MOOC semaine 3

Variable et méthode de classe

Surcharge des opérateurs

Buts:

- Intérêt des variables et méthodes de classe ?
- Quelle justification pour la surcharge des opérateurs ?

Plan:

- Les trois usages du mot-clef static
- Extension de la notion de surcharge
- Exemple de l'opérateur d'égalité

Les 3 usages de **static**

1) **static** associé à une **variable locale**

- Portée limitée au bloc de sa déclation
- Durée de vie permanente
- N'est initialisée qu'une seule fois
- si la valeur initiale n'est pas précisée,
  - -> est initialisé avec des 0
  - -> ou par le constructeur par défaut

⇒ Conserve sa valeur d'un appel au suivant

⇒ Privilégier cet usage de **static** par rapport aux deux autres car ***c'est le plus local***

Ex: variable mémorisant l'état courant de la lecture d'un fichier (cf série « automate de lecture »)

```
...
unsigned int nb_appel()
{
    static unsigned int count(0);
    return ++count;
}

int main()
{
    cout << nb_appel() << endl;
    cout << nb_appel() << endl;
    cout << nb_appel() << endl;
```

Affiche :

```
1
2
3
```

Les 3 usages de `static` (2)

2) **static** associé à une **variable globale** rend cette variable ***confidentielle au module***

- Portée limitée au module = *espace de noms non-nommé*
- Durée de vie permanente
- Accessible par toutes les fonctions du module
- n'est initialisée qu'une seule fois
- si la valeur initiale n'est pas précisée,
  -> est initialisé avec des 0
  -> ou par le constructeur par défaut

- Ne JAMAIS mettre dans l'interface d'un module
- Utile pour des constantes locales au module
- Sinon, ***à limiter au strict nécessaire***
  OK pour petit module mono-classe
  Ex : ensemble des instances d'une classe,
  reste caché dans l'implémentation ->

myclass.cc

```
...
#include "myclass.h"

static vector<MyClass> tab;// vide

// externalisation de la définition
// des méthodes de la classe MyClass

MyClass::Myclass() ...
{...}

...
```

Aussi utilisable pour une fonction

Les 3 usages de **static (3)**

2) **static** associé à un **attribut** partage la valeur de cet attribut avec toutes les instances

- Portée limitée à la classe
- Durée de vie permanente ; pas besoin d'instance
- Accessible par toutes les méthodes de la classe
- DOIT être initialisé en dehors de la classe
- si la valeur initiale n'est pas précisée,
  -> est initialisé avec des 0
  -> ou par le constructeur par défaut

- Est visible dans l'interface d'un module
- NE PAS RENDRE **public**
- Utile pour des paramètres communs
- Sinon, à limiter au strict nécessaire

Aussi OK pour gérer l'ensemble
des instances d'une classe Si
le but de cette classe est de gérer
un seul ensemble d'instances.

Aussi utilisable pour une méthode

myclass.h

```
...
class MyClass {
private:
    static vector<MyClass> tab;
...
```

myclass.cc

```
...
#include "myclass.h"

MyClass::vector<MyClass> tab;// vide

// externalisation de la définition
// des méthodes de la classe MyClass

MyClass::Myclass() ...
{...}
...
```

Extension de la notion de **surcharge**

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

<u>Règle de base:</u>
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-dela des types de base et de la bibliothèque standard

Exemple1:
Pourquoi ne dispose-t-on pas de l'opérateur d'égalité par défaut 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`	`cout.operator+(b)`	`operator<<(cout,b)`
`++a`	`a.operator++()`	`operator++(a)`

Surcharge: **interne** ou **externe ?** **(2)**

Méthode de classe

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)
```

Préférable:
- si un **nouvel** objet est créé
- si on peut éviter l'usage de **`friend`**

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`

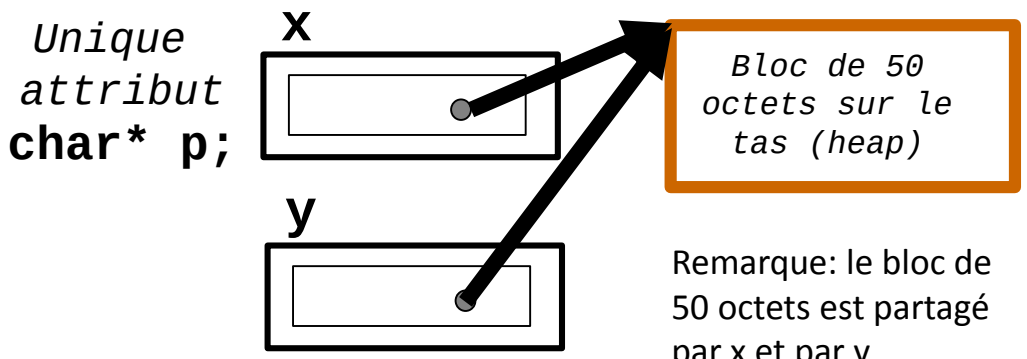
```
operator<<(cout,b)
```

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)
```



Remarque: le bloc de 50 octets est partagé par x et par y

```
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).

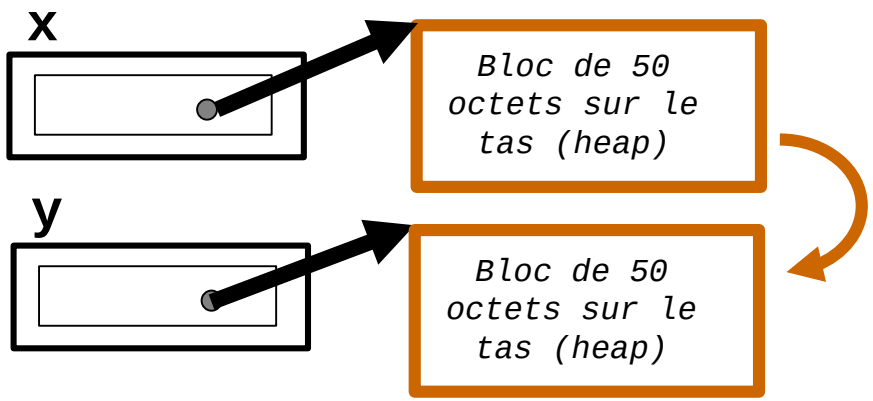
Exemple: surcharge **interne** de l'opérateur d'égalité

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

Scénario 1: Classe avec allocation dynamique et copie **profonde**

```
CADP <==> Classe avec Allocation Dynamique et copie Profonde
CADP x(50);  // constructeur avec allocation
CADP y(x);   // constructeur de copie (copie profonde)
```

Unique attribut **char* p;**



le bloc de 50 octets de x est copié
dans le nouveau bloc de 50 octets de y

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

Qu'obtient-on pour l'expression **x == y** avec le scénario de copie profonde ?

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