

R2.01 : Développement Orienté Objets

C++

François Delobel, Anaïs Durand, Abdel Hasbani, Laurent Provot, Carine Simon

La Surcharge

- ▶ La surcharge indique la possibilité pour plusieurs fonctions de porter le même nom (fonctions homonymes)
- ▶ Le mécanisme qui permet au compilateur de distinguer les fonctions surchargées est basé sur la comparaison des types et du nombre des arguments effectifs et des arguments formels (la signature de la fonction)

La Surcharge (suite)

- ▶ Les fonctions surchargées peuvent être des fonctions standards ou bien des fonctions membres.
- ▶ Les opérateurs (comme par exemple `+`, `<`, `++`, ...) peuvent aussi être surchargés
- ▶ Une application courante de la surcharge consiste à surcharger un constructeur pour créer un nouvel objet de plusieurs manières.
- ▶ Le compilateur choisit la méthode qui correspond le mieux

La Surcharge (suite)

En pratique :

```
int max(int, int);  
int max(int, int, int) ;  
string max(string, string);
```

```
j=max(3,2);  
j=max(10,5,11) ;  
s=max("baleine", "souris");
```

```
double surface(Triangle t) ;  
double surface(Carre c) ;
```

La Surcharge (suite)

Quelques problèmes

- ▶ nécessité de tenir compte des conversions implicites effectuées par le compilateur :

```
void f(float, float){};  
void f(double, double){};  
int main() {  
    f(3,3);  
    return 0 ;  
}
```

```
t.cpp:3: error: call of overloaded 'f(int, int)' is ambiguous  
t.cpp:1: error: candidates are: void f(float, float)  
t.cpp:2: error: candidates are: void f(double, double)
```

copie des objets

La copie d'objet est omniprésente en C et C++ :

- ▶ Affectation explicite (avec écrasement) (`a = b;`)
- ▶ Passage de paramètre (sur la pile) \Rightarrow création d'un nouvel objet (`f(a)`)
- ▶ Retour de valeur (sur la pile) \Rightarrow création d'objet transitoire (`return tmp;`)

Donc deux cas:

- ▶ Écraser le contenu d'un objet avec une nouvelle valeur
Affectation (opérateur de copie ou d'affectation)
- ▶ Construire un objet à partir d'un autre objet de même type "Clôner" l'autre objet (constructeur de copie)

copie des objets(suite)

Constructeur de copie

- ▶ Idée: « clôner » un objet
- ▶ Version par défaut:
 - ▷ Appelle l'opérateur de copie de chacun des attributs
 - Types de base: copie « bit à bit »
 - Amène un problème de partage si pointeur ou référence

copie des objets(suite)

Constructeur de copie

Exemple : pour la classe Contact de la semaine dernière :

```
Contact(const Contact&);
```

Remarque :

- ▶ Usage obligatoire d'une référence
- ▶ Toujours le même prototype

Usage :

- ▶ Appel par `Contact c2{c1}` ou `Contact c2=c1;`
- ▶ Mais pas par `c2=c1;` (affectation, vue plus tard)

Surcharge d'opérateurs

- ▶ Intérêt : Étendre l'utilisation des opérateurs usuels

Exemple avec les nombres complexes :

```
class Complex{ ...} ;  
Complexe z1{1.0, 2.0}, z2{3.0, 4.0}, z3 ;
```

Il est plus naturel d'écrire :

```
z3 = z1 + z2 ;  
cout<<"z3 = "<< z3 << endl ;
```

Que d'écrire : $z3 = \text{add}(z1, z2)$;

ou

```
z3 = z1.add(z2) ;  
afficher(z3) ;
```

Surcharge d'opérateurs

- ▶ Un appel à un opérateur est un appel à une méthode spécifique :
 - ▷ $a \text{ Op } b \Rightarrow \text{operatorOp}(a, b)$ ou $a.\text{operatorOp}(b)$;
 - ▷ $\text{Op } a \Rightarrow \text{operatorOp}(a)$ ou $a.\text{operatorOp}()$;

Exemples :

$a + b \Rightarrow \text{operator}+(a, b)$ OU $a.\text{operator}+(b)$;

$--a \Rightarrow \text{operator}--(a)$ OU $a.\text{operator}--()$;

$\text{cout} \ll a \Rightarrow \text{operator} \ll (\text{cout}, a)$ OU $\text{cout}.\text{operator} \ll (a)$;

$a = b \Rightarrow a.\text{operator}=(b)$;

Surcharge d'opérateurs

De la même manière que la surcharge de fonctions, on peut surcharger tout opérateur (sauf ., ::, #) :

```
Complexe operator+(Complexe, Complexe) ;  
Duree operator+(Duree, Duree) ;  
Matrice operator+(Matrice, Matrice) ;
```

Surcharge d'opérateurs

La surcharge des opérateurs peut être réalisée de deux façons différentes :

- ▶ la **surcharge externe** : utilise une **fonction (amie !)** à l'extérieur de la classe

```
Complexe operator+(Complexe, Complexe) ;  
z3 = operator+(z1, z2) ; //c-à-d z3 = z1 + z2 ;
```

- ▶ la **surcharge interne** : utilise une **méthode** à l'intérieur de la classe

```
class Complexe{  
public :  
    Complexe operator+(Complexe) const ;  
    ...  
};  
z3 = z1.operator+(z2) ;
```

Surcharge des opérateurs << et >>

L'opérateur << est surchargé pour afficher les éléments d'une instance d'une classe en faisant référence à une instance (cout) de la classe de sortie (ostream).

Il renvoie une référence sur le flux de sortie (cout) ce qui permet l'appel du même opérateur et donc l'enchaînement des opérations d'affichage.

```
class Point {
private :
    int x, y ;
public :
    ...
    friend ostream& operator<<(ostream&, const Point & ) ;
};
ostream& operator<<(ostream& s, const Point & p){
    s <<"abs= " <<p.x<<" " <<"ord= " <<p.y<<"\n " ;
    return s ;
}
```

Forme Canonique (de Coplien)

Une classe A est sous forme canonique ssi elle définit :

- ▶ Un constructeur par défaut (`A::A()`)
- ▶ Un constructeur de copie (`A::A(const A&)`)
- ▶ Un destructeur (`A::~~A()`)
- ▶ Un opérateur d'affectation (`A::operator=(const A&)`)

Le compilateur ajoute une version simplifiée de ces méthodes si elles ne sont pas définies par l'utilisateur (cas particulier du constructeur par défaut)

Parfois indispensable de les redéfinir si les versions par défaut ne suffisent pas

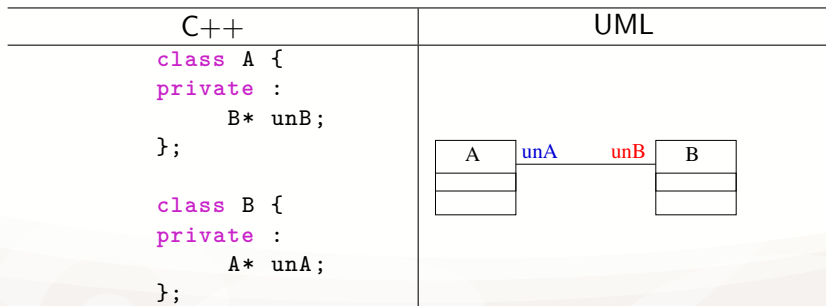
Relations entre objets : UML/C++

5 types de relations de base entre les classes :

- ▶ 4 relations durables
 - ▷ l'association (trait plein avec ou sans flèche)
 - ▷ la composition (trait plein avec ou sans flèche et un losange plein)
 - ▷ l'agrégation (trait plein avec ou sans flèche et un losange vide)
 - ▷ la relation de généralisation ou d'héritage (flèche fermée vide)
- ▶ relation temporaire
 - ▷ la dépendance (flèche pointillée)

Relations entre objets : UML/C++

Relation d'association




- Ici, la relation est bidirectionnelle (pas de fêche), on a une navigabilité dans les deux sens. A « connaît » B et B « connaît » A.

Relations entre objets : UML/C++

Relation d'agrégation

Dans une agrégation, le composant peut être partagé entre plusieurs composites ce qui entraîne que, lorsque le composite A sera détruit, le composant B ne le sera pas forcément.

C++	UML
<pre>class A { private : B* unB; }; class B { private : };</pre>	

Relations entre objets : UML/C++

Relation de composition

Une composition est une agrégation plus forte signifiant « est composée d'un » et impliquant :

- ▶ une partie ne peut appartenir qu'à un seul composite (agrégation non partagée)
- ▶ la destruction du composite entraîne la destruction de toutes ses parties (il est responsable du cycle de vie de ses parties).

