

Module M5: Structures de données et programmation C++ Élément 2: Programmation C++

**Filière : Génie Informatique, Semestre 2
Année Universitaire 2021-2022**

Pr. Rachid AIT DAOUD

Introduction

- ✓ Le langage C++ a été conçu à partir de 1982 par **Bjarne Stroustrup**, avec un objectif précis : ajouter au langage C des classes analogues à celles du langage **Simula**.
- ✓ *Pourquoi du C++*: c'est avant tout une nécessité de répondre à des besoins générés par de gros projets.
 - L'exactitude : aptitude d'un logiciel à fournir les résultats voulus, dans des conditions normales d'utilisation.
 - La robustesse: aptitude d'un logiciel à bien réagir dans des conditions anormales d'utilisation « les exceptions ».
 - **L'extensibilité**: facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des besoins.
 - **La réutilisabilité**: possibilité d'utiliser certaines parties (modules) du logiciel dans une autre application.
 - L'efficacité: temps d'exécution, taille mémoire.

Introduction

- Ceci est assez limité lorsqu'on emploie un langage simplement structuré tel que **C** ou **Turbo Pascal**.
- La programmation structurée reposait sur ce que l'on nomme souvent "l'équation de Wirth", à savoir :

Algorithmes + Structures de données = Programmes

- C++ ajoute au langage C des **spécificités supplémentaires** en lui permettant d'appliquer les concepts de la **P**rogrammation **O**rientée **O**bjets « P.O.O qui se base sur la notion d'objet. ».

Historique de la P.O.O

- La programmation orientée objet est apparue très tôt en informatique dans les années 1960 avec **Simula67**.
- Elle a trouvée son voie dans l'industrie au milieu des années 1980 avec les langages comme **Smalltalk** et **C++**.
- Les méthodes d'analyses sont arrivées à maturité relativement tardivement vers 1990: UML ...
- Ce type de programmation se base sur la notion d'objet.

Qu'est ce qu'un **Objet** ?

Chapitre 1: Spécificités de C++, notions Objet et Classe

Définition d'objet

- Un objet est une entité formée par les **données** (les champs) et des **fonctions** (méthodes) qui agissent sur ces données (notion de spécialisation des fonctions).
 - Objet = Méthodes + Champs "avec leur valeur"
 - Objet = Méthodes + Attributs (où les attributs sont les données avec leur valeur à un moment de l'application)
- Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la P.O.O. est :

$$\text{Données} + \text{Méthodes} = \text{Objet}$$

- La P.O.O permet de définir et de **modéliser une application** en terme d'objets qui collaborent entre eux. (gs)
- Exemple d'objets de la vie courante:
 - Un téléphone, ville, voiture, client, le bulletin de salaire, etc..

Attribut et État d'un objet

- Un objet est caractérisé par:
 - Les services proposés: càd les méthodes de l'objet
 - ses champs, son État et son nom.
- les champs
 - Information qui qualifie l'objet qui le contient;
 - Peut être une constante.
- État
 - Valeurs instantanées de tous les attributs d'un objet
 - Évolue au cours du temps
 - Dépend de l'histoire de l'objet

Identité d'objet

Champ variable

Champ constant

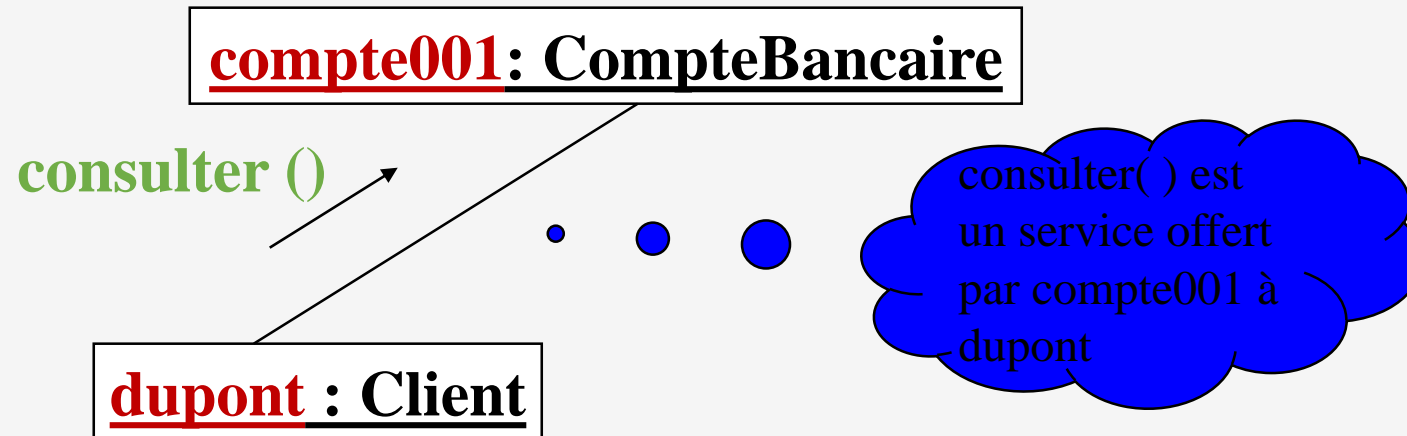
compte001 : CompteBancaire

solde : 10

DEBITAUTORISE : 1

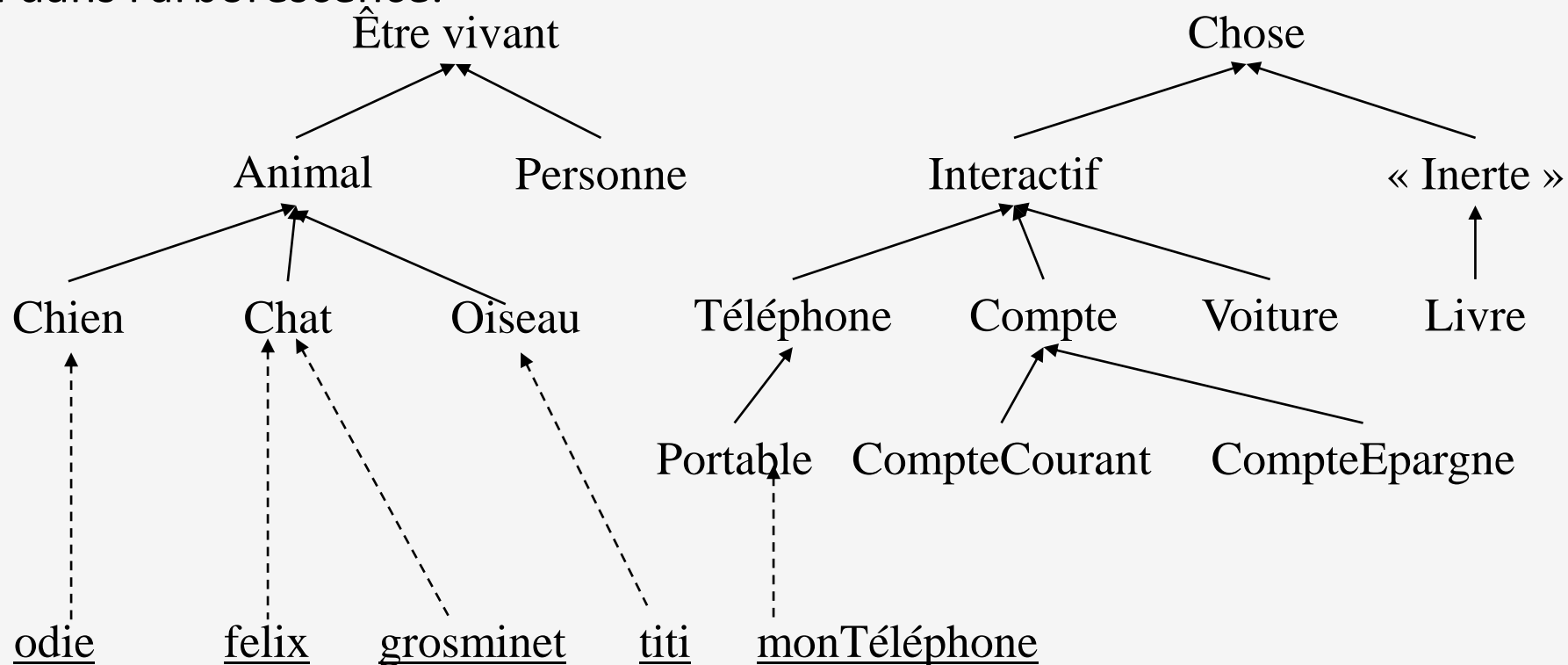
Comportement d'un objet

- Le comportement d'un objet décrit les **actions** et les **réactions** d'un objet.
 - En terme général, le comportement d'un objet représente les services offerts par un objet.
- Service = opération = méthode.
- Un comportement est déclenché par un message (invocation de service) d'un objet par un autre objet.



Les objets sont rattachés à leur famille "classe"

- On classe les objets par **famille** en faisant abstraction de quelques différences. On élabore ainsi une définition théorique de ce que doit être un objet de cette famille ou de cette **classe**.
- les familles d'objets **partagent** un ensemble **d'attributs** et de **services**, en fonction de leur position dans l'arborescence.



Les classes en C++

- Une classe est une description d'une famille d'objets, qui ont :
 - ✓ Mêmes attributs
 - ✓ Mêmes méthodes

Spécification
- Une classe est une moule d'un ensemble d'objets qui partagent une structure commune (les attributs) et un comportement commun (les méthodes).
- Une classe est une entité qui permet de regrouper un ensemble de données et les méthodes qui leurs sont attachées.
- Une classe correspond aussi a un type de donnée comme int, double, elle correspond au type des objets qui lui sont liés.

Les classes en C++

- Pour créer une classe, on utilise le **mot-clé class suivi du nom de la classe** qui commence par une **majuscule**, qui n'est pas obligatoire mais ceci rend la distinction entre les noms des classes et des objets plus simple. La définition de la classe s'écrit entre les accolades. On liste les attributs puis les méthodes dont a besoin la classe. Il y a un **point-virgule après l'accolade fermante**.
- **Une classe est constituée de** variables appelées **attributs** (ou variables membres) et de fonctions appelées **méthodes** (ou fonctions membres).

Les classes en C++

- Chaque **attribut** et chaque **méthode** d'une classe peut **posséder son propre droit d'accès**.
On peut citer les trois droits d'accès (modes d'accès) :
 - public: l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
 - private: l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. **Par défaut, tous les éléments d'une classe sont private.**
 - protected: l'attribut ou la méthode peut être appelé par les classes héritant de la classe de base.
- Les mots clés **public**, **private** et **protected** peuvent **apparaître à plusieurs reprises** dans la définition d'une classe.

Les classes en C++

- **Exemple:** Classe CompteBancaire

<u>Nom de la classe</u>	CompteBancaire	
<u>Attributs</u>	solde numCompte	private protected
<u>Opérations</u>	deposer() retirer()	public

Les classes en C++

Une classe se déclare par le mot-clé class
(définit un nouveau type de variable)

Déclaration des attributs

Syntaxe

```
type nomAttribut;
```

L'accès aux valeurs des attributs d'une instance

Exemple

```
class Rectangle {  
    ...  
};
```

```
class Rectangle {  
    double hauteur;  
    double largeur;  
};
```

```
nom_instance.nom_attribut
```

Définition des méthodes

Syntaxe

```
typeRetour nomMethode (typeParam1 nomParam1, ...)  
{  
    // corps de la méthode  
    ...  
}
```

Exemple

```
class Rectangle {  
    double hauteur;  
    double largeur;  
    double surface() {  
        return hauteur * largeur;  
    }  
};
```

Portée des attributs

Les attributs d'une classe constituent des variables **directement accessibles dans toutes les méthodes** de la classe. On parle de « portée de classe » (« **variables globales** à la classe »).

=> Il n'est donc pas nécessaire de les passer comme arguments des méthodes.

Les méthodes sont :

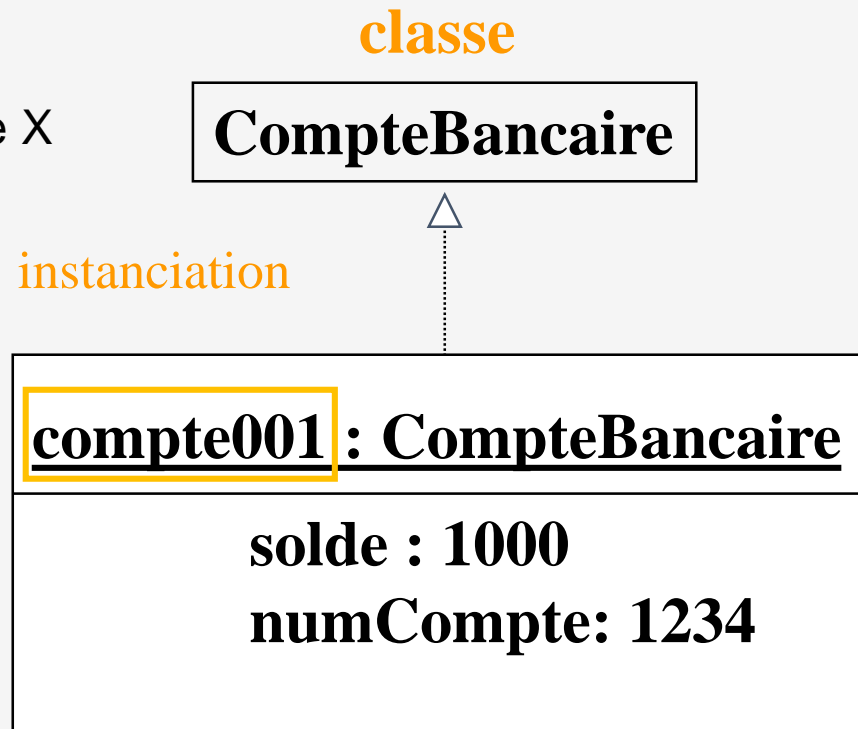
- des fonctions propres à la classe
- qui ont accès aux attributs de la classe

Notion d'instance

- Chaque objet appartient à une classe
- On dit que:
 - obj1 est un **objet** de la classe Xou
 - Obj1 est une **instance** de la classe X

Instance ou objet

Compte001 est un objet de type CompteBancaire
on dit aussi qu'on a instancié un objet ou crée
une instance de la classe CompteBancaire dont
l'identifiant est **compte001**



Notion d'instance

1. Définition de la classe

Les droits d'accès

Chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès: public, private et protected

```
class Rectangle {  
    public:  
        double hauteur;  
        double largeur;  
        double surface() {  
            return hauteur * largeur;  
        }  
};
```

2. Déclaration d'une instance (un objet)

(se fait de façon similaire à la déclaration d'une variable)

Syntaxe

```
nom_classe nom_instance;
```

Exemple

```
Rectangle rect;
```

3. Appels aux méthodes

Syntaxe

```
nom_instance.nom_methode(val_arg1, ...)
```

Exemple

```
rect.surface(4,2)
```


Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

- Par défaut, tous les attributs sont privés ?
- Et si on a besoin de les utiliser depuis l'extérieur de la classe ?!
- Si le programmeur *le juge utile*, il **inclut les méthodes publiques nécessaires** ...

1. Manipulateurs (« méthodes set ») :

- Modification (action)
- Affectation de l'argument à une variable d'instance précise
 - **void** setHauteur(**double** h) { hauteur = h;}
 - **void** setLargeur(**double** L) { largeur = L;}

2. Accesseurs (« méthodes get ») :

- Consultation (prédicat)
- Retour de la valeur d'une variable d'instance précise
 - **double** getHauteur() { return hauteur;}
 - **double** getLargeur() { return largeur;}

Méthodes «get» et «set» (« accesseurs » et « manipulateurs »)

Exemple de programme

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

class Rectangle {
private:
double hauteur; double largeur;
public:
double surface() { return hauteur * largeur; }
double getHauteur() { return hauteur; }
double getLargeur() { return largeur; }

void setHauteur(double h){ hauteur = h; }
void setLargeur(double l){ largeur = l; }
};
```

```
int main()
{
Rectangle rect1;

rect1.setHauteur(3.0); rect1.setLargeur(4.0);

cout << "hauteur : "<< rect1.getHauteur()<< endl;

return 0;
}
```

Les spécificités de C++

C++ présente par rapport au C, des extensions qui ne sont pas véritablement orientées P.O.O. Voici un bref résumé :

- Surdéfinition des fonctions : attribution d'un même nom à différentes fonctions, la reconnaissance de la fonction réellement appelée se faisant d'après **le type et le nombre des arguments figurant dans l'appel** (on parle parfois de signature).
- Nouveaux opérateurs de gestion dynamique de la mémoire : *new* et *delete*.
- Plus grande liberté dans l'emplacement des déclarations.
- Notion de référence facilitant la mise en œuvre de la transmission d'arguments par adresse.

Les spécificités de C++ (Déclaration de fonctions)

En C++, toute fonction utilisée dans un fichier source doit obligatoirement avoir fait l'objet :

- soit d'une **déclaration** sous forme d'un **prototype** (il précise à la fois le nom de la fonction, le type de ses arguments éventuels et le type de sa valeur de retour), comme dans cet exemple :

float fexp (int, double, char *) ;

- soit d'une **définition** préalable au sein du même fichier source (ce dernier cas étant d'ailleurs peu conseillé, dans la mesure où des problèmes risquent d'apparaître dès lors qu'on sépare ladite fonction du fichier source en question).

Les spécificités de C++ (Notation de référence)

- En C, les arguments et la valeur de retour d'une fonction sont transmis par valeur. Pour simuler en quelque sorte ce qui se nomme "transmission par adresse" dans d'autres langages, il est alors nécessaire de "jongler" avec les pointeurs (la transmission se fait toujours par valeur mais, dans ce cas, il s'agit de la valeur d'un pointeur).
- En C++, le principal intérêt de la notion de référence est qu'elle permet de laisser le compilateur mettre en œuvre les "bonnes instructions" pour assurer un transfert par adresse. Autrement dit, C++ permet de demander au compilateur de prendre lui même en charge la transmission des arguments par adresse : on parle alors de transmission **d'argument par référence**.

Les spécificités de C++ (Notation de référence)

- On considère les déclarations suivantes:

`int n=1,`

`int &p = n; // initialisation et déclaration de référence`

- ✓ p est une référence à la variable n.
- ✓ p est un synonyme ou alias de la variable n.
- ✓ n et p ont même type et même place mémoire.

`p=3;`

`cout << n << endl;`

- ✓ // toute modification de n est répercutée sur p et réciproquement
- ✓ Résultat: n=3

- Une fois une référence est déclarée et initialisée, on ne peut la modifier: p est une référence sur n et non sur autre variable.

`int &p = 3 // Incorrecte!!` initialisation d'une référence par une constante

`float x = 5; p = x; // interdit` car p une référence à un int

Les spécificités de C++ (Notation de référence)

Exemple:

Ecrire un programme qui permet de faire :

- Déclarer et initialiser un entier
- Déclarer une référence vers cet entier
- Déclarer un pointeur vers cet entier

Dans les deux cas, imprimer la variable, l'adresse de la variable, la valeur pointée

```
#include <iostream>

using namespace std;

int main()
{
    int AA=10;
    int &bb = AA; // bb est une référence à la variable AA
    int *cc = &AA; // cc est un pointeur qui pointe vers AA
    cout << "valeur de bb = " << bb << " adr de bb = " << &bb << endl;
    cout << "valeur de cc = " << cc << " adr de cc = " << &cc << " val de la variable pointée par cc = " << *cc << endl;
    return 0;
}
```

Les spécificités de C++ (Notation de référence)

- Le programme suivant montre comment appliquer un tel mécanisme à une fonction de permutation :

la notation **int & a** signifie que **a** est une information de type int transmise par référence.

1- Autrement dit, il suffit d'avoir fait ce choix de transmission par référence au niveau de **l'en-tête de la fonction** pour que le processus soit entièrement pris en charge par le compilateur.

2- Le même phénomène s'applique au niveau de l'utilisation de la fonction. Il suffit en effet d'avoir spécifié, dans le **prototype**, les arguments (ici, les deux) que l'on souhaite voir transmis par référence.

3- Au niveau de l'appel, nous n'avons plus à nous préoccuper du mode de transmission utilisé.

Remarque:

- a et b sont des références, alias ou synonymes des variables réels n et p.
- Le compilateur, lui-même, transmet les adresses des paramètres réels lors de l'appel de la fonction

```
#include <iostream>
using namespace std ;
main()
{ void echange (int &, int &) ;
  int n=10, p=20 ;
  cout << "avant appel : " << n << " " << p << "\n" ;
  echange (n, p) ; // attention, ici pas de &n, &p
  cout << "apres appel : " << n << " " << p << "\n" ;
}

void echange (int & a, int & b)
{ int c ;
  cout << "debut echange : " << a << " " << b << "\n" ;
  c = a ; a = b ; b = c ;
  cout << "fin echange : " << a << " " << b << "\n" ;
}

avant appel : 10 20
debut echange : 10 20
fin echange : 20 10
apres appel : 20 10
```

Utilisation de la transmission d'arguments par référence en C++¹⁴

Les spécificités de C++ (Les entrées/Sorties)

❑ les entrées/sorties

Pour effectuer **les entrées/sorties (flux)** dans les programmes, on utilise les fonctions

- **cout** (pour afficher à l'écran des caractères)
- **cin** (pour lire sur le clavier: entrer des valeurs ou caractère)

Remarques:

il faut inclure le fichier `<iostream.h>` au début du programme pour pouvoir utiliser les entrées/sorties

- L'opérateur d'insertion `<<` permet d'envoyer des valeurs dans un flot de sortie
- L'opérateur d'extraction `>>` permet d'extraire des valeurs d'un flot d'entrée.

Exemple

```
#include<iostream.h>
```

```
void main(){  
    int a, b, c;  
    cout<< "Entrer la valeur de a"<< endl;  
    cin >> a;  
    cout<< "Entrer la valeur de b"<< endl;  
    cin >>b;  
    if (a >b) c = a + b;  
    if (a == b) c = a + 1;  
    if (a < b) c = a - b;  
    cout <<"la valeur de c est: " << c<< endl;  
}
```

Les spécificités de C++ (Visibilité des variables)

❑ Définition des variables

En C++, on peut déclarer les variables ou fonctions n'importe où dans le code.

La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.

❑ Visibilité des variables

L'opérateur de résolution de portée `::` permet d'accéder aux variables globales plutôt qu'aux variables locales.

Remarque:

- En fait, on utilise beaucoup cet opérateur pour **définir hors d'une classe les fonctions membres** ou pour accéder à un identificateur dans un espace de noms.
- L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.

```
#include <iostream.h>
int i = 11; // variable globale
void main() {
    int i = 34;
    { int i = 23;
      ::i = ::i + 1;
      cout << ::i << " " << i << endl;
    }
    cout << ::i << " " << i << endl;
}
```

```
/*-- résultat de l'exécution --*/
12 23
12 34
```

Les spécificités de C++ en termes POO (membres données, fonctions membres)

- C++ dispose de la notion de classe (généralisation de la notion de type défini par l'utilisateur).

Une classe comportera :

- la description d'une structure de données (les attributs)
- des méthodes
- Sur le plan du vocabulaire, C++ utilise des termes qui lui sont propres. On parle en effet de :
 - "**membres données**" pour désigner les différents membres de la structure de données associée à une classe,
 - "**fonctions membres**" pour désigner les méthodes.
- A partir d'une classe, on pourra "instancier" des objets (nous dirons généralement créer des objets) :
 - soit par des déclarations usuelles (de type classe),
 - soit par allocation dynamique, en faisant appel au nouvel opérateur ***new***.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

□ Encapsulation des données:

- Ceci signifie qu'il n'est pas possible pour l'utilisateur d'un objet, **d'accéder directement aux données** (les modes d'accès protected et private).
- L'utilisateur doit passer par des méthodes spécifiques écrites par le concepteur de l'objet, et qui servent **d'interface** entre l'objet et ses utilisateurs.
- L'intérêt de cette technique est évident. L'utilisateur ne peut pas intervenir directement sur les données d'un objet, ce qui diminue les risques d'erreur.
→ En général cela permet de protéger les variables membres contre des modifications incohérentes.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

Exemple: soit la déclaration de la structure **point** suivante:

En C++ il est possible dans une structure, d'associer aux données constituées par ses membres des méthodes qu'on nommera "fonctions membres".

N.B: les données ne sont pas encapsulées dans une structure.

```
struct point  
{ int x ;  
  int y ;  
};
```

Q1- Déclarer une structure comportant des fonctions membres (Supposons que nous souhaitons associer à la structure point précédente trois fonctions):

1. **initialise** pour attribuer des valeurs aux "coordonnées" d'un point ;
2. **deplace** pour modifier les coordonnées d'un point ;
3. **affiche** pour afficher un point : ici, nous nous contenterons, par souci de simplicité, d'afficher les coordonnées du point.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

Q1- Déclarer une structure comportant des fonctions membres (Supposons que nous souhaitons associer à la structure point précédente trois fonctions):

1. ***initialise*** pour attribuer des valeurs aux "coordonnées" d'un point ;
2. ***deplace*** pour modifier les coordonnées d'un point ;
3. ***affiche*** pour afficher un point : ici, nous nous contenterons, par souci de simplicité, d'afficher les coordonnées du point.

```
struct point
{ int x ;
  int y ;
};
```

```
/* ----- Déclaration du type point ----- */
struct point
{ /* déclaration "classique" des données */
  int x ;
  int y ;
  /* déclaration des fonctions membre
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
```

Notions d'association et interface (méthodes) */

La méthode membre *initialise* va affecter aux membres x et y les valeurs reçues en arguments.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

Q2- En utilisant l'opérateur de "résolution de portée", définir les trois fonctions membres.

```
/* --- Définition des fonctions membres du type point -- */
#include <iostream>
using namespace std ;
void point::initialise (int abs, int ord)
{
    x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
    x += dx ; y += dy ;
}
void point::affiche ()
{
    cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Dans l'en-tête, le nom de la fonction est :

point::initialise

Le symbole **::** sert à modifier la portée d'un identificateur. Ici, il signifie que l'identificateur **initialise** concerné est celui défini dans la structure ***point***.

Q3- Qu'est ce qu'il va passer à l'absence du préfix (***point::***)

Réponse: En l'absence de ce "préfixe" (point::), nous définirions effectivement une fonction nommée initialise, mais celle-ci ne serait plus associée à point ; il s'agirait d'une fonction "ordinaire" nommée initialise, et non plus de la fonction membre initialise de la structure point.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

```
/* --- Définition des fonctions membres du type point -- */
#include <iostream>
using namespace std ;
void point::initialise (int abs, int ord)
{
  x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{
  x += dx ; y += dy ;
}
void point::affiche ()
{
  cout << "Je suis en " << x << " " << y << "\n" ;
}
```

Q4- Que représente les symboles *abs* et *x*?

Réponse: Le symbole *abs* désigne, classiquement, la valeur reçue en premier argument. Mais *x*, quant à lui, n'est ni un argument ni une variable locale. En fait, *x* désigne le membre *x* correspondant au type ***point*** (cette association étant réalisée par le `point::` de l'en-tête).

Q5- Déclarer deux structures nommées *a* et *b* de type ***point***.

```
point a, b;
```

Q6- Comment peut 'on accéder aux membres *x* et *y* de nos structures *a* et *b*?

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

Q5- Déclarer deux structures nommées a et b de type *point*. `point a, b;`

Q6- Comment peut 'on accéder aux membres x et y de nos structures a et b?

1 `a.x = 5 ;`

2 `a.initialise (5,2) ;`

1^{er} cas: L'accès aux membres x et y de nos structures a et b pourrait se dérouler comme en C. Accès direct aux données, sans passer par l'intermédiaire des méthodes. Certes, nous ne respecterions pas le principe d'encapsulation, mais dans ce cas précis (de structure et pas encore de classe), ce serait accepté en C++.

2^{ème} cas: Appeler la fonction membre **initialise** pour la structure **a**, en lui transmettant en arguments les valeurs 5 et 2. le préfixe **a** va préciser à la fonction membre **initialise** quelle est la structure sur laquelle elle doit opérer.

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

❑ Encapsulation des données:

Remarque importante:

1. Un appel tel que `a.initialise (5,2)` ; pourrait être remplacé par :

`a.x = 5 ; a.y = 2 ;`

Nous verrons précisément qu'il n'en ira plus de même dans le cas d'une (**vraie**) **classe**, pour peu qu'on y ait convenablement **encapsulé** les données.

2. En jargon P.O.O., on dit également que `a.initialise (5, 2)` constitue **l'envoi d'un message** (`initialise`, accompagné des informations 5 et 2) à **l'objet a**.

Les spécificités de C++ en

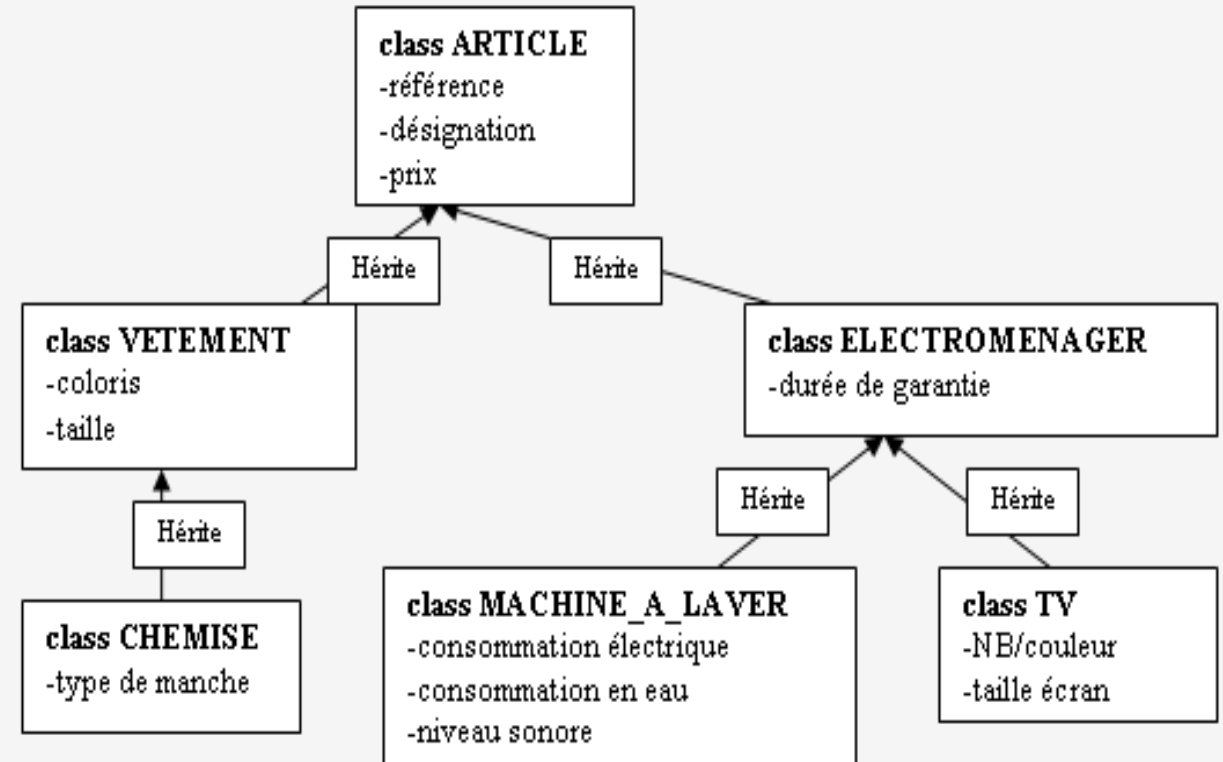
Exemple récapitulatif

```
#include <iostream>
using namespace std ;
/* ----- Déclaration du type point ----- */
struct point
{ /* déclaration "classique" des données */
int x ;
int y ;
/* déclaration des fonctions membres (méthodes) */
void initialise (int, int) ;
void deplace (int, int) ;
void affiche () ;
} ;
/* ----- Définition des fonctions membres du type point ---- */
void point::initialise (int abs, int ord)
{ x = abs ; y = ord ;
}
void point::deplace (int dx, int dy)
{ x += dx ; y += dy ;
}
void point::affiche ()
{ cout << "Je suis en " << x << " " << y << "\n" ;
}
main()
{ point a, b ;
a.initialise (5, 2) ; a.affiche () ;
a.deplace (-2, 4) ; a.affiche () ;
b.initialise (1,-1) ; b.affiche () ;
}
```

Les spécificités de C++ en termes POO (l'Encapsulation, l'Héritage et le Polymorphisme)

□ Héritage:

- L'héritage permet la définition d'une nouvelle classe à partir d'une classe **existante**.
- Il est alors possible de lui adjoindre de **nouvelles données, de nouvelles fonctions membres** (méthodes) pour la **spécialiser**.
- L'héritage multiple: permet à une classe d'hériter **simultanément de plusieurs** autres classes.




Les spécificités de C++ en termes POO (Gestion dynamique de la mémoire)

□ Gestion dynamique de la mémoire:

- En C++, les fonctions malloc, calloc... et free sont remplacées avantageusement par les opérateurs *new* et *delete*.
- Si **type** représente la description d'un type absolument quelconque et si **n** représente une expression d'un type entier (short, long, ...), l'expression : **new type [n]**, alloue l'emplacement nécessaire pour **n** éléments du type indiqué et fournit en résultat un pointeur (de type **type ***) sur le premier élément.

Exemple:

add = malloc (sizeof (double) * 100) ;  add = new double [100] ;

Les spécificités de C++ en termes POO (Constructeur, Destructeur)

- Comme la plupart des langages objets, C++ permet de définir ce que l'on nomme des "**constructeurs**" de classe.
- Un constructeur est une **fonction membre particulière** qui est **exécutée** au moment de **la création d'un objet de la classe**.
- Le constructeur peut notamment prendre en charge **l'initialisation** d'un objet (c'est-à-dire sa mise dans un état initial permettant son bon fonctionnement ultérieur).
- L'existence d'un constructeur garantit que l'objet sera toujours initialisé, ce qui constitue manifestement une sécurité.
- De manière similaire, une classe peut disposer d'un "**destructeur**", fonction membre exécutée **au moment de la destruction d'un objet**. Celle-ci présentera surtout un intérêt dans le cas d'objets effectuant des allocations dynamiques d'emplacements ; ces derniers pourront être libérés par le destructeur.

Les spécificités de C++ en termes POO (Fonctions amies)

- Une des originalités de C++ par rapport à d'autres langages de P.O.O. réside dans la possibilité de définir des "**fonctions amies** d'une classe".

Fonctions amies ??

- Les fonctions amies sont des fonctions "usuelles" (qui ne sont donc pas des fonctions membres d'une classe) qui sont autorisées (par une classe) à accéder aux données (encapsulées) de la classe.

→ **Certes, le principe d'encapsulation est violé, mais uniquement par des fonctions dûment autorisées à le faire.**

Chapitre 2:

- **Constructeur, destructeur**
- **Constructeur par copie**
- **Objets transmis en argument d'une fonction membre**
- **Fonctions et classes amies**
- **Surdéfinition d'opérateurs**

Chapitre 2: A- Notion de Constructeur/Destructeur

- ❑ Un constructeur est une fonction membre *systématiquement exécutée* lors de la création d'un objet (en statique ou en dynamique).
 - Un constructeur porte le même nom que la classe dans laquelle il est défini.
 - Un constructeur n'a pas de type de retour (même pas void).
 - Un constructeur peut contenir ou pas des arguments.
 - Un constructeur permet d'initialiser les champs de la classe.
- ** Attention aux pointeurs**, aux tableaux et tout objet qui doit s'allouer dynamiquement de la mémoire, il faut toujours **les initialiser dans les constructeurs**.
-
- ❑ le constructeur par défaut (constructeur sans arguments) est appelé automatiquement à la création d'un objet (si pas de constructeur avec arguments).

Exemple 0

```
class Point {  
    double X; // coordonnées x et y  
    double Y;  
    double LongueurVecteurAssocie;  
    public :  
    Point(){ // constructeur par défaut (sans arguments)  
        X = 0;  
        Y = 0;  
        LongueurVecteurAssocie = module(); // appel fonction membre pour  
        initialisé un champs  
    } // remarquez que le constructeur appelle une fonction membre dont le  
    prototype est fourni plus bas  
    double module();  
};
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Constructeur sans arguments

Dans l'exemple de la structure *point*, le constructeur remplace la fonction membre *initialise*.

Exemple 1:



```
#include <iostream> // constructeur
#include <conio.h>
using namespace std;
class point
{
    int x,y;
public:
    point(); // noter le type du constructeur (pas de "void")
    void deplace(int,int);
    void affiche();
};
////////////////////////////////////
point::point() // initialisation par défaut grace au constructeur
{x = 20; y = 10;}
////////////////////////////////////
void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}
////////////////////////////////////
void point::affiche()
{cout<<"Je suis en "<<x<<" "<<y<<endl;}
////////////////////////////////////
main()
{
    point a,b; // les deux points sont initialisés en 20,10
    a.affiche();
    a.deplace(17,10);
    a.affiche();
    b.affiche();
    getch();
}
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Constructeur avec arguments

Exemple 2:



```
class point
{
    int x,y;
public:
    // noter le type du constructeur (pas de "void")
    point(int,int);
    void deplace(int,int);
    void affiche();
};
point::point(int abs,int ord)
{x = abs; y = ord;}
```

Initialisation par défaut grâce au constructeur, ici paramètres à passer

```
main()
{
    point a(20,10),b(30,20);
```

Les deux points sont initialisés : **a** en 20,10 **b** en 30,20

Chapitre 2: A- Notion de Constructeur/

Surdéfinition de constructeur

Nous avons déjà vu comment C++ nous autorise à surdéfinir les fonctions ordinaires. Cette possibilité s'applique également aux fonctions membres d'une classe, y compris au constructeur.
→ **Surdéfinition d'un constructeur permet de créer un objet de plusieurs façons différentes.**

```
Je suis en : 0 0
Point b - Je suis en : 5 5
Hello ---- Je suis en : 3 12

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
public :
point () ; // constructeur 1 (sans arguments)
point (int) ; // constructeur 2 (un argument)
point (int, int) ; // constructeur 3 (deux arguments)
void affiche () ; // fonction affiche 1 (sans arguments)
void affiche (char *) ; // fonction affiche 2 (un argument 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 () // fonction affiche 1
{ cout << "Je suis en : " << x << " " << y << "\n" ;
}
void point::affiche (char * message) // fonction affiche 2
{ cout << message ; affiche () ;
}
main()
{ point a ; // appel constructeur 1
a.affiche () ; // appel fonction affiche 1
point b (5) ; // appel constructeur 2
b.affiche ("Point b - ") ; // appel fonction affiche 2
point c (3, 12) ; // appel constructeur 3
c.affiche ("Hello ---- ") ; // appel fonction affiche 2
}
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Constructeur avec valeurs par défaut

Définition du constructeur avec valeurs par défaut

```
Point (double abs = 0, double ord = 0){  
    X = abs;  
    Y = ord;  
}
```

→ Avec cette déclaration, l'utilisation d'un constructeur par défaut n'est pas utile, car les valeurs par défaut des champs de la classe sont données par ce constructeur.

Chapitre 2: A- Notion de Constructeur/Destructeur

Appel du constructeur

L'appel du constructeur se fait lors de la création de l'objet. De ce fait, l'appel du constructeur est différent selon que l'objet est créé de façon **statique** ou **dynamique** :

- **en statique** : le constructeur est appelé grâce à une instruction constituée du nom de la classe, suivie par le nom que l'on donne à l'objet, et les paramètres entre parenthèses.

`point a(20,10);`

- **en dynamique** : le constructeur est appelé en **définissant un pointeur vers un objet du type désiré** puis en lui affectant la valeur retournée par l'opérateur *new*.

`point *pa;`

`pa = new point(20,10);`

Chapitre 2: A- Notion de Constructeur/Destructeur

Exercice 1

Utiliser la classe « point » Exemple 2.

Ecrire une fonction de prototype ***void test()*** dans laquelle on déclare un point ***p***, on l'initialise, on l'affiche , on le déplace et on l'affiche à nouveau. Le programme principal ***main*** ne contient que l'appel à ***test***.

Chapitre 2: A- Notion de Constructeur/Destructeur

Solution:

```
#include <iostream>
#include <conio.h>
using namespace std;
class point
{
    int x,y;
public: point(int,int); // noter le type du constructeur (pas de "void")
    void deplace(int,int);
    void affiche();
};
point::point(int abs,int ord) // initialisation par défaut
{x = abs; y = ord;} // grâce au constructeur, ici paramètres à passer
void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}
void point::affiche()
{cout<<"Je suis en "<<x<<" "<<y<<"\n";}
void test()
{
    point p(1,4);
    p.affiche();
    p.deplace(17,10);
    p.affiche();
}
main()
{test();
getch();
}
```


Chapitre 2: A- Notion de Constructeur/Destructeur

Destructeur

- Un destructeur est une fonction membre qui est automatiquement appelée **au moment de la destruction d'un objet**, avant la libération de l'espace mémoire associée à l'objet.
- La libération de la mémoire allouée **dynamiquement** étant confié dans ce cas au destructeur.
- le destructeur est indispensable dès qu'un **objet** ou **champs** est amené à allouer dynamiquement de la mémoire.

Par convention:

- Il porte le même nom que la classe précédé du signe "*tilde*" *~*.

Par définition:

- Un destructeur ne prend aucun paramètre et ne renvoie aucune valeur de retour.
- Sa déclaration se fait de la façon suivante :

```
class MaClasse {  
    public:  
        ~ MaClasse(); // destructeur  
};
```

Chapitre 2: A- Notion de Constructeur/Destructeur

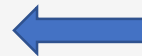
Destructeur

Le destructeur, comme dans le cas du constructeur, est appelé différemment selon que l'objet auquel il appartient a été créé de façon *statique* ou *dynamique*.

- Le destructeur d'un objet créé de façon statique est appelé de façon **implicite** dès que le programme quitte **la portée dans lequel l'objet existe**.
- Le destructeur d'un objet créé de façon dynamique doit être appelé grâce au mot clé ***delete***, qui permet de libérer la mémoire occupée par l'objet.

```
Je suis en 1 4
Je suis en 3 7
Frapper une touche...
destruction du point x =3 y=7
Je suis en 5 10
Frapper une touche...
destruction du point x =5 y=10
Frapper une touche...
destruction du point x =1 y=4

Process returned 0 (0x0)   execution time : 8.534 s
Press any key to continue.
```



```
#include <iostream>
#include <conio.h>
using namespace std;
class point
{
int x,y;
public:
point(int,int); //constructeur
void deplace(int,int);
void affiche();
~point(); // noter le type du destructeur
};
// initialisation par défaut grace au
//constructeur, ici paramètres à passer
point::point(int abs,int ord)
{x = abs; y = ord;}
////////////////////////////////////
void point::deplace(int dx,int dy)
{x = x+dx; y = y+dy;}
////////////////////////////////////
void point::affiche()
{cout<<"Je suis en "<<x<<" "<<y<<"\n";}
////////////////////////////////////
point::~~point()
{cout<<"Frapper une touche...\n";getch();
cout<<"destruction du point x ="<<x<<" y="<<y<<"\n";}
////////////////////////////////////
void test()
{
point u(3,7);
u.affiche();
}
////////////////////////////////////
main()
{point a(1,4);a.affiche();
test();
point b(5,10);b.affiche();
getch();}
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Les membres données statiques

Le qualificatif **static** pour un membre donnée

- Lors de la création de différents objets d'une même classe, chaque objet possède ses propres membres données.
- Par exemple, si nous avons défini une classe myclass par :

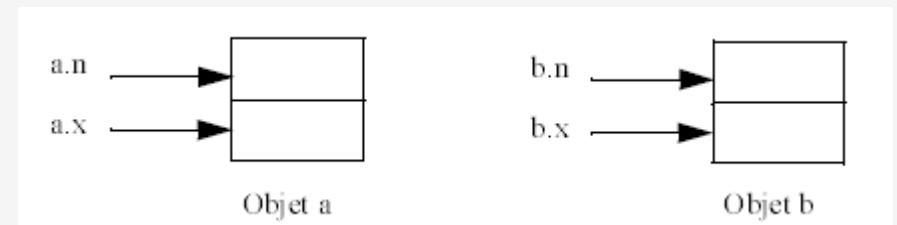
1

```
class myclass
{
  int n ;
  float x ;
  .....
} ;
```

2

une déclaration telle que
:
myclass a, b ;

3



Chapitre 2: A- Notion de Constructeur/Destructeur

Les membres données statiques

Le qualificatif **static** pour un membre donnée

Pour permettre à plusieurs objets de partager des données il suffit de déclarer avec le qualificatif **static** les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe.

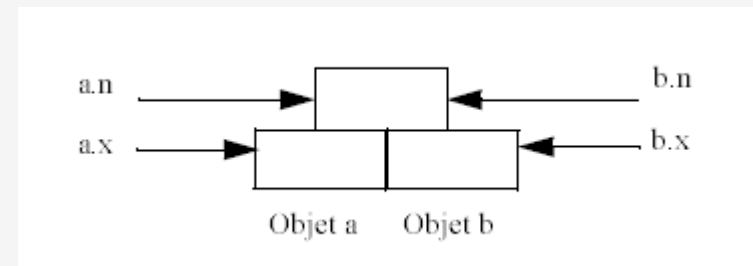
1

```
class myclass
{
static int n ;
float x ;
.....
} ;
```

2

une déclaration telle que
:
myclass a, b ;

3



1. Une variable statique: c'est une variable qui est partagée par tout objet de la classe
2. Une variable statique doit être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que : `int myclass::n = 7;`

Chapitre 2: A- Notion de Constructeur/Destructeur

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>
using namespace std ;
class point // Une classe point contenant seulement :
{ int x, y ;
public :
point (int abs=0, int ord=0) // Un constructeur ("inline")
{ x=abs; y=ord ; }
void affiche () ; // Une fonction affiche
} ;
void point::affiche ()
{ cout << "Adresse : " << this << " - Coordonnees " << x << " " << y << "\n" ;
}
main() // Un petit programme d'essai
{ point a(5), b(3,15) ;
a.affiche () ;
b.affiche () ;
}
```

```
Adresse : 0x22fe48 - Coordonnees 5 0
Adresse : 0x22fe40 - Coordonnees 3 15
```

```
Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Exemple: utilisation de destructeur

```
#include <iostream>
using namespace std;
class Point2{
int X, Y;
static int ctr; // compteur du nombre d'objet créés.
public:
Point2(); // constructeur par défaut
~Point2(); // destructeur
};
// initialisation de la variable statique à l'extérieur du constructeur
int Point2::ctr=0;
Point2::Point2(){
X=0;
Y = 0;
cout << "++ construction: il y a maintenant" << ++ctr << "objet(s)" << endl;
Point2::~~Point2(){
cout << "-- destructeur: il y a maintenant " << --ctr << " objet(s)" << endl;
cout << " adresse de l'objet détruit est: "<<this << endl;}
// fonction qui crée deux objets (création statique )
void fct(){
Point2 u, v;
cout << " sortie de la fonction" << endl;}
int main(){
Point2 U; // le constructeur crée 1 objet
cout << "appel de la fonction creation" << endl;
fct(); // la fonction crée deux objets
Point2 V; // le constructeur crée 1 objet
return 0;
}
```

```
++ construction: il y a maintenant1objet(s)
appel de la fonction creation
++ construction: il y a maintenant2objet(s)
++ construction: il y a maintenant3objet(s)
sortie de la fonction
-- destructeur: il y a maintenant 2 objet(s)
adresse de l objet détruit est: 0x22fde0
-- destructeur: il y a maintenant 1 objet(s)
adresse de l objet détruit est: 0x22fde8
++ construction: il y a maintenant2objet(s)
-- destructeur: il y a maintenant 1 objet(s)
adresse de l objet détruit est: 0x22fe30
-- destructeur: il y a maintenant 0 objet(s)
adresse de l objet détruit est: 0x22fe38
Process returned 0 (0x0) execution time : 0.012 s
Press any key to continue.
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Exemple Allocation dynamique

En utilisant la classe suivante, définir ses fonctions membres, le programme principal **main** doit contenir la déclaration et l'affichage des objets créés.

Le constructeur calcul(int, int) initialise la table de multiplication d'un nombre entier.

Par exemple la déclaration *calcul suite1(10, 5)* donne la table de multiplication du nombre 5 (0,5,10,15, ..., 50)

```
class calcul
{int nbval,*val;
public:
calcul(int,int); // constructeur
~calcul(); // destructeur
void affiche();
};
```

Chapitre 2: A- Notion de Constructeur/Destructeur

Exemple Allocation dynamique: Calcul table de multiplication

A retenir:

- Lorsque les membres données d'une classe sont des *pointeurs*, le constructeur est utilisé pour **l'allocation dynamique de mémoire sur ce pointeur**.
- Le destructeur est utilisé pour **libérer la place**.

```
// Allocation dynamique de données membres
#include <iostream>
#include <conio.h>
using namespace std;
class calcul
{int nbval,*val;
public: calcul(int,int); // constructeur
~calcul(); // destructeur
void affiche();
};

////////////////////////////////////
calcul::calcul(int nb,int mul) //constructeur
{int i;
nbval = nb;
val = new int[nbval]; // reserve de la place
for(i=0;i<nbval;i++)
    val[i] = i*mul;
}

////////////////////////////////////
calcul::~~calcul()
{delete val;} // abandon de la place reservee
////////////////////////////////////
void calcul::affiche()
{int i;
for(i=0;i<nbval;i++)cout<<val[i]<<" ";
cout<<"\n";
}

main()
{
calcul suite1(10,4);
suite1.affiche();
calcul suite2(6,8);
suite2.affiche();
getch();}
```


Chapitre 2: B- PROPRIETES DES FONCTIONS MEMBRES

❑ FONCTIONS MEMBRES « EN LIGNE »

- Le langage C++ autorise la description des fonctions membres dès **leur déclaration** dans la classe. On dit que l'on écrit une fonction « **inline** ».
- Une fonction en ligne se définit et s'utilise comme une fonction ordinaire, à la seule différence qu'on fait précéder son en-tête de la spécification **inline**.

La présence du mot **inline** demande au compilateur de traiter une telle fonction différemment d'une fonction ordinaire. A chaque appel de cette fonction, le compilateur devra incorporer au sein du programme les instructions correspondantes (le corps de la fonction). Autrement dit, à chaque appel, il y a **génération** du code de la fonction et **non appel** à un sous-programme.

- Le mécanisme habituel de gestion de **l'appel** et du **retour** n'existera plus (il n'y a plus besoin de sauvegardes, copies...), ce qui permet une économie de temps.
- En revanche, les instructions correspondantes seront générées à chaque appel, ce qui consommera une quantité de mémoire croissante avec le nombre d'appels.

Chapitre 2: B- PROPRIETES DES FONCTIONS MEMBRES

❑ FONCTIONS MEMBRES « EN LIGNE »

- Pour rendre **en ligne** une fonction membre, on peut :

- Soit fournir directement la définition de la fonction dans la déclaration même de la classe (dans ce cas, le qualificatif ***inline*** n'a pas à être utilisé)

- Soit procéder comme pour une fonction ordinaire en fournissant une définition en dehors de la déclaration de la classe ; dans ce cas, le qualificatif ***inline*** doit apparaître **à la fois** devant la déclaration et devant l'en-tête.

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
public :
    point () { x = 0 ; y = 0 ; }           // constructeur 1 "en ligne"
    point (int abs) { x = y = abs ; }      // constructeur 2 "en ligne"
    point (int abs, int ord) { x = abs ; y = ord ; } // constructeur 3 "en ligne"
    void affiche (char * = "") ;
} ;
```

```
class point
{ .....
public :
    inline point () ;
    .....
} ;
inline point::point () { x = 0 ; y = 0 ; }
.....
```

Chapitre 2: B- PROPRIETES DES FONCTIONS MEMBRES

❑ OBJETS TRANSMIS EN ARGUMENT D'UNE FONCTION MEMBRE

- Dans les exemples précédents, les fonctions membres recevaient :
 - un **argument implicite** du type de leur classe, à savoir **l'adresse** de l'objet l'ayant appelé,
 - un certain nombre d'arguments qui étaient d'un type "ordinaire" (c'est-à-dire autre que classe).

Exemple:

```
point a;  
a.deplace(2,10)
```

- Une fonction membre peut, outre l'argument implicite, recevoir un ou plusieurs **arguments du type de sa classe**.

Objets transmis en arguments
d'une fonction membres

=

Quand on passe comme paramètre à une fonction membre un objet de la classe à laquelle appartient cette fonction:

Exemple:

```
point a;  
point b(4,7)  
a.compare(b)
```

- a: **objet fourni implicitement** lors de l'appel de la fonction membre (ses membres sont désignés, comme d'habitude, par **x** et **y**)
- b: **objet fourni en argument** dont les membres sont désignés par b.x et b.y

Chapitre 2: B- PROPRIETES DES FC

❑ Mode de transmission des objets en argu

1- Passage par valeur

Quel est le résultat de ce programme ?

```
a et b:0 ou 0  
a et c:1 ou 1
```

N.B: La notation « pt.x » ou « pt.y » n'est autorisée qu'à l'intérieur d'une fonction membre (x et y membres privés de la classe).

```
#include <iostream>
#include <conio.h>
using namespace std;
class point
{
    int x,y;
public:
    point(int abs = 0,int ord = 2) // constructeur
    {x=abs;y=ord;}
    int coincide(point);
};

////////////////////
int point::coincide(point pt)
{if ((pt.x == x) && (pt.y == y)) return(1);else return(0);}
main()
{
    int test1,test2;
    //a(0,2)  b(1,0)  c(0,2)
    point a,b(1),c(0,2);
    test1 = a.coincide(b);
    test2 = b.coincide(a);
    cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
    test1 = a.coincide(c);
    test2 = c.coincide(a);
    cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
    getch();
}
```

Chapitre 2: B- PROPRIETES DES FONCTIONS MEMBRES

❑ Mode de transmission des objets en argument

2- Passage par adresse

Modifier la fonction membre coincide de l'exemple précédent de sorte que son prototype devienne

*int point::coincide(point *adpt).* Ré-écrire le programme principal en conséquence.

```
#include <iostream>
#include <conio.h>
using namespace std;
// objets transmis en argument d'une fonction membre - transmission de l'adresse
class point
{
    int x,y;
public: point(int abs = 0,int ord = 2){x=abs;y=ord;}// constructeur
    int coincide(point *);
};
int point::coincide(point *adpt)
{if ((adpt->x == x) && (adpt->y == y)) return(1);else return(0);}
// noter la dissymetrie des notations pt->x et x
main()
{point a,b(1),c(0,2);
int test1,test2;
test1 = a.coincide(&b);
test2 = b.coincide(&a);
cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
test1 = a.coincide(&c);
test2 = c.coincide(&a);
cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
getch();}
```

Chapitre 2: B- PROPRIETES DES FONCTIONS MEMBRES

❑ Mode de transmission des objets en argument

3- Passage par référence

Modifier à nouveau la fonction membre coincide de sorte que son prototype devienne *int point::coincide(point &pt)*. Ré-écrire le programme principal en conséquence.

```
#include <iostream>
#include <conio.h>
using namespace std;
// objets transmis en argument d'une fonction membre - transmission par reference
class point
{
    int x,y;
public: point(int abs = 0,int ord = 2){x=abs;y=ord;}// constructeur
    int coincide(point &pt);
};
int point::coincide(point &pt)
{if (pt.x == x && pt.y == y) return(1);else return(0);}
// noter la dissymetrie des notations pt.x et x
main()
{point a,b(1),c(0,2);
int test1,test2;
test1 = a.coincide(b);
test2 = b.coincide(a);
cout<<"a et b:"<<test1<<" ou "<<test2<<"\n";
test1 = a.coincide(c);
test2 = c.coincide(a);
cout<<"a et c:"<<test1<<" ou "<<test2<<"\n";
getch() ;}
```

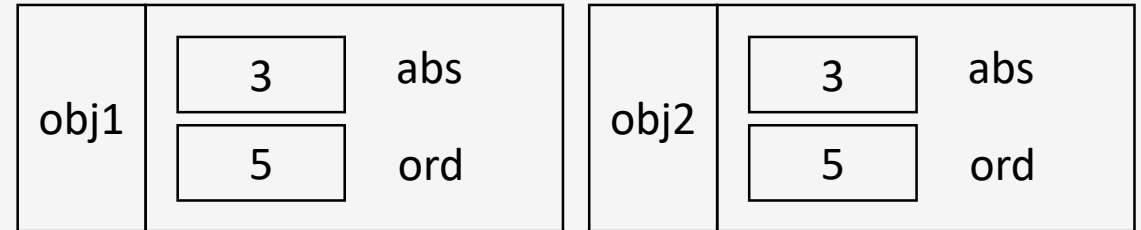
Chapitre 2: C- Constructeur par recopie

C++ offre un moyen de créer la copie d'une instance (copie d'objet): **le constructeur de copie**.

Autrement dit, **une initialisation par recopie d'un objet** est la création d'un objet par recopie d'un objet existant de même type.

Exemple 1:

```
point obj1(3,5);  
point obj2=obj1; //initialisation
```



→ **obj1** et **obj2** sont deux instances distinctes mais ayant des mêmes valeurs pour leur attributs.

Pour réaliser une telle initialisation, C++ a prévu d'utiliser un constructeur particulier dit **constructeur de recopie**. Si un tel constructeur n'existe pas, un traitement par défaut est prévu ; on peut dire, de façon équivalente, qu'on utilise **un constructeur de recopie par défaut**.

Exemple 2:

```
int f1 (point obj);  
...  
R = f1(obj1)
```

Rappel: Passage par valeur → les paramètres effectifs sont donc évalués et seront recopiés dans les paramètres de la fonction.

Chapitre 2: C- Constructeur par recopie

Dans toute situation **d'initialisation par recopie** il y toujours appel d'un constructeur de recopie, mais il faut distinguer **deux** cas principaux.

- Il n'existe pas de constructeur prévoyant ce cas de déclaration: dans ce cas un traitement par défaut est prévu (constructeur de recopie par défaut).
- Il existe un constructeur prévoyant ce cas de déclaration et il sera alors utilisé (constructeur de recopie).

Chapitre 2: C- Constructeur par copie

❑ Cas 1: S'il n'existe pas de constructeur approprié:

- Cela signifie qu'il n'existe pas, dans la classe point, de constructeur à un **seul argument de type point**:
 - le compilateur va donc mettre en place une copie des valeurs de l'objet **obj1** aux valeurs de l'objet **obj2**, après création de celui ci (comme l'affectation entre variable de même type).
- Un problème se pose lorsque les objets contiennent des pointeurs sur des emplacements alloués dynamiquement :
 - les valeurs des pointeurs seront copiées mais pas les emplacements pointés (c.à.d: les pointeurs pointent vers la même adresse sachant qu'ils ont eux des adresses différents).
- Dans ce cas, C++ utilise un **constructeur par copie par défaut** effectuant la simple copie des membres données.

Remarque

Même s'il existe un constructeur usuel pour la classe, c'est le constructeur par copie par défaut qui sera utilisé.

Chapitre 2: C- Constructeur par recopie

❑ Cas 2: S'il existe un constructeur approprié

- Le C++ imposant à ce constructeur(constructeur par recopie):

- Que son unique argument soit transmis par référence.
 - Il sera donc de la forme: `point(point &)`.

- Syntaxe:

NomClasse(NomClasse const& objrecu) {...}

- Ce constructeur sera appelé de façon habituelle (après création de l'objet):

- Mais cette fois ci, aucune recopie n'est mise en place automatiquement
 - C'est au constructeur de recopie de la prendre en charge.

Exemple 1: Constructeur de recopie par défaut

```
#include <iostream>
using namespace std;
class tab{
int nbelem;      // nombre d'élément dans le tableau
int *adr;
public:
tab(int n) {
nbelem = n;
adr = new int[nbelem];    // allocation dynamique du tableau
cout << "++ constructeur usuel | adresse de l'objet cree = " << this << endl;
cout << "- adresse tableau = " << adr << endl;
}
~tab() {
cout << "----->destructeur | adresse objet detruit = " << this << endl;
cout << "- adresse tableau = " << adr << endl;
delete adr;
}
};

int main()
{
    tab t1(5);    // création de t1 par le constructeur usuel
    tab t2 = t1;  // création de t2 par appel du constructeur de recopie par défaut
    // sur l'objet t1 ou bien, tab t2(t1) ;
    return 0;
}
```

```
++ constructeur usuel | adresse de l'objet cree = 0x22fe30
- adresse tableau = 0x5fac60
----->destructeur | adresse objet detruit = 0x22fe20
- adresse tableau = 0x5fac60
----->destructeur | adresse objet detruit = 0x22fe30
- adresse tableau = 0x5fac60
```

```
Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

Exemple 1: Constructeur de copie par défaut

D'après les résultats de l'exécution du programme, on constate que l'objet t2 a bien été créé, avec recopie des valeurs de l'objet t1.

A la fin du programme principal, l'appel du destructeur libère l'emplacement pointé par adr pour l'objet t2, puis libère le même emplacement pour l'objet t1 : le même emplacement mémoire est donc libéré deux fois, ce qui est complètement anormal !!!

```
++ constructeur usuel : adresse de l'objet cree = 0x22fe30
- adresse tableau = 0x5fac60
----->destructeur : adresse objet detruit = 0x22fe20
- adresse tableau = 0x5fac60
----->destructeur : adresse objet detruit = 0x22fe30
- adresse tableau = 0x5fac60

Process returned 0 (0x0)   execution time : 0.058 s
Press any key to continue.
```

Schématiser la situation étudiée:

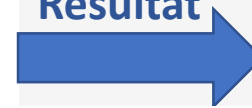
Exemple 2: Constructeur de recopie par défaut

```
class vect
{   int nelem ;           // nombre d'éléments
    double * adr ;        // pointeur sur ces éléments
public :
    vect (int n)           // constructeur "usuel"
    {   adr = new double [nelem = n] ;
        cout << "+ const. usuel - adr objet : " << adr << " - adr vecteur : " << adr << "\n" ;
    }
    ~vect ()               // destructeur
    {   cout << "- Destr. objet - adr objet : "
        << this << " - adr vecteur : " << adr << "\n" ;
        delete adr ;
    }
} ;

void fct (vect b)
{   cout << "*** appel de fct ***\n" ;
}

main()
{   vect a(5) ;
    fct (a) ;
}
```

Résultat



crée l'Objet 1

des l'Objet 2

des l'Objet 1

```
+ const. usuel - adr objet : 006AFDE4 - adr vecteur : 007D0320
*** appel de fct ***
- Destr. objet - adr objet : 006AFD90 - adr vecteur : 007D0320
- Destr. objet - adr objet : 006AFDE4 - adr vecteur : 007D0320
```

Explication: A la fin de l'exécution de la fonction fct, le destructeur ~vect est appelé pour b, ce qui libère l'emplacement pointé par adr ; à la fin de l'exécution de la fonction main, le destructeur est appelé pour a, ce qui libère... le même emplacement.
Cette tentative constitue une erreur d'exécution dont les conséquences varient avec l'implémentation.

Définition d'un constructeur de recopie

Cette situation montre également que:

- Toute modification du tableau de l'objet t1 entraînera donc une modification du tableau de l'objet t2 et inversement (exemple 1).

→ Pour éviter ce problème, il faut créer un **constructeur par recopie**.

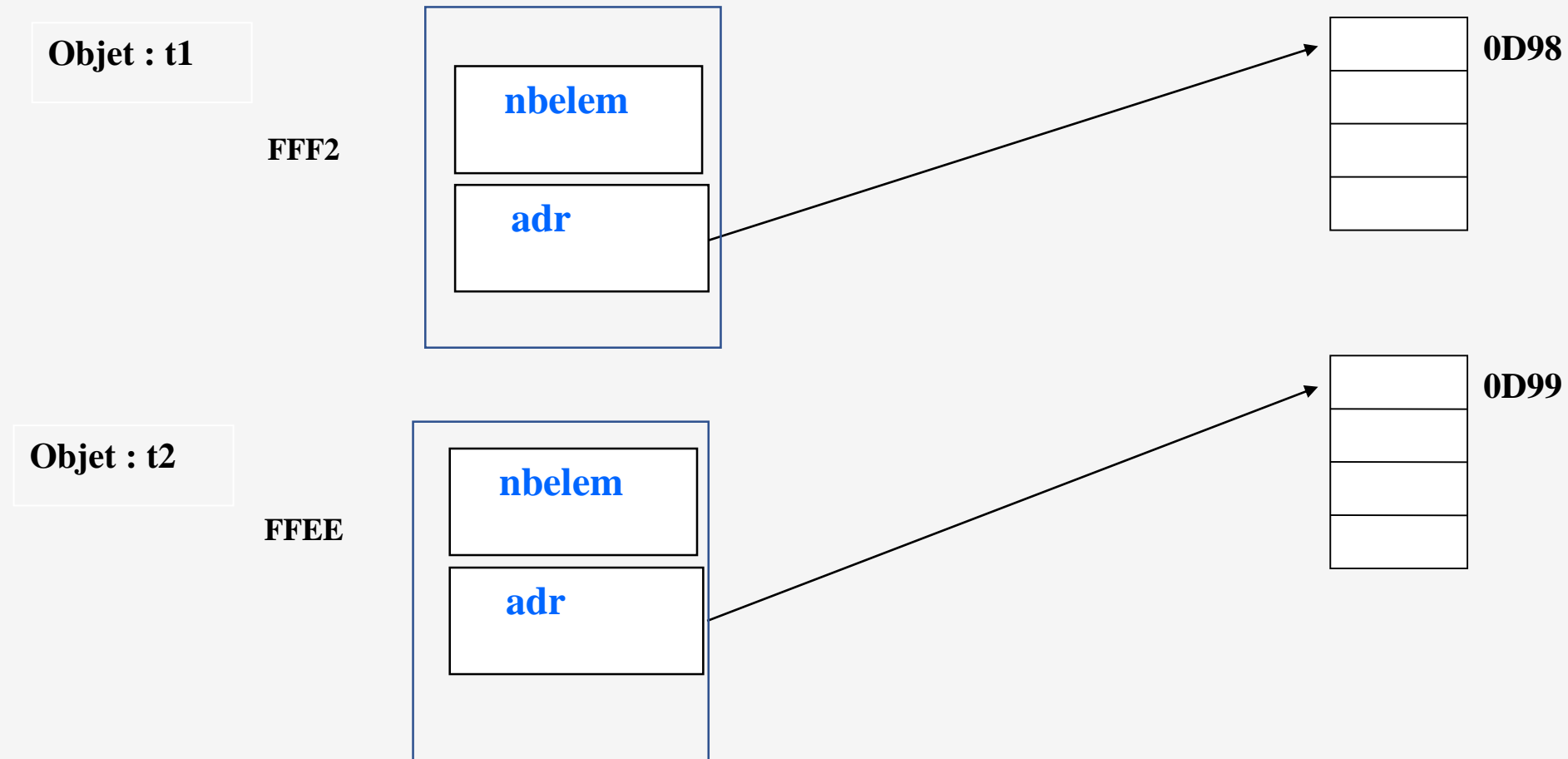
Le constructeur par recopie devra:

+ Créer un nouvel objet avec recopie des membres données, mais surtout **avec un nouvel emplacement pour le tableau**.

Autrement dit, l'instruction ***tab t2 = t1*** doit conduire à créer "intégralement" un nouveau objet de type tab, avec ses membres données nelem et adr, mais aussi son propre emplacement de stockage des valeurs du tableau.

Définition d'un constructeur de copie

La situation deviendra alors la suivante :



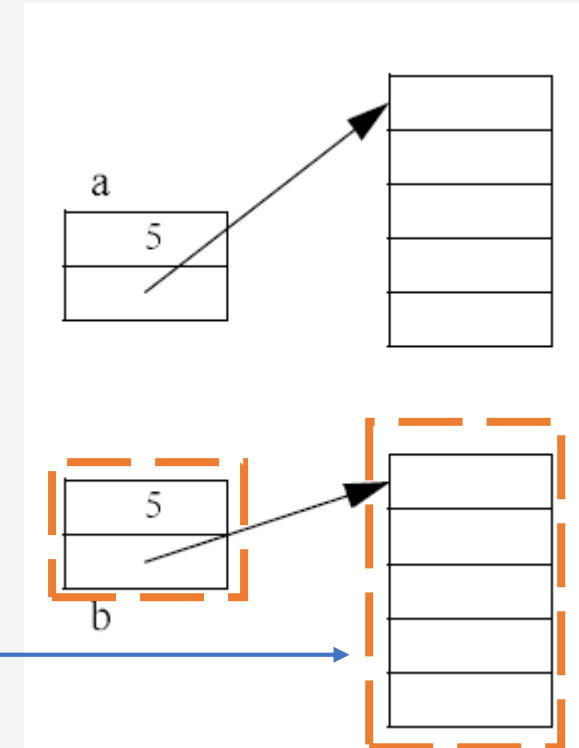
Définition d'un constructeur de recopie

Pour ce faire, nous définissons, au sein de la classe vect (exemple 2), un constructeur par recopie de la forme :

`vect (const vect &) ; // ou, a la rigueur vect (vect &)`

Ce constructeur (appelé après la création d'un nouveau objet) doit :

- Créer dynamiquement **un nouvel emplacement** dans lequel il recopie les valeurs correspondant à l'objet reçu en argument.
- Renseigner convenablement les membres données du nouvel objet
 - nelem = valeur du membre nelem de l'objet reçu en argument.
 - adr = adresse du nouvel emplacement.



→ Introduisons ce constructeur de recopie dans l'exemple précédent :

```
#include <iostream>
using namespace std ;
class vect
{
int nelem ; // nombre d'éléments
double * adr ; // pointeur sur ces éléments
public :
vect (int n) // constructeur "usuel"
{ adr = new double [nelem = n] ;
cout << "+ const. usuel - adr objet : " << this
<< " - adr vecteur : " << adr << "\n" ;
}
vect (vect & v) // constructeur de recopie
{
    nelem = v.nelem;
    adr = new double [nelem] ; // création nouvel emplacement pour le vecteur
int i ; for (i=0 ; i<nelem ; i++) adr[i]=v.adr[i] ; // recopie de l'ancien
cout << "+ const. recopie - adr objet : " << this
<< " - adr vecteur : " << adr << "\n" ;
}
~vect () // destructeur
{ cout << "- Destr. objet - adr objet : "
<< this << " - adr vecteur : " << adr << "\n" ;
delete adr ;
} ;
void fct (vect b)
{ cout << "**** appel de fct ****\n" ; }
int main()
{ vect a(5); //creation object a [appel constructeur usuel]
  fct (a) ; //appel constructeur de recopie
return 0;
}
```

```
+ const. usuel - adr objet : 0x22fe20 - adr vecteur : 0x675ca0
+ const. recopie - adr objet : 0x22fe30 - adr vecteur : 0x675d10
*** appel de fct ***
- Destr. objet - adr objet : 0x22fe30 - adr vecteur : 0x675d10
- Destr. objet - adr objet : 0x22fe20 - adr vecteur : 0x675ca0

Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.
```

Définition d'un constructeur de copie

- Remarque

Le constructeur par copie permet ici de régler le problème de l'initialisation d'un objet par un autre objet de même type **au moment de sa déclaration**.

Liste d'Initialisation d'un constructeur

Soit la classe :

```
class Y { /* ... */  
};
```

```
class X {  
    const int _x;  
    Y _y;  
    int _z;  
public:  
    X(int a, int b, Y y);  
    ~X(); // ....  
};
```

```
X::X(int a, int b, Y y) {  
    _x = a; // ERREUR: l'affectation à une constante est interdite  
    _z = b; // OK : affectation };
```

Questions:

- Comment initialiser la donnée membre constante `_x` ?
- Comment appeler le constructeur de la classe Y afin d'initialiser l'objet membre `_y` ?

→ Réponse : la liste d'initialisation.

Liste d'Initialisation d'un constructeur

- La phase d'initialisation d'un objet utilise:
 - une liste d'initialisation qui est spécifiée dans la définition du constructeur.
- Syntaxe :
 - `nom_classe::nom_constructeur(args ...) : liste_d_initialisation { // corps du constructeur};`
- Exemple:

```
X::X(int a, int b, Y y) : _x( a ) , _y( y ) , _z( b ) {  
    // rien d'autre à faire d'autre pour l'initialisation }
```

 - **L'expression `_x(a)`**: indique au compilateur d'initialiser la donnée membre `_x` avec la valeur du paramètre `a`.
 - **L'expression `_y(y)`**: indique au compilateur d'initialiser la donnée membre `_y` par un appel au constructeur (avec l'argument `y`) de la classe `Y`.
 - Avec `y` est un objet de la classe `Y`

Ou

 - `y` est un ensemble de valeur des champs de la classe `Y`

Liste d'Initialisation d'un constructeur

Exemple 1

```
class truc
{ const int n ;
public :
  truc () ;
  .....
} ;
```

il n'est pas possible de procéder ainsi pour initialiser n dans le constructeur de truc :

```
truc::truc() { n = 12 ; } // interdit : n est constant
```

En revanche, on pourra procéder ainsi :

```
truc::truc() : n(12) { ..... }
```

Exemple 2

```
class point
{ const int abs;
  int ord ;
public :
  point (point &) ;
  .....
} ;
```

```
point::point(point &pt) : abs(pt.abs), ord(pt.ord)
{ }
```

Chapitre 2: D- Les fonctions et classes amies

La propriété d'encapsulation en C++ impose que:

- Les membres privés ne sont pas accessibles à l'extérieur que via des ***accesseurs***.

Exemple

```
Cout<<"X ="<<pt1.Getabscisse()<<"Y="<<pt1.Getordonne()<<endl;
```

- Ce même principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe.

Exemple: soit deux classes: **vecteur** et **matrice**

Question: A partir de ces classes créer deux objets de type vecteur et matrice puis définir une fonction qui permet de calculer leur produit.

Chapitre 2: D- Les fonctions et classes amies

Avec les contraintes imposées par le C++ (principe de l'encapsulation), nous ne pourrions définir cette fonction ni comme fonction membre de la classe vecteur, ni comme fonction membre de la classe matrice, et encore moins comme fonction indépendante (c'est-à-dire membre d'aucune classe).

Solutions classiques:

Solutions 1: Rendre publiques les données de deux classes,

- **Inconvénients** → Perte de protection de données membres.

Solutions 2: Introduction dans les deux classes des fonctions publiques permettant d'accéder aux données.

- **Inconvénients** → Le temps d'exécution lors d'utilisation des accesseurs, car cela demande de nombreux appels si on a besoin des valeurs des membres privés.

Solution optimale:

→ **Utilisation des fonctions ou classe amies**

N.B Cette notion de fonction amies propose une solution intéressante, sous la forme d'un **compromis** entre encapsulation formelle des données privées et des données publiques.

Chapitre 2: D- Les fonctions et classes amies

Caractéristiques:

- Les fonctions amies (ou classes amies) d'une classe ont libre accès aux membres privés ou protégés d'une autre classe.
- Lors de la définition d'une classe, il est en effet possible de déclarer qu'une ou plusieurs fonctions (extérieures à la classe) sont des "amies" ; une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.*
- Les fonctions amies d'une classe ne sont pas **des fonctions membres**.

Syntaxe

```
class MyClass{  
....  
friend returnType fonctionName(liste des arguments);  
};
```


Chapitre 2: D- Les fonctions et classes amies

Types d'amitiés:

Il existe plusieurs situations d'amitiés :

- fonction indépendante, amie d'une classe,
- fonction membre d'une classe, amie d'une autre classe,
- fonction amie de plusieurs classes,
- toutes les fonctions membres d'une classe, amies d'une autre classe.

Exemple de fonction indépendante amie d'une classe

Exemple 1

```
#include <iostream>
#include<string>
using namespace std;
class client{
private:
    int idclient;
    string nom, prenom;
public:
    client(int i, string n, string p):idclient(i), nom (n), prenom(p){};
    string getName() { return nom; }
    void setName(string n) { nom = n;}
    friend void affiche(client clt); //informer le compilateur qu'une fonction indépendante sera
    // existe à l'extérieure et qui aura le droit d'accéder aux données privées de la classe client.
};
//définition de la fonction indépendante en dehors de la classe
void affiche(client clt){
    cout<<"Affichage des données des clients"<<endl;
    cout<<"ID = "<<clt.idclient<<" , nom = "<<clt.nom<<" , prenom = "<<clt.prenom<<endl;
}

int main()
{
    client clt1(100, "Ahmed", "Omar");
    affiche(clt1);
    return 0;
}
```



```
Affichage des données des clients
ID = 100 , nom = Ahmed , prenom = Omar
```

Chapitre 2: D- Les fonctions et classes amies

Exemple de fonction indépendante amie d'une classe

Exemple 2: Soit la fonction coincide que nous avons vu précédemment, et qui examine la "coïncidence" de deux objets de type point.

Résoudre le même problème, en faisant cette fois de la fonction coincide une fonction indépendante amie de la classe point.

1. Introduire dans la classe point la déclaration d'amitié appropriée.

```
friend int coincide (point, point) ;
```

2. Définition de la fonction coincide en **dehors** de la classe avec un passage en argument de deux points. (Pas d'argument implicite (this), car il ne s'agit pas d'une fonction membre).

```
int coincide (point p, point q) // définition de coincide
{ if ((p.x == q.x) && (p.y == q.y)) return 1 ;
  else return 0 ;
}
```

3. Créer dans la fonction principale des objets et faire les tests nécessaires.

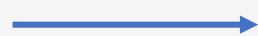
Chapitre 2: D- Les fonctions et classes amies

Exemple de fonction indépendante amie d'une classe

Exemple de fonction indépendante (coincide) amie de la classe point

```
#include <iostream>
using namespace std ;
class point
{ int x, y ;
public :
    point (int abs=0, int ord=0)        // un constructeur ("inline")
        { x=abs ; y=ord ; }
    // déclaration fonction amie (indépendante) nommée coincide
    friend int coincide (point, point) ;
};
int coincide (point p, point q)        // définition de coincide
{   if ((p.x == q.x) && (p.y == q.y)) return 1 ;
    else return 0 ;
}
main()                                // programme d'essai
{   point a(1,0), b(1), c ;
    if (coincide (a,b)) cout << "a coincide avec b \n" ;
        else cout << "a et b sont differents \n" ;
    if (coincide (a,c)) cout << "a coincide avec c \n" ;
        else cout << "a et c sont differents \n" ;
}
```

Résultat



```
a coincide avec b
a et c sont differents
```

Chapitre 2: D- Les fonctions et classes amies

Les différentes situations d'amitié

D'autres situations d'amitié sont possibles ; fondées sur le même principe (cas d'une fonction indépendante amie d'une classe), elles peuvent conduire à des déclarations d'amitié très légèrement différentes.

→ Fonction membre d'une classe, amie d'une autre classe

Il s'agit un peu d'un cas particulier de la situation précédente. En fait, il suffit simplement de préciser, dans la déclaration d'amitié, la classe à laquelle appartient la fonction concernée, à l'aide de l'opérateur de résolution de portée (::).

Chapitre 2: D- Les fonctions et classes amies

Les différentes situations d'amitié

→ Fonction membre d'une classe, amie d'une autre classe

Par exemple, on considère deux classes A et B

Nous avons besoin **dans B** d'une fonction membre **f**, de prototype : ***int f(char, A);***

```
class A{  
private:  
    int xa;  
public:  
    A(){};  
};
```

```
class B{  
private:  
    int xb;  
public:  
    B(){};  
};
```

Cela signifie que, la fonction **f** doit pouvoir accéder aux membres privés de A.

→ Il faut rendre la fonction membre **f** de la classe B, amie de la classe A, Pour cela:
- On ajoute dans la définition de la **classe A** la ligne:

```
class A  
{  
    // partie privée  
    .....  
    // partie publique  
    friend int B::f (char, A) ;  
    .....  
};
```

```
class B  
{  
    .....  
    int f (char, A) ;  
    .....  
};  
int B::f (char ..., A ...)  
{  
    // on a accès ici aux membres privés  
    // de tout objet de type A  
}
```

Fonction (f) d'une classe (B), amie d'une autre classe (A)

Chapitre 2: D- Les fonctions et classes amies

Les différentes situations d'amitié

→ Fonction amie de plusieurs classes

Une même fonction *f* (qu'elle soit indépendante ou fonction membre) peut faire l'objet de déclarations d'amitié dans différentes classes.

Pour se faire:

- On ajoute dans la classe A
friend int *f*(A a, B b);
- On ajoute dans la classe B
friend int *f*(A a, B b)

```
class A
{    // partie privée
    .....
    // partie publique
    friend void f(A, B) ;
    .....
} ;
```

```
class B
{ // partie privée
    .....
    // partie publique
    friend void f(A, B) ;
    .....
} ;
```

```
void f(A..., B...)
{ // on a accès ici aux membres privés
  // de n'importe quel objet de type A ou B
}
```

Chapitre 2: D- Les fonctions et classes amies

Les différentes situations d'amitié

→ Toutes les fonctions d'une classe amies d'une autre classe

Pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on place, dans la classe A, la déclaration :

```
friend class B;
```

Remarque:

- Ce type de déclaration d'amitié évite de fournir les en-têtes des fonctions concernées.
- L'amitié n'est pas transmissible: Une fonction amie de la classe de base (notion héritage) ne sera amie de la classe dérivée que si elle a été déclarée amie dans la classe dérivée.

Chapitre 2: D- Les fonctions et classes amies

Exercice

Réaliser une fonction permettant de déterminer le produit d'un vecteur (dimension 3) (objet de classe *vect*) par une matrice (3*3) (objet de classe *matrice*).

Par souci de simplicité, les fonctions membres de deux classes sont limités aux:

- un constructeur pour *vect* et pour *matrice*,
- une fonction d'affichage (*affiche*) pour *matrice*.

A- fonction amie nommée *prod* **indépendante** et amie des deux classes *vect* et *matrice*.

B- fonction amie nommée *prod* est **membre** de la classe *matrice* et amie de la classe *vect*.

Solution: A- fonction amie nommée **prod indépendante** et amie des deux classes **vect** et **matrice**.

```
#include<iostream>
using namespace std;
class matrice;
class vecteur{
private:
    float v[3];
public:
    vecteur(float v1=0, float v2=0, float v3=0){
        v[0] = v1; v[1] = v2; v[2] = v3;
        cout<<"Appel constructeur:objet de type vecteur a ete initialise avec succes"<<endl;
    }
    friend vecteur prod(matrice mat, vecteur vect);
    void affiche(){
        for(int i=0; i<3; i++){
            cout<<v[i]<<endl;
        }
    }
};

class matrice{
private:
    float m[3][3];
public:
    matrice(){
        int i,j,v=1;
        for(i=0; i<3; i++){
            for(j=0; j<3; j++){
                m[i][j]=v++;
            }
        }
        cout<<"Appel constructeur: objet de type matrice 3*3 a ete initialise avec succes"<<endl;
    }
    friend vecteur prod(matrice mat, vecteur vect);
    void affiche(){
        cout<<"contenu de la Matrice"<<endl;
        for(int i=0; i<3; i++){
            for(int j=0; j<3; j++){
                cout<<m[i][j]<<" ";
            }
        }
        cout<<"\n";
    }
};
```

```
vecteur prod(matrice mat, vecteur vect)
{
    int i,j;
    float som;
    vecteur res;
    for(i=0; i<3; i++){
        for(j=0, som=0; j<3; j++){
            som = som + (mat.m[i][j]*vect.v[j]);
        }
        res.v[i] = som;
    }
    return res;
}

int main()
{
    matrice mat1;
    vecteur vect1(1,1,1);
    mat1.affiche();
    vecteur produit;
    produit = prod(mat1, vect1);
    produit.affiche();

    return 0;
}
```

Solution: fonction amie nommée **prod** est **membre** de la classe *matrice* et amie de la classe *vect*

```
#include <iostream>
using namespace std;
class vecteur;
class matrice{
private:
    int m[3][3];
public:
    matrice(){
        int i,j;
        int v=1;
        for(i=0; i<3; i++){
            for(j=0; j<3; j++){
                {m[i][j] = v++; }
            }
        }
        cout<<"Initialisation matrice"<<endl;
    }
    vecteur prod(vecteur vect);
};

class vecteur{
private:
    float v[3];
public:
    vecteur(float v1=0, float v2=0, float v3=0)
    {
        v[0] = v1; v[1] = v2; v[2] = v3;
        cout<<"Initialisation vecteur"<<endl;
    }
    friend vecteur matrice::prod(vecteur vect);
    void affiche(){
        for(int i=0; i<3; i++){
            cout<<v[i]<<endl;
        }
    }
};
```

```
vecteur matrice::prod(vecteur vect){
    int i,j;
    float som;
    vecteur res;
    for(i=0; i<3; i++){
        for(j=0, som=0; j<3; j++){
            som = som + m[i][j]*vect.v[j];
        }
        res.v[i] = som;
    }
    return res;
}

int main()
{
    vecteur vect1(1,1,1);
    matrice mat1;
    vecteur produit;
    produit = mat1.prod(vect1);
    produit.affiche();
    return 0;
}
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Le C++ autorise:

- la **surdéfinition** de fonctions: des fonctions membres ou fonctions indépendantes.
- Mais aussi, surdéfinition des **opérateurs** (considérés comme des fonctions).
- Les opérateurs: + * - == = < <= ou autres

	Opérateur	Type
Opérateur unaire	++, --	Opérateur unaire
Opérateur binaire	+, -, *, /, %	Opérateur arithmétique
	<, <=, >, >=, ==, !=	Opérateur relationnel (comparaison)
	&&, , !	Opérateur logique
	=, +=, -=, *=, /=, %=	Opérateur d'assignation
Opérateur ternaire	?:	Opérateur ternaire ou conditionnel Exemple: MAX = (A > B) ? A : B;

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int a = 2, b=5;
    cout<<a + b<<endl;
    string nom = "Amer";
    string prenom = "Ahmed";
    cout<<nom + prenom<<endl;
    return 0;
}
```



```
?
AmerAhmed
Process returned 0 (0x0)
Press any key to continue.
```

- Deux opérandes de type **entier**, l'opérateur **+** a retourné leur somme.
- Type **string** → Concaténation

Comment pouvez-vous expliquer ce mécanisme ??

→ Même principe de surdéfinition des fonctions:

- l'opérateur **+** désigne le nom de la fonction

- Le **type des opérandes** indique au compilateur la fonction qui doit d'être appelée (somme/concaténation).

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets:

Objectif: Création d'une fonction qui calcule la somme des coordonnées de deux points:

Solution 1:

- Création d'une fonction membre appelée somme qui prend comme argument un point.
- Le résultat renvoyé par cette fonction est de type point.

```
#include <iostream.h>
classe Point
{
private :
    int x,y ;
public :
    Point(int, int);
    Point somme(Point) ;
    void affiche() ;
} ;
```

Déclaration de la classe Point

```
Point ::Point(int abs, int ord){
x=abs ; y=ord ;}
Point Point ::somme(Point p) {
Point p1;
p1.x = x+p.x;
p1.y = y+p.y;
return p1;};
void Point ::affiche() {
cout << "(" << x << " , " << y <<
")" <<endl;}
```

Définition des fonction membres de la classe Point

```
void main()
{
Point a(1,2), b(5,10);
Point c=a.somme(b);
cout<<"a"<<a.affiche()<<
endl;
cout<<"b"<<b.affiche()<<
endl;
cout<<"c"<<c.affiche()<<
endl;
}
```

Programme principal

Quelles sont les instructions nécessaires pour pouvoir écrire $c = a + b$ au lieu de $c = a.somme(b)$?

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets (suite):

- ❑ A la place de l'instruction `c=a.somme(b)`
 - Il serait plus judicieux de pouvoir écrire `c=a+b`. C'est plus naturel.
 - Il faut pour cela **surdéfinir** (surcharger) l'opérateur `+` pour pouvoir effectuer directement une addition entre **deux objets appartenant à la classe Point**.
- ❑ La surdéfinition d'un opérateur consiste à:
 - définir une fonction portant le même nom que l'opérateur précédé du mot clé ***operator***.

Syntaxe:

TypeRetour ***operator*** + (Argument);

Le programme précédent devient alors →

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets (*Opérateur binaire*):

```
#include <iostream.h>
classe Point
{
private :
    int x,y ;
public :
    Point(int, int);
    /* surdéfinition d'opérateur +, definie comme fonction
    membre*/
    Point operator+(Point);
    /*surdéfinition d'opérateur avec fonction externe,
    indépendante ou globale */
    friend operator+(Point, Point);
    void affiche() ;
} ;
```

Déclaration de la classe Point avec surdéfinition de l'opérateur +

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets (*Opérateur binaire*):

Définition des fonction membres de la classe Point

```
Point::Point(int abs, int ord){
x=abs ; y=ord ;}
Point Point ::operator+(Point p){
Point p1;
p1.x = x+p.x; /* x et y sont les coordonnées de l'objet a */
p1.y = y+p.y;
return p1;};
/* dans le cas de fonction externe*/
Point operator+(Point p1, Point p2)
{
Point pt;
pt.x = p1.x + p2.x;
pt.y = p1.y + p2.y;
return pt;
}
void Point ::affiche(){
cout << "(" << x << ", " << y << ")" << endl;}
```

Programme principal

```
void main()
{
Point a(1,2), b(5,10);
Point c = a + b;
cout<<"c"<<c.affiche()<<
endl;
}
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Remarques:

- ☐ Les opérateurs surdéfinis gardent leur priorité et leur associativité habituelles.
- ☐ La surdéfinition des opérateurs `=`, `<<` et `>>` est différente des autres opérateurs et fait appel à des nouvelles notions du C++ .
- ☐ Ce que nous avons mis en oeuvre pour l'opérateur d'addition, nous pouvons en faire de même pour tous les autres opérateurs "*simples*".
- ☐ On peut utiliser La surdéfinition des opérateurs avec des fonctions membres ou des fonctions externes, en utilisant la notion de fonctions amies.

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets (*Opérateur binaire*):

A- Surdéfinition des opérateurs de bases comme des fonctions membres

```
class Point
{
    int x;
    int y;

public :
    Point(){} //Constructeur par défaut
    Point(int a, int b){ x=a; y=b; }
    Point operator+(const Point&);
    Point operator-(const Point&);
    int operator==(const Point&);
    int operator*(const Point&); //calcul produit scalaire
    Point operator/(int);
    void Affiche()
    {
        cout << this << "->" << x << ", " << y << endl;
    }
};
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Exemple avec l'utilisation des objets (*Opérateur binaire*):

A- Surdéfinition des opérateurs de bases comme des fonctions membres

```
Point Point::operator+ (const Point & a)
{ // Addition de 2 points
    Point p;
    p.x = x + a.x;
    p.y = y + a.y;
    return p;
}
```

```
Point Point::operator- (const Point & a)
{ // Soustraction de 2 points
    Point p;
    p.x = x - a.x;
    p.y = y - a.y;
    return p;
}
```

```
int Point::operator==(const Point & p)
{ // Egalité de 2 points
    if( x==p.x && y==p.y )
        return 1;
    else
        return 0;
}

int Point::operator* (const Point & p)
{ // produit scalaire de deux points
    int prod = x * p.x + y * p.y;
    return prod;
}

Point Point::operator/ (int diviseur)
{ /*division d'un point ou vecteur par un int*/
    Point p;
    p.x = x/diviseur ;
    p.y = y/diviseur ;
    return p;
}
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

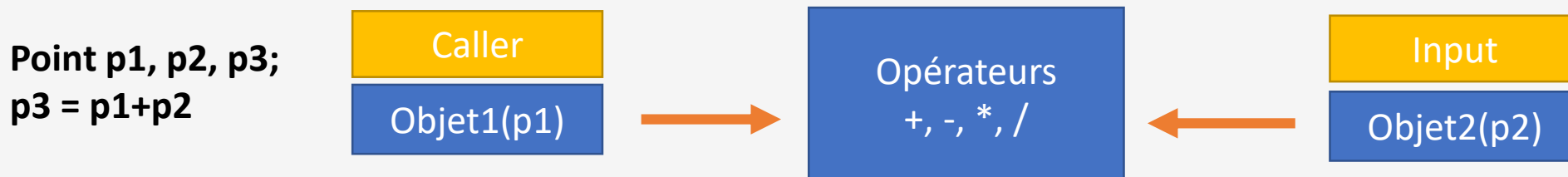
```
int main()
{
    Point p(1,2), nul(0,0), pp(3,4);
    p.Affiche();
    pp.Affiche();
    Point pt = p+pp; // appel à p.operator+(pp);
    pt.Affiche();
    if( p==pp ) // appel à p.operator==(pp);
        cout<<"p==pp" <<endl;
    else cout<<"p!=pp"<<endl;
    p = pp;
    p.Affiche();
    pp = p-pt; // appel à p.operator-(pt);
    pp.Affiche();
    if( p==pt)
        cout << "p==pt" << endl;
    else cout << "p!=pt" << endl;
    if( pt== nul)
        cout << "produit scalaire de p par un vecteur
        nul\n" << p*pt;
    else cout << "produit scalaire de deux points:
    "<<p*pt<<"\n";
    (p/4).Affiche(); // appel à p.operator/(4);
    return 0;
}
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Remarques:

- ❑ Quand on a le choix, l'utilisation d'une fonction membre pour surcharger un opérateur est préférable.
- ❑ Une fonction membre renforce l'encapsulation.
- ❑ Les opérateurs surchargés par des fonctions membres se transmettent aussi par héritage **sauf l'opérateur d'affectation**.

Appel des fonctions pour la surdéfinition:




Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Surdéfinition **opérateur unaire** (Opérateur ++, --)

```
class Compteur {
private:
int cpt;
public:
Compteur(): cpt(0){}
Compteur(int c): cpt(c){}
void get_cpt(){ cout<<this<<"->"<<cpt<<endl; }
Compteur operator ++ ()
{
    ++cpt; // prefix
    return Compteur(cpt);
}
Compteur operator -- ()
{
    --cpt; //prefix
    return Compteur(cpt);
}
};
```

```
int main()
{
Compteur c1, c2(7), c3(c2);
Compteur c4=++c1;
Compteur c5=--c2;
c1.get_cpt();
c2.get_cpt();
c3.get_cpt();
c4.get_cpt();
c5.get_cpt();
    return 0;
}
```



```
0x22fe4c->1
0x22fe48->6
0x22fe44->7
0x22fe40->1
0x22fe3c->6

Process returned 0 (0x0)   execut
Press any key to continue.
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Surdéfinition **opérateur unaire** (Opérateur ++, --)

```
int main()
{
    Compteur c1, c2(7), c3(c2);
    Compteur c4=++c1; //prefix
    Compteur c5=--c2;
    Compteur c6=c1++; //postfix
    c1.get_cpt();
    c2.get_cpt();
    c3.get_cpt();
    c4.get_cpt();
    c5.get_cpt();
    return 0;
}
```

```
error: no 'operator++(int)' declared for postfix '++' [-fpermissive]
```


Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

Solution

```
Compteur operator ++ ()  
{  
    ++cpt;  
    return Compteur(cpt);  
}  
  
Compteur operator ++ (int )  
{  
    cpt++;  
    return Compteur(cpt);  
}  
  
Compteur operator -- ()  
{  
    --cpt;  
    return Compteur(cpt);  
}  
  
Compteur operator -- (int )  
{  
    cpt--;  
    return Compteur(cpt);  
}  
};
```

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

opérateur d'affectation (=)

- L'opérateur d'affectation "=" est assez particulier.

En effet,

- On retrouve le même problème que lors de la construction par recopie :
 - il est toujours possible d'effectuer une affectation entre deux objets (de même type, **qui sont déjà créés**).
 - mais que se passe-t-il s'ils contiennent des pointeurs? → problème de recopie
 - C'est pourquoi, il est souvent important d'implémenter ce type d'opérateur.
- A la différence du constructeur de recopie,
 - l'objet que l'on manipule dans l'opérateur d'affectation est déjà construit et a probablement des attributs dynamiques déjà alloués.
 - Il faut dans ce cas
 1. libérer la mémoire réservée à cet objet en faisant un **delete**,
 2. Ceci avant d'allouer à nouveau les attributs dont on a besoin.
 3. Retourner l'objet lui-même et pas une copie.

Chapitre 2: E- La surcharge (ou surdéfinition) des opérateurs

opérateur d'affectation (=)

Exerice: Soit la classe Vecteur suivante, Définir les fonctions membres et tester le programme.

```
class Vecteur
{
    int taille;
    int *pt;
public :
    Vecteur(int);
    Vecteur(Vecteur &);
    ~Vecteur();
    Vecteur& operator =(const Vecteur & );
};
```

Chapitre 2: E- La surcharge (ou surdéfin

opérateur d'affectation (=)

```
//Constructeur usuel avec argument
Vecteur::Vecteur(int n){
    taille = n;
    pt = new int[taille];
    for(int i=0;i<taille;i++)
        pt[i]=2*i;
}
//Constructeur de recopie
Vecteur::Vecteur(const Vecteur & v) {
    taille = v.taille;
    pt = new int[taille];
    for( int i = 0; i < taille; i++ )
        pt[i]=v.pt[i];
}
//Destructeur
Vecteur::~Vecteur()
{
    delete[] pt;
}
```

```
//Surcharge de l'opérateur d'affectation =
Vecteur& Vecteur::operator =(const
Vecteur & v)
{
    // On vérifie qu'on fait pas une affectation de
    // l'objet sur lui-même !
    if( this != &v )
    {
        // Effacement de la partie dynamique du vecteur
        // (son historique: les valeurs des champs)
        delete[] pt;
        taille = v.taille;
        // Allocation dynamique de la mémoire
        pt = new int[taille];
        // Recopie des valeurs
        for( int i=0; i<taille; i++ )
            pt[i]=v.pt[i];
    }
    return *this;
    // une référence sur l'objet est retourné, car on
    // doit rendre l'objet lui-même et non une copie.
}
```

```
int main(){
    Vecteur v(10);
    Vecteur vv(5);
    vv = v; // appel du l'opérateur d'affectation
    Return 0; }
```