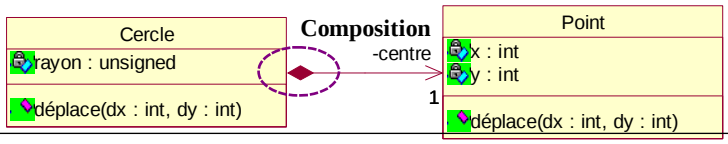
Un objet peut être constitué d'attributs de n'importe quel type. Du coup, il est possible qu'il soit lui même composé d'autres objets. C'est ce que nous appelons la composition ou l'agrégation par valeur. L'étude qui suit va nous permettre de maîtriser la création complète de l'objet conteneur associé à ses objets membres. Pour aborder ces différents thèmes, nous allons nous servir de classes simplifiées, afin de comprendre rapidement les mécanismes mis en jeu. Attention, il ne s'agit pas ici de l'agrégation par référence qui est souvent représenté par un pointeur sur un objet. Nous avons déjà évoqué le problème des pointeurs dans l'étude précédente.



Création et utilisation des objets

Par rapport à ce contexte, nous allons analyser plusieurs cas de figure, notamment le rôle joué par chacun des objets durant la phase de création. Dans la représentation UML, les constructeurs sont très rarement représentés. En effet, le but de la phase de conception, est de déterminer la structure du logiciel représenté par ses différents objets. Durant cette phase, les détails de construction ne présente aucun intérêt. Par contre, au moment de l'implémentation, vous devez vous assurer que vos objets seront correctement créés. C'est à ce moment là, que vous déciderez du ou des constructeurs à mettre en place.

Première situation - aucune des deux classes ne possèdent de constructeurs :

Malgré cette annonce, n'oubliez pas que les objets disposent d'un comportement minimum et que le langage C++ propose un certain nombre de méthodes qui sont créées automatiquement grâce au canevas proposé par la forme canonique des classes. Ainsi, même si nous ne proposons pas de constructeurs, de toute façon, il existe les constructeurs par défaut (et qui ne font rien, par défaut). Je rappelle que les constructeurs par défaut ne possèdent pas de paramètres.

```
//-----------------------------------------------------------------
class Point
{
  int x, y;
public:
  void deplace(int dx, int dy) { x += dx; y += dy; }

  Point() {} // constructeur par défaut
  ~Point() {}
  Point(const Point& p) { x = p.x; y = p.y;  }
  Point& operator= ( const Point& p) { x = p.x; y = p.y;  }
};
//-----------------------------------------------------------------
```

Forme canonique de la classe "Point"

```
//-----------------------------------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  void deplace(int dx, int dy) { centre.deplace(dx, dy); }

  Cercle() {} // constructeur par défaut
  ~Cercle() {}
  Cercle(const Cercle& c) { centre = c.centre; rayon = c.rayon; }
  Cercle& operator= (const Cercle& c) { centre = c.centre; rayon = c.rayon; }
};
//-----------------------------------------------------------------
```

Forme canonique de la classe "Cercle"

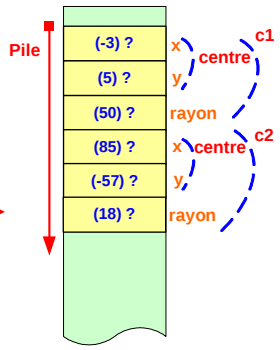
```
//-----------------------------------------------------------------
int main()
{
  Cercle c1, c2;   // création de l'objet c1 et de l'objet c2
  Cercle c3 = c1;  // création de l'objet c3 par copie de l'objet c1
  c1 = c2;         // affectation de c1 par copie de l'objet c2
  ...
  c1.deplace(-5, 3);
  ...
  return 0;
}
//-----------------------------------------------------------------
```

Afin de bien maîtriser les différents enchaînements, nous allons détailler tout le processus mis en jeu, en analysant le comportement global à chacune des lignes. Nous en profitons pour contrôler le fonctionnement d'un objet durant toute sa vie : création, utilisation, destruction.

--------Cercle c1, c2 ;----------

Création des objets « *c1* » et « *c2* ». Cette phase fait systématiquement appel à un constructeur. C'est donc le constructeur par défaut qui est sollicité vu que les objets ne possèdent pas d'arguments. Toutefois, avant que ce constructeur ne soit exécuté, un certain nombre d'événements doivent se produire.

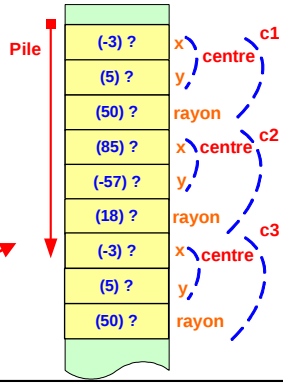
1. **Comme pour tout autre variable, il faut d'abord allouer la mémoire nécessaire en rapport à la dimension des objets. Comme nous sommes en présence de variables locales, cette allocation se fait sur la pile.**
2. **Une fois que l'emplacement mémoire est constitué, chacun des objets internes, s'ils existent, doivent être construits. Sauf indication contraire, c'est le constructeur par défaut qui est sollicité. Dans notre cas, c'est le constructeur par défaut de la classe « *Point* » qui est appelé. Vu que nous n'avons pas fait de redéfinition, ce constructeur ne fait rien par défaut. Du coup l'emplacement mémoire réservé pour les attributs « *x* » et « *y* » gardent les valeurs qui se trouvaient au préalable. Donc, les attributs « *x* » et « *y* » ont des valeurs aléatoires.**



Dès que les objets internes sont définitivement construits, c'est au tour du constructeur de la classe conteneur de prendre le relais afin de terminer la création des objets « *c1* » et « *c2* ». En effet, « *Cercle* » possède en plus un « *rayon* » qui, certe n'est pas un objet, mais qui doit tout de même être géré. Ceci dit, puisqu'il s'agit du constructeur proposé par défaut, lui non plus ne modifie pas la valeur de l'attribut. Du coup, « *rayon* » aura également une valeur aléatoire.

-------- Cercle c3 = c1 ; ----------

Il s'agit encore une fois d'une création. Elle concerne l'objet « *c3* ». Cette création s'effectue au moyen de l'objet « *c1* ». Cette fois-ci, c'est le constructeur de copie qui est sollicité. Son comportement par défaut est de proposer une copie entre chacun des membres des objets respectifs. Comme un des membres est un objet, ce dernier va également sollicité son propre constructeur de copie. Finalement, c'est le centre du cercle qui va d'abord être copié, viendra ensuite le rayon.
Ce comportement par défaut est très agréable puisque, sans que nous ayons écris une seule ligne de codes supplémentaires, l'objet « *c3* » est une copie conforme de l'objet « *c1* ».

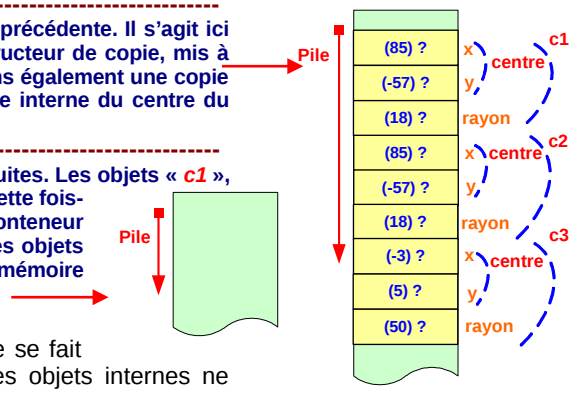


-------- c1 = c2 ; ----------

Attention, même s'il s'agit du même opérateur, cela n'a rien à voir avec la ligne précédente. Il s'agit ici d'une affectation. Ceci dit, nous obtenons le même scénario que pour le constructeur de copie, mis à part que c'est l'affectation par défaut qui est sollicitée. Finalement, nous obtenons également une copie de tous les membres de « *c2* » vers « *c1* » en passant au préalable par la copie interne du centre du cercle grâce également à son opérateur d'affectation par défaut.

-------- } ----------

Au moment de sortir de la fonction, toutes les variables locales doivent être détruites. Les objets « *c1* », « *c2* », « *c3* » font appel respectivement à leur destructeurs. L'enchaînement est cette fois-ci inversé par rapport au constructeur. En effet, le destructeur de la classe conteneur s'occupe d'abord des attributs classiques avant de solliciter les destructeurs des objets internes. Ici, rien ne se passe en particulier, si ce n'est la libération de l'espace mémoire sollicité au moment de la création, et donc, en sens inverse.



Les objets internes sont des attributs comme les autres. L'accessibilité ne se fait qu'au travers des méthodes de la classe englobante. Les méthodes des objets internes ne seront donc utilisées que par les méthodes de la classe conteneur. Ainsi, pour déplacer un cercle, il suffit de déplacer le centre du cercle. Cela se traduit par ; lorsque j'utilise la méthode « ***déplace*** » de la classe « ***Cercle*** », celle-ci utilise la méthode « ***déplace*** » de la classe « ***Point*** ».

```
//-----------------------------------------------------
int main()
{
  Cercle c1, c2;    // création de l'objet c1 et de l
  Cercle c3 = c1;   // création de l'objet c3 par cop
  c1 = c2;          //
  ...
  c1.deplace(-5, 3);
  ...
  return 0;
}
//-----------------------------------------------------
```

```
//-----------------------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  void deplace(int dx, int dy) { centre.deplace(dx, dy); }

  Cercle() {} // constructeur par défaut
  ~Cercle() {}
  Cercle(const Cercle& c) { centre = c.centre; rayon = c.rayon; }
  Cercle& operator= (const Cercle& c) { centre = c.centre; rayon = c.rayon; }
};
//-----------------------------------------------------
```

Forme canonique de la classe "Cercle"

```
//-----------------------------------------------------
class Point
{
  int x, y;
public:
  void deplace(int dx, int dy) { x += dx; y += dy; }

  Point() {} // constructeur par défaut
  ~Point() {}
  Point(const Point& p) { x = p.x; y = p.y;  }
  Point& operator= ( const Point& p) { x = p.x; y = p.y;  }
};
//-----------------------------------------------------
```

Forme canonique de la classe "Point"

Vous remarquez le rôle du conteneur par rapport aux objets internes. C'est lui qui doit s'occuper de tout. Il est normal de procéder de cette façon, puisque les objets intégrés sont utilisés de façon particulière par rapport à leur conteneur.

Deuxième situation - les deux classes redéfinissent les constructeur par défaut :

Finalement, tout fonctionne très bien sans avoir écrit beaucoup de lignes. Ce qui est embêtant, tout de même, c'est que l'état de tous ces objets est totalement aléatoire. Il serait peut être plus judicieux d'imposer un état par défaut, en proposant, par exemple, une valeur nulle à chacun des attributs.

On pourrait se demander pourquoi le système ne le fait pas automatiquement. Il ne faut pas oublier que ce langage a été conçu pour être le plus performant possible en terme de rapidité au détriment de la validité des valeurs. Du coup, le système ne va pas perdre son temps à placer des valeurs sur la pile, surtout, qu'il est préférable que se soit le programmeur qui fasse son choix.

Nous allons donc redéfinir les constructeurs par défaut représentant un cercle placé à l'origine avec un rayon de 10 pixels. Dès que nous définissons un nouveau constructeur, le constructeur d'origine est irrémédiablement inactif.

```
//-----------------------------------------------------
int main()
{
  Cercle c1, c2;    // création de l'objet c1 et de
  Cercle c3 = c1;   // création de l'objet c3 par co
  c1 = c2;          // a
  ...
  c1.deplace(-5, 3);
  ...
  return 0;
}
//-----------------------------------------------------
```

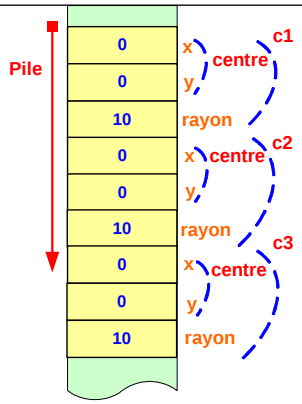
```
//-----------------------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  void deplace(int dx, in

  Cercle() {}
  Cercle() { rayon = 10; }
  ~Cercle() {}
  Cercle(const Cercle& c) { centre = c.centre; rayon = c.rayon; }
  Cercle& operator= (const Cercle& c) { centre = c.centre; rayon = c.rayon; }
};
//-----------------------------------------------------
```

```
//-----------------------------------------------------
class Point
{
  int x, y;
public:
  void deplace(int dx, int dy) { x += dx; y += dy; }

  Point() {}                  // constructeur par défaut d'origine
  Point() { x = y = 0; }      // nouveau constructeur par défaut
  ~Point() {}
  Point(const Point& p) { x = p.x; y = p.y;  }
  Point& operator= ( const Point& p) { x = p.x; y = p.y;  }
};
//-----------------------------------------------------
```

Seule la première ligne du programme principal va subir le changement de comportement. En effet, les autres lignes font appel aux anciennes méthodes, donc, le comportement global reste identique. Les objets « ***c1*** » et « ***c2*** » utilisent tous les deux le constructeur par défaut de la classe « ***Cercle*** » que nous venons de mettre en place. Souvenez-vous, qu'avant que le constructeur de la classe « ***Cercle*** » exécute ce qui lui est demandé, il attend que l'objet interne « ***centre*** » soit correctement construit grâce également à l'appel de son nouveau constructeur par défaut. Du coup, le constructeur de la classe « ***Cercle*** » ne s'occupe que de l'initialisation du « ***rayon*** ».



Troisième situation – La classe « Point » possède un seul constructeur avec plusieurs paramètres, alors que la classe « Cercle » n'a pas de constructeur :

Cette situation n'est pas envisageable. En effet, la classe « ***Point*** » attend des valeurs précises pour sa construction. Si il n'existe pas de constructeur pour la classe « ***Cercle*** », cela veut dire que les objets créés dans le programme principal seront des objets par défaut, c'est-à-dire, des objets sans arguments. Dans ce cas là, la classe « ***Point*** » n'arrive pas à se construire puisqu'elle attend des arguments, et personne ne lui en fournit. Même, si nous désirons conserver la possibilité d'avoir des objets par défaut, il est quand même nécessaire que la classe « ***Cercle*** » s'occupe de l'initialisation de son centre.

> ***Règle***
> Lorsqu'un objet interne possède un ou plusieurs constructeurs mais pas de constructeur par défaut, il est impératif que la classe conteneur définisse au moins un constructeur. Ce ou ces constructeurs doivent absolument faire appel explicitement a un des constructeurs de l'objet interne. Il faut, tout simplement, être sûr que l'objet interne soit correctement créé.

Quatrième situation – les deux classes définissent de nouveaux constructeurs :

Imaginons que la classe « ***Point*** » possède un seul constructeur qui permet de récupérer les coordonnées. Nous allons définir un constructeur par défaut pour la classe « ***Cercle*** » afin que le centre soit explicitement défini. Il faut alors maîtriser l'appel explicite et ne pas laisser le système utiliser son comportement automatique qui consiste à ce que chacun des objets membres fassent appel directement à leurs constructeurs par défaut.

```
class Point
{
  int x, y;
public:
  Point(int x, int y) { this->x = x;  this->y = y; }
  ...
};
//-----------------------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  Cercle() : centre(0, 0) { rayon = 10; }
  ...
};
//-----------------------------------------------------
int main()
{
  Cercle c1;
  ...
  return 0;
}
```

Initialisation des coordonnées

Corps du constructeur

Initialisation explicite de l'objet "centre"

Création de l'objet c1 avec appel du constructeur par défaut de "Cercle"

> ***Liste d'initialisation***
> Il existe une syntaxe appropriée pour les appels explicites. C'est une liste d'initialisation des membres qui suit la signature du constructeur et qui débute par un deux-points « ***:*** ». Le nom du membre est spécifié, suivi par les valeurs initiales entre parenthèses, à l'identique de la syntaxe de la création d'un objet avec paramètres. Si vous possédez plusieurs objets membres, chacun devra être initialisé et donc faire parti de la liste d'initialisation. La séparation entre les différents objets s'effectue à l'aide de l'opérateur virgule « ***,*** ». Attention, l'ordre d'initialisation des objets n'est pas du tout imposé par l'ordre de la liste d'initialisation mais uniquement par celui de la déclaration des membres de la classe.

Lorsque « ***c1*** » est en phase de création, le constructeur par défaut est sollicité. Vu qu'il possède une liste d'initialisation, il s'occupe immédiatement de l'objet désigné dans la liste (un seul élément dans cette liste). « ***centre*** » fait alors un appel au constructeur possédant deux arguments. Quand la liste d'initialisation est entièrement gérée, c'est au tour du corps du constructeur de prendre le relais. Dès lors, toutes les instructions proposées en son sein sont exécutées.

En toute rigueur, toutes les instructions qui se situent dans le corps du constructeur sont des instructions de calcul plutôt qu'une initialisation proprement dite. Ainsi, lorsque nous écrivons « ***rayon = 10 ;*** », il s'agit d'une affectation et non pas d'une initialisation. Dans le cas du rayon, cela ne présente de problème majeur (bien que !?), et c'est généralement l'écriture que nous adopterons. Toutefois, il existera des cas de figure ou cette distinction aura toute son importance.

Si nous voulons être d'une extrême rigueur, il serait peut-être souhaitable que l'attribut « ***rayon*** » soit initialisé plutôt que de proposer cette affectation. En effet, n'oubliez pas que les types dits primitifs, sont finalement considérés comme des classes et que pour eux, les constructeurs existent également :

- Les types primitif ont un ***constructeur par défaut*** qui, soit ne fait rien, soit propose une valeur nulle pour les variables statiques.
- Les types primitifs possèdent également un ***constructeur avec un seul argument*** dont le type de paramètre correspond au type de la variable.

Imaginons que nous déclarions un objet de type « ***Cercle*** » dans la zone statique de la mémoire. Souvenez-vous que cette zone est particulière puisque la valeur des cases mémoires n'est jamais aléatoire, il doit toujours exister une valeur précise. Si aucune valeur n'est proposée, la case mémoire prend alors une valeur nulle.

```
//----------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  Cercle() : centre(0, 0) { rayon = 10; }
  ...
};
//----------------------------------------
```

« ***rayon*** » n'est pas proposé dans la liste d'initialisation. Donc, c'est le constructeur par défaut qui est sollicité (toujours, même s'il ne fait rien). Dans le cas d'un objet statique, le constructeur par défaut propose la valeur nulle. Donc, à cet instant là, « ***rayon ← 0*** ».

Plus tard, dans le corps du constructeur, nous demandons un changement de valeur pour « ***rayon*** », puisque nous lui proposons une affectation. Finalement « ***rayon ← 10*** ».

Ce scénario nous montre que nous perdons du temps en proposant deux valeurs successives pour la même variable. Pour ce cas très précis, il est donc préférable de proposer l'initialisation plutôt que l'affectation.

```
//----------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  Cercle() : centre(0, 0), rayon(10) { }
  ...
};
//----------------------------------------
```
Initialisation explicite de "rayon"

Plus rien dans le corps du constructeur

Nous gardons très souvent la première syntaxe, puisque même si tous les constructeurs par défaut sont sollicités, généralement, ils ne font rien. Donc, nous ne perdons pas de temps. Toutefois, nous venons de voir que dans le cas exceptionnel d'un objet statique, cela peut porter un préjudice.

Cinquième situation – les deux classes possèdent plusieurs constructeurs :

Finalement, ce principe de liste d'initialisation est très simple d'utilisation puisque nous avons le loisir de choisir notre constructeur pour chacun des objets membres. Toutefois, sans précision particulière, c'est le constructeur par défaut qui est pris en compte. Nous remarquons finalement que le constructeur par défaut joue un rôle prépondérant dans bien des situations.

```
6  //----------------------------------------
7  class Cercle {
8    Point centre;
9    unsigned rayon;
10 public:
11   Cercle();
12 };
13 //----------------------------------------
14 inline Cercle::Cercle() : centre(0, 0), rayon(10) { }
15 //----------------------------------------
```
Définition du constructeur à l'extérieur de la déclaration de la classe

Attention

Par rapport au scénario ci-contre, l'écriture « ***C4 = 5 ;*** » est interdite puisque aucun constructeur ne gère ce genre de situation.

```
//----------------------------------------
class Point
{
  int x, y;
public:
  Point() { y = x = 0; }
  Point(int x, int y) { this->x = x;  this->y = y; }
  ...
};
//----------------------------------------
class Cercle
{
  Point centre;
  unsigned rayon;
public:
  Cercle() : rayon(10) { }
  Cercle(int x, int y, int r = 10) : centre(x, y), rayon(r) {}
  ...
};
//----------------------------------------
int main()
{
  Cercle c1, c2(5, -3, 20), c3(2, 2);
  ...
```
Appel implicite du constructeur par défaut de "Point"

Paramètre par défaut

Appel du constructeur qui possède deux paramètres

Possibilités d'utilisation

Attributs constants

Dans quelques cas, vous pouvez avoir besoin d'attributs particuliers comme des attributs constants ou alors des attributs qui fassent référence à d'autres éléments extérieurs. La déclaration de ces attributs particuliers reste classique, toutefois, il est impératif que ces éléments soient toujours initialisés avant leurs utilisations.

```
//----------------------------------------
class Cercle
{
  Point centre;
  const unsigned rayon;
public:
  Cercle(int r=10) { rayon = r; }
  ...
};
//----------------------------------------
```
Attribut constant

Attention, c'est une affectation, cette écriture n'est pas admise puisque il s'agit d'un membre constant

Imaginons, par exemple, qu'une fois que le cercle est dessiné, il soit possible de le déplacer, mais que le rayon garde toujours la même valeur, quelque soit la situation. Il serait alors judicieux de proposer à l'attribut « ***rayon*** », le qualificatif de constant, et c'est au moment de la création que nous décidons de la valeur constante à donner. Si vous écrivez le code ci-contre, vous obtiendrez une erreur de compilation.

En effet, lorsque vous êtes à l'intérieur du constructeur, la phase d'initialisation proprement dite est terminée et vous effectuez une affectation sur un élément constant, ce qui est totalement interdit. Encore une fois, la liste d'initialisation, comme son nom l'indique, permet de palier à ce problème. C'est effectivement au moment de l'initialisation qu'il faut préciser la valeur de la constante et pas plus tard.

```
//----------------------------------------
class Cercle
{
  Point centre;
  const unsigned rayon;
public:
  Cercle(int r=10) : rayon(r) { }
  ...
};
//----------------------------------------
```
Initialisation de la constante

Finalement, dans le cas d'un attribut primitif non constant, nous avons le choix entre la liste d'initialisation ou l'affectation directe pour imposer une valeur. Dans le cas où cet attribut est constant, il n'y a pas d'alternative, le seul choix possible est l'initialisation explicite.

Agrégation - attributs de type référence

Le procédé reste le même si vous devez implémenter des attributs de type « référence ». En effet, souvenez-vous qu'une référence correspond à un autre nom donné à une autre variable déjà existante. Il faut que puissions localiser son adresse, et donc y faire référence. Cela impose que toute référence doit être également initialisée avant son utilisation, d'où l'utilisation impérative de la « liste d'initialisation » durant la phase de construction.

```
10 class Cercle
11 {
12   Point centre;
13   unsigned &rayon;
14 public:
15   Cercle(int x, int y, unsigned &r) : centre(x, y), rayon(r) {}
16 };
17 //-----------------------------------------------------------
18 int main()
19 {
20   unsigned rayon = 50;
21   Cercle c1(3, 5, rayon), c2(-2, 0, rayon);
22   rayon *= 2;
23   return 0;
24 }
```

Si l'attribut est une référence, le paramètre doit l'être aussi

Pour une référence, la liste d'initialisation est impérative

Ici, le rayon pour c1 et c2 est de 50

Ici, le rayon pour c1 et c2 est de 100

Imaginons, par exemple que nous ayons besoin d'un groupe de cercle possédant systématiquement le même rayon fixé par une variable externe à la classe. Vous avez ci-contre le programme correspondant. Attention, il faut que le paramètre du constructeur qui récupère l'adresse de la variable externe pour initialiser la référence soit également une référence, sinon cela n'aurait pas de sens.

Ce que nous avons mis en œuvre sur les attributs de type primitif fonctionne également sur les attributs qui sont des objets constants ou des attributs faisant référence à d'autres objets. Je dirais même que cela conforte d'autant plus l'utilisation de la « liste d'initialisation ».

Imaginons, par exemple, que nous ayons besoin d'un ensemble de cercles concentriques qu'il est possible de déplacer à notre convenance par rapport à un point de référence.

```
10 class Cercle
11 {
12   Point &centre;
13   const unsigned rayon;
14 public:
15   Cercle(Point &c, unsigned r) : centre(c), rayon(r) {}
16 };
17 //-----------------------------------------------------------
18 int main()
19 {
20   Point p(3, 5);
21   unsigned rayon = 30;
22   Cercle c1(p, rayon), c2(p, rayon+10);
23   p.deplace(-5, 2);
24   return 0;
25 }
```

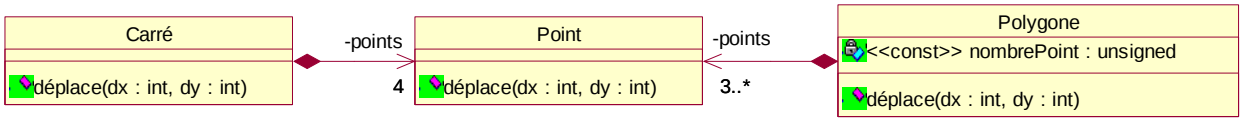
Ici, les deux cercles sont centrés sur <3, 5>

Ils sont maintenant centrés sur <-2, 7>

Composition de plusieurs objets de même nature

Les règles que nous avons établies concernaient la composition d'un seul objet. Qu'en est-il de la composition lorsque qu'un attribut représente une collection d'objet ? Devons-nous également utiliser la liste d'initialisation ?

Pour comprendre de quoi il s'agit, il faut travailler avec des exemples concrets. Nous pouvons, par exemple, fabriquer un carré en estimant qu'il est composé précisément de quatre points. De même, nous pouvons estimer qu'un polygone est composé d'un ensemble de points dont le nombre est supérieur ou égal à trois.



Pour le carré, cela se traduit par un attribut de type tableau d'objet :

```
class Carré
{
    Point points[4] ;
    ...
} ;
```

Pour le polygone, il s'agit cette fois-ci d'un pointeur qui fera référence (indirectement) à un tableau dynamique d'objets :

```
class Polygone
{
    Point *points ;
    const unsigned nombrePoint ;
    ...
} ;
```

Comment initialiser précisément ces attributs ? Est-ce que la liste d'initialisation va convenir dans ce contexte ?

La classe « Carré »

Pour bien comprendre les mécanismes mis en jeu, il est nécessaire de se rappeler le fonctionnement du tableau statique et dynamique en dehors de la notion d'attribut, c'est-à-dire, de revoir la déclaration de ces tableaux en dehors de la classe. Ainsi, la seule solution pour initialiser le tableau au moment de la déclaration est la suivante :

```
Point points[4] = { Point(2, 2), Point(2, -2), Point(-2, -2), Point(-2, 2) } ;
```

Malheureusement, cette écriture n'est pas possible lorsqu'il s'agit d'attributs. Nous n'avons pas le droit d'écrire la suite des valeurs directement dans la déclaration de l'attribut. Nous devons, au mieux, passer par la liste d'initialisation.

Le problème, c'est que nous devons récupérer les valeurs au travers des paramètres proposés par le constructeur. Ce qui suppose que ces valeurs soient déjà créées bien avant la phase de construction. Ce qui est logique puisque c'est au moment de la phase de création que l'utilisateur décide de la valeur de chacun des points qui constituera le carré.

Deux cas peuvent alors se présentés, soit nous proposons les quatre points séparément, soit l'ensemble des quatre points sous forme de tableau. Par rapport à ces deux cas de figure, nous allons donc voir ce qu'il se passe au niveau de la liste d'initialisation. Prenons le premier cas, et voici éventuellement le constructeur que nous pourrions proposer :

```
class Carré
{
    Point points[4] ;
public :
    Carré(Point p1, Point p2, Point p3, Point p3) : points( ? ) { }
        …
} ;
```

Nous sommes incapable de proposer une seule entité à partir des quatre objets proposés en argument du constructeur.

Dans l'attribut « *points* » explicité dans la liste d'initialisation, nous ne savons pas quoi mettre. En fait, pour ce cas là, la seule solution est de proposer successivement l'initialisation de chacun des points respectivement à chacune des cases du tableau. Du coup, cela ne peut se faire qu'à l'intérieur du constructeur puisqu'il s'agit précisément d'un ensemble d'affectations.

```
class Carré
{
    Point points[4] ;
public :
    Carré(Point p1, Point p2, Point p3, Point p3)
    {
        points[0] = p1 ;
        points[1] = p2 ;
        points[2] = p3 ;
        points[3] = p4 ;
    }
        …
} ;
```

Cette écriture ressemble à ce que nous ferions en dehors de la classe, c'est-à-dire, d'abord la déclaration suivi des affectations.

```
Point points[4] ;
points[0] = p1 ;
points[1] = p2 ;
points[2] = p3 ;
points[3] = p4 ;
```

Conclusion

Dans ce cas là, la liste d'initialisation ne fonctionne pas.

Voyons si nous avons plus de chance avec le deuxième cas de figure :

```
class Carré
{
    Point points[4] ;
public :
    Carré(Point p[ ]) : points(p) { }
        …
} ;
```

En dehors de la classe, cette écriture est équivalente à :

```
Point points[4] ;
Point p[ ] = {p1, p2, p3, p4} ;
points = p ;  // cette écriture est rigoureusement interdite
```

Cette fois-ci, nous savons quoi mettre dans l'attribut « *points* » écrit dans la liste d'initialisation. Pourtant, une erreur de compilation apparaît si nous tentons d'exécuter ce programme. En effet, nous tentons de proposer une affectation entre deux tableaux, ce qui est formellement interdit.

N'oubliez pas qu'un tableau est avant tout un pointeur constant et que l'affectation d'un tableau envers un autre ne consiste pas à copier le contenu de leurs cases, mais à changer l'adresse de localisation. Ce genre de traitement n'est pas permis puisque cette localisation est définie une fois pour toute et qu'il est impossible de la changer à cause du pointeur constant.

Du coup, si nous désirons quand même conserver notre tableau de points comme argument du constructeur, nous devons réaliser la copie de chacune des cases du tableau afin de bien initialiser notre attribut.

```
class Carré
{
    Point points[4] ;
public :
    Carré(Point p[ ])
    {
        for (int i=0 ; i<4 ; i++)  points[ i ] = p[ i ] ;
     }
    …
} ;
```

Conclusion

Dans ce cas là aussi, la liste d'initialisation ne fonctionne pas.

Finalement, pour ces deux cas de figure, la liste d'initialisation n'est pas utilisée. Il n'empêche, et ne l'oubliez pas, que lorsque nous rentrons dans le constructeur, l'attribut qui représente le tableau d'objet est déjà créé. ***Cela suppose l'utilisation du constructeur par défaut de la classe « Point » pour chacune des cases du tableau. Si un tel constructeur n'existe pas, les tableaux d'objets ne sont pas permis.***

Nous avions déjà beaucoup insisté sur l'importance de mettre en œuvre un constructeur par défaut lorsque nous manipulons les tableaux. C'est d'autant plus vrai pour les attributs de type tableau d'objets.

Et pour la classe « *Polygone* », comment doit-on procéder ? Pouvons-nous proposer l'écriture suivante ?

```
class Polygone
{
    Point *points ;
    const unsigned nombrePoint ;
public :
    Polygone(unsigned n) : nombrePoint(n) { … }
    …
} ;
```

Le compilateur accepte tout à fait cette écriture. Cela ne le gène absolument pas. Dès que vous vous trouvez avec des pointeurs, vous pouvez ne pas les initialiser. Un pointeur reste un pointeur même si celui-ci pointe vers un objet. Dans la leçon précédente, nous avons d'ailleurs déjà traités ce cas de figure sans nous préoccuper de la liste d'initialisation.

Pourrons-nous quand même utiliser la liste d'initialisation ?

Oui, mais cela n'arrange pas grand-chose. Nous pouvons écrire indifféremment :

```
Polygone(unsigned nombre) : nombrePoint(nombre), points(new Point[nombre]) { }
```

Que :

```
Polygone(unsigned nombre) : nombrePoint(nombre)
{
    points = new Point[nombre] ;
}
```

> ***Conclusion***
> Dans ce cas là aussi, la liste d'initialisation n'est pas très utile.

Et encore, dans ces écritures, je ne me suis préoccuper d'initialiser que le pointeur. Il faut compléter également le tableau dynamique qui est derrière à l'aide des objets « *points* » représentant le polygone. Ainsi, le code devient, par exemple celui-ci :

```
Polygone(Point p[ ], unsigned nombre) : nombrePoint(nombre)
{
    points = new Point[nombre] ;
    for (int i=0 ; i<nombre ; i++)  points[ i ] = p[ i ] ;
}
```

Souvenez-vous d'ailleurs que pour un tableau dynamique, il n'est pas possible d'initialiser ses cases en même temps que l'allocation mémoire. Par contre, si l'attribut est un pointeur qui représente un objet dynamique, cette fois-ci, nous pouvons initialiser l'objet avec une valeur particulière. Du coup, il est judicieux et avantageux d'utiliser la liste d'initialisation. Par exemple, en prenant l'attribut « *centre* » qui est un pointeur vers un objet dynamique représentant un « *Point* » pour la classe « *Cercle* », nous pouvons parfaitement écrire :

```
class Cercle
{
    Point *centre ;
    const unsigned rayon ;
public :
    Cercle(int x, int y, unsigned r) : centre(new Point(x, y)),
    rayon(r) { }
    …
} ;
```

Synthèse - Quand utiliser la liste d'initialisation ?

1. ***L'attribut est un objet*** : A moins que l'objet possède un constructeur par défaut, il est très souvent judicieux d'utiliser la liste d'initialisation. Cette liste est de toute façon nécessaire si nous désirons préciser une valeur particulière autre que celle prévue par défaut. Surtout, si l'objet ne possède pas de constructeur par défaut, la liste d'initialisation est obligatoire pour bien spécifier le constructeur à prendre.
2. ***Constante vers un type primitif ou vers un objet*** : Il est absolument nécessaire d'utiliser la liste d'initialisation. Une constante doit être précisée avant sa consultation. Rappelez-vous qu'il n'est plus permis ultérieurement de changer sa valeur.
3. ***Référence vers un type primitif ou vers un objet*** : même remarque que pour la constante.
4. ***Tableau d'objets*** : La liste d'initialisation est inutilisable. Il est impératif qu'il existe un constructeur par défaut pour l'objet représenté par chacune des cases du tableau, sinon cette écriture n'est pas possible.
5. ***Pointeur d'objet(s)*** : Généralement utilisé pour représenter un objet dynamique ou plus souvent encore un tableau dynamique d'objets. Dans ce dernier cas, l'initialisation des cases du tableau en même temps que l'allocation dynamique n'est pas possible. La liste d'initialisation n'a donc aucun intérêt. Par contre, dans le cas d'un pointeur vers un seul objet, la liste doit être utilisée lorsque nous proposons une valeur particulière au constructeur.

Retour sur le constructeur de copie

Nous avons écrit au tout début de cette étude que la construction de copie se réalisée de la façon suivante :

```
Cercle(const Cercle& c)  { centre = c.centre ;   rayon = c.rayon ; }
```

En fait, ce n'est pas très rigoureux. Un constructeur de copie est un constructeur comme un autre, et maintenant que nous connaissons la liste d'initialisation, nous devons écrire :

```
Cercle(const Cercle& c) : centre(c.centre), rayon(c.rayon ) { }
```