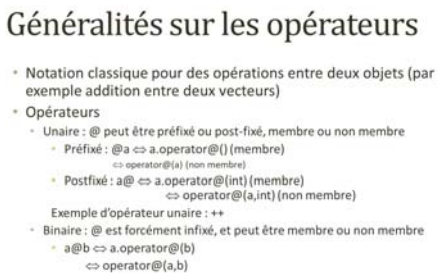
Chapitre 5. La surcharge d'opérateurs

Introduction

La surcharge (fait d'écrire une fonction portant le même nom qu'une autre, mais avec des paramètres différents en type et/ou en nombre) a l'intérêt de simplifier la rédaction du code.

On a déjà vu que l'on peut surcharger des fonctions membres de classes ou des fonctions non membres. On peut aller plus loin et surcharger des opérateurs (addition, soustraction, etc.) pour permettre d'utiliser des notions classiques entre deux objets ou deux types construits (par exemple une affectation directe entre deux structures ou deux objets d'une même classe).

Généralités concernant les opérateurs



Slide 2

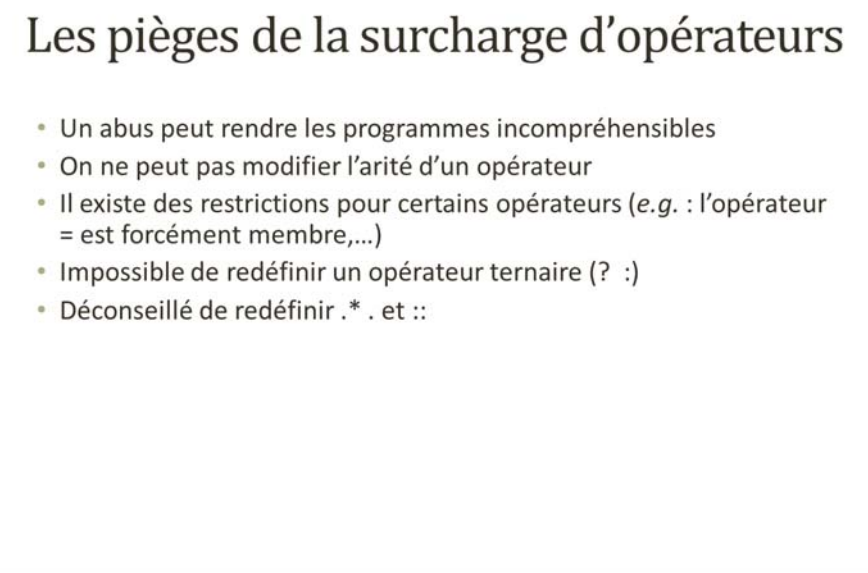
Il existe différents types d'opérateurs applicables entre deux objets, on distingue notamment opérateurs binaires et unaires, ceux qui sont préfixés, postfixés ou infixés.

1) Les opérateurs unaires n'impliquent qu'un seul objet. Ils peuvent être préfixés ou postfixés.
   Exemple d'opérateur unaire qui peut être préfixé ou postfixé :
   l'opérateur ++
   - préfixé : ++i se traduit par operator++( )
   on pourrait l'appeler ainsi i.operator++( )
   incrémente i.
   La valeur de l'expression est celle de i après incrémentation.
   - postfixé : i++ se traduit par operator++(int)
   on pourrait l'appeler ainsi operator++(i)
   incrémente i.
   La valeur de l'expression est celle de i avant incrémentation.
2) Les opérateurs binaires, forcément infixés (voir notation sur le slide)

Chacun de ces types d'opérateur peut être surchargé pour être adapté à des types de données définis par l'utilisateur, par exemple.

Si l'on voit bien l'intérêt de surcharger les opérateurs classiques pour des types qui ne seraient pas des types de base, notamment dans le but de simplifier et rendre plus intuitif le code, il y a tout de même quelques pièges à l'usage de la surcharge.

Les pièges de la surcharge d'opérateurs



Slide 3

Exemple de surcharge pour l'opérateur de comparaison

```
class Complexe {
double re, im ;
public :
        Complexe(double r, double i=0) : re(r), im(i) {}
        double real() const {return re ;}
        double imag() const {return im ;}
        bool operator==(Complexe c2) {
                return this->re==c2.real() && this->im==c2.imag();};
        };

int main() {
Complexe c(4.0);
cout << (c == 4.0) << endl; // (1)
cout << (5.0 == c) << endl; // (2)
cout << c == 0 << endl; // (3) obligation de rajouter des parenthèses sinon
                                //notre type Complexe ne peut être traité par l'opérateur <<
return 0;
}
```

Dans le cas des opérations suivantes : 5.0 == C et C == 4.0 le constructeur de la classe Complexe est utilisé pour convertir un double en complexe avant comparaison (on a spécifié une valeur par défaut pour la partie imaginaire).
Dans le cas (3) :
error: no match for 'operator<<' in 'std::cout << c'
/usr/include/c++/4.3/ostream:112: note: candidates are:
cout << c <=> std::cout.operator<<(c)

La surcharge de ==

```
class complexe {
  double re, im ;
public :
  complexe(double r, double i=0) { /*…*/ }
  double reelle() const {return re ;}
  double imag() const {return im ;}
};

bool operator==(complexe c1, complexe c2)
{ return c1.real()==c2.real() && c1.imag()==c2. imag (); }
```

- Opérations permises : `c1==c2; 1.0==c1; c1==1.0;`
- Utilisation du constructeur pour les conversions
- Remarque : la surcharge de l'opérateur + fonctionne comme celle de == (sauf pour la valeur de retour)
- Attention aux conversions implicites

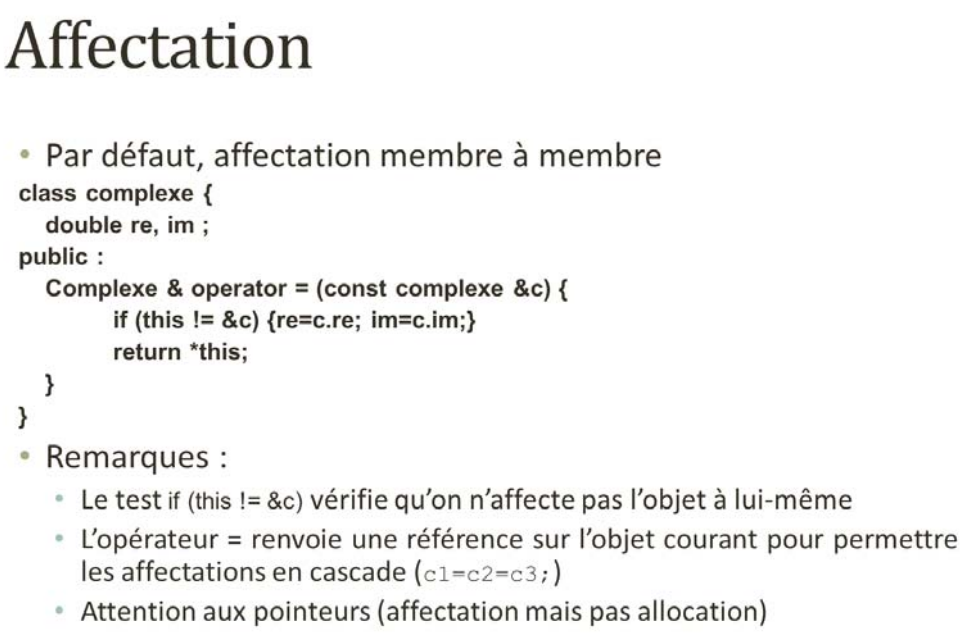
slide 4

Opérateur d'affectation

Par défaut, l'affectation se fait membre à membre, ce qui est suffisant pour pas mal de types, mais peut poser problème pour les structures contenant des pointeurs, par exemple.
La surcharge proposée ici pour la classe Complexe reproduit le comportement de l'opérateur d'affectation par défaut.

```
class complexe {
        double re, im ;
public :
        Complexe & operator = (const complexe &c) {
                if (this != &c) {re=c.re; im=c.im;}
                return *this;
        }
}
```

Dans ce contexte, :
- on utilise l'auto-référence, this de façon à éviter de faire une auto-affectation
- on renvoie une référence sur un objet de type Complexe, ce qui permet d'effectuer des affectations en cascade



Slide 5

La particularité du constructeur de copie

Le constructeur de copies est modifiable à la création de la classe. Il n'est pas appelé explicitement dans le code mais lorsqu'on construit une instance à partir d'une autre.
Le constructeur de copies porte le même nom que la classe, mais prend en paramètre une référence sur un objet du type de cette classe : il permet de construire une nouvelle instance de classe à partir de celle passée en paramètre.
Par défaut, le constructeur de copie copie les champs de l'instance modèle dans l'instance à créer… dans le cas des pointeurs, il copie le pointeur mais ne fait pas l'allocation mémoire : il faut donc impérativement définir le constructeur de copies quand il y a des membres de type pointeur.

Exemple sur la classe Complexe :

Constructeur de copie

- Construit une instance à partir d'une instance existante
- Il en existe un par défaut
- Il faut impérativement le définir quand il y a des membres de type pointeur
- Exemple :

```
class complexe {
        double re, im ;
        public :
                //...
                complexe(const complexe &c) : re(c.re), im(c.im) {}
                //...
}
```

- Remarques :
  - Passage de c par référence pour éviter un appel infini au constructeur de copies
  - Accès aux composants de c (privés) car le contrôle se fait au niveau de la classe.

Slide 6

On a accès aux composants privés car le contrôle d'accès se fait au niveau de la classe et non au niveau des objets.
La règle des 3-5-0 (Ste Trinité) mentionnée dans les chapitres précédents s'applique au constructeur de copies.

Exercice collectif

Affectation ou constructeur de copies ?

```
complexe c1(1.0,2.0) ;
complexe c2=c1 ;
c2=c1 ;
```

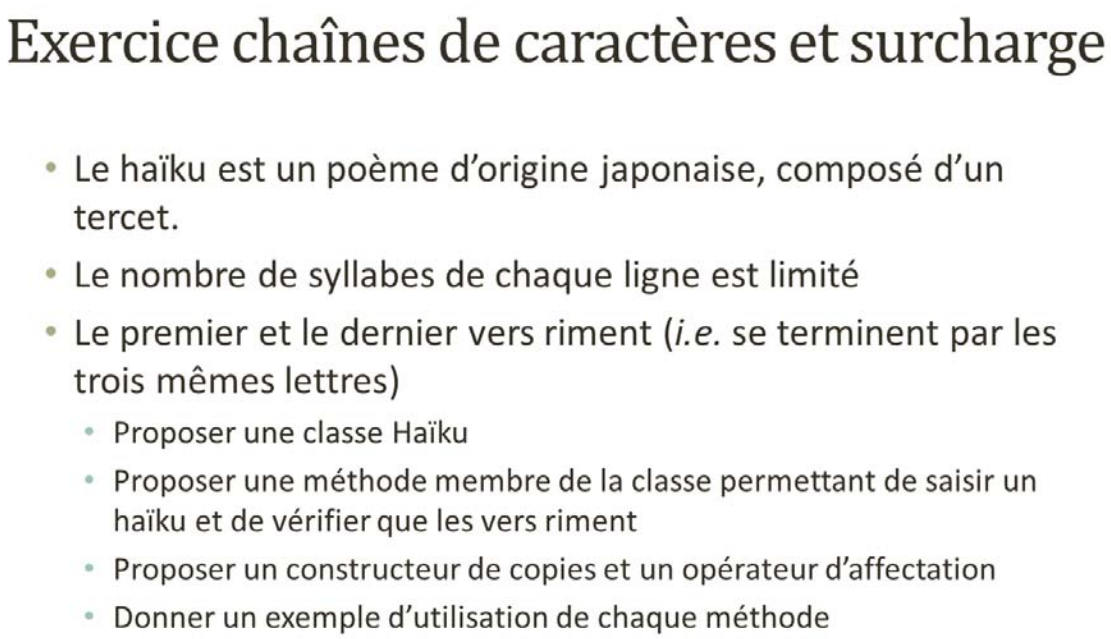
Où est utilisé le constructeur de copies, où est utilisée l'affectation ?

Slide 7

Solution :
Complexe C1(1.2,2.0) ; // constructeur avec paramètres
Complexe C2=C1 ; // constructeur de copies équivalent à Complexe C2(C1)
C2=C1 ; // opérateur affectation, la copie se fait membre à membre si l'opérateur d'affectation n'a pas été surchargé au préalable.

Pour aller plus loin et coder…

Un cas d'école avec des chaînes de caractères et de la surcharge



Slide 8