

LE LANGAGE C++ MASTER 1 LA SURCHARGE D'OPÉRATEUR

Jean-Baptiste.Yunes@u-paris.fr
U.F.R. d'Informatique
Université de Paris

10/2021



GÉNÉRALITÉS

- Les opérateurs surchargeables sont :

+	-	*	/	%
^	&		~	!
=	<	>	+=	-=
*=	/=	%=	^=	&=
=	<<	>>	>>=	<<=
==	!=	<=	>=	&&
	++	--	->*	,
->	[]	()	new	new[]
delete	delete[]	(<i>type</i>)		

- Tous sauf :
 - l'opérateur de portée ::
 - la sélection de membre .
 - la sélection *via* pointeur .*
 - l'opérateur conditionnel ? :
 - le calcul de taille `sizeof`
 - la réflexion `typeid`

**TECHNIQUE OPÉRATEUR
COMME FONCTION MEMBRE
(MÉTHODE D'INSTANCE)**

But : donner un sens à des expressions utilisant un ou des opérateurs du C++ qui combinent des valeurs dont au moins l'un des types est défini par l'utilisateur.

Autoriser l'écriture d'expression en s'exprimant dans le langage du domaine

- `Matrix M, M1, M2;`
`M = M1*M2;`

- `Gateau g; Farine f; Oeuf o; Beurre b; Sucre s;`
`g = f+o+b+s;`

- En première approximation

`o1 + o2`

peut se lire

`o1.+(o2)`

- Ainsi l'opérateur `+` peut être vu comme une méthode de la classe de l'objet `o1`

- C'est la première technique de surcharge des opérateurs

Les opérateurs vus comme méthode d'instance

- La convention est que la méthode surchargeant l'opérateur *op* s'appelle `operatorop`, ainsi :

`o1+o2` est traduit en

`o1.operator+(o2)`

- Remarque : les **deux** écritures sont valides

- Ainsi :

```
class Nombre {  
private:  
    int valeur;  
public:  
    Nombre(int v=0) { valeur = v; };  
    Nombre operator+(const Nombre &) const;  
};
```

```
Nombre Nombre::operator+(const Nombre &n) const {  
    return Nombre(this->valeur+n.valeur);  
}
```

```
void main() {  
    Nombre n, n1(123), n2(256);  
    n = n1+n2;  
    // que l'on peut écrire aussi !?  
    n = n1.operator+(n2);  
}
```

- C'est bien entendu un idéal, il existe des exceptions à cette *traduction* littérale.
- D'abord, certains opérateurs ne peuvent être surchargés (`.`, `?:`, `sizeof`, `::`, `typeid`, `*` et les `_cast`), donc tous les autres peuvent l'être...
- Les opérateurs unaires préfixes : `~`, `++`, `--`, `-`, `+`, etc. Pour lesquels la *traduction* de *op variable* est *variable.operatorop()*

- Les opérateurs unaires suffixes : ++, -- :

ici il s'agit de distinguer la forme préfixe de la forme suffixe!!! On introduit donc !? un argument fictif !? et

variable++

est capturé par la définition de

variable.operator++(int)

L'argument `int` ne sert à rien et il ne faut pas l'utiliser!

- Les opérateurs parenthésés : `()`, `[]`
- dans le cas des crochets, l'argument est habituellement de type `int` et correspond à l'*indice*

```
variable[i]
```

se traduit par

```
variable.operator[ ](i)
```

- Note : on peut surcharger cet opérateur avec différents types pour l'*indice*

- Ainsi :

```
class Collection {  
private:  
    Element *lesElements;  
    int nombreDElements;  
public:  
    Element operator[](int) const;  
};
```

```
Element Collection::operator[](int indice) const {  
    if (indice < 0 || indice >= nombreDElements)  
        throw OutOfBounds(indice, nombreDElements);  
    return lesElements[indice];  
}
```

```
void main() {  
    Collection c;  
    Element e = c[10];  
    c[4] = e; // interdit  
}
```

- Ainsi :

```
class Collection {
private:
    Element *lesElements;
    int nombreDElements;
public:
    Element &operator[](int) const;
};
```

```
Element &Collection::operator[](int indice) const {
    if (indice<0 || indice>=nombreDElements)
        throw OutOfBounds(indice,nombreDElements);
    return lesElements[indice];
}
```

```
void main() {
    Collection c;
    Element e = c[10];
    c[4] = e;
}
```

```
class AssociativeArray {
private:
    int t[3];
public:
    int &operator[](const string &k) {
        if (k=="zero") return t[0];
        if (k=="un") return t[1];
        return t[2];
    }
};
```

```
[yunes] ./assoc
123
321
[yunes]
```

```
int main() {
    AssociativeArray t;
    t["un"] = 123;
    t["deux"] = 321;
    cout << t["un"] << endl;
    cout << t["deux"] << endl;
}
```

- dans le cas des parenthèses, on peut obtenir la surcharge avec un nombre d'arguments quelconque :

`objet(), objet(23), objet(5,4,9)`

```
class Convertisseur {
private:
    double taux;
public:
    Convertisseur(double tx) { taux = tx; };
    void changeTaux(double tx) { taux = tx; }
    double operator()(double) const;
    double operator()(double, double) const;
};
```

```
double Convertisseur::operator()(double v) const {
    return v*taux;
}
double Convertisseur::operator()(double v, double tx) const {
    changeTaux(tx); return v*taux;
}
```

```
void main() {
    Convertisseur euroDollar(1.4306);
    cout << euroDollar(15) << endl;
    euroDollar.changeTaux(1.4208);
    cout << euroDollar(15) << endl;
}
```

- L'opérateur de conversion :

- Une conversion de la forme (à la C)

(type) expression

- peut s'écrire en notation fonctionnelle (à la C++)

type(expression)

- et s'implémente comme une fonction membre de la classe de l'expression

```
operator type() const { ... }
```

```
class Fraction {
private:
    int numérateur, dénominateur;
public:
    Fraction(int n,int d) {
        numérateur = n;
        dénominateur = d;
    }
    operator double() {
        return (double)numérateur/dénominateur;
    }
};

int main() {
    Fraction unDemi(1,2);
    double d = (double)unDemi; // d = double(unDemi)
    cout << d << endl;
}
```

- On peut alors remarquer qu'un constructeur joue aussi (dans certains cas) le rôle d'un opérateur de conversion :
- un constructeur à un argument permet de créer à partir d'une expression d'un type donné un objet d'un autre type
- c'est si proche de la définition d'une conversion que le C++ considère que c'est effectivement une conversion

```

class Fraction {
private:
    int numérateur, dénominateur;
public:
    Fraction(int n) {
        numérateur = n;
        dénominateur = 1;
    }
    Fraction(int n,int d) {
        numérateur = n;
        dénominateur = d;
    }
    operator double() {
        return (double)numérateur/dénominateur;
    }
};

int main() {
    Fraction f(25,12);
    f = 2;
    Fraction unDemi(1,2);
    double d = (double)unDemi;
    cout << d << endl;
    double d2 = (double)f;
    cout << d2 << endl;
}

```

Conversion de 2 en Fraction
par appel à Fraction(2)

```

[yunes] ./main
0.5
2
[yunes]

```

**TECHNIQUE OPÉRATEUR
COMME FONCTION
(ORDINAIRE)**

- Il existe des cas dans lesquels la technique *opérateur comme méthode de classe* se révèle impossible par exemple lorsque l'opérande gauche (ou principale) est d'un type dont on ne contrôle pas la définition, *i.e.* type primitif (même pas une classe) ou type classe mais donné sous forme de bibliothèque ou dont on ne contrôle pas le code source

Les opérateurs vus comme fonctions statiques

- En ce cas, il suffit de considérer, par exemple, que $o_1 + o_2$ se lit $+(o_1, o_2)$
- Remarque : l'addition est bien une fonction à deux opérands (arité 2)

- Ainsi :

```
class Nombre {  
private:  
    int valeur;  
public:  
    Nombre(int v) { valeur = v; };  
    int getValeur() const { return valeur; }  
};
```

```
Nombre operator+(const Nombre &n1, const Nombre &n2) {  
    return Nombre(n1.getValeur()+n2.getValeur());  
}
```

```
void main() {  
    Nombre n, n1(123), n2(256);  
    n = n1+n2;  
    // que l'on peut écrire aussi !?  
    n = operator+(n1,n2);  
}
```

- L'usage premier et le plus courant de la surcharge concerne les entrées/sorties. Il s'agit d'uniformiser les affichages des objets avec ceux des types primitifs...

```
Matrix m;
```

```
cout << "Voici la matrice " << m << endl;
```

- Tout d'abord, l'affichage de *base*

```
cout << m;
```

- **La classe de cout est ostream :**

```
class Nombre {  
private:  
    int valeur;  
public:  
    Nombre(int v) { valeur = v; };  
    int getValeur() const { return valeur; }  
};
```

```
void operator<<(ostream &os, const Nombre &n) {  
    os << n.getValeur();  
}
```

- **Attention : le flux est modifié donc pas const...**

```
int main() {  
    Nombre n(256);  
    cout << n;  
}
```

- **Le problème est que**

```
cout << "Voici un nombre " << n;
```

fonctionne mais pas

```
cout << n << endl;
```

- **Rappel : l'opérateur << est associatif à gauche, donc l'expression**

```
((cout << n) << endl);
```

se traduit par

```
operator<<(operator<<(cout,n),endl);
```

- **il faut donc renvoyer une valeur!**

- Il faut donc renvoyer le flux lui-même et par référence ! :

```
class Nombre {  
private:  
    int valeur;  
public:  
    Nombre(int v) { valeur = v; };  
    int getValeur() const { return valeur; }  
};
```

```
ostream &operator<<(ostream &os, const Nombre &n) {  
    os << n.getValeur();  
    return os;  
}
```

```
int main() {  
    Nombre n(256);  
    cout << n << endl;  
}
```

- Attention, la technique fonction (ordinaire) ne peut être employée
- ni pour les opérateurs parenthésés,
- ni pour les opérateurs d'affectation...

**DES SURCHARGES
D'OPÉRATEURS ÉTRANGERS...**

- l'opérateur ,

```

class A {
private:
    int x, y;
public:
    A(int x,int y) : x(x), y(y) {};
    A() : x(0), y(0) {};
    A &operator=(int x) { this->x = x; return *this; }
    void operator,(int y) { this->y = y; }
    friend ostream &operator<<(ostream &o,const A &a);
};
ostream &operator<<(ostream &o,const A &a) {
    return o << "(x:" << a.x << ",y:" << a.y << ')';
}

```

```

int main() {
    A a;
    cout << a << endl;
    a = 3,4;
    cout << a << endl;
    return 0;
}

```

```

[yunes] ./opcomma
(x:0,y:0)
(x:3,y:4)
[yunes]

```

- les opérateurs \rightarrow et $*$

```
class A;
class PointerToA {
private:
    A *a;
    PointerToA(A *a) { this->a = a; }
public:
    PointerToA() { this->a = 0; }
    A &operator*() { /*if (a==NULL) {...} else*/ return *a; }
    A *operator->() { return a; }
    A *operator=(A *a) { this->a = a; return a; }
    friend class A;
};
```

```
class A {
private:
    int x, y;
public:
    A(int x,int y) { this->x = x; this->y = y; }
    int getX() { return x; }
    int getY() { return y; }
    PointerToA operator&() { return PointerToA(this); }
};
```

```

int main() {
    A a(5,6), b(8,9);
    PointerToA p(&a);
    cout << p->getX() << ' ' << p->getY() << endl;
    cout << (*p).getX() << ' ' << (*p).getY() << endl;
    p = &b;
    cout << p->getX() << ' ' << p->getY() << endl;
    cout << (*p).getX() << ' ' << (*p).getY() << endl;
    return 0;
}

```

```
[yunes] ./arrow
```

```
5 6
```

```
5 6
```

```
8 9
```

```
8 9
```

```
[yunes]
```

**DES SURCHARGES
VRAIMENT SPÉCIALES...**

- `new` **et** `delete` :
- l'idée est de contrôler finement les allocations et désallocations d'objets
- usages courants :
 - *pool* d'objets pré-alloués
 - singleton

```
class MaClasse {
private:
    static MaClasse unique; // déclaration
public:
    void *operator new(size_t taille) {
        return &unique;
    }
};
MaClasse MaClasse::unique; // définition

int main() {
    MaClasse *p1 = new MaClasse;
    MaClasse *p2 = new MaClasse;
    cout << p1 << " : " << p2 << endl;
}
```

```
[yunes] ./main
0x1000010f0 : 0x1000010f0
[yunes]
```

- il est possible d'ajouter des arguments
au `new`
- pour en obtenir des variantes...

```
class A {
public:
    void *operator new(size_t taille) {
        return ::new char[taille];
    }
    void *operator new(size_t taille, string m) {
        cout << m << endl;
        return ::new char[taille];
    }
    void *operator new[](size_t taille, string m) {
        cout << "[]" << m << endl;
        return ::new char[taille];
    }
};
```

```
int main() {  
    A *p = new A;  
    A *p2 = new ("coucou") A;  
    A *p3 = new ("coucou") A[10];  
}
```

```
[yunes] ./alloc  
coucou  
[ ]coucou  
[yunes]
```

- si besoin est on peut utiliser l'opérateur `new` global qui est disponible par `::new`
- l'opérateur `void *operator new (size_t s)` peut être redéfini :
 - en ce cas, l'allocation doit être réalisée *in fine* par le système... sinon récursion sans fin
 - attention aussi aux appels de constructeurs
 - une instruction `new` : un appel à `new + ctor!!!`

LES PIÈGES DE LA SURCHARGE

- Les pièges de la surcharge proviennent de l'usage habituellement fait des opérateurs et qui sous-entendent un comportement attendu :
 - Propriétés attendues : symétrie, commutativité, etc.
 - Usage dans des expressions multi-opérateurs
 - Le sens donné à l'opérateur doit être *naturel*
- Donc, il ne faut surcharger que si la sémantique existe dans le domaine du problème

- Symétrie/Commutativité : Soient A a et B b
- Si $a+b$ est de type R , il serait bien surprenant que $b+a$ n'existe pas et ne soit pas aussi de type R ...
- ex : Nombre n ; $3+n$; $n+3$;
- contre-ex : produit de matrices...

```
Nombre operator+(Nombre n1,int i) {  
    return Nombre(n1.getValeur()+i);  
}  
Nombre operator+(int i,Nombre n1) {  
    return n1+i; // appel de l'autre opérateur...  
}
```

```
bool operator<(Nombre n1,Nombre n2) {
    return n1.getValeur()<n2.getValeur();
}
bool operator==(Nombre n1,Nombre n2) {
    return n1.getValeur()==n2.getValeur();
}
// La définition des autres opérateurs se fait à l'aide des précédents
bool operator<=(Nombre n1,Nombre n2) {
    return (n1<n2) || (n1==n2);
}
bool operator>(Nombre n1,Nombre n2) {
    return ! (n1<=n2);
}
bool operator>=(Nombre n1,Nombre n2) {
    return ! (n1<n2);
}
bool operator!=(Nombre n1,Nombre n2) {
    return ! (n1==n2);
}
```

- **Multi-opérateurs :**
 - L'exemple a déjà été donné pour les opérateurs d'entrées/sorties << et >>
 - Pour les opérateurs pour lesquels la sémantique est la construction d'une nouvelle valeur, il faut renvoyer une valeur du type, sinon renvoyer une référence éventuellement constante
 - Attention aux priorités et aux associativités...

- L'opérateur d'affectation...
- Sa définition peut être **absolument** nécessaire

C'est la tripléte infernale :

- Constructeur par copie
- Destructeur
- Opérateur d'affectation

- L'opérateur d'affectation... définition naïve...

```
class Tableau {
private:
    int *elements;
    int nElements;
public:
    Tableau(int taille) {
        elements = new int[nElements=taille];
    }
    ~MonTableau() { delete elements; }
    void operator=(const Tableau &t) {
        delete elements;
        elements = new int[nElements=t.nElements];
        for (int i=0; i<nElements; i++)
            elements[i] = t.elements[i];
    }
};
```

- Problèmes : optimisation (pas trop grave !?), auto-affectation (problématique), associativité (problématique)

- Auto-affectation... Il n'est pas interdit d'écrire `t=t`. Moi ? Jamais! Ah ?

```
void f(Tableau &t1, Tableau &t2) {  
    ...  
    t1 = t2;  
    ...  
}  
  
...  
    f(t,t); // ???  
...  

```

```
class Tableau {  
    ...  
    void operator=(const Tableau &t) {  
        if (*this==t) return; // Auto-affectation  
        delete elements;  
        elements = new int[nElements=t.nElements];  
        for (int i=0; i<nElements; i++)  
            elements[i] = t.elements[i];  
    }  
};
```

- Associativité... Il n'est pas interdit d'écrire $t1=t2=t3$ qui se lit $t1.operator=(t2.operator=(t3))$, ainsi
- le type de retour doit être celui de l'objet appelant
- le retour peut se faire par référence constante pour éviter une copie...

```
class Tableau {  
    ...  
    const Tableau &operator=(const Tableau &t) {  
        if (*this==t) return *this; // Auto-affectation  
        delete elements;  
        elements = new int[nElements=t.nElements];  
        for (int i=0; i<nElements; i++)  
            elements[i] = t.elements[i];  
        return *this; // On renvoie l'objet lui-même  
    }  
};
```

- Optimisation... Éviter une allocation dynamique lorsque le tableau est déjà de la bonne dimension...

```
class Tableau {
    ...
    const Tableau &operator=(const Tableau &t) {
        if (*this==t) return; // Auto-affectation
        if (nElements!=t.nElements) { // Contrôle de l'allocation
            delete elements;
            elements = new int[nElements=t.nElements];
        }
        for (int i=0; i<nElements; i++)
            elements[i] = t.elements[i];
        return *this; // On renvoie l'objet lui-même
    }
};
```

- Sémantique *naturelle* :
 - ex : addition de nombres, incrémentation d'un itérateur ou d'un compteur, etc.
 - ex : si o_{1+o2} est défini et $o_{1=o2}$ aussi alors l'utilisateur s'attendra à ce que $o_1 += o_2$ le soit et fonctionne bien (sur les valeurs produites) comme $o_1 = o_{1+o2}...$
 - contre-ex : additionner une montre et un cheval pour fabriquer une liste ??? Pour un informaticien peut-être mais sinon c'est louche...

- Puisque les constructeurs peuvent être employés pour créer automatiquement des objets (dans le cadre de conversions), il peut être utile d'empêcher cela :
- en obligeant l'utilisateur à appeler lui-même explicitement le constructeur qui doit être qualifié par :

```
class MaClasse {
    public:
        explicit MaClasse(int v) { ... }
};
int main() {
    MaClasse c(4);
    c = 45; // interdit!
    c = MaClasse(45); // OK
}
```

DÉFINITION CANONIQUE D'UNE CLASSE

- **Rule of Three**

- copy-ctor, copy-assignment-ctor, dtor

- Canonical class form

- RoT, dft-ctor, output-op

- Nous avons maintenant tout en main pour définir la forme canonique d'une classe :

pour les tableaux

pour les copies

pour les destructions polymorphes

```
class ClasseCanon {
public:
    ClasseCanon();
    ClasseCanon(const ClasseCanon &);
    virtual ~ClasseCanon();
    ClasseCanon &operator=(const ClasseCanon &);
    friend ostream
        &operator<<(ostream &, const ClasseCanon &);
};
```

pour les entrées/sorties

pour les affectations

- Attention, c++11 définit Rule of Five
 - RoT + move semantic (késako?)