

Chapitre 8

Les Objets

La vie d'un objet se résume à trois phases fondamentales :

- sa création,
- son utilisation,
- puis sa destruction.

La création et la destruction font appel à des méthodes particulières : respectivement les constructeurs et le destructeur.

1- Création d'objets : les constructeurs

✓ La création d'objets repose sur des méthodes spéciales nommées *constructeurs*.

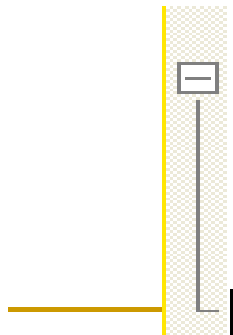
✓ Celles-ci ont pour but de réaliser la partie « instanciation » de la création d'un objet, c'est à dire, le positionnement de la valeur initiale des variables d'instance.

- ✓ Les constructeurs sont faciles à repérer : ils portent le même nom que la classe.

Par exemple:

Si l'on reprend la classe Point, son constructeur est facile à repérer :

Point::Point(int absc=0, int ordo=0);



```
Point::Point(int absc=0, int ordo=0)
{
    abscisse_ = absc;
    ordonnee_ = ordo;
    NbPoints_++;
}
```

2- Instanciation des objets

La création d'objets (opération d'instanciation) se fait en appelant le constructeur. La syntaxe diffère selon la classe d'allocation de l'objet.

2.1- Les classes d'allocations

Il y a trois classes d'allocation :

Allocation statique

- Elle concerne les objets placés en variable globale.
- Les objets explicitement mis en allocation statique dans une méthode ou fonction à l'aide du modificateur d'allocation **static**.
- Rappelons que dans ce cas, ils gardent leur valeur d'un appel de la fonction/méthode sur l'autre.
- Ces objets sont créés dès le début du programme, ils sont détruits automatiquement à la fin de l'exécution de celui-ci sans que vous ayez à vous en occuper.

Allocation automatique

- Tout objet créé sur la pile est dit à allocation automatique.
- Cela concerne donc les variables temporaires dans une fonction/méthode ainsi que les objets renvoyés par une fonction/méthode qui sont momentanément stockés sur la pile.
- Un objet automatique est créé à chaque lancement de la fonction/méthode à l'intérieur de laquelle il est déclaré et détruit automatiquement sans que vous ayez à vous en soucier dès le retour à l'appelant.

Allocation dynamique

- Il s'agit des objets créés sur le tas et auxquels vous ne pouvez accéder qu'au travers d'un pointeur.
- Ils sont créés par un appel à l'opérateur **new** et doivent être détruits explicitement par un appel à l'opérateur **delete**.

2.2- Création d'instances simples

✓ La création d'une instance automatique ou statique est simple et répond à la syntaxe suivante :

IdentificateurClasse identificateurObjet(parametres d'un constructeur);

✓ Dans le cas d'une instance dynamique, il faut commencer par déclarer un pointeur, puis appeler **new** suivi du nom du constructeur avec ses paramètres. Ce qui peut se faire sur une ou plusieurs lignes

IdentifClasse *identifPointeurObjet ;
IdentifPointeurObjet = new IdentifClasse(parametres d'un constructeur);

```
int main(int, char **)  
{  
    Point p1;          // Utilise le constructeur Point::Point(int, int)  
                        // avec les valeurs par default des arguments  
    Point p2(10,20);   // Crée un nouveau point avec x=10 et y=20  
    Point *p3, *p4;     // Déclaration d'un pointeur sur un objet de type Point  
  
    p3=new Point(20,30); // Instanciation de l'objet dynamique avec arguments explicites  
    p4=new Point;        // instanciation de l'objet dynamique avec arguments par défaut  
  
    delete p3;          // On n'oublie pas de rendre la mémoire !  
    delete p4;          // Idem => voir section suivante !  
    return 0;  
}
```

3- Appel de méthodes

La syntaxe d'appel d'une méthode se différencie selon deux modalités :

- le type de la méthode (instance ou méthode de classe)
- et, dans le cas d'une méthode d'instance, la classe d'allocation de l'objet cible.

3.1 Appel de méthodes d'instance

- ✓ Le code se différencie selon que vous utilisez un objet d'allocation dynamique ou non.
- ✓ Pour les objets à classe d'allocation statique ou automatique, la syntaxe d'appel est la suivante :

objet.methodeInstance(paramètres);

Par exemple:

✓ Appliquons les opérations suivantes à l'objet de classe Point nommé p1 telle qu'il a été instancié au paragraphe précédent :

- afficher son abscisse,
- le déplacer de 10 unités sur x et 20 unités sur y relativement à sa position actuelle,
- afficher son ordonnée.

▪

nous obtenons le code suivant :

```
cout << p1.x() << endl;  
p1.deplacerDe(10,20);  
cout << p1.y();
```

- ✓ Pour un objet dynamique, ce n'est guère plus compliqué.
- ✓ Il faut juste changer le "." par une flèche "->".
- ✓ Ainsi, si l'on reprend l'exemple précédent avec l'objet p4 alloué dynamiquement, on obtient :

```
cout << p4->x() << endl;  
p4->deplacerDe(10,20);  
cout << p4->y();  
|
```

3.2- Appel de méthodes de classe

- ✓ Les méthodes de classe ne sont pas invoquées à travers un objet mais à l'aide du nom de la classe.
- ✓ La syntaxe générale est la suivante :

```
NomClasse::NomMethodeClasse (paramètres) ;
```

- ✓ Ainsi, si vous souhaitez afficher le nombre d'objets graphiques dans votre système en invoquant la méthode **NbPoints**, vous devrez utiliser :

```
cout << "Nombre de points présents : " << Point::NbPoints() << endl;
```

3.3- Accès aux attributs

- Tout d'abord, notons que : aucun attribut ne doit être visible de l'extérieur.
- Donc si vous accédez à un attribut ça ne peut être que dans 2 conditions bien particulières :
 - ✓ Vous êtes dans une méthode de la classe A qui accède aux attributs d'un autre objet de la classe A que l'objet cible
 - ✓ Vous êtes dans une méthode friend (nous aborderons cela plus tard).
- Quoiqu'il en soit, la syntaxe est la même. On accède à un attribut de la même manière que l'on accède aux méthodes en notation pointée pour les instances statiques ou avec une flèche pour les instances dynamiques.

4- Appels de méthodes et objets constants

- ✓ Les règles suivantes s'appliquent aux objets constants :
 - On déclare un objet constant avec le modificateur *const*.
 - On ne peut appliquer que des *méthodes constantes* sur un objet *constant*
 - Un objet passé en paramètre sous forme de référence constante est considéré comme constant.
- ✓ Une méthode constante c'est tout simplement une méthode qui ne modifie aucun des attributs des objets.
- ✓ Elle est donc tout à fait applicable sur un objet constant ! D'où la règle importante suivante :

Il est impératif de déclarer constante toute méthode ne modifiant pas l'état des objets auxquels elle s'applique
- ✓ Si vous appliquez ce principe, vous pourrez toujours passer en référence constante les objets que vous ne souhaitez pas modifier.

Comment l'on rend une méthode constante ?

- ✓ Il suffit de faire suivre son prototype du mot clef `const`, que ce soit lors de la déclaration ou lors de la définition. Ainsi le code devient :

```
class Chose
{
    // Code omis

    void metConstDep(int i, double d) const;
    void metConstInl(int j, const char *z) const
    {
        // Tout plein de code vachement utile
    }
    void metNonConst(double h)
    {
        // Code non moins utile
    }
};

void Chose::metConstDep(int i, double d) const
{
    // Code vachement important lui aussi
}
```

Utilisons maintenant ce code !

```
int main(int, char **)
{
    const Chose OBJET_CONSTANT(params d initialisation); // Objet constant !

    OBJET_CONSTANT.metConstDep(1,2.5); //Ok, methode constante sur objet constant

    • OBJET_CONSTANT.metNonConst(35.5); //NON ! metNonConst est une méthode non constante
      //qui ne peut en aucun cas être appliquée sur un objet constant

    OBJET_CONSTANT.metConstInk(3,"coucou"); //Ok, methode constante sur objet constant

    return 0;
}
```

5- Surdéfinition des fonctions membres.

- ✓ La surcharge est un mécanisme qui permet de donner différentes signatures d'arguments à une même fonction.
- ✓ Plus précisément, vous pouvez nommer de la même façon des fonctions de prototypes différents i.e ; modification des squelettes des méthodes (arguments différents soit en nombre ou en type).
- ✓ L'intérêt est de pouvoir nommer de la même manière des fonctions réalisant la même opération à partir de paramètres différents.
- ✓ En utilisant la propriété de surcharge des fonctions du C++, on peut définir plusieurs constructeurs, ou bien plusieurs fonctions membres, différentes, mais portant le même nom.

Exemple 1:

```
#include <iostream> // Surcharge de fonctions
using namespace std;
class point
{
private:
    int x,y;
public:
    point(); // constructeur 1
    point(int abs); // constructeur 2
    point(int abs,int ord); // constructeur 3
    void affiche();
};

point::point() // constructeur 1
{ x=0; y=0; }
point::point(int abs) // constructeur 2
{ x = abs; y = abs; }
point::point(int abs,int ord) // constructeur 3
{ x = abs; y = ord; }
void point::affiche()
{
    cout<<"Je suis en "<<x<<" "<<y<<"\n";
}

int main()
{
    point a, b(5), c(3,12);
    clrscr();
    a.affiche();
    b.affiche();
    c.affiche();
    return 0;
}
```

Exemple 2:

```
#include <iostream> // Surcharge de fonctions
using namespace std;
class point
{
private:
    int x,y;
public:
    point(); // constructeur 1
    point(int abs); // constructeur 2
    point(int abs,int ord); // constructeur 3
    void affiche();
    void affiche(char *message); // argument de type chaîne
};

point::point() // constructeur 1
{ x=0; y=0; }
point::point(int abs) // constructeur 2
{ x=y=abs; }
point::point(int abs,int ord) // constructeur 3
{ x = abs; y = ord; }
void point::affiche() // affiche 1
{ cout<<"Je suis en "<<x<<" "<<y<<"\n"; }
void point::affiche(char *message) // affiche 2
{ cout<<message;
  cout<<"Je suis en "<<x<<" "<<y<<"\n"; }

void main()
{
    point a,b(5),c(3,12); char texte[10] = "Bonjour";
    a.affiche();
    b.affiche("Point b:");
    c.affiche(texte)
}
```

6- Mort des objets

✓ La mort des objets s'accompagne de l'appel d'une méthode spéciale : *le destructeur*.

✓ Notons immédiatement que si un constructeur peut être surchargé, le destructeur lui ne peut pas l'être. En effet, son appel étant automatique.

✓ La syntaxe du destructeur est simple:
Classe::~~Classe

✓ On reconnaît ici le symbole ~ associé à l'opérateur binaire **NON !** A l'instar du constructeur, le destructeur ne renvoie rien ; en outre, il ne prend pas de paramètre.

- ✓ **Le but du destructeur est de procéder à toutes les opérations de nettoyage nécessaires à la destruction correcte d'un objet.**
- ✓ **Par exemple, si le constructeur ou une autre méthode réalise des allocations dynamiques, le destructeur veillera à rendre la mémoire afin d'éviter que celle-ci ne soit perdue.**
- ✓ **Il est important de noter que vous n'avez pas besoin de spécifier un destructeur si votre classe ne nécessite pas de nettoyage spécifique.**
- ✓ **Si vous ne spécifiez pas explicitement un destructeur, un destructeur par défaut est créé automatiquement par le compilateur.**

Exemple:

```
class Tableau
{
public:
    Tableau(int laTailleDuTableau=5)
    {
        taille_=laTailleDuTableau;
        if (_taille)
            tab_=new int [taille_];
        else
            tab_=NULL;
    }

    ~Tableau(void)
    {
        delete [] tab_;
    }

private:
    int  taille_;
    int *tab_;
};
```

✓ **Lorsqu'un objet a les classes d'allocation statique (variable globale) ou automatique (variable locale dans une fonction/méthode), il est détruit automatiquement dès qu'il sort de sa portée, c'est à dire :**

- **A la fin du programme pour une variable globale.**
- **Dès que la fonction/méthode rend la main à l'appelant pour une variable locale.**

✓ **Il en est tout autre pour une variable dynamique, c'est à dire un objet créé dynamiquement à l'aide de new ou de new [].**

✓ **En effet, tout objet instancié dynamiquement doit être explicitement détruit :**

- **à un appel de new doit correspondre un appel de delete**
- **un ensemble d'objets créés à l'aide de new [] doit être détruit par delete []**

7- Objet retourné par une fonction membre

7.1- Retour par valeur

Exemple : (la fonction concernée est la fonction symetrique)

▪

```
#include <iostream>      // La valeur de retour d'une fonction est un objet
using namespace std;    // Retour par valeur
class point
{
private:
int x,y;
public:
point(int abs = 0,int ord = 0)
{
x=abs; y=ord; // constructeur
}
point symetrique();
void affiche();
};

point point::symetrique()
{
point res;
res.x = -x; res.y = -y;
return res;
}

void point::affiche()
{
cout<<"Je suis en "<<x<<" "<<" "<<y<<"\n";
}

int main()
{
point a,b(1,6);
a=b.symetrique();
a.affiche();b.affiche();
return 0;
}
```


7.2- Retour par adresse

Exemple :

```
#include <iostream>
using namespace std;
// La valeur de retour d'une fonction est l'adresse d'un objet
class point
{
private:
int x,y;
public:
point(int abs = 0,int ord = 0)
{
x=abs; y=ord; // constructeur
}
point *symetrique();
void affiche();
};
point *point::symetrique()
{
point *res;
res = new point;
res->x = -x; res->y = -y;
return res;
}
void point::affiche()
{
cout<<"Je suis en "<<x<<" "<<y<<"\n";
}
int main()
{
point a,b(1,6);
a = *b.symetrique();
a.affiche();b.affiche();
return 0;
}
```

7.3- Retour par référence

Exemple:

```
#include <iostream>
using namespace std;
// La valeur de retour d'une fonction est la référence d'un objet
class point
{
private:
int x,y;
public:
point(int abs = 0,int ord = 0)
{
x=abs; y=ord; // constructeur
}
point &symetrique();
void affiche();
};
point &point::symetrique()
// La variable res est obligatoirement static
// pour retourner par référence
static point res;
res.x = -x; res.y = -y;
return res;
}
void point::affiche()
{
cout<<"Je suis en "<<x<<" "<<y<<"\n";
}
int main()
{
point a,b(1,6);
a=b.symetrique();
a.affiche();b.affiche();
return 0;
}
```

Remarque:

" res " et " b.symetrique " occupent le même emplacement Mémoire, car " res " est une référence à " b.symetrique ". On déclare donc " res " comme variable **static, sinon, cet objet n'existerait plus après être sorti de la fonction.**

8- Passage d'objets par référence

Considérons la fonction suivante :

```
void affichageObjetPoint (Point obj)
{
    obj.affiche();
}
```

Examinons ce qui se passe lors de l'appel de cette fonction dans le cas suivant :

```
int main (int, char **)
{
    Point unObjet (...);

    affichageObjetPoint (unObjet);
    return 0;
}
```

- ✓ Rappelons que dans le cas du passage de paramètre par valeur, il y a recopie du paramètre sur la pile.
- ✓ Dans le cas d'un objet, cette recopie nécessite un appel à un constructeur particulier, le constructeur de recopie.
- ✓ Cette opération peut s'avérer particulièrement coûteuse.
- ✓ En outre, l'objet temporaire créé sur la pile devra être détruit à la fin, ce qui implique un appel à un destructeur.
- ✓ Avec un appel par référence, il n'y a pas de recopie car l'on passe l'adresse.
- ✓ Au point de vue du code généré, passer un objet par référence ou un pointeur sur cet objet par valeur revient exactement au même.

✓ Les références sont une facilité apportée au programmeur afin de limiter les erreurs.

✓ Avec un passage par référence, la fonction précédente et son appel s'écrivent :

```
void affichageObjetPoint (Point &obj)
{
    obj.affiche();
}

...

int main (int, char **)
{
    Point unObjet (...);

    affichageObjetPoint (unObjet);
    return 0;
}
```

✓ **Trois points méritent à signaler :**

- **Typiquement l'opérateur "&" servait à prendre une adresse. Ici, il désigne une référence sur un objet.**
- **Du point de vue de l'appelant, rien ne permet de distinguer que l'on a passé l'objet par référence plutôt que par valeur.**
- **A l'intérieur de la fonction, on manipule la référence comme l'on manipulerait un objet**

✓ **Dans le cas où un objet n'est pas destiné à être modifié on utilise ce qu'on appelle une référence constante.**

✓ **Il est très important de noter qu'un objet passé par référence constante dans une fonction ou une méthode y est considéré comme constant. On peut donc appliquer que des méthodes constantes.**

9- Le mot clé " this "

Ce mot désigne l'adresse de l'objet invoqué. Il est *utilisable uniquement* au sein d'une fonction membre.

Exemple:

```
#include <iostream.h>
using namespace std;
// le mot clé this : pointeur l'objet sur l'instance de point
// utilisable uniquement dans une fonction membre
class point
{
private:
    int x,y;
public:
    point(int abs=0, int ord=0) // constructeur en ligne
    {
        x=abs; y=ord;
    }
    void affiche();
};

void point::affiche()
{
    cout<<"Adresse: "<<this<<" - Coordonnées: "<<x<<" "<<y<<"\n";
}

void main()
{
    point a(5),b(3,15);
    a.affiche();
    b.affiche();
}
```