

# MOOC semaine 3: Extension de la notion de **surcharge**

Déjà connu pour les **fonctions** (sem1) et les **méthodes** (sem2)

Règle de base:

Si plusieurs fonctions/méthodes ont le même nom, le compilateur sait laquelle doit être appelée car il peut les différencier grâce à leur **signature**.

*La signature est limitée au **nombre** et aux **types** des **paramètres***

*La signature n'inclut PAS le type de retour*

BOOC sem1 p40

```
int    affiche(int) ;    // ok
double affiche(int) ;    // erreur car même signature
int    affiche(double) ; // ok
```

# Extension de la notion de **surcharge** (2)

En C++ on peut aussi surcharger la plupart des symboles des **opérateurs**

## Motivation:

- Implémenter une solution avec une plus grande *concision de l'écriture*
- Bénéficier de la *sémantique familière* associée aux symboles des opérateurs pour l'élargir au-delà des types de base et de la bibliothèque standard

### **Exemple1:**

*Pourquoi ne dispose-t-on pas de l'opérateur d'égalité sur des objets ?*

*On doit écrire laborieusement une méthode pour tester l'égalité de 2 instances...*

### **Exemple2:**

*Ça serait tellement pratique de faire afficher une instance avec un seul << ...*

### **Exemple3:**

*Gros potentiel pour les applications orientées vers le calcul, la géométrie, etc*

# Extension de la notion de **surcharge** (3)

Risques de confusion si mal employé (*obfuscation* du code)

Perte d'information du fait du remplacement des noms de fonction par des symboles qui n'apportent pas suffisamment d'information sur le BUT de la fonction

Mettre en œuvre la **ré-utilisation** pour la surcharge des opérateurs ayant un lien sémantique fort, par exemple `==` et `!=` ou `+` et `+=` etc...

Take-home message:

- La surcharge des opérateurs est une option intéressante du langage
- C'est possible mais pas obligatoire / *on ne force personne pour le projet*
- Il est bon de connaître ce mécanisme pour comprendre du code que l'on n'a pas écrit

Surcharge: **interne** ou **externe ?**



Méthode  
de classe



Fonction

`a + b`

`a.operator+(b)`

`operator+(a,b)`

`a += b`

`a.operator+=(b)`

`operator+=(a,b)`

`a = b`

`a.operator=(b)`

`a == b`

`a.operator==(b)`

`operator==(a,b)`

`cout << b`

`operator<<(cout,b)`

`++a`

`a.operator++()`

`operator++(a)`

Surcharge: **interne**



Méthode  
de classe

ou **externe ? (2)**



Fonction

Préférable:

- si l'opérande gauche est modifié
- si l'accès aux attributs est requis

ex: modifie **a** donc *surcharge interne*

**a += b**

**a.operator+=(b)**

Obligatoire si l'opérande gauche :

- est un type de base
- n'appartient pas à la classe dont  
on veut surcharger l'opérateur

ex: **cout** est **ostream**

**cout << b**

**operator<<(cout,b)**

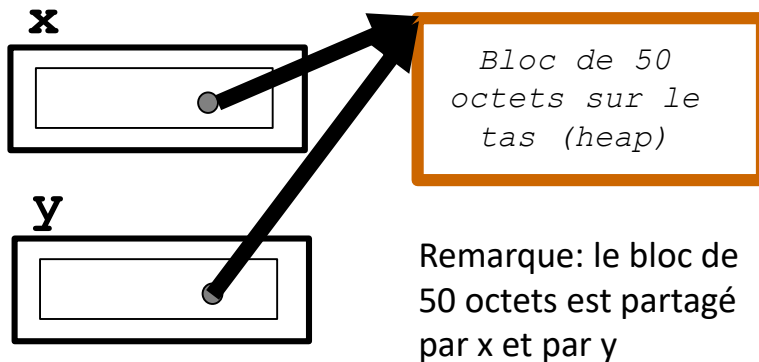
# Exemple: surcharge **interne** de l'opérateur d'égalité

car l'accès à tous les attributs est nécessaire pour les comparer terme à terme

## Scénario 1: Classe avec Allocation Dynamique et copie **Superficielle**

```
CADS <==> Classe avec Allocation Dynamique et copie Superficielle  
CADS x(50); // constructeur avec allocation  
CADS y(x); // constructeur de copie (copie superficielle)
```

Unique  
attribut  
`char* p;`



```
bool operator==(CADS const& b)  
{  
    return p == b.p;  
}
```

on obtient **true** pour l'expression `x == y` avec le scénario de copie superficielle car l'attribut de x et de y ont la même valeur d'adresse de bloc (OK).

# Rappel

Si on modifie l'une des trois méthodes ci-dessous,  
il FAUT vérifier si les deux autres doivent être aussi adaptées

Constructeur de copie

Destructeur

Surcharge de l'opérateur d'affectation (interne seulement)

+ réfléchir à la sémantique des opérateurs d'égalité / différence