

LANGUAGE OBJ. AV.(C++) MASTER 1

Yan Jurski

U.F.R. d'Informatique
Université de Paris Cité

LA SURCHARGE D'OPÉRATEURS

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

Les opérateurs surchargeables sont :

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

Quasiment tous sauf :

`::` (portée)

`.` (selection)

`?:` (expression conditionnelle)

`:` (initialisation, boucles foreach)

et quelques rares autres

Les opérateurs surchargeables sont :

+	-	*	/	%
^	&		~	!
=	<	déjà rencontré (pour l'affichage)		-=
*=	/=	%=	<=	&=
++	<<	>>	>>=	<<=
:	!=	<=	>=	&&
déjà vu aussi (une version)		--	->*	,
[]	[]	()	new	new[]
delete	delete[]	(type)		

Peut-on redéfinir ses propres opérateurs (autres que les précédents) ?

NON !

exemple pas d'union : $a \cup b$

On se contente de ceux là

Il y a deux techniques à connaître.

On peut redéfinir les opérateurs :

- comme fonction membre
- comme fonction extérieure

La plupart du temps les deux sont possibles.

Technique avec fonction extérieure

```
class A {  
public :  
    int x;  
    A(int );  
};  
ostream & operator<<(ostream &, const A&);
```

```
ostream & operator<<(ostream &o, const A& a) {  
    o<< a.x << endl;  
    return o;  
}
```

cout << a ;

s'interprète comme : operator<<(cout, a) ;

Technique avec fonction membre

```
class A {  
public :  
    int x;  
    A(int );  
    A & operator=(const A &);  
};
```

```
A & A::operator=(const A &a) {  
    this->x = a.x;  
    return *this;  
}
```

a=b ;

s'interprète comme :

a.operator=(b) ;

Les redéfinitions sont justifiables en fonction de l'usage que vous souhaitez en faire.

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

redéfinir cette multiplication se justifie assez bien.

C'est une opération assez complexe (en $O(n^3)$) et naturelle.

Mais

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

se justifie peut être un peu moins ...

par exemple parce que l'associativité pose la question de la nature des objets intermédiaires

(existe t'il qq chose entre ingrédient simple et gateau ?)

Les redéfinitions sont justifiables en fonction de l'usage que vous souhaitez en faire. `Matrix M, M1, M2; M = M1*M2;`

redéfinir cette multiplication se justifie assez bien.

C'est une opération assez complexe (en $O(n^3)$) et naturelle.

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

se justifie peut être un peu moins ...

par exemple parce que l'associativité pose la question de la nature des objets intermédiaires

(existe t'il qq chose entre ingrédient simple et gateau ?)

Quoi que ... pour le fun on pourrait imaginer :

`pieceJeu = briqueBase + déplacement (droite) + rotation(horaire)`

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

Remarque sur la surcharge :

cas de la construction implicite

```
int main() {  
    "moi"+"toi";  
    string m{"moi"};  
    string t{"toi"};  
    m+t;  
}
```

ne compile pas :
invalid operands of types
'const char [4]' and 'const char [4]'
to binary 'operator+'

compile ! C'est un cas où
l'opérateur + a été redéfini
pour l'objet string

Remarque sur la surcharge :

cas de la construction implicite

```
int main() {  
    "moi"+"toi";  
    string m{"moi"};  
    string t{"toi"};  
    m+t;  
    "toi"+m;  
    m+"toi";  
}
```

ne compile pas :
invalid operands of types
'const char [4]' and 'const char [4]'
to binary 'operator+'

compile ! C'est un cas où
l'opérateur + a été redéfini
pour l'objet string

compilent tous les deux !
mais pour une autre raison

aucune surcharge de + ne correspond strictement aux deux dernières signatures, mais un constructeur permet de passer de "toi" à string. Lorsque c'est simple, et non ambigu il est invoqué...

Remarque sur la surcharge :

cas de la construction implicite

```
int main() {  
    "moi"+"toi";  
    string m{"moi"};  
    string t{"toi"};  
    m+t;  
    string{"toi"}+m;  
    m+string{"toi"};  
}
```

dans ce cas le compilateur a appelé (implicitement) le constructeur
de signature : `string (const char [])`

Remarque sur la surcharge :

cas de la construction implicite

```
int main() {  
    "moi"+"toi";  
    string m{"moi"};  
    string t{"toi"};  
    m+t;  
    string{"toi"}+m;  
    m+string{"toi"};  
}
```

dans ce cas le compilateur a appelé (implicitement) le constructeur de signature : `string (const char [])`

Pour `"moi"+"toi"`, déjà il faudrait 2 constructions ...

« trop compliqué ?! » ... apparemment, mais on en reparle bientôt

Remarque sur la surcharge :

cas de la construction implicite

Cette transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe.

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
void f(A a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  

```

moi

Remarque sur la surcharge :

cas de la construction implicite

Cette transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe.

```
int main() {  
    f("moi");  
}
```

```
void f(A a) {  
    cout << a.nom << endl;  
}
```

```
class A{  
    public :  
    string nom;  
    A(string );  
};
```

```
A::A(string x):nom{x}{}  
A::A():nom{""}{}  
A::A(char*){}
```

```
error: could not convert  
from 'const char*' to 'A'
```

Elle est limitée en profondeur (c.a.d malgré que `string{char []}` existe)

Remarque sur la surcharge :

cas de la construction implicite

Cette transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe.

```
int main() {  
    string m{moi};  
    f(m,m);  
}
```

```
void f(A a,A b) {  
    cout << a.nom << b.nom << endl;  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
A::A(string x):nom{x}{}  

```

moi moi

Mais pas limitée en "largeur" : chaque expression est testée pour un type fixé, et convertie si c'est possible

Remarque sur la surcharge :

cas de la construction implicite

Cette transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe.

```
int main() {  
    "moi"+"toi";  
}
```

mais alors, "moi" et "toi" pourraient donc être convertis chacun ... qu'est ce qui est « trop compliqué » ?

```
error: invalid operands of types 'const char [4]' and  
'const char [4]' to binary 'operator+'
```

Le problème est que + est déjà beaucoup surchargé.
La résolution pourrait ici prendre plusieurs directions.

Remarque sur la surcharge :

cas de la construction implicite

Exemple :

```
int main() {  
    cout << "moi" + 1;  
}
```

??

Difficile d'avoir une certitude à priori sur le résultat ...
... mais ça compile

Remarque sur la surcharge :

cas de la construction implicite

Exemple :

```
int main() {  
    cout << "moi" + 1;  
}
```

oi

Difficile d'avoir une certitude à priori sur le résultat ...

...mais ça compile ...

avec ici une opération sur l'adresse du const char[]

Dans "moi"+"toi"; il faudrait arbitrer :

un char[] est transformable en string ou en entier d'adresse.

Comment doit-on voir « toi » ?

impossible de choisir d'où l'erreur...

Remarque sur la surcharge :

cas de la construction implicite

La transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe. On peut aussi l'interdire (la neutraliser). On avait :

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
void f(A a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  

```

moi

Remarque sur la surcharge :

cas de la construction implicite

La transformation est invoquée au moment de la résolution de type, arbitrant entre surcharges quand un constructeur adéquat existe. On peut aussi l'interdire (la neutraliser). Et ici :

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
    string nom;  
    explicit A(string );  
};
```

```
void f(A a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  
A::A():nom{}
```

```
error: could not convert  
from string to 'A'
```

Remarque sur la surcharge :

cas de la construction implicite

Cas des références ?

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
void f(A &a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  

```

??

Remarque sur la surcharge :

cas de la construction implicite

Cas des références ?

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
void f(A &a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  
}
```

error: cannot bind
reference of type 'A&' to
an rvalue of type 'A'

Naturellement l'objet construit est anonyme, cet alias est refusé

Remarque sur la surcharge :

cas de la construction implicite

Cas des références ?

```
int main() {  
    string m{"moi"};  
    f(m);  
}
```

```
class A{  
    public :  
        string nom;  
        A(string );  
};
```

```
void f(const A &a) {  
    cout << a.nom << endl;  
}
```

```
A::A(string x):nom{x}{}  

```

moi

il faut bien sûr un const pour l'alias vers cette r-value, mais c'est ok

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

Etude de cas

nous allons réfléchir un peu :

faut il être plutôt en faveur d'une redéfinition avec la technique d'une fonction membre ?

Ou plutôt avec celle d'une fonction externe ?

Et déjà, l'idée de la redéfinition d'un opérateur est-elle bien fondée ?

Etude de cas

Illustration avec l'addition sur des objets « carnet d'adresse »

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse& operator+(string x);  
    CarnetAdresse& operator+(CarnetAdresse & x);  
};
```

au sens de
l'ajout

au sens de
l'union

```
ostream& operator<<(ostream& o, const CarnetAdresse& c) {  
    for (string x:c.known) o << x << " ";  
    o<< endl;  
    return o;  
}
```

pour voir qq chose

```
CarnetAdresse& CarnetAdresse::operator+(string x) {  
    known.push_back(x);  
    return *this;  
}
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    cout << c;  
}
```

moi toi

notez que l'associativité à une importance puisque "moi"+"toi" ne compilait pas

L'associativité ainsi que les priorités entre opérateurs sont fixées. Impossible de toutes les connaître, il faut se référer à la doc.


```
CarnetAdresse& CarnetAdresse::operator+(string x) {  
    known.push_back(x);  
    return *this;  
}
```

```
int main() {  
    CarnetAdresse c;  
    (c+"moi")+"toi";  
    cout << c;  
}
```

moi toi

interprété implicitement ainsi,
car + est associatif à gauche

L'associativité ainsi que les priorités entre opérateurs sont fixées. Impossible de toutes les connaître, il faut se référer à la doc.

```
CarnetAdresse& CarnetAdresse::operator+(string x) {  
    known.push_back(x);  
    return *this;  
}  
CarnetAdresse& CarnetAdresse::operator+(CarnetAdresse& c) {  
    for(string x:c.known) (*this)+x;  
    return *this;  
}
```

pas mal.
ici l'addition précédente !
mais ...
n'est pas assez prudent.
Il y a une situation délicate

```
CarnetAdresse& CarnetAdresse::operator+(string x) {  
    known.push_back(x);  
    return *this;  
}  
CarnetAdresse& CarnetAdresse::operator+(CarnetAdresse& c) {  
    for(string x:c.known) (*this)+x;  
    return *this;  
}
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    cout << c;  
    c+c;  
    cout << c;  
}
```

```
moi toi  
moi toi moi
```

il faut penser à tout
si on fourni un service.
L'itérateur est perturbé

```
CarnetAdresse& CarnetAdresse::operator+(string x) {  
    known.push_back(x);  
    return *this;  
}  
  
CarnetAdresse& CarnetAdresse::operator+(CarnetAdresse & c) {  
    vector<string> copie{c.known};  
    for(string x:copie) (*this)+x;  
    return *this;  
}
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    cout << c;  
    c+c;  
    cout << c;  
}
```

```
moi toi  
moi toi moi toi
```

il faut penser à tout
si on fourni un service.
L'itérateur est perturbé

```

CarnetAdresse& CarnetAdresse::operator+(string x) {
    known.push_back(x);
    return *this;
}
CarnetAdresse& CarnetAdresse::operator+(CarnetAdresse & c) {
    vector<string> copie{c.known};
    for(string x:copie) (*this)+x;
    return *this;
}

```

```

int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    cout << c;
    c+c;
    cout << c;
}

```

```

moi toi
moi toi moi toi

```

il faut penser à tout
si on fourni un service.
L'itérateur est perturbé

Rq: (parce que vous demandez souvent à quoi sert const ...)

si l'argument avait été déclaré const on aurait eu un message d'erreur à l'écriture de c+c.

Etude de cas (suite)

Dans ce cadre :

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse& operator+(string x);  
    CarnetAdresse& operator+(CarnetAdresse & x);  
};
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    c+c;  
    "eux"+c;  
}
```

une opération
reste impossible

error: no match for 'operator+'
(operand types are
'const char [4]' and
'CarnetAdresse')

Essayons de faire accepter "eux"+c
en ne considérant que les additions en fonctions membres.
On pense ici à faire transformer "eux" en Carnet implicite

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse(string);  
    CarnetAdresse &operator+(CarnetAdresse & x);  
};
```

```
CarnetAdresse::CarnetAdresse( string x) : known{x} {}
```

```
int main() {  
    CarnetAdresse c;  
    "eux"+c;  
}
```

cet conversion est insuffisante
(la distance entre char [] et
CarnetAdresse est 2)

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse( const char x[] );
    CarnetAdresse(string);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
```

```
CarnetAdresse::CarnetAdresse( string x) : known{x} {}
CarnetAdresse::CarnetAdresse( const char x[])
    : CarnetAdresse{string{x}} {}
```

```
int main() {
    CarnetAdresse c;
    c="test";
    "eux"+c;
}
```

La conversion marche, comme en témoigne `c="test"` qui est `c.operator=("test")` et va correspondre à l'affectation par défaut entre carnets.

Mais cela « n'aboutit pas » pour `"eux"+c`


```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse( const char x[] );
    CarnetAdresse(string);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
```

```
CarnetAdresse::CarnetAdresse( string x) : known{x} {}
CarnetAdresse::CarnetAdresse( const char x[])
    : CarnetAdresse{string{x}} {}
```

```
int main() {
    CarnetAdresse c;
    "eux"+c;
}
```

On pourrait émettre l'hypothèse que la conversion, si réalisée, serait anonyme (vers un const CarnetAdresse) et que l'opérateur+ ne peut s'appliquer car il n'est pas signé par un const à la fin ...
écartons cette idée, en testant :

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse( const char x[] );
    CarnetAdresse(string);
    CarnetAdresse &operator+(CarnetAdresse & x) const;
    // naturellement avec ce test ne ferons rien dans +
};
```

```
CarnetAdresse::CarnetAdresse( string x) : known{x} {}
CarnetAdresse::CarnetAdresse( const char x[])
    : CarnetAdresse{string{x}} {}
```

```
int main() {
    CarnetAdresse c;
    "eux"+c;
}
```

ne marche toujours pas.

L'idée d'une conversion de "eux" char [] en const CarnetAdresse, puis l'appel de operator+(...) const est donc écartée

Explication :

Avec des fonctions membres (ici on parle d'operator+), l'argument à gauche est nécessairement un objet de la classe.

Le compilateur envisage une conversion implicite pour les arguments seulement.

Si aucune conversion implicite ne correspond, la recherche de la bonne fonction se poursuit parmi les fonctions non membres.

Explication :

Avec des fonctions membres (ici on parle d'operator+), l'argument à gauche est nécessairement un objet de la classe.

Le compilateur envisage une conversion implicite pour les arguments des fonctions seulement.

Si aucune conversion implicite ne correspond, la recherche de la bonne fonction se poursuit parmi les fonctions non membres.

Ici, on n'a pas écrit d'operator+ non membre : échec

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

moi toi eux

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c,
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

cherche d'abord des correspondances
dans les « fonctions membres » de
char [] : aucune à priori

moi toi eux

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

passé ensuite à la liste des
fonctions externes

moi toi eux

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

celle ci pourrait correspondre si une conversion implicite existe entre char [] et string : oui !

moi toi eux

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse &operator+(string x);  
    CarnetAdresse &operator+(CarnetAdresse & x);  
};  
CarnetAdresse& operator+(string x, CarnetAdresse& c) {  
    c+x;  
    return c;  
}
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    "eux"+c;  
    cout << c;  
}
```

on parle à présent d'autres datas.
Clairement une méthode membre existe.

moi toi eux

Une solution : ici on va mélanger les 2 approches : fonction membre, et externe, avec conversion de "eux" en string

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse &operator+(string x);  
    CarnetAdresse &operator+(CarnetAdresse & x);  
};  
CarnetAdresse& operator+(string x, CarnetAdresse& c) {  
    c+x;  
    return c;  
}
```

celle là

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    "eux"+c;  
    cout << c;  
}
```

moi toi eux

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

Parfait ?

Il reste une anomalie conceptuelle
dans $c1+c2$: $c1$ est modifié
dans $"eux" + c$: c est modifié

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse &operator+(string x);  
    CarnetAdresse &operator+(CarnetAdresse & x);  
};  
CarnetAdresse& operator+(string x, CarnetAdresse& c) {  
    c+x;  
    return c;  
}
```

on retourne c1 à cause de
cette syntaxe

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    "eux"+c;  
    cout << c;  
}
```

Parfait ?

Il reste une anomalie conceptuelle
dans `c1+c2` : `c1` est modifié
dans `"eux" + c` : `c` est modifié

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

et on retourne c2 à cause
de cette syntaxe

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

Parfait ?

Il reste une anomalie conceptuelle
dans `c1+c2` : `c1` est modifié
dans `"eux" + c` : `c` est modifié

```
class CarnetAdresse {
public :
    vector <string> known;
    CarnetAdresse();
    CarnetAdresse &operator+(string x);
    CarnetAdresse &operator+(CarnetAdresse & x);
};
CarnetAdresse& operator+(string x, CarnetAdresse& c) {
    c+x;
    return c;
}
```

```
int main() {
    CarnetAdresse c;
    c+"moi"+"toi";
    "eux"+c;
    cout << c;
}
```

Il faut trancher : retournons une copie,
inutile de modifier les arguments

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse();  
    CarnetAdresse operator+(string x) const;  
    CarnetAdresse operator+(const CarnetAdresse & x) const ;  
};  
CarnetAdresse operator+(string x, const CarnetAdresse& c) {  
    return c+x;  
}
```

```
int main() {  
    CarnetAdresse c;  
    c+"moi"+"toi";  
    "eux"+c;  
    cout << c;  
}
```

Il faut trancher : retournons une copie,
inutile de modifier les arguments

```
CarnetAdresse CarnetAdresse::operator+(string x) const {  
    vector<string> copie {known};  
    copie.push_back(x);  
    return CarnetAdresse{copie}; // il manque ce constructeur  
}
```


On l'ajoute :

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse (vector<string> = vector<string>{});  
    ...  
};
```

```
CarnetAdresse::CarnetAdresse (vector<string> x)  
    : known{x} {}
```

```
CarnetAdresse CarnetAdresse::operator+(string x) const {  
    vector<string> copie {known};  
    copie.push_back(x);  
    return CarnetAdresse{copie};  
}
```

Le return est à présent opérationnel,
mais on peut faire une remarque ici

```
CarnetAdresse CarnetAdresse::operator+(string x) const {  
    vector<string> copie {known};  
    copie.push_back(x);  
    // return CarnetAdresse{copie};  
    return copie;  
}
```



oui, on peut aussi ici utiliser
cette notation (avec conversion)

On continue à dérouler l'exemple, reste à écrire :

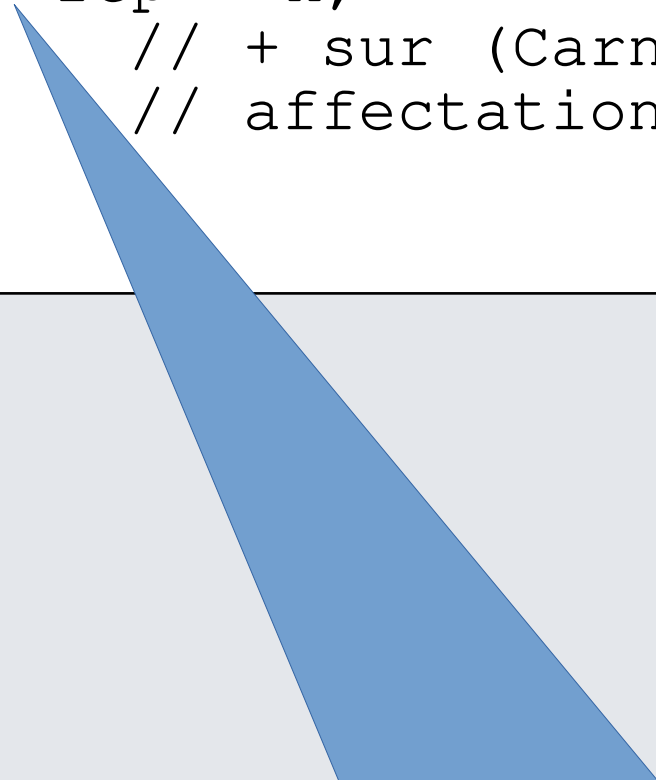
```
CarnetAdresse CarnetAdresse::operator+(const CarnetAdresse &c) const{
    CarnetAdresse rep{*this}; // copie par défaut ok ? (oui !)
    for(const string &x:copie) rep = rep + x;
                                // + sur (Carnet, string)
                                // affectation par défaut ok

    return rep;
}
```

On continue de dérouler l'exemple, reste à écrire :

```
CarnetAdresse CarnetAdresse::operator+(const CarnetAdresse &c) const{
    CarnetAdresse rep{*this}; // copie par défaut ok ? (oui !)
    for(const string &x:copie) rep = rep + x;
                                // + sur (Carnet, string)
                                // affectation par défaut ok

    return rep;
}
```



mais ... on ne peut pas
utiliser un += là ?
non, car c'est un
opérateur différent ...

Etude de cas : bilan, et vérification finale :

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse(vector<string> = vector<string>{});  
    CarnetAdresse operator+(string x) const;  
    CarnetAdresse operator+(const CarnetAdresse & x) const ;  
};
```

```
CarnetAdresse operator+(string x, const CarnetAdresse& c) {  
    return c+x;  
}
```

Etude de cas : bilan, et vérification finale :

```
class CarnetAdresse {  
public :  
    vector <string> known;  
    CarnetAdresse(vector<string> = vector<string>{});  
    CarnetAdresse operator+(string x) const;  
    CarnetAdresse operator+(const CarnetAdresse & x) const ;  
};
```

```
CarnetAdresse operator+(string x, const CarnetAdresse& c) {  
    return c+x;  
}
```

```
int main() {  
    CarnetAdresse c;  
    // il faut passer par = (ok par défaut)  
    c=c+"moi"+"toi";  
    cout << c;  
    c=c+c;  
    cout << c;  
    c="eux"+c;  
    cout << c;  
}
```

```
moi toi  
moi toi moi toi  
moi toi moi toi eux
```

Notez que :

- les fonctions membres ne peuvent être utilisées qu'avec une instance directe de la classe (sans conversion). Mais les fonctions externes peuvent être utilisées avec des types implicitement convertis car la conversion se fait sur les arguments. En ce sens surcharger les opérateurs binaires en tant que fonctions externe les rend plus flexibles.
- les opérateurs internes (membres) sont prioritaires pendant la résolution sur les externes. On peut travailler sur leur visibilité, leur virtualité.
- s'il existe 2 techniques de mises en places, il ne faut pas créer de confusions pour l'utilisateur !!

Quid de la **virtualité** des opérateurs fonctions membres ?

- pour les opérateurs binaires, souvent il y a une symétrie, voir une commutativité : $a+b$ et $b+a$ devraient être les mêmes. Or la liaison dynamique ne se ferait qu'à gauche. Sans très bonne raison, on va donc renoncer à la virtualité.
- pour les opérateurs unaires, rien ne s'y oppose à priori.

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

Les opérateurs unaires

Les opérateurs unaires préfixes : (\sim , ++, --, -, +, ...)

ont aussi 2 traductions possibles :

- en fonction membre ex : `type operator++()`
- en fonction externe ex : `type operator++(MaClasse x)`

Les opérateurs unaires

Les opérateurs unaires préfixes : (~, ++, --, -, +, ...)

ont aussi 2 traductions possibles :

- en fonction membre ex : `typeRet operator++()`
- en fonction externe ex : `typeRet operator++(MaClasse x)`

Pour les opérateurs unaires postfixes : (++, --)

on ne peut pas utiliser :

- en fonction membre ex : `typeRet operator++()`
- en fonction externe ex : `typeRet operator++(MaClasse x)`

Car ce serait ambigu : même signatures que les précédentes

Les opérateurs unaires

Les opérateurs unaires préfixes : (`~`, `++`, `--`, `-`, `+`, ...)

ont aussi 2 traductions possibles :

- en fonction membre ex : `typeRet operator++()`
- en fonction externe ex : `typeRet operator++(MaClasse x)`

Pour les opérateurs unaires postfixes : (`++`, `--`)

on ne peut pas utiliser :

- en fonction membre ex : `typeRet operator++(int)`
- en fonction externe ex : `typeRet operator++(MaClasse x, int)`

Car ce serait ambigu : même signatures que les précédentes

L'arbitrage de c++ consiste à introduire un argument fictif...

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

il permet juste de distinguer les signatures.

Petites réflexions sur les types retour de ++

L'usage habituel :

```
int x{0};  
cout << x++;  
cout << x;
```

```
0  
1
```

```
int x{0};  
cout << ++x;  
cout << x;
```

```
1  
1
```

Petites réflexions sur les types retour de ++

Avec un objet ...

```
class A{  
    public :  
        int val;  
}
```

```
A x{0};  
cout << x++;  
cout << x;
```

```
A x{0};  
cout << ++x;  
cout << x;
```

```
0  
1
```

```
1  
1
```

Comment cela se traduit-il dans les signatures des opérateurs ?

Petites réflexions sur les types retour de ++

Avec un objet ...

```
A x{0};  
cout << x++;  
cout << x;
```

0
1

```
A x{0};  
cout << ++x;  
cout << x;
```

1
1

```
class A{  
public :  
    int val;  
    ??? operator++(int ); // le postfixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
??? A::operator++(int ) {  
    val++;  
    return ???  
}
```


Petites réflexions sur les types retour de ++

Avec un objet ...

```
A x{0};  
cout << x++;  
cout << x;
```

0
1

```
A x{0};  
cout << ++x;  
cout << x;
```

1
1

```
class A{  
public :  
    int val;  
    ??? operator++(int ); // postfixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
??? A::operator++(int ) {  
    val++;  
    return ???  
}
```

c'est l'ancienne valeur,
pas l'actuelle.
C'est donc une copie.

Petites réflexions sur les types retour de ++

Avec un objet ...

```
A x{0};  
cout << x++;  
cout << x;
```

0
1

```
A x{0};  
cout << ++x;  
cout << x;
```

1
1

```
class A{  
    public :  
        int val;  
        A operator++(int ); // le postfixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
A A::operator++(int ) {  
    A old(*this);  
    val++;  
    return old;  
}
```

Petites réflexions sur les types retour de ++

Avec un objet ...

```
A x{0};  
cout << x++;  
cout << x;
```

```
0  
1
```

```
A x{0};  
cout << ++x;  
cout << x;
```

```
1  
1
```

```
class A{  
public :  
    int val;  
    ??? operator++(); // le prefixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
??? A::operator++() { // A ou A& ?  
    val++;  
    return *this;  
}
```

Petites réflexions sur les types retour de ++

Avec un objet ...

```
A x{0};  
cout << x++;  
cout << x;
```

0
1

```
A x{0};  
cout << ++x;  
cout << x;
```

1
1

```
class A{  
public :  
    int val;  
    A& operator++(); // le prefixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
A& A::operator++() { // plutôt A&, on peut se passer de copie  
    val++;  
    return *this;  
}
```

Info : post++ est prioritaire sur ++préf. Résultat ici ?

```
A x{0};  
cout << x++;  
cout << ++x;  
cout << ++x++;  
cout << x;
```

??

```
class A{  
    public :  
        int val;  
        A& operator++(); // le prefixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
A& A::operator++() {  
    val++;  
    return *this;  
}
```

Info : post++ est prioritaire sur ++préf. Résultat ici ?

```
A x{0};  
cout << x++;  
cout << ++x;  
cout << (x.operator++(0)).operator++();  
cout << x;
```

??

```
class A{  
    public :  
        int val;  
        A& operator++(); // le prefixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
A& A::operator++() {  
    val++;  
    return *this;  
}
```

Info : post++ est prioritaire sur ++préf. Résultat ici ?

```
A x{0};  
cout << x++;  
cout << ++x;  
cout << ++x++;  
cout << x;
```

0
2
3
3

```
class A{  
    public :  
        int val;  
        A& operator++(); // le prefixe  
}  
ostream & operator<<(ostream& o, const A&x) {  
    o<< x.val << endl;  
    return o;  
}
```

```
A& A::operator++() {  
    val++;  
    return *this;  
}
```

Version fonction externe :

```
A x{0};  
cout << x++;  
cout << ++x;  
cout << ++x++;  
cout << x;
```

??

```
class A{  
    public :  
        int val;  
}  
A operator++(A &x, int) {  
    A old{x};  
    x.val++;  
    return old;  
}  
A& operator++(A &x) {  
    x.val++;  
    return x;  
}
```


Version fonction externe :

```
A x{0};  
cout << x++;  
cout << ++x;  
//cout << ++x++;  
cout << x;
```

```
0  
2  
// echec  
2
```

```
class A{  
public :  
    int val;  
}  
A operator++(A &x, int) {  
    A old{x};  
    x.val++;  
    return old;  
}  
A& operator++(A &x) {  
    x.val++;  
    return x;  
}
```

priorité de post++,
c'est la première copie
(anonyme)

ne peut pas s'aliasser
avec l'anonyme

Dans la version membre, les objets invoquent des méthodes

```
cout << (x.operator++(0)).operator++();
```

Dans la version fonction externe, les objets, en argument, passent par un aliasing

```
cout << operator++(operator++(x, 0));
```

L'usage habituel :

```
int x{0};  
cout << ++x++;  
cout << x;
```

error: lvalue required
as increment operand

C'est la preuve que c++ surcharge ++ en fonction externe pour int
(après tout c'est un type de base, il n'a pas de membre)

Exercice :

```
class A{
public :
    int val;
    A(int x) : val{x}{}

    A operator++(int );           // le postfixe
    A& operator++();              // le prefixe
    A operator+(const A &a) ;     // le binaire
}

A A::operator++(int ){ A old(*this); val++; return old; }
A& A::operator++(){ val++; return *this; }
A operator+(const A &a){ return A{val + a.val}; }
```

```
int main() {
    A a{10};
    A b{20};
    cout << ++a+++b++ ;
    cout << a ;
    cout << b ;
}
```

??

Sachant que : (++post) prioritaire sur (++pref) prioritaire sur (+binaire) ...

Exercice :

```
class A{
public :
    int val;
    A(int x) : val{x}{}

    A operator++(int );           // le postfixe
    A& operator++();             // le prefixe
    A operator+(const A &a) ;    // le binaire
}

A A::operator++(int ){ A old(*this); val++; return old; }
A& A::operator++(){ val++; return *this; }
A operator+(const A &a){ return A{val + a.val}; }
```

```
int main() {
    A a{10};
    A b{20};
    cout << ++a+++b++ ;
    cout << a ;
    cout << b ;
}
```



```
31
11
21
```

Sachant que : (++post) prioritaire sur (++pref) prioritaire sur (+binaire) ...

Vérifiez chez vous : (++(a++))+ (b++)

Exercice : (pire ?)

```
class A{
public :
    int val;
    A(int x) : val{x}{}
    A operator+(const A &a=A{0}) ;    // avec valeur par défaut
}
A operator+(const A &a){ return A{val + a.val}; }
```

```
int main() {
    A a{10};
    a+ ; // ???
    cout << a ;
}
```

?

Exercice : (pire ?)

```
class A{
public :
    int val;
    A(int x) : val{x}{}
    A operator+(const A &a=A{0}) ;    // avec valeur par défaut
}
A operator+(const A &a){ return A{val + a.val}; }
```

```
int main() {
    A a{10};
    a+ ; // ???
    cout << a ;
}
```

```
error:
`A A::operator+(const A&) '
cannot have default arguments
```

(Heureusement) presque tous les opérateurs ne peuvent pas être définis avec un argument par défaut.

Si cela vous amuse, vous pouvez combiner pour la classe A, un ++ membre, et un ++ externe pour permettre ++x++.
(Le nombre de combinaisons est limité)

La question intéressante derrière cette expérience serait de savoir si on peut ainsi changer la priorité entre post++ et pre++ : ils ne sont plus au même niveau syntaxique, l'un étant membre et l'autre externe, cette précedence l'emporte t'elle sur la priorité des opérateurs ?

(spoil : non c.à.d que la syntaxe est résolue avant la surcharge)

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

L'opérateur `[]`

- n'a qu'un seul argument
- ne peut pas avoir d'argument par défaut
- habituellement `int` (mais pas nécessairement)
- est obligatoirement fonction membre
- souvent se combine

```
x[2]++;  
x[2] = qq chose
```

L'opérateur []

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (int) ;  
};
```

pour proposer un exemple original, disons que `a[n]` élève la valeur de `a` à la puissance `n`.

```
A A::operator[] (int x) {  
    return A{pow(val,x)};  
}
```

```
int main() {  
    A a{3};  
    cout << a[2];  
    cout << a;  
}
```



9
3

L'opérateur []

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (int) ;  
};
```

```
A A::operator[] (int x) {  
    return pow(val,x);  
}
```

Rq : on peut aussi utiliser
une conversion implicite

```
int main() {  
    A a{3};  
    cout << a[2];  
    cout << a;  
}
```

9
3

L'opérateur []

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (int) ;  
};
```

```
A A::operator[] (int x) {  
    return pow(val,x);  
}
```

```
int main() {  
    A a{3};  
    cout << a[2][2];  
    cout << a;  
}
```

81
3

L'opérateur []

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (int) ;  
};
```

Version 2 (l'objet est modifié)

```
A A::operator[] (int x) {  
    val=pow(val,x);  
    return *this; // ou val ..  
}
```

```
int main() {  
    A a{3};  
    cout << a[2]  
    cout << a[2];  
    cout << a;  
}
```



9
81
81

L'opérateur []

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (int) ;  
};
```

Version 3 (l'objet est modifié, et retour par référence)

```
A& A::operator[] (int x) {  
    val=pow(val,x);  
    return *this; // pas val !  
}
```

possible :
-conversion de 5 vers A
-affectation entre A(défaut)
peu d'intérêt conceptuel ici

```
int main() {  
    A x{3};  
    cout << x[2];  
    x[2]=5; // why ?  
    cout << x;  
}
```

9
5

L'opérateur []

Mais parfois on veut naturellement une référence, c'est ce qui est fait avec vector :

```
int main() {  
    vector<int> v{1,1,1};  
    v[1]++;  
    cout << v[1] << endl;  
}
```



2

L'opérateur []

on peut surcharger avec des types énumérés par ex :

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (Operation) ;  
};
```

```
enum Operation {  
    Nullify,  
    Pow2,  
    Pow3  
};
```

```
A A::operator[] (Operation x) {  
    switch(x) {  
        case Pow2 : return val=pow(val,2);  
        case Pow3 : return val=pow(val,3);  
        default   : return val=0;  
    }  
}
```

```
int main() {  
    A x{3};  
    cout << x[Pow2][Pow3];  
    cout << x[Nullify];  
}
```

729
0

L'opérateur []

on peut surcharger avec des types énumérés par ex :

```
class A {  
    public :  
        int val;  
        A(int=0);  
        A operator[] (Operation) ;  
};
```

```
enum Operation {  
    Nullify,  
    Pow2,  
    Pow3  
};
```

```
A A::operator[] (Operation x) {  
    switch(x) {  
        case Pow2 : return val=pow(val,2);  
        case Pow3 : return val=pow(val,3);  
        default   : return val=0;  
    }  
}
```

rq, syntaxe concise :
- type retour de =
- conversion

```
int main() {  
    A x{3};  
    cout << x[Pow2][Pow3];  
    cout << x[Nullify];  
}
```

729
0

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

L'opérateur ()

- peut avoir plusieurs arguments
- permet de "considérer" l'objet comme une fonction

```
a(2);  
a(2, 4, "truc");
```

- est obligatoirement fonction membre
- accepte les valeurs par défaut

```
class Convertisseur {
private:
    double taux;
public:
    Convertisseur(double tx): taux{tx} {};
    double operator()(double=1) const;
    double operator()(double montant, double nouvTaux);
};
```

```
double Convertisseur::operator()(double m) const {
    return m*taux;
}
double Convertisseur::operator()(double m, double tx) {
    taux=tx;
    return (*this) (m);
}
```

```
void main() {
    Convertisseur euroDollar{1.03};
    cout << euroDollar();
    cout << euroDollar(50);
    cout << euroDollar(50,1);
}
```

```
1.03
51.5
50
```

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

L'opérateur de casting

- notation à la C : `x= (int) y;`
- notation à la C++ : `x= int(y);`

Il s'implémente comme fonction membre de la classe de y :

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

L'opérateur de casting

- notation à la C : `x= (int) y;`
- notation à la C++ : `x= int(y);`

Il s'implémente comme fonction membre de la classe de `y` :

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

```
class A {  
    public :  
        int val;  
        A(int=0);  
        operator int() const;  
};
```

```
A::operator int() const {return val;}
```


L'opérateur de casting

- notation à la C : `x= (int) y;`
- notation à la C++ : `x= int(y);`

Il s'implémente comme fonction membre de la classe de y :
`operator int() const { ... }`

```
class A {  
    public :  
        int val;  
        A(int x):val{x}{}  
        operator int() const;  
};
```

```
A::operator int() const {return val;}
```

```
int main() {  
    A a{3};  
    int x;  
    x=(int)a;  
    x=int(a);  
    x=int{a};  
    x=a;  
    int y{a};  
    cout << x;  
}
```

L'opérateur de casting

généralisable à nos propres types

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const;  
};
```

```
A::operator B() const {  
    string rep="";  
    for (int i=0; i<val; i++) rep+="x";  
    return B{rep};  
}
```

```
int main() {  
    A a{3};  
    cout << (B) a;  
    cout << B(a);  
}
```

???

L'opérateur de casting

généralisable à nos propres types

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
};
```

```
int main() {  
    A a{3};  
    cout << (B) a;  
    cout << B(a);  
}
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const;  
};
```

```
xxx  
xxx
```

```
A::operator B() const {  
    string rep="";  
    for (int i=0; i<val; i++) rep+="x";  
    return B{rep};  
}
```

L'opérateur de casting

généralisable à nos propres types

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
};
```

```
int main() {  
    A a{3};  
    cout << B{a};  
}
```

perturbant,
ce n'est pas un
constructeur

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const;  
};
```

xxx

```
A::operator B() const {  
    string rep="";  
    for (int i=0; i<val; i++) rep+="x";  
    return B{rep};  
}
```

L'opérateur de casting

généralisable à nos propres types

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const;  
};
```

```
A::operator B() const {  
    string rep="";  
    for (int i=0; i<val; i++) rep+="x";  
    return B{rep};  
}
```

```
int main() {  
    A a{3};  
    cout << (B)a;  
}
```

perturbant,
pas de lien
d'héritage
entre A et B

xxx

L'opérateur de casting

rappel : construction implicite vue au début de cette séance

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
        B(const A &a) : cont{"x"} {}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
};
```

```
void f(const B &b) {  
    cout << b;  
}
```

```
int main() {  
    string s="abc";  
    f(s);  
    A a;  
    f(a);  
    f((B)a);  
    f(B(a));  
    f(B{a});  
    B b=a;  
    f(b);  
    f(static_cast<B>(a));  
}
```

nous avons vu que toutes les instructions du main font appel implicitement à l'un des constructeurs

L'opérateur de casting

a exactement le même comportement :

```
class B {  
    public :  
        string cont;  
        B(string x) : cont{x} {}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const {return "y";}  
};
```

```
void f(const B &b) {  
    cout << b;  
}
```

```
int main() {  
    string s="abc";  
    f(s);  
    A a;  
    f(a);  
    f((B)a);  
    f(B(a));  
    f(B{a});  
    B b=a;  
    f(b);  
    f(static_cast<B>(a));  
}
```

marche de la même façon

L'opérateur de casting

VS

la construction implicite ?

```
class A;  
class B {  
    public : string cont;  
    B(string s) : cont{s} {};  
    B(const A &a):cont{"x"}{}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x):val{x}{}  
        operator B() const {return "y";}  
};
```

```
void f(const B &b) {  
    cout << b;  
}
```

```
int main() {  
    A a;  
    f(a);  
    f((B)a);  
    f(B(a));  
    f(B{a});  
    B b=a;  
    f(b);  
    f(static_cast<B>(a));  
}
```

mais alors, si les deux co-existent ?

L'opérateur de casting

VS

la construction implicite ?

```
class A;  
class B {  
    public : string cont;  
    B(string s) : cont{s} {};  
    B(const A &a):cont{"x"}{}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x):val{x}{}  
        operator B() const {return "y";}  
};
```

```
void f(const B &b) {  
    cout << b;  
}
```

```
int main() {  
    A a;  
    f(a);  
    f((B)a);  
    f(B(a));  
    f(B{a});  
    B b=a;  
    f(b);  
    f(static_cast<B>(a));  
}
```



x
x
x..

c++ arbitre en faveur de la construction

L'opérateur de casting

VS

la construction implicite ?

c++ arbitre en faveur de la construction

L'argument est que la personne qui a écrit B et ses constructeurs est considéré comme le « propriétaire intellectuel » de B.

Alors que celui qui a écrit, dans A, le casting de A vers B est un utilisateur.

Dans le cas de ce conflit l'auteur de B est plus fiable

L'opérateur de casting

VS

la construction implicite ?

de la même façon que pour la construction implicite,
on peut neutraliser avec « explicit » les cast implicites

L'opérateur de casting

VS

la construction implicite ?

de la même façon que pour la construction implicite, on peut neutraliser avec « explicit » les cast implicites

Pour conclure avec notre exemple d'étude, puisque la construction est prioritaire, imposons qu'elle soit explicite, cela fera apparaître les cas restants, qui seront des conversions.

L'opérateur de casting

VS

la construction implicite ?

```
class A;  
class B {  
    public : string cont;  
    B(string s) : cont{s} {};  
    explicit B(const A &a) : cont{"ctr"}  
{}  
};
```

```
class A {  
    public :  
        int val;  
        A(int x) : val{x} {}  
        operator B() const {return "cast";}  
};
```

```
void f(const B &b) {  
    cout << b;  
}
```

```
int main() {  
    A a;  
    f(a);  
    f((B)a);  
    f(B(a));  
    f(B{a});  
    B b=a;  
    f(b);  
    f(static_cast<B>(a));  
}
```

```
cast  
ctr  
ctr  
ctr  
cast  
ctr
```

on peut donc les distinguer ... mais c'est plus dangereux qu'autres chose ...

Les opérateurs `->` `*` `&` , `new` `delete`

On ne peut pas voir ensemble toutes les redéfinitions, mais on peut donner quelques justifications :

- avec `new` et `delete`, dans un "garbage collector" maison, on pourrait écouter toutes les créations de pointeurs ; retirer les objets `deleted` explicitement par le programmeur, pour ne se concentrer que sur ses "oublis".
- surcharger `->` `*` et l'adressage `&` permet de définir des "smart pointers" (des objets qui maîtrisent qui les connaît)

Plan :

- Présentation / justification
- Résolution de surcharge : cas de la construction implicite
- Redéfinition membre ou externe ? Une étude de cas
- Les opérateurs unaires (préfixe/suffixe)
- Exemple de []
- Exemple de ()
- L'opérateur de casting
- Illustration avec les itérateurs

Le cas des itérateurs est une bonne illustration

On en a déjà rencontrés (masqués) avec vector

```
int main() {  
    vector<int> v = { 1, 2, 3 };  
    for (int x : v) cout << x;  
}
```

1 2 3

Le cas des itérateurs

On en a déjà rencontrés avec vector

```
int main() {  
    vector<int> v = { 1, 2, 3 };  
    for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)  
        cout << *i;  
}
```

1 2 3

La forme explicite ...

Le cas des itérateurs

On en a déjà rencontrés avec vector

```
int main() {  
    vector<int> v = { 1, 2, 3 };  
    for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)  
        cout << *i;  
}
```



1 2 3

La forme explicite fait apparaître :

- vector<int>::iterator on reconnaît une classe interne
- ++i une redéfinition de l'opérateur ++prefixe
- *i une redéfinition de l'opérateur *
- != une redéfinition de l'opérateur !=

Un itérateur est une « sorte » de pointeur, qui va pointer successivement sur les éléments du vecteur, de la liste, ...

Le cas des itérateurs - Exercice

Définissons une classe A qui représente un intervalle [x,y] associé à un 'step'.

On veut que ces intervalles A possèdent des itérateurs :

```
int main() {  
    A a{20,1,4}; // intervalle de 1 à 20 par pas de 4  
  
    for (A::iterator i=a.begin(); i!=a.end() ;++i) cout << *i ;  
    for (int x:a) cout << x << " ";  
}
```

```
1 5 9 13 17  
1 5 9 13 17
```

```
class A {  
public:  
    int x, y;  
    int step;  
    A(int x,int y,int step=1);  
};
```

```
A::A(int x,int y, int s): x{min(x,y)}, y{max(x,y)}, step{s} {};
```

```
class A {
public:
    int x, y;
    int step;
    A(int x,int y,int step=1);
    class iterator {
        int i;
        A & source;
        iterator(A& x, int i);
    public :
        iterator& operator++();
        int operator*();
        bool operator!=(const iterator &) const;
        friend class A; // pour begin et end qui construisent
    };
    A::iterator begin();
    A::iterator end();
};
```

```
class A {
public:
    int x, y;
    int step;
    A(int x,int y,int step=1);
    class iterator {
        int i;
        A & source;
        iterator(A& x, int i);
    public :
        iterator& operator++();
        int operator*();
        bool operator!=(const iterator &) const;
        friend class A; // pour begin et end qui construisent
    };
    A::iterator begin();
    A::iterator end();
};
```

```
A::iterator::iterator(A& x, int i):source{x},i{i} {}
```

```

class A {
public:
    int x, y;
    int step;
    A(int x,int y,int step=1);
    class iterator {
        int i;
        A & source;
        iterator(A& x, int i);
    public :
        iterator& operator++();
        int operator*();
        bool operator!=(const iterator &) const;
        friend class A; // pour begin et end qui construisent
    };
    A::iterator begin();
    A::iterator end();
};

```

```

A::iterator A::begin() { return A::iterator{*this,x}; }
A::iterator A::end()   { return A::iterator{*this,y+1}; }

```

```

class A {
public:
    int x, y;
    int step;
    A(int x,int y,int step=1);
    class iterator {
        int i;
        A & source;
        iterator(A& x, int i);
    public :
        iterator& operator++();
        int operator*();
        bool operator!=(const iterator &) const;
        friend class A; // pour begin et end qui construisent
    };
    A::iterator begin();
    A::iterator end();
};

```

```

A::iterator& A::iterator::operator++() {
    i=min(i+source.step , source.y+1);
    return *this;
}
int A::iterator::operator*() {return i;}

```



```

class A {
public:
    int x, y;
    int step;
    A(int x,int y,int step=1);
    class iterator {
        int i;
        A & source;
        iterator(A& x, int i);
    public :
        iterator& operator++();
        int operator*();
        bool operator!=(const iterator &) const;
        friend class A; // pour begin et end qui construisent
    };
    A::iterator begin();
    A::iterator end();
};

```

```

bool A::iterator::operator!=(const iterator &x) const {
    return (x.i!=i) || (&source != &(x.source)); // y avez vous pensé ?
}

```

On a bien :

```
int main() {  
    A a{20,1,4}; // intervalle de 1 à 20 par pas de 4  
  
    for (A::iterator i=a.begin(); i!=a.end() ;++i) cout << *i ;  
    for (int x:a) cout << x << " ";  
}
```

1	5	9	13	17
1	5	9	13	17

Conclusion - Surcharge des opérateurs :

- attention à respecter un sens proche des habitudes (ne pas redéfinir $*$ pour faire $\&$ et inversement)
- penser à la commutativité,
- associativité gauche ou droite fixée
- combinaisons d'opérateurs (coordonnées avec priorités)
- si vous redéfinissez $<$ il faudra aussi redéfinir $>$
- si vous redéfinissez $==$ il faudra aussi redéfinir $!=$
- si vous redéfinissez $<$ et $==$ il faudra redéfinir
 $<=$ et $>=$ et $>$ et $!=$
- si vous redéfinissez $=$ et $+$ pensez aussi à $+=$