



## Les opérateurs

### Chapitre 2



## Opérateurs et expressions

- Une expression renvoie un résultat. Elle contient un opérateur représenté par un symbole pour effectuer une opération
  - Opérateurs arithmétiques : +, -, \*, /, %  
 $1 + 2$
  - Opérateurs logiques : &&, ||, !  
 $!trouvé \ \&\& \ (i < 20)$
  - Opérateurs de comparaison : ==, !=, >=, <=, <, >
  - Opérateurs d'incrément : ++, --

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

2



## Autres opérateurs

- Opérateurs simplifiés : +=, -=, ...
- Opérateurs binaires (manipulation de bits) :  
&, |, ^ (ou exclusif), ~(compl. à 1),  
<< (décalage de n bits à gauche), >>
- Opérateur ternaire  
 $z = (a > b) ? a : b ;$
- Opérateur sizeof  
`int i; sizeof(i); sizeof(long);`

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

3



## Surcharge des opérateurs

- Tous les opérateurs disponibles en C (+, -, ==, ...) peuvent être surchargés.  
-> ils peuvent être reprogrammés pour s'adapter au type des opérandes  
  
 $// \text{comparaison de motos}$   
 $Moto \ Kawa(80), \ BMW(100);$   
 $if (Kawa < BMW) \ cout << "...; "$
- Pas de combinaison possible (\*\* pour exponentielle est impossible)
- On ne peut pas changer le nombre d'arguments requis par un opérateur
- Un opérande au moins doit être d'un type classe

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

4



## L'opérateur + pour l'addition de deux nombres complexes

Complexe C1(1.0), C2(2.0), C3;  
C3 = C1 + C2;

- S'il est codé par une **fonction globale** (fonction non membre de la classe), il sera interprété par :

C3 = operator+ (C1, C2);

- S'il est codé par une **méthode de classe** (fonction membre de la classe), il sera interprété par :

C3 = C1.operator+ (C2);



## Operator + Surcharge par une méthode de la classe

```
class Complexe { float re, im;
public :
    Complexe (float = 0.0, float = 0.0);
    Complexe operator+(const Complexe &);
    float Re() {return re;}
    ... };
```

```
Complexe Complexe::operator+(const Complexe & c2)
{
    Complexe res;
    res.re= re + c2.re;
    res.im = im + c2.im;
    return res;
    // return Complexe( re + c2.re, im + c2.im);
}
```



## Operator + Surchargé par une fonction globale

```
class Complexe { float re, im;
public :
```

// accès aux attributs privés grâce à friend

**friend Complexe operator+(const Complexe &, const Complexe &);**

};

// fonction globale

Complexe operator+(Complexe &c1, Complexe &c2)

{

return Complexe( c1.re + c2.re, c1.im + c2.im);

}



## Quel type de surcharge choisir?

Tout dépend du type du premier opérande

- Si le premier opérande est un type de base (int, ostream, ...), l'opérateur sera codé par une fonction globale.

```
Complexe C1(1.0), C3;
C3 = 3.14 + C2; // double + Complexe
```

```
Livre lion(« Le lion », »Kessel »);
cout << lion; // ostream << Livre
```

```
string s(« il fait beau »);
Chaine c(« aujourd'hui »);
s + c; // string << Chaine
```

## double + Complexe

- Codage par une fonction globale

```
class Complexe {
    float re, im;
    public :
    ....
};

// surcharge de l'opérateur +
Complexe operator+(double nb, Complexe &c2) {
    Complexe res;
    res.re= nb + c2.re;
    res.im = nb + c2.im;
    return res;
}
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

9

## string + Chaine

- Codage par une fonction globale

```
class Chaine { char *pt;
    public:
        Chaine(int);
        Chaine(char *);
        Chaine(Chaine &);
        int longueur();
        void ajouter(string);
        void ajouter(Chaine&);
};

// surcharge de l'opérateur +
Chaine operator+(string mot1, Chaine &mot2) {
    Chaine res(mot1.size()+mot2.longueur());
    res.ajouter(mot1);
    res.ajouter(mot2);
    return res;
}
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

10

## ostream << Livre

- Codage par une fonction globale

```
Livre.h

class Livre {
    // attributs
    public :
    // méthodes
    friend ostream& operator<<(ostream& out, Livre& l);
};

// Surcharge de l'opérateur d'affichage <<
ostream& operator<< (ostream& out, Livre& l) {
    out << « titre » << l.titre << endl;
    out << « auteur » << l.auteur << endl;
    return out; }
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

11

## Quel type de surcharge choisir?

- Si le premier opérande est une classe (Complexe, Livre, Chaine, ...), l'opérateur sera codé par une fonction de la classe.

```
Complexe C1(1.0), C2(2.0), C3;
C3 = C1 + C2; // Complexe + Complexe
```

```
Livre lion(« Le lion », »Kessel »);
lion + 1997; // Livre + int
```

```
Chaine s(« il fait beau »);
Chaine c(« aujourd'hui »);
s + c; // Chaine + Chaine
```

```
Moto Kawa(80), BMW(100);
if (Kawa < BMW) ... // Moto < Moto
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

12

## Livre + int

- Codage par une fonction de la classe Livre

```
class Livre {  
    string titre, auteur; int annee;  
public :  
    // méthodes  
    void operator+(int);  
};
```

Livre.h

```
// Surcharge de l'opérateur +  
void Livre::operator+(int an) {  
    annee = an;  
}
```

Livre.cpp

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

13

## Chaine + Chaine

- Codage par une fonction de la classe Chaine

```
class Chaine { char *pt;  
public:  
    Chaine(int);  
    Chaine(char *);  
    Chaine(Chaine &);  
    int longueur();  
    void ajouter(char *);  
    void ajouter(Chaine&);  
    Chaine operator+ (Chaine&);  
};
```

```
// surcharge de l'opérateur +  
Chaine Chaine::operator+(Chaine &mot2) {  
    Chaine res(this->longueur() + mot2.longueur());  
    res.ajouter(*this);  
    res.ajouter(mot2);  
    return res;  
}
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

14

## Moto < Moto

- Codage par une fonction de la classe Moto

```
class Moto { int puissance;  
public:  
    Moto (int);  
    bool operator< (Moto&);  
};
```

```
// surcharge de l'opérateur <  
bool Moto ::operator<(Moto & m2) {  
    if (this->puissance < m2.puissance)  
        return true;  
    else return false;  
}
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

15

## Modification de l'objet courant

- l'objet courant contiendra le résultat de l'opération
- Retour d'une référence

```
Complexe & Complexe::operator+(const Complexe &c2) {  
    re+= c2.re; im += c2.im;  
  
    // retour de l'objet courant modifié  
    return *this;  
}
```

2011/2012

M1 - MCPOOA - Chap.2 - Opérateurs

16

## Résultat dans un nouvel objet

- Retour d'une zone mémoire réservée dans la fonction (danger! Il faudra penser à la libérer!)

```
Complexe & Complexe::operator+(const Complexe &c2)
{ // objet courant non modifié
  Complexe &res = *new Complexe (*this);
  // objet courant modifié
  // Complexe &res = *this;
  res.re+= c2.re; res.im += c2.im;
  return res; }
```

## L'opérateur d'affectation : operator=

- Pour modifier la valeur d'un objet existant

Tableau t1,t2;  
t1 = t2;

```
Classe& Classe::operator= (const Classe & copie )
{ if (this != &copie) // éviter l'auto-affectation
  // modifier l'objet courant pour qu'il soit égal à source
  return *this;
}
```

## Exemple

```
Tableau & Tableau::operator=(const Tableau & t2)
{
  if (this != &t2) {
    delete [] valeurs;
    taille = t2.taille; nb = t2.nb; valeurs=new int [taille];
    for (int i=0;i<taille;i++)
      valeurs[i] = t2.valeurs[i];
  }
  return *this;}
```

## Attention !

- Contrairement au constructeur copie, dans lequel on construit un nouvel objet, l'objet existe déjà.
- Génération d'un opérateur= par défaut qui effectue une simple affectation, données membre à données membre (copie mémoire)  
=> insuffisant si on utilise les pointeurs
- Retourner le résultat de l'affectation (return \*this) ce qui permet des affectations multiples telles que

A = B = C;



## Forme canonique d'une classe

- Quand la classe contient des pointeurs

```
class T {  
    public:  
        T(...);  
        T(const T &);  
        ~T();  
        T& operator= (const T &);  
        ...  
};
```



## Opérateur d'indice [ ]

- Mécanisme d'indexation similaire à celui des tableaux mais s'appliquant à un objet  
**TYPE operator [ ] (INDICE);**
- Accès à une composante par :  
**OBJET [ INDICE]**
- Contrôle la validité d'un indice
- Différents types d'indice : entier, chaîne de caractères, ...



## Exemple

```
// en lecture seulement  
Livre Biblio::operator[] (int i) const  
{ return tab[i]; }  
  
// en lecture / écriture  
Livre & Biblio::operator[] (int i)  
{ return tab[i]; }  
  
Livre L1, L2; Biblio B;  
B[0] = L1; // écriture  
L2 = B[0]; // lecture
```



## Les opérateurs -- et ++

- **TYPE operator++ ();**  
Préincrémenté pour ++OBJET,  
le retour est OBJET + 1
- **TYPE operator++ (int);**  
Postincrémenté pour OBJET++  
int est fictif, il surcharge la fonction,  
le retour est OBJET



## Exemple

- Post-incrémentation

```
Complexe Complexe::operator++(int)
{ Complexe old(*this);
  re++; return old;}
```

- Pré-incrémentation

```
Complexe Complexe::operator++()
{ re++; return *this;}
```



## L'opérateur fonction ()

- Opérateur n-aire (n arguments)
- Privilégie une fonction particulière de la classe en simplifiant l'accès  
TYPE operator () (Paramètres formels);  
OBJET (Paramètres effectifs)
- Réaliser des objets qui se comportent comme des fonctions (foncteurs)  
Matrice m(2,3);  
m(i,j) = 6;
- Permet l'implantation d'itérateurs
- L'itérateur est un objet permettant d'accéder successivement aux différents éléments d'un objet composite auquel il est connecté (généralement des listes).



## Exemple

```
class Iterateur {
  Cell *p;
public :
  Iterateur (List &l) {p = l.tete; }
  Elem operator() () {return p->val; }
  void operator++ () { p = p -> suiv; }
  operator int () {return (int) p ; }
};
```



## Utilisation

```
Liste l;
for (Iterateur it(l) ; it ; ++it)
  cout << it();
```

## Conversions de type

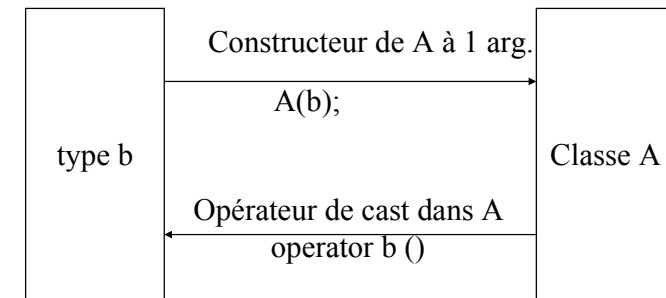
- Conversion explicite

```
int n; double z;  
z = double (n);
```

- Conversions implicites du compilateur dans :

- les affectations `z = n ;`
- les appels de fonctions
- les expressions `z + n`

## Conversion de type



## Conversion classe -> type

`operator T ();`

- Retourne une expression de type T
- Un opérateur de cast est toujours défini comme une fonction membre
- Le type de la valeur de retour n'est pas mentionnée

## Conversion Complexe -> float

```
class Complexe {  
    ...  
    operator float() { return re;}  
};  
  
Complexe c (2.0);  
float f;  
f = c; // appel implicite de float()
```