MEMBRES AMIS : FRIEND

Les **fonctions** (ou opérateurs) ou les **classes** déclarées **amies** dans la **déclaration** d'une classe peuvent accéder aux membres privés de cette classe.

```
class Complexe {
    // Déclaration des fonctions et classes amies
    friend void testFct(Complexe *);
    // Ceci déclare que la fonction
    // testFct est fonction amie de la classe complexe.
    friend ostream & operator<<(ostream &,
                                const Complexe &);
    // operator<< avec un complexe comme argument
    // peut accéder à im et re;
    friend class X;
    // Toutes fonctions membres de X peuvent accéder à
    // im et re
    ...
private:
    float re, im; // PRIVES
};
// Définition de la fonction amie testFct
void testFct(Complexe * pc) { pc->re = 5 ; }
// Définition de l'opérateur ami operator<<
ostream & operator<<(ostream &s, const Complexe & z) {
    s << z.re << z.im;      // l'accès aux membres
    // privés est utorisé car
    // ostream & operator<<(ostream &s,const Complexe& z)
    // est une « fonction amie » de Complexe.
    return s;
}
int main() {
    Complexe c(1,3) ;
    cout << c ;
    cout << c.re() << c.im()  ;          etc.
}
// Définition de la classe X
class X {
    ...
public:
    void essai(Complexe * a) { a->re=0; a->im=15; }
    // Autorisé car X est une "classe amie" de Complexe.
};
```

SURCHARGE D'OPERATEURS (1)

La plupart des opérateurs peuvent être surchargés et avoir ainsi une signification particulière pour une classe donnée.

L'action de la fonction n'est pas obligatoirement identique à celui de l'opérateur initial (+ peut ne pas être une addition)

Un opérateur conserve son arité (ie le nombre d'opérandes)

Un opérateur binaire (*resp* unaire) est implanté soit par une fonction membre avec un (*resp* zéro) paramètre, soit par une fonction amie avec 2 (*resp* 1) paramètres, mais pas les deux à la fois.

➢ Syntaxe

```
operator xxx      où xxx est un symbole parmi
+ - * / % ^ & | ~ ! << >> = != == || && ++ -- [] ()
new delete toutes_les_affectations_composées (+= ..)
tous_les_opérateurs_relationnels (< <= ...)
```

➢ Exemple

```
class Complexe {
  friend Complexe operator+ ( const Complexe&,
                              const Complexe&);
private: float re,im;
public:
   Complexe operator- (const Complexe & c) const;// *this-y
   Complexe operator- () const;                  // -this
};
Complexe Complexe::operator- (const Complexe & b) const {
   Complexe r; r.re=re-b.re; r.im=im-b.im; return r;}
Complexe Complexe::operator- () const {
   Complexe r; r.re=-re; r.im=-im; return r;}
}

// fonction « operator + »
Complexe operator+ (const Complexe &a, const Complexe & b) {
   Complexe r; r.re=a.re+b.re; r.im=a.im+b.im; return r;

int main() { Complexe x,y,z;  z = x+y; x = -y; y = x-z; …}
```

SURCHARGE D'OPERATEURS (2)

Rq: opérateurs non surchargeables :

```
::.     .*      sizeof   ?:
```

`.*` = pointeur sur membre

Rq: les opérateurs **`= [] () ->`** doivent être des fonctions membres non statiques et non des fonctions amies

Rq: les fonctions opérateurs peuvent être appelées explicitement, bien que cela ne soit pas l'usage courant

Exemple: `z = a.operator+(b); // idem z=a+b;`

Rq: les opérateurs `++ et --` sont l'incrémentation et la décrémentation. Distrinction entre entre pré (++i) et post (i++) incrémentation : ce sont deux opérateurs distincts

```
class X {
public:
  X& operator++();
     // Pré-incrémentation : fonction appelée pour ++a
  X& operator++(int); // paramètre int Obligatoire
     // Post-incrémentation : fonction appelée pour a++
};
int main() {
  X b;
  ++b; // b.operator++()       :   1ère  fonction
  b++; // b.operator++(0)      :   2ième fonction
  return 0;
}
```

Conseils avec les opérateurs

- Pour tous les opérateurs usuels (affectation, crochet...) pensez à utiliser les signatures conseillées...
- Visez la complétude, eg opérateur =, deux versions de l'opérateur [] ...

SURCHARGE DE ->

** Classe qui se comporte comme un pointeur.

** Contrôle d'accès à un membre à travers les pointeurs (problème de typage statique & dynamique) : debogage, protection, réflexe

** Opérateur **unaire** de nom : `operator->()`

** Syntaxe d'appel : `expression->nom_de_membre`

** Evalué comme : `(expression.operator->())->nom_de_membre`

** Doit retourner un **pointeur** ou une **référence sur un objet comportant un mebre**

Exemple:

```
#include <iostream>
using namespace std;

class X {  int data;
  public:
  X(int a) : data(a) {};
  void f() { cout << "Appel de f :"<<++data<<endl;  }
};

class Ptr{   X* pointeur;
  public:
  Ptr(X* p=NULL) : pointeur(p) {}
  X* operator->() { cout << "Point. intelligent"<< endl;
    if (pointeur) return pointeur;
    else { cout << "Erreur: non valide:"<<endl;
      exit(1);
    }
  }
};

int main() { X x1(2);
  Ptr p1(&x1), p2;;
  p1->f();   p2->f();  p1->f();
  return 0;
}
```

SURCHARGE DE [ ]

Opérateur **binaire** de nom : `<type> operator[](int)`
Utilisé pour trouver un élément dans une collection, e.g. : `t[i]`

Pour écrire : `t[i]=0;` [ ] retourne une **Lvalue** (un objet) et non une valeur

En général en deux versions :

- version *const* qui retourne une copie ou une référence const *pour appeler [ ] sur un objet const*
- version *non const* qui retourne une référence non const *pour écrire* `t[i]=0;`

➢ Exemple

Le programme suivant affiche « v1 D v2 v2 Z »

```
class Chaine {    char *p;
public:
    Chaine(const char *s) {  p=strdup(s);  }

    // V1 -  Operateur[ ]const. Retourne copie ou const &
    char operator[](int i) const {
      cout << "v1" << endl;     return p[i] ;
    }
    //ou bien : const char & operator[](int i) const {...}


    // v2 -  Operateur[ ]non const. Return une & !
    char & operator[](int i) {
      cout << "v2" << endl;     return p[i];
    }
};

int main() {
    const Chaine s1 ("ABSDEF");
    cout << s1[3] << endl; // Pas de problème : v1

    Chaine s2 ("ABSDEF");
    s2[3] = 'Z';            // Pas de problème : v2
    cout << s2[3] << endl; // Pas de problème : v1
    …
}
```

CONVERSION DEFINIE PAR L'UTILISATEUR

➢ Conversion via un constructeur

Exemple: conversion réel-> complexe

```
Complexe::Complexe (double reel) {re=reel; im=0; }
main() {
    complexe x;
    x = 2.0; // idem x = complexe(2.0);
}
```

Usage très limité :

- Conversion vers un type de base impossible
- Pas de différence construction-conversion

➢ Opérateur de conversion

Syntaxe : fonction membre operator de même nom que le type de base
Pas de type de retour précisé.

Exemple: conversion complexe -> réel

```
class Complexe {
    float re, im;
public:
    Complexe(double a, double b = 0. ) { re=a; im=b;}
    operator double () const    ;
};

Complexe::operator double () const { return(re); }

main() {
    Complexe x(2.0,3.0);
    double a;
    a = x; // a=2.0 :valeur retournée par la conversion
}
```

Remarque : il est possible de définir un tel opérateur de conversion vers non pas un type de base mais un objet.

FONCTION AMIE *VS* FONCTION MEMBRE (1)

- Une **fonction** (ou opérateur) qui modifie l'état de l'objet sur lequel elle travaille doit de préférence être une fonction membre.
  L'appel se fait par `id_variable.id_fonction(paramètres)`

Exemple *:*

```
class complexe { float re,im;
  complexe (float a, float b=0) { re=a; im=b; }

  // operateur += pour l'addition « en place »
  complexe & operator+= (const complexe& other) {
    this->re += other.re ;
    this->im += other.im ;
    return *this ;
  }
};
main() {
    complexe c (10,20);
    complexe c2(20, 10) ;
    c2 += c ;
}
```

Considérons par contre l'exemple suivant :

```
class complexe { float re,im;
  complexe (float a, float b=0) { re=a; im=b; }

  // operateur pour faire c + 1.0 par exemple
  complexe operator+ (double x) {
    complexe other( this.re + x, this.im) ;
    return other;
  }
};
int main() {
    complexe c (10,20);
    complexe c2 = c + 1.0 ; // ca ca marche
    complexe c2 = 1.0 + c ; // mais là ca coince !
    …
}
```

Avec cette définition de l'opérateur `operator+ (double x)`, il n'est impossible de faire **`1.0+c`**.
Pour l'opérateur +, on préfèrera donc écrire des opérateurs amis, comme ci dessous.

FONCTION AMIE *VS* FONCTION MEMBRE (2)

- Une **fonction commutative** qui nécessite des conversions éventuelles de ses paramètres doit de préférence être une fonction amie.

  L'appel se fait par `id_fonction(id_variable,paramètres)`

Exemple : `affiche2(x);`

```
class complexe { float re,im;
  public :
  friend complexe operator+ (const complexe &,
                             const complexe &);
  // Pour x+y

  friend complexe operator+ (const complexe&,float);
  // Pour x+1.0

  friend complexe operator+ (double,const complexe&);
  // Pour 1.0+x
};

complexe operator+ (const complexe&a, const complexe &b){
  complexe r; r.re=a.re+b.re; r.im=a.im+b.im; return r;}
complexe operator+ (const complexe&a, float x){
  complexe r; r.re=a.re+x; r.im=a.im; return r;}
complexe operator+ (float x, const complexe&a){
  complexe r; r.re=a.re+x; r.im=a.im; return r;}


int main() {
     complexe x,y,z;
     z=x+y; x=y+1; y=2+z;
     …
}
```