

POO avec C++

chapitre 3

Surcharge des opérateurs



2. Table des matières

1. Introduction

- 2. Techniques de surcharge
- 3. Surcharge de `<<` et `>>`
- 4. Surcharge de `=`
- 5. Autres surcharges (`++`, `--`, ...)
- 6. Résumé

```
Complex c1 (2,3) , c2 (1,1) , c3 ;
```

- Comment additionner c1 et c2 ?

```
c3 = Complex::add(c1, c2) ;
```

```
c3 = c1.add(c2) ;
```

```
Complex c1 (2,3) , c2 (1,1) , c3;  
c3 = c1 + c2;  
cout << c1 << endl
```

3.1 Surcharge des opérateurs

Pour manipuler les objets **comme des types simples**

Exemple: surcharge des opérateurs +, << dans la classe `Complex`

```
Complex c1 (2,3) , c2 (1,1) , c3;  
c3 = c1 + c2;  
cout << c1 << endl
```

- On ne peut **pas redéfinir** les opérateurs sur les types primitifs
 - Seulement pour les classes/struct définies par le programmeur
 - **Seuls les opérateurs existants peuvent être surchargés**
- ✓ Essayer de conserver la **sémantique** initiale de l'opérateur!

3.1 Surcharge des opérateurs

Ces 53 opérateurs peuvent être surchargés:

=	+	-	*	/	^	%
==	+=	-=	*=	/=	^=	%=
<	<=	>	>=	<<	>>	<<=
>>=	++	--	&		!	&=
=	!=	,	[]	()	&&	
~	xor	xor_eq	and	and_eq	or	or_eq
not	not_eq	bitand	bitor	compl		

-> ->* new new[] delete delete[]

Dangereux!

Ces 5 opérateurs ne peuvent pas être surchargés:

. .* :: ?: sizeof

Interdit!

3.1 Surcharge des opérateurs

Pour chaque opérateur ♦, il existe une méthode **operator♦ (...)**

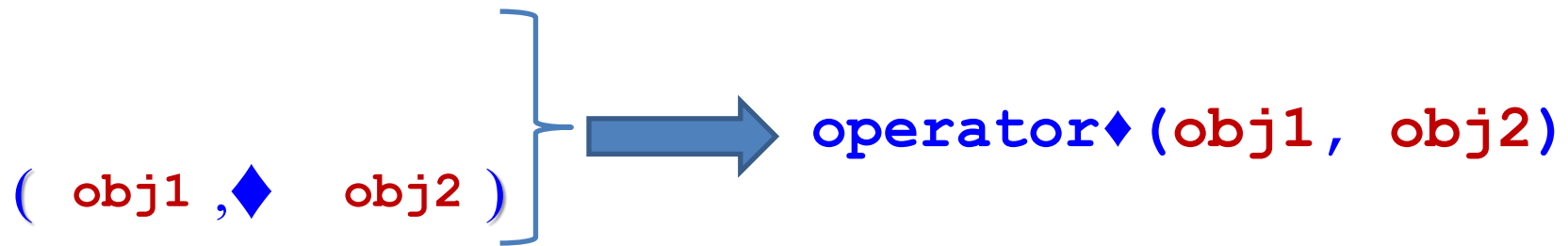
c1+c2; est transformé en : **c1.operator+** (c2) ;

c1=c2; est transformé en : **c1.operator=** (c2) ;

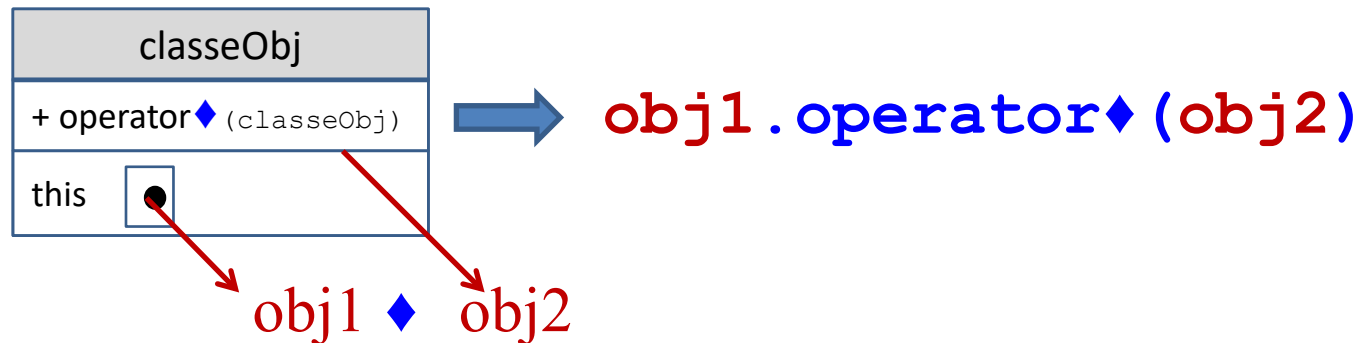
p1=p2=p3; → **p1.operator=** (p2.operator= (p3)) ;

3.2 Surcharge des opérateurs

Surcharge d'un opérateur binaire par une **fonction non membre**



Surcharge d'un opérateur binaire par une **fonction membre**





3.2 Surcharge des opérateurs unaires

Exemples d'opérateurs unaires: **!x**, **x++**, **~x**

Surcharge d'un opérateur unaire par:

a) une **fonction membre**:

obj♦ ou **♦obj** \equiv **obj.operator♦()**

Exemple: **!x** \equiv **x.operator!()**

b) une **fonction non membre (de la classe)**:

obj♦ ou **♦obj** \equiv **operator♦(obj)**

Exemple: **!x** \equiv **operator!(x)**

3.2 Surcharge des opérateurs

Opérateur binaire: $c1+c2$, $c2/c3$, ...

Surcharge d'un opérateur binaire par:

A) une **fonction membre**

$\llbracket \text{obj1} \blacklozenge \text{obj2} \rrbracket \equiv \llbracket \text{obj1.operator} \blacklozenge (\text{obj2}) \rrbracket$

Exemple: $c1+c2 \equiv c1.operator+(c2)$

B) une **fonction non membre**

$\llbracket \text{obj1} \blacklozenge \text{obj2} \rrbracket \equiv \llbracket \text{operator} \blacklozenge (\text{obj1}, \text{obj2}) \rrbracket$

Exemple: $c1+c2 \equiv \text{operator}+(c1, c2)$

3.2 Surcharge des opérateurs

A) Surcharge d'un opérateur par une **fonction membre**

Complex.h

```
class Complex
{
    double r, i;
public:
    ...
    Complex operator+(const Complex & ) const;
};
```

L'objet passé en argument n'est pas modifiable

L'objet courant (*this) n'est pas modifiable

Complex.cpp

```
Complex Complex::operator+(const Complex &c) const
{
    return Complex(r + c.r, i + c.i);
}
```

3.2 Surcharge des opérateurs

A) Surcharge d'un opérateur par une **fonction membre**

main.cpp

```
Complex c1, c2, c;  
int x, y;  
  
c = c1 + c2; // OK: c = c1.operator+(c2);  
c = c1 + y; // OK: c = c1.operator+(Complex(y));  
c = x + c2; // Erreur : x n'est pas un objet Complex  
               x.operator() n'existe pas!
```

Appel implicite à un constructeur de
conversion (s'il existe!)

Problème: Opération non commutative !!



3.2 Surcharge des opérateurs

Solution pour avoir une opération commutative

B1) Surcharge d'un opérateur par une **fonction non membre**

```
Complex operator+(const Complex &x,  
                  const Complex &y) {  
    return Complex(x.getR() + y.getR(), x.getI() + y.getI());  
}
```

B2) Surcharge d'un opérateur par une **fonction non membre AMIE**

Dans la classe *Complex*:

```
friend Complex operator+(const Complex&,  
                          const Complex&);
```

Hors de la classe *Complex*:

```
Complex operator+(const Complex &x,  
                  const Complex &y) {  
    return Complex(x.r + y.r, x.i + y.i);  
}
```

3.2 Surcharge des opérateurs

B) Surcharge d'un opérateur par une **fonction non membre**

Complex **operator+**(const Complex &x, const Complex &y);

main.cpp

```
Complex p, q, r;
```

```
int x, y;
```

```
c = c1 + c2; // OK: c = operator+(c1, c2);
```

```
c = c1 + y; // OK: c = operator+(c1, Complex(y));
```

```
c = x + c2; // OK: c = operator+(Complex(x), c2);
```

```
c = x + y; // ?
```

Appel implicite à un constructeur de conversion, pour autant qu'il existe

3.3 Exemples courants (<<, >>)

Les opérateurs >> et << permettent d'insérer et d'extraire des objets dans les flux

Fonction non membre:

```
ostream& operator<<(ostream &s, const Point &p)
{
    return s << '(' << p.X() << ',' << p.Y() << ') ' ;
}
```

- L'opérande de gauche est une **référence** sur le flux `ostream&`
- Cette fonction ne pourra pas être membre de la classe `Point`
- Le renvoi d'une référence permet de l'utiliser comme *Lvalue*. On pourra donc faire des appels en cascade :

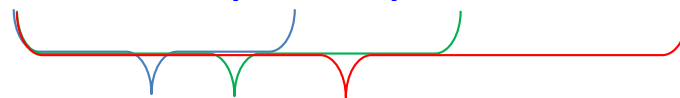
```
cout << p1 << p2 << endl;
```

```
operator<<(operator<<(operator<<(cout, p1), p2), endl);
```

`ostream&`
`ostream&`

3.3 Exemples courants (<<, >>)

```
cout << p1 << p2 << endl;
```



```
ostream&  
operator<<(operator<<(operator<<(cout, p1), p2), endl);
```



```
ostream& operator<<(ostream &s, const Point &p);
```




3.3 Exemples courants (<<, >>)

Exemple d'utilisation d' **operator>>**

```
void main()
{
    Point p;
    cout << " Donner un point au format (x,y) : ";
    cin >> p;
    cout << "le point est : " << p << "\n";
}
```

On veut que l'utilisateur respecte un format prédéfini:
parenthèse-X-virgule-Y-parenthèse

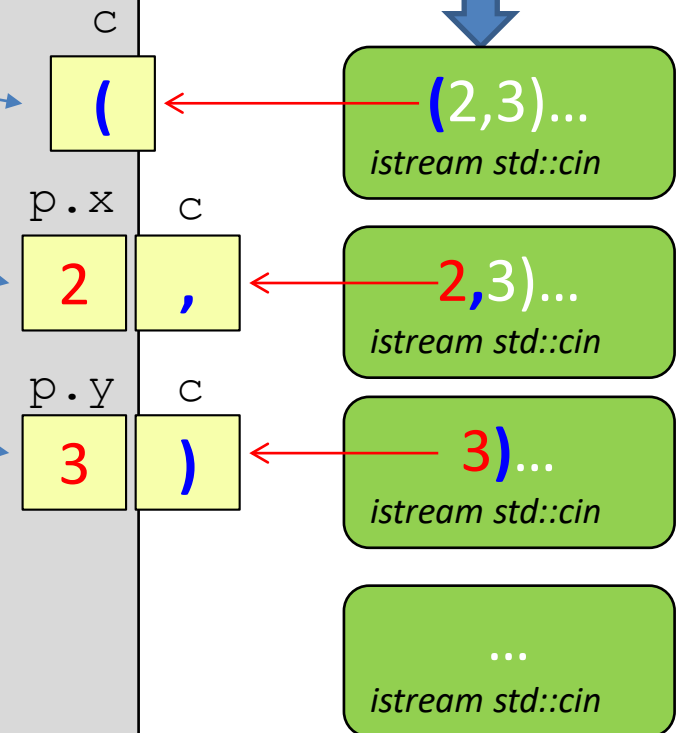
```
Donner un point : (2,3)
le point est : (2,3)
```

3.3 Exemples courants (<<, >>)

```
istream& operator>>(istream& i, Point& p)
{
    char c;
    i >> c;
    if (c == '(')
    {
        i >> p.x >> c;
        if (c == ',')
        {
            i >> p.y >> c;
            if (c == ')')
                return i;
        }
    }
    cerr << "Erreur de lecture.\n";
    exit(-1);
}
```



Enter



3.4 Surcharge de l'opérateur d'affectation

- L'affectation est une opération prédéfinie
- Copie de surface (copie membre à membre)

Exemple:

```
class NamedPoint
{
public:
    NamedPoint(int = 0, int = 0, char* = nullptr);
    NamedPoint(const NamedPoint &);
    ~ NamedPoint();
private:
    int x, y;
    char* name;
};
```



3.4 Surcharge de l'opérateur d'affectation



```
NamedPoint :: NamedPoint(int a, int b, char* s)
{
    x = a; y = b;
    name = new char[strlen(s) + 1];
    strcpy(label, s);
}

NamedPoint::NamedPoint(const NamedPoint& p) //constructeur ...
{
    x = p.x; y = p.y;
    name = new char[strlen(p.name) + 1];
    strcpy(name, p.name);
}

NamedPoint ::~ NamedPoint()
{
    delete [] name;
}
```



3.4 Surcharge de l'opérateur d'affectation



```
void main()
{
    NamedPoint pt(1, 4, "P1"), copyPt;
    NamedPoint copyPt2 = p;
    copyPt = pt;
}
```

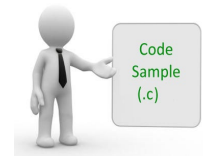
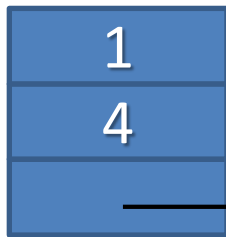
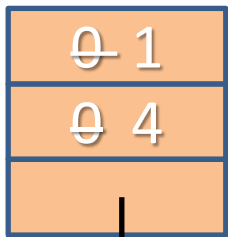
Constructeur par recopie
différent de ...

Affectation

copyPt

pt

copyPt2



02_SurchargeAffectation

3.4 Surcharge de l'opérateur d'affectation

→ On doit le **redéfinir** pour une copie en **profondeur**

A éviter →

```
void NamedPoint::operator=(const NamedPoint & p)
{
    if (this != &p)
    {
        x = p.x; y = p.y;
        if (name != nullptr)
            delete [] name;
        name = new char[strlen(p.name)+1];
        strcpy(name, p.name);
    }
}
```

! Evite une copie inutile

Que se passe-t-il avec?

```
p1 = p2 = p3;
```

```
p1.operator=(p2.operator=(p3)) ;
```

3.4 Surcharge de l'opérateur d'affectation

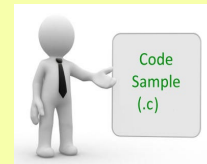
➔ On doit le **redéfinir** pour une copie en **profondeur**

Retour par
référence

```
Point & Point::operator=(const Point & p)
{
    if(this != &p)
    {
        x = p.x; y = p.y;
        if(name != 0)
            delete [] name;
        name = new char[strlen(p.name)+1];
        strcpy(name, p.name);
    }
    return *this;
}
```



Surcharge.pdf



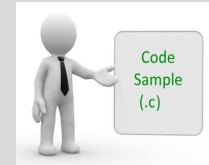
02_SurchargeAffectation

3.5 Surcharge des opérateurs de conversion

```
class Point
{
    int x, y;

    ...
    // Conversion int → Point Constructeur
    Point(int a) { x = a; y = 0; }

    // Conversion Point → int Surcharge op
    operator int() { return abs(x) + abs(y); }
};
```



05_SurchargeConversion



Série 3.1

Série 3.2

3.5 Surcharge des opérateurs ++, --

Cas particulier : des opérateurs unaires préfixés et postfixés.

Préfixé: `++p; // \Leftrightarrow c.operator++()`

`Classe & operator++();`

Retour par référence

Pas d'argument

Postfixé: `p++; // \Leftrightarrow c.operator++(1)`

`Classe operator++(int);`

Retour par valeur

Argument

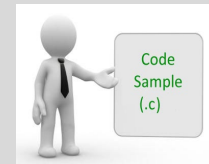
Astuce pour avoir de signatures différentes

3.5 Surcharge des opérateurs ++, --

```
class XYZ
{
    int x,y,z;

    XYZ& operator++()          // pré incrémentation ++x
    {
        ++x;
        return *this;
    }

    XYZ operator++(int)        // post incrémentation x++
    {
        XYZ temp(*this);      // XYZ temp = *this;
        ++x;
        cout << "Copie de retour ";
        return temp;
    }
}
```



04_SurchargeOpérateurPrePostInc

3.6 Surcharge d'opérateurs (résumé)

```
class Classe
```

```
{ ...
```

```
// Prototype de l'opérateur Op
```

```
type_retour operatorOp (type_argument) ;
```

```
...
```

```
};
```

```
// Définition de l'opérateur Op membre
```

```
type_retour Classe::operatorOp (type_argument arg) { ... }
```

```
// Opérateur externe (non membre)
```

```
type_retour operatorOp (type_argument arg1, Classe & arg2) { ... }
```

3.6 Surcharge d'opérateurs (résumé)

Quelques exemples de prototypes :

`bool operator==(const Classe&) const;`

`bool operator<(const Classe&) const;`

`Classe& operator=(const Classe&);`

`Classe& operator+=(const Classe&);`

`Classe& operator++();`

`Classe operator++(int);`

`Classe& operator*=(const autre_type);`

`Classe operator-() const;`

`<type>& operator[](int);`

exemple

`// p == q`

`// p < q`

`// p = q`

`// p += q`

`// ++p`

`// p++`

`// p *= x`

`// q = -p`

`// vecteur[5]`

Opérateurs externes :

`ostream& operator<<(ostream&, const Classe&);`

`Classe operator*(const autre_type, const Classe&);`

`Classe operator-(const Classe &, const Classe&);`

`// std::cout << p`

`// 3.14 * p`

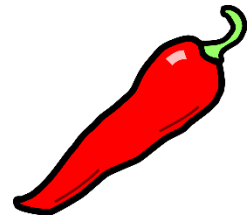
`// r = p - q`



The BIG RULE OF THREE

*«Si vous **devez** définir l'une des trois fonctions de base que sont le **constructeur de copie**, l'**opérateur d'affectation** ou le **destructeur**, alors, vous devrez définir les trois.»*

Foncteurs



Surcharger ... `operator()` (...)

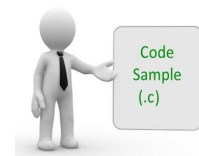
permet d'utiliser une classe comme une fonction
(*FunctionObjects*)

```
class Linear
{
    double a, b;

public:
    Linear(int _a, int _b) : a(_a), b(_b) {}

    double operator()(double x) const
    {
        return a * x + b;
    }
};
```

```
Linear f(2, 1); // 2x + 1
f(3.5);        // 8
```



06_ADV_Foncteurs