n'aurait aucun sens.




## Comportements par défaut et héritage

**Constructeur de copie – écriture implicite :**
Par défaut, le constructeur de copie propose une copie membre à membre. Les attributs de l'objet à créer sont initialisés par rapport aux attributs de l'objet (qui sert de copie) passé en argument. Le même comportement par défaut reste vrai pour un objet de la classe dérivée.
Un constructeur de classe de base est toujours invoqué avant l'exécution du constructeur de la classe dérivée. Le constructeur de copie est également un constructeur. Il ne fait donc pas exception à cette règle, ce qui est logique. Nous retrouvons donc les quatre mêmes phases pour la création d'un objet par un autre.

```
1  class Forme
2  {
3    int x, y;
4  public:
5    Forme(int x, int y)             { this->x = x; this->y = y; }
6    void deplace(int dx, int dy)    { x+=dx; y+=dy; }
7  };
8  //-----------------------------------------------------------
9  class Cercle : public Forme
10 {
11   int rayon;
12 public:
13   Cercle(int x, int y, int rayon) : Forme(x, y) { this->rayon = rayon;  }
14   void agrandir(int dr) { rayon+=dr; }
15 };
16 //-----------------------------------------------------------
17 int main()
18 {
19   Cercle c1(12, -5, 50);
20   Cercle c2 = c1;
21   return 0;
22 }
```

Création du cercle c2 par rapport au cercle c1

Appel du constructeur de copie

Lorsque nous écrivons les lignes de code ci-contre, avec la création d'un objet cercle par rapport à un autre objet cercle, le constructeur de copie par défaut de la classe « *Cercle* » est sollicité. Rien n'a été spécialement écrit explicitement dans aucune des deux classes, mais cela correspond à ce qui vous est montré ci-dessous.

```
class Forme
{
  int x, y;
public:
  Forme(int x, int y)             { this->x = x; this->y = y; }
  Forme(const Forme &forme)       { x = forme.x; y = forme.y; }
  void deplace(int dx, int dy)    { x+=dx; y+=dy; }
};
//-----------------------------------------------------------
class Cercle : public Forme
{
  int rayon;
public:
  Cercle(int x, int y, int rayon) : Forme(x, y) { this->rayon = rayon;  }
  Cercle(const Cercle &cercle) : Forme(cercle) { rayon = cercle.rayon; }
  void agrandir(int dr) { rayon+=dr; }
};
//-----------------------------------------------------------
int main()
{
  Cercle c1(12, -5, 50);
  Cercle c2 = c1;
  return 0;
}
```

Constructeur de copie par défaut implicite

Constructeur de copie par défaut implicite

Comme je viens de le rappeler, un constructeur de copie reste un constructeur. De ce fait, lorsque le constructeur de copie de la classe dérivée fait référence au constructeur de la classe de base, il passe par la liste d'initialisation.

Par ailleurs, quand le constructeur de copie de la classe dérivée est sollicité, il doit faire référence également au constructeur de copie de la classe de base. Effectivement, le constructeur de copie de la classe dérivée doit s'occuper d'exécuter la copie de chacun de ces membres, alors que la classe de base s'occupe de copier ses propres membres.

Dans ces conditions, on voit que le constructeur de copie de la classe de base « *Forme* » doit recevoir en argument, non pas l'objet « *cercle* » tout entier, mais seulement ce qui, dans « *cercle* » représente la « *Forme* ». C'est là qu'intervient la possibilité de conversion implicite d'une classe dérivée vers une classe de base comme nous l'avons découvert dans le chapitre précédent. Nous pouvons donc passé directement l'objet « *cercle* » au constructeur de copie de la classe de base « *Forme* » prévu dans la liste d'initialisation.

### Constructeur de copie – écriture explicite :

Vous savez que le comportement par défaut n'est pas toujours souhaitable, notamment lorsque nous disposons de variables dynamiques au sein même de la classe. L'héritage n'exclut pas cette problématique, il faut alors gérer la situation. En fait trois cas peuvent se présenter :

1. *La classe de base possède au moins une variable dynamique, mais pas la classe dérivée* : Dans ce cas, il faut uniquement redéfinir le constructeur de copie de la classe de base. Lorsque nous tenterons de créer un objet de la classe dérivée par copie, l'appel du constructeur de copie de la classe de base se fera implicitement sans aucun problème particulier.
2. *La classe de base et la classe dérivée possèdent toutes les deux des variables dynamiques* : Dans ce cas, il faut redéfinir, bien entendu, les deux constructeurs de copie. *Mais attention, lors de la définition du constructeur de copie de la classe dérivée, vous devez impérativement faire un appel explicite au constructeur de copie de la classe de base grâce à la liste d'initialisation. L'appel implicite au constructeur de copie de la classe de base ne marche pas dès que vous redéfinissez le constructeur de copie de la classe dérivée*.
3. *La classe dérivée possède une variable dynamique, mais pas la classe de base* : *Vous êtes donc obligé de redéfinir le constructeur de copie de la classe dérivée, ce qui implique que là aussi, vous devez faire un appel explicite au constructeur de copie par défaut à l'aide de la liste d'initialisation.*

Pour illustrer ces propos, nous allons nous servir des classe « *Personne* » et « *Elève* ». Pour la classe « *Elève* », nous allons gérer l'ensemble des notes au travers d'un tableau classique. Puisque le nombre de notes est variable, nous sommes obligé de prendre un pointeur. Du coup, nous avons effectivement une variable dynamique (si nous avions pris une classe « *vector* » pour gérer les notes, nous n'aurions pas ce genre de problème). Cela peut paraître dommage de faire un appel explicite au constructeur de copie de la classe de base lorsque nous redéfinissons le constructeur de copie de la classe dérivée. Ce que nous pouvons dire, c'est que, d'une part, c'est très vite écrit, d'autre part, nous avons la possibilité de choisir un autre constructeur que le constructeur de copie.

```
4  class Personne
5  {
6    string nom, prenom;
7  public:
8    Personne(string n, string p) : nom(n), prenom(p) { }
9  };
10 //-----------------------------------------------------------
11 class Eleve : public Personne
12 {
13   unsigned nombreNotes;
14   double *notes;
15 public:
16   Eleve(string n, string p, unsigned nombre=10) : Personne(n, p)
17   { notes = new double[nombreNotes = nombre]; }
18   Eleve(const Eleve &eleve) : Personne(eleve) { /* ... */ }
19 };
20 //-----------------------------------------------------------
21 int main( )
22 {
23   Eleve eleve1("Lagafe", "Gaston", 15);
24   Eleve eleve2 = eleve1;
25   return 0;
26 }
```

Le constructeur de copie par défaut va être utilisé pour la classe de base (non visisble puisque c'est celui par défaut qui est pris)

Appel explicite obligatoire du constructeur de copie de la classe de base

Construction par copie

En effet, puisque nous passons par une liste d'initialisation, nous avons la liberté de décider du constructeur à prendre. Ainsi, par rapport à l'exemple précédent, nous aurions pu aussi écrire ce qui vous est présenté dans le codage ci-contre.

### Opérateur d'affectation :

Pour l'opérateur d'affectation, nous devons avoir le même type de raisonnement que pour le constructeur de copie.

Lorsque nous utilisons l'affectation par défaut, une copie membre à membre est réalisée. Puisque que la classe dérivée hérite de tous les attributs de la classe de base, lorsque nous effectuons une affectation entre deux objets de la classe dérivée, l'ensemble des attributs est bien copié d'un objet vers l'autre, avec d'abord, la copie des attributs relatifs à la classe de base.

Si nous avons à gérer des variables dynamiques au sein d'une classe ou des deux classes, nous devons procéder exactement de la même façon que pour le constructeur de copie. Toutefois, il existe une différence majeure lorsque vous devez redéfinir l'opérateur d'affectation de la classe dérivée. Un constructeur dispose d'une liste d'initialisation, mais pas l'opérateur d'affectation. Du coup, le mécanisme de transfert d'argument qui permettait l'appel vers la bonne méthode ne peut-être utilisé. Vous devez donc écrire explicitement, à l'intérieur de la méthode définissant l'opérateur d'affectation, tout ce qui concerne l'affectation de l'objet dérivée, y compris les attributs de la classe de base. Dans ce cas de figure, il est souhaitable que les attributs de la classe de base aient un statut protégé.

### Destructeur :

Les différentes phases qui constituent la destruction de l'objet s'effectuent dans l'ordre inverse de la construction, c'est-à-dire :

1. *Appel du destructeur de la classe dérivée.*
2. *Exécution du destructeur de la classe dérivée.*
3. *Appel et exécution du destructeur de la classe de base.*
4. *Libération de la mémoire utilisée par l'objet.*

Ces quatre phases existent que le ou les destructeurs soient redéfinis ou pas. Les destructeurs sont à redéfinir dans le cas où les classes disposent de variables dynamiques. Dans ce cas là, les appels se feront implicitement sans soucis particuliers.

```
4  class Personne
5  {
6  protected:
7    string nom, prenom;
8  public:
9    Personne(string n, string p) : nom(n), prenom(p) { }
10 };
11 //-----------------------------------------------------------------
12 class Eleve : public Personne
13 {
14   unsigned nombreNotes;
15   double *notes;
16 public:
17   Eleve(string n, string p, unsigned nombre=10) : Personne(n, p)
18   { notes = new double[nombreNotes = nombre]; }
19   Eleve(const Eleve &eleve) : Personne(eleve.nom, eleve.prenom) { /* ... */ }
20 };
21 //-----------------------------------------------------------------
22 int main( )
23 {
24   Eleve eleve1("Lagafe", "Gaston", 15);
25   Eleve eleve2 = eleve1;
26   return 0;
27 }
```

*Attention, il est nécessaire de prendre le statut protégé pour que nom et prenom soient accessibles par la classe dérivée*

*Cette fois-ci, c'est le constructeur normal qui est demandé*

*Construction par copie*

```
4  class Personne
5  {
6  protected:
7    string nom, prenom;
8  public:
9    Personne(string n, string p) : nom(n), prenom(p) { }
10 };
11 //-----------------------------------------------------------------
12 class Eleve : public Personne
13 {
14   unsigned nombreNotes;
15   double *notes;
16 public:
17   Eleve(string n, string p, unsigned nombre=10) : Personne(n, p)
18   { notes = new double[nombreNotes = nombre]; }
19   Eleve(const Eleve &eleve) : Personne(eleve.nom, eleve.prenom) { /* ... */ }
20   Eleve& operator=(const Eleve &eleve) {
21     nom = eleve.nom;
22     prenom = eleve.prenom;
23     // codage relatif à la gestion de nombreNotes et notes
24 };
25 //-----------------------------------------------------------------
26 int main( )
27 {
28   Eleve eleve1("Lagafe", "Gaston", 15);
29   Eleve eleve2 = eleve1;
30   eleve1 = eleve2;
31   return 0;
32 }
```

*Nécessaire pour permettre à la classe dérivée d'accéder aux attributs de la classe de base*

*Affectation des attributs de la classe de base*

*Affectation*

## Redéfinition des méthodes

***ATTENTION***

Il ne faut pas mélanger la redéfinition et la sur-définition.

⇒ Une ***sur-définition*** (ou surcharge) permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe, avec une signature différente, pour que le système puisse s'y retrouver.

⇒ Une ***redéfinition*** permet de fournir une nouvelle définition d'une méthode d'une classe ascendante et ainsi de substituer la description qui en été faite. Nous avons également le même nom que la méthode parente mais surtout avec une signature rigoureusement identique. La redéfinition constitue la base du polymorphisme.

***Finalement, dans les exemples précédents, nous avons surtout utilisé la sur-définition puisque nous avions plusieurs constructeurs pour une même classe. Dans ce chapitre, nous allons nous préoccuper de la redéfinition en sachant, par ailleurs, que la redéfinition et la sur-définition peuvent coexister sans problème.***

Reprenons l'exemple de la classe « *Elève* » issue d'une classe « *Personne* ». Nous pouvons avoir besoin, pour notre application, d'afficher la description, soit de la personne, soit de l'élève. Nous disposons donc d'une méthode « *affiche* » sur chacune de ces deux classes (voir page suivante). Remarquez que nous avons choisi le même nom, il s'agit donc bien d'une

redéfinition. Il est effectivement nécessaire de redéfinir la méthode de la classe de base puisque cette dernière affiche uniquement l'identité de la personne, alors que la méthode « *affiche* » de la classe dérivée doit proposer en plus l'affichage de l'ensemble des notes.

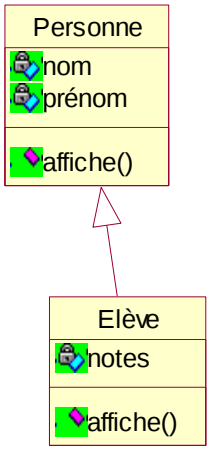
Ceci dit, la méthode « *affiche* » de la classe « *Elève* » doit également proposer l'affichage de l'identité de l'élève. Dans la méthode « *affiche* » de la classe « *Elève* », nous pouvons donc réécrire ce qui a déjà été écrit dans la méthode « *affiche* » de la classe « *Personne* ». Le problème, c'est qu'il faut savoir ce qui est déjà écrit, ce qui n'est pas toujours évident et surtout, c'est une perte de temps. Le mieux c'est de faire appel à la méthode « *affiche* » de la classe « *Personne* ». Par contre, il faut bien préciser qu'il s'agit effectivement de la méthode « *affiche* » de la classe « *Personne* », sinon nous aurons un appel récursif. Rappelez-vous que lorsque nous devons spécifier une méthode par rapport à une classe, c'est l'opérateur de portée « *::* » qui sert de qualificateur.

```
4  class Personne
5  {
6  protected:
7    string nom, prenom;
8  public:
9    Personne(string n, string p) : nom(n), prenom(p) { }
10   void affiche() {
11     cout << "Nom : " << nom << endl;
12     cout << "Prénom : " << prenom  << endl;
13   }
14 };
15 //-----------------------------------------------------------------
16 class Eleve : public Personne
17 {
18   vector<double> notes;
19 public:
20   Eleve(string n, string p) : Personne(n, p) { }
21   Eleve& ajoutNote(double note) { notes.push_back(note); return *this; }
22   void affiche() {
23     Personne::affiche();
24     for (unsigned i=0; i<notes.size(); i++) cout << notes[i] << ' ';
25     cout << endl;
26   }
27 };
28 //-----------------------------------------------------------------
29 int main( )
30 {
31   Eleve eleve("Lagafe", "Gaston");
32   eleve.ajoutNote(15).ajoutNote(8).ajout(12);
33   eleve.affiche();
34   eleve.Personne::affiche(); // affiche que l'identité de l'élève
35   return 0;
36 }
```

*Appel de la méthode affiche de la classe Personne*

*Appel de la méthode affiche de la classe Eleve*

*Appel explicite de la méthode affiche de la classe parente Personne*




```
Nom : Lagafe
Prénom : Gaston
15 8 12
Nom : Lagafe
Prénom : Gaston
```

Même si c'est très rarement utilisé, vous pouvez faire un appel explicite à une méthode d'une classe ascendante.

## C++11 - Redéfinition et sur-définition

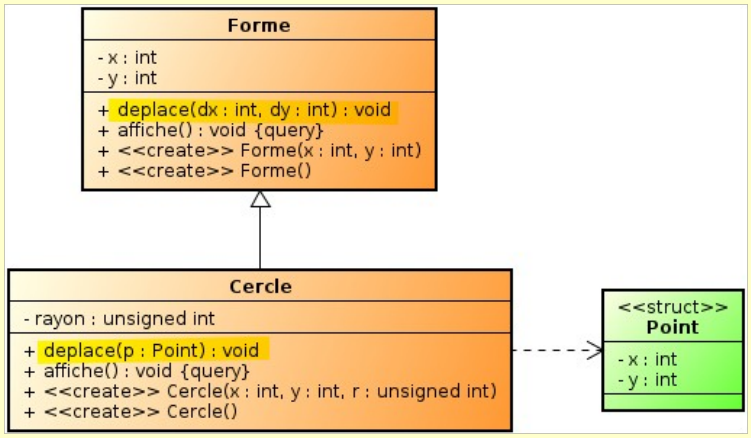
Nous venons de le voir, lorsqu'une méthode est redéfinie dans une classe dérivée, elle masque la méthode de même signature que la classe de base, mais pas seulement. En effet, par défaut, la méthode définie dans la classe dérivée masque toutes les méthodes qui portent le même nom dans la classe de base quelque soit leurs signatures (peu importe leurs arguments et leur valeur de retour). Par défaut, à priori, il n'est donc pas permis de sur-définir des méthodes sur des portées différentes.

***Autrement dit, la recherche d'une méthode (sur-définie ou non) se fait systématiquement dans une seule portée, soit celle de la classe concernée (dérivée par exemple), soit celle de la classe de base, mais jamais dans plusieurs classes à la fois (plusieurs niveaux hiérarchiques).***

**Exemple de sur-définition sur des portées différentes**

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

struct Point
{
  int x, y;
};

class Forme
{
  int x, y;
```

```
public:
  Forme(int x, int y) : x(x), y(y) {}
  Forme() : Forme(0, 0) {}
  void deplace(int dx, int dy) { x+=dx; y+=dy; }
  void affiche() const
  {
    cout << "(x=" << x << ", y=" << y << ")";
  }
};

class Cercle : public Forme
{
  unsigned rayon = 50;
public:
   Cercle(int x, int y, unsigned r) : Forme(x, y),
rayon(r) {}
  Cercle() = default;
  // sur-définition
  void deplace(Point p) { Forme::deplace(p.x, p.y); }
  void affiche() const  // redéfinition
  {
    Forme::affiche();
    cout << "\b, rayon=" << rayon << ")" << endl;
  }
};

int main()
{
  Cercle c1, c2(20, 30, 100);
  c2.deplace(5, 8);
  c1.affiche();
  return 0;
}
```




**Erreur de compilation : Cette méthode deplace() de la classe Forme n'est pas accessible puisque l'objet c2 fait partie de la classe Cercle qui elle-même propose une méthode du même nom avec une signature différente (sur-définition sur des portées de niveau différent).**

***Heureusement, il est possible d'imposer que la recherche d'une méthode sur-définie s'étende sur plusieurs niveaux hiérarchiques en utilisant la directive « using ».***

## Utilisation de la directive using afin de permettre l'utilisation de la méthode deplace() de la classe Forme

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

struct Point
{
  int x, y;
};

class Forme
{
  int x, y;
public:
  Forme(int x, int y) : x(x), y(y) {}
  Forme() : Forme(0, 0) {}
  void deplace(int dx, int dy) { x+=dx; y+=dy; }
  void affiche() const { cout << "(x=" << x << ", y=" << y << ")"; }
};

class Cercle : public Forme
{
  unsigned rayon = 50;
public:
  Cercle(int x, int y, unsigned r) : Forme(x, y), rayon(r) {}
  Cercle() = default;
  using Forme::deplace;
  void deplace(Point p) { Forme::deplace(p.x, p.y); }
  void affiche() const
  {
    Forme::affiche();
    cout << "\b, rayon=" << rayon << ")" << endl;
  }
};

int main()
{
  Cercle c1, c2(20, 30, 100);
  c2.deplace(5, 8);
  c1.affiche();
  c2.affiche();
  return 0;
}
```

```
(x=0, y=0, rayon=50)
(x=25, y=38, rayon=100)
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Dans les faits, nous rencontrerons très rarement ce genre de situation. Ici par exemple, ce n'est pas cohérent de proposer une méthode ***deplace()*** dans la classe ***Cercle***. Cette notion de déplacement doit être associée systématiquement à la classe de base, la classe ***Forme***, puisque c'est elle qui dispose des coordonnées. Surtout, n'importe quelle forme doit pouvoir se déplacer. Ainsi, si vous désirez créer une nouvelle classe dérivée, ***Carré*** par exemple, cette nouvelle classe pourra ainsi bénéficier des deux types de déplacement.

***En résumé, il est généralement préférable, sauf situation particulière, de sur-définir vos méthodes dans une même classe (même portée), si possible dans la classe de base, pour éviter que les méthodes soient masquées par défaut les unes par rapport aux autres, et surtout pour permettre à toutes les classe filles de les utiliser naturellement sans contrainte particulière.***

```
class Forme
{
  int x, y;
public:
  Forme(int x, int y) : x(x), y(y) {}
  Forme() : Forme(0, 0) {}
  void deplace(int dx, int dy) { x+=dx; y+=dy; }
  void deplace(Point p) { x=p.x, y=p.y); }
  void affiche() const { cout << "(x=" << x << ", y=" << y << ")"; }
};

class Cercle : public Forme
{
  unsigned rayon = 50;
public:
  Cercle(int x, int y, unsigned r) : Forme(x, y), rayon(r) {}
  Cercle() = default;
  void affiche() const
  {
    Forme::affiche();
    cout << "\b, rayon=" << rayon << ")" << endl;
  }
};
```

**Sur-définition des méthodes** : Lorsque les méthodes sont sur-définies dans une même portée, elles deviennent accessibles, sans contrainte, à toutes les classes filles.

## C++11 - Héritage des constructeurs

Comme nous venons de le démontrer, les règles relatives à la sur-définition des méthodes (constructeurs compris) font que, par défaut, une méthode redéfinie dans une classe dérivée masque les méthodes de même nom des classes ascendantes.

Nous aimerions souvent pouvoir initialiser une classe dérivée avec le même ensemble de constructeurs que la classe de base. Jusqu'à présent, la seule manière de faire consistait à redéclarer tous les constructeurs en les redirigeant sur ceux de la classe de base (liste d'initialisation).

***Le compilateur peut maintenant faire le travail ; il en résulte moins d'erreurs et l'intention est rendue bien visible, grâce à la norme C++11 qui étend la signification de la directive « using » en ce sens. Il acquiert ainsi une nouvelle signification, puisqu'il permet d'hériter tous les constructeurs d'une classe mère donnée, évitant d'avoir à les coder de nouveau.***

**Utilisation de la directive using afin d'hériter de tous les constructeurs de la classe Forme**

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

struct Point
{
  int x, y;
};

class Forme
{
protected:
  int x, y;
public:
  Forme(int x, int y) : x(x), y(y) {}
  Forme() : Forme(0, 0) {}
  void deplace(int dx, int dy) { x+=dx; y+=dy; }
  void deplace(Point p) { x=p.x, y=p.y); }
  void affiche() const { cout << "(x=" << x << ", y=" << y << ")"; }
};

class Cercle : public Forme
{
  unsigned rayon = 50;
  using Forme::Forme;
public:
  Cercle(int x, int y, unsigned r) : Forme(x, y), rayon(r) {}
  Cercle() = default;
  Cercle(const Cercle&) = delete;

  void affiche() const
  {
    Forme::affiche();
    cout << "\b, rayon=" << rayon << ")" << endl;
  }
};

int main()
{
  Cercle c1, c2(15, 20), c3 = {14, 18}, c4(20, 30, 100);
  // Cercle c5=c2; // interdit (delete)
  c3.deplace(5, 8); // ou c3.deplace({5, 8}) ;
  c1.affiche();
  c2.affiche();
  c3.affiche();
  c4.affiche();
  return 0;
}
```

**Héritage des constructeurs** : Grâce à cette directive, nous pouvons utiliser les deux constructeurs définis dans la classe de base, ce qui est particulièrement utile pour la création des objets c1, c2 et c3.

**Constructeur par défaut** : Le mot réservé « default » permet de préciser que nous utilisons le comportement par défaut. Ici, l'attribut rayon sera initialiser à 50 et les attributs x et y seront initialisés automatiquement par le constructeur par défaut de la classe Forme par héritage.

**Blocage de la construction par copie** : Le mot réservé « delete » permet d'empêcher d'utiliser une méthode particulière, comme c'est les cas ici avec le constructeur de copie.

```
(x=0, y=0, rayon=50)
(x=15, y=20, rayon=50)
(x=19, y=26, rayon=50)
(x=20, y=30, rayon=100)
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```