Séance 5
Surcharge d'opérateurs

Surcharge d'un opérateur

```
class Complexe
{
public :
   …
   Complexe operator+(const Complexe& c2) const;

private :
   float re, im;
};
```

Complexe A(..), B(..);
But : écrire,
Complexe C = A + B;

=> ajouter une fonction membre operator+ :

- objet courant : 1er opérande
- paramètre : 2ème opérande
- retourne un Complexe résultat

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

Le compilateur traduit A + B par A.operator+(B)

Opérateurs : possibilités et limitations

On peut :

- surcharger tout opérateur du langage
  en lui donnant la signification qu'on veut
  important : donner aux opérateurs des significations intuitives

- écrire des expressions
  ex. si operator+ et operator* dans Complexe,
  on peut écrire C1 + C2 * C3 + C4 + C5

On ne peut pas :

- utiliser un symbole qui n'est pas un opérateur
  ex. operator$

- changer le nb d'opérandes d'un opérateur donné
  ex. :
  A++ oui, A ++ B non ∀ signification donnée à ++

- changer la priorité
  * plus prioritaire que + ∀ leur signification

Quelques opérateurs : opérateurs de comparaison
operator== 2 opérandes, retourne bool
utilisation : if (A == B)
idem !=, <, >, <=, >=

Plusieurs formes d'un même opérateur

```
class Complexe
{
public :
   …
   Complexe operator+(const Complexe& c2) const;  (1)
   Complexe operator+(float x) const;  (2)

private :
   float re, im;
};
```

```
Complexe A(..);
Complexe C = A + 3.5;  (2)
```

traduit par :

```
A.operator+(3.5);  (2)
```

```
Complexe  Complexe::operator+(float x) const {  (2)
   Complexe R(re + x, im);
   return R;
}
```

Opérateur en tant que fonction non membre

```
class Complexe
{
public :
  …
  Complexe operator+(const Complexe& c2) const; (1)
  Complexe operator+(float x) const; (2)
 friend Complexe operator+(float x, const Complexe& c2);(3)

private :
  float re, im;
};
```

```
Complexe B(..);
Complexe C = 2.1 + B;   (3)
```

traduit par :

```
operator+(2.1, B);  (3)
```

fonction non membre => opérandes tous en paramètre

```
Complexe  operator+(float x, const Complexe& c2) { (3)
   Complexe R(x + c2.re, c2.im);
   return R;
}
```

Règle : quand le 1er opérande n'est pas un objet de la classe
l'opérateur doit être une fonction non membre (et amie).

Remarque : dans les autres cas on a le choix, fonction membre ou non,
ex. pour l'opérateur (1),
friend Complexe operator+(const Complexe& c1, const Complexe& c2);

Opérateur = (affectation)

Il existe d'office pour toute classe.

```
Complexe A(4, 6), B(7, 5);
…
A = B;       copie les champs de B dans ceux de A
```

Pour certaines classes (ex. String) l'opérateur = créé automatiquement par le compilateur n'est pas approprié => il faut le réécrire.

s1 = s2; deux opérandes

```
String&  String::operator=(const String& str2) {
   objet courant : destination de la copie, objet en paramètre : source de la copie
}
```

retourne String& pour comportement standard (x = y retourne une référence sur x)
=> return *this

Opérateurs +=, -=, *=, /=

L'opérateur += n'existe pas automatiquement même si opérateurs + et = existent.

```
Complexe A(..), B(..);
A += B;
```

```
Complexe&  Complexe::operator+=(const Complexe& c2) {

   on modifie l'objet courant en lui ajoutant c2

   return *this;       pour avoir le comportement de += standard
}
```

Opérateurs d'affichage (<<) et de saisie (>>)

cin : un objet de type istream
cout : un objet de type ostream

```
class Complexe
{
public :
   …
   friend  ostream& operator<<(ostream& flux,
                                const Complexe& c);
   friend istream& operator>>(istream& flux, Complexe& c);
   …
private :
   float re, im;
};
```

opérateurs << et >> fonctions amies car le 1er opérande (cout ou cin) n'est pas un objet Complexe

Complexe A(..);
cout << A …
cin >> A …

```
ostream&  operator<<(ostream& flux,
                                const Complexe& c) {
    flux << c.re << " + i " << c.im;
    return flux;
}
```

```
istream& operator>>(istream& flux, Complexe& c) {
    flux >> c.re >> c.im;
    return flux;
}
```

retour de flux (cout ou cin) pour pouvoir enchaîner sur un autre << ou >>