

Attributs de classe

Revenons à notre exemple du *comptage des instances* :

```
int compteur(0); // Hmm...

class Rectangle {
    // constructeur par défaut
    Rectangle(): hauteur(0.0), largeur(0.0)
    { ++compteur; }

    // destructeur
    ~Rectangle() { --compteur; }
    //...
```

Oh horreur !.. Nous avons utilisé une *variable globale* !

C'est une **très mauvaise solution** ! (contraire au principe d'encapsulation, effets de bord, mauvaise modularisation)

Attributs de classe (2)

La solution à ce problème consiste à utiliser un **attribut de classe** :

```
class Rectangle {
private:
    double hauteur, largeur;
    static int compteur;
    //...
};
```

- ▶ La déclaration d'un attribut de classe est précédée du mot clé **static**
- ▶ Un attribut de classe est **partagé par toutes les instances** de la même classe (on parle aussi d'« *attribut statique* »)
- ▶ Il existe même lorsqu'*aucune* instance de la classe n'est déclarée
- ▶ Un attribut de la classe peut être *privé* ou *public*

Initialisation des attributs de classe

Un attribut de classe doit être initialisé explicitement à l'extérieur de la classe

```
/* Initialisation de l'attribut de classe dans le fichier de
   définition de la classe, mais HORS de la classe.
*/

int Rectangle::compteur(0);

/* Rectangle::compteur existe même si l'on n'a déclaré
   aucune instance de la classe Rectangle */
```

Les attributs de classe sont très pratiques lorsque *différents* objets d'une classe doivent accéder à une *même* information.

Ils permettent notamment d'*éviter* que cette information soit *dupliquée* au niveau de chaque objet.

- ☛ Concrètement : réserver cet usage à des **constantes** utiles pour toutes les instances de la classe.

Méthodes de classe

Similairement, si on ajoute **static** à une méthode :

- ▶ on peut accéder aussi à la méthode *sans* objet, à partir du nom de la classe et de l'opérateur de résolution de portée « **::** »

```
class A {
public:
    static void methode1() { cout << "Méthode 1" << endl; }
    void methode2() { cout << "Méthode 2" << endl; }
};

int main () {
    A::methode1(); // OK
    A::methode2(); // ERREUR
    A x;
    x.methode1(); // OK
    x.methode2(); // OK
}
```

Restrictions sur les méthodes de classe

Puisqu'une méthode de classe peut être appelée sans objet :

- ▶ elles n'ont pas le droit d'utiliser de méthode ni d'attribut d'instance (y compris `this`)
- ▶ elles ne peuvent accéder *seulement* **qu'**à d'autres méthodes ou attributs *de classe*

☞ Ce sont simplement des *fonctions usuelles* mises dans une classe.

Le recours à des méthodes de classe ne se justifie que dans des situations **très** particulières :

- ▶ affichage spécifique d'attributs de classe
- ▶ manipulation d'attributs de classe non constants et `private`

☞ Préférez toujours les fonctions usuelles et évitez absolument la prolifération de `static` !

Intérêt ?

Exemple avec les nombres complexes :

```
class Complexe { ... };
Complexe z1, z2, z3, z4;
```

Il est quand même plus naturel d'écrire :

```
z4 = z1 + z2 + z3;
```

que `z3 = addition(addition(z1, z2), z3);`

De même, on préférera unifier l'affichage :

```
cout << "z3 = " << z3 << endl;
```

plutôt que d'écrire :

```
cout << "z3 = ";
affiche(z3);
cout << endl;
```

En pratique, quelle surcharge des opérateurs ?

Dans votre pratique du C++, vous pouvez, [en fonction de votre niveau](#) :

- ① ne pas faire de surcharge des opérateurs ;
- ② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...) ;
surcharger l'opérateur d'affichage (<<) ;
- ③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :
- ④ faire la surcharge avec valeur de retour des opérateurs d'« auto-affectation » :

```
void operator+=(Classe const&);
```

```
Classe& operator+=(Classe const&);
```

Opérateur ?

Rappel : un opérateur est une opération sur un ou entre deux opérande(s) (variable(s)/expression(s)) :

opérateurs arithmétiques (+, -, *, /, ...), opérateurs logiques (and, or, not),

opérateurs de comparaison (==, >=, <=, ...), opérateur d'incrément (++), ...

Un appel à un opérateur est un appel à une fonction ou une méthode spécifique :

`a Op b` \longrightarrow `operatorOp(a, b)` ou `a.operatorOp(b)`

`Op a` \longrightarrow `operatorOp(a)` ou `a.operatorOp()`

Exemples d'appels d'opérateurs

<code>a + b</code>	correspond à	<code>operator+(a, b)</code>	ou	<code>a.operator+(b)</code>
<code>b + a</code>		<code>operator+(b, a)</code>		<code>b.operator+(a)</code>
<code>-a</code>		<code>operator-(a)</code>		<code>a.operator-()</code>
<code>cout << a</code>		<code>operator<<(cout, a)</code>		<code>cout.operator<<(a)</code>
<code>a = b</code>		—		<code>a.operator=(b)</code>
<code>a += b</code>		<code>operator+=(a, b)</code>		<code>a.operator+=(b)</code>
<code>++a</code>		<code>operator++(a)</code>		<code>a.operator++()</code>
<code>not a</code>		<code>operator not(a)</code>		<code>a.operator not()</code>
	ou	<code>operator!(a)</code>		<code>a.operator!()</code>

Surcharge ?

Rappel : surcharge de fonction

☞ deux fonctions ayant le même nom mais pas les mêmes paramètres

Exemple :

```
int max(int, int);  
double max(double, double);
```

De la même façon, on va pouvoir écrire plusieurs fonctions pour les opérateurs ;
par exemple :

```
Complexe operator+(Complexe, Complexe);  
Matrice operator+(Matrice, Matrice);
```

Surcharge interne et surcharge externe

Presque tous les opérateurs sont surchargeables
(sauf, parmi ceux que vous connaissez, `::` et `.`)

La surcharge des opérateurs peut être réalisée

- ▶ soit à l'**extérieur**,
- ▶ soit à l'**intérieur**

de la classe à laquelle ils s'appliquent.

```
Complexe operator+(Complexe, Complexe);
```

```
class Complexe {  
public:  
    Complexe operator+(Complexe) const;  
};
```

☞ Les opérateurs externes sont des *fonctions* ;
les opérateurs internes sont des *méthodes*.

Exemple de surcharge externe

```
Complexe z1;  
Complexe z2;  
Complexe z3;  
// ...  
z3 = z1 + z2;
```

```
class Complexe {  
public:  
    Complexe(double abscisse, double ordonnee)  
        : x(abscisse), y(ordonnee) {}  
    // ...  
    double get_x() const;  
    double get_y() const;  
    // ...  
private:  
    double x;  
    double y;  
};  
  
const Complexe operator+(Complexe z1, Complexe const& z2)  
{  
    Complexe z3( z1.get_x() + z2.get_x(),  
                 z1.get_y() + z2.get_y() );  
    return z3;  
}
```

Choix du prototype

► base :

```
Complexe operator+(Complexe z1, Complexe z2);
```

► optimisation :

```
const Complexe operator+(Complexe const& z1, Complexe const& z2);
```

► avancé et C++11 :

```
const Complexe operator+(Complexe z1, Complexe const& z2);  
// const Complexe operator+(Complexe const& z1, Complexe&& z2);
```

Nécessité de la surcharge externe

La surcharge **externe** est nécessaire pour des opérateurs concernés par une classe, mais pour lesquels la classe en question **n'est pas** l'**opérande de gauche**.

Exemples :

1. multiplication d'un nombre complexe par un double :

```
double x;  
Complexe z1, z2;  
// ...  
z2 = x * z1;
```

`z2 = x.operator*(z1);`
ou `z2 = operator*(x, z1);`

Le premier **n'a pas de sens** (x n'est pas un objet, mais de type élémentaire `double`)

2. écriture sur `cout` : `cout << z1;`

`cout.operator<<(z1);` ou `operator<<(cout, z1);`,
mais on souhaite le surcharger pour la classe `Complexe` et non pas dans la classe de `cout` (`ostream`).

Exemple de la multiplication externe

```
double x;  
Complexe z1, z2;  
// ...  
z2 = x * z1;
```

```
const Complexe operator*(double a, Complexe const& z)  
{  
    /* Soit l'écrire explicitement,  
       soit, quand c'est possible, utiliser l'opérateur interne :  
    */  
    return z * a;  
}
```

Exemple de l'opérateur d'affichage

Exemple (d'appel) :

```
cout << z1; // appel équivalent : operator<<(cout, z1);
```

dont le prototype est (*hors* de la classe) :

```
ostream& operator<<(ostream&, Complexe const&);
```

Définitions de l'opérateur d'affichage

Via des accesseurs :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << '(' << z.get_x() << ", " << z.get_y() << ')';
    return sortie;
}
```

Via une autre méthode :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << z.to_string()
    return sortie;
}
```

OU :

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    return z.affiche(sortie);
}
```

Surcharge externe / friend

Parfois, il peut être nécessaire d'autoriser les opérateurs externes d'accéder à certains éléments **private**.

(mais *préférez passer par les accesseurs* !)

Dans ce cas, ajoutez, **dans** la définition de la classe, leur prototype précédé du mot clé **friend** :

```
friend const Complexe operator*(double, Complexe const&);
friend ostream& operator<<(ostream&, Complexe const&);
```

Le mot clé **friend** signifie que ces fonctions, bien que *ne faisant pas* partie de la classe, peuvent avoir accès aux attributs et méthodes **private** de la classe.

Remarque : les définitions restent **hors** de la classe (et **sans** le mot clé **friend**)

friend : exemple

```
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << '(' << z.x << ", " << z.y << ')';
    return sortie;
}

// ...

class Complexe {
    friend ostream& operator<<(ostream& sortie, Complexe const& z);
    // ...
private:
    double x;
    double y;
};
```

Surcharge interne et surcharge externe

Presque tous les opérateurs sont surchargeables
(sauf, parmi ceux que vous connaissez, `::` et `.`)

La surcharge des opérateurs peut être réalisée

- ▶ soit à l'extérieur,
- ▶ soit à l'intérieur

de la classe à laquelle ils s'appliquent.

```
Complexe operator+(Complexe, Complexe);
```

```
class Complexe {  
public:  
    Complexe operator+(Complexe) const;  
};
```

- ☞ Les opérateurs externes sont des *fonctions*;
les opérateurs internes sont des *méthodes*.

Surcharge interne des opérateurs

Pour surcharger un opérateur *Op* dans une classe *NomClasse*, il faut **ajouter la méthode** `operatorOp` dans la classe en question :

```
class NomClasse {  
    ...  
    // prototype de l'opérateur Op  
    type_retour operatorOp(type_parametre);  
    ...  
};  
  
// définition de l'opérateur Op  
type_retour NomClasse::operatorOp(type_parametre)  
{  
    ...  
}
```

Rappel : les méthodes ne doivent pas recevoir l'instance courante en paramètre

Surcharge interne des opérateurs : exemple

```
Complexe z1, z2;  
// ...  
z1 += z2;
```

```
class Complexe {  
public:  
    // ...  
    void operator+=(Complexe const& z2); // z1 += z2;  
    // ...  
};  
  
void Complexe::operator+=(Complexe const& z2) {  
    x += z2.x;  
    y += z2.y;  
}
```

Retour sur l'addition externe

```
class Complexe {  
public:  
    // ...  
    void operator+=(Complexe const& z2); // z1 += z2;  
    // ...  
};  
  
const Complexe operator+(Complexe z1, Complexe const& z2) {  
    z1 += z2; // utilise l'opérateur += redéfini précédemment  
    return z1;  
}
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : S0IT z3 = z1.operator+(z2);
                //                S0IT z3 = operator+(z1, z2);

class Complexe {
public:
    const Complexe operator+(Complexe const& z2) const;
    // ....
};
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : S0IT z3 = z1.operator+(z2);
                //                S0IT z3 = operator+(z1, z2);

const Complexe operator+(Complexe z1, Complexe const& z2);
```

Surcharge interne ou surcharge externe ?

Les opérateurs propres à une classe peuvent être surchargés en interne ou en externe :

```
z3 = z1 + z2; // appel équivalent : S0IT z3 = z1.operator+(z2);
                //                S0IT z3 = operator+(z1, z2);

const Complexe operator+(Complexe z1, Complexe const& z2);

// ...

class Complexe {
    friend const Complexe operator+(Complexe z1, Complexe const& z2);
    // ...
};
```

Surcharge interne ou surcharge externe ?

- ▶ préférez la *surcharge externe* chaque fois que vous pouvez le faire **SANS** `friend`
c.-à-d. chaque fois que vous pouvez écrire l'opérateur à l'aide de l'interface de la classe
(et sans copies inutiles)
- ▶ si l'opérateur est « proche de la classe », c.-à-d. nécessite des accès internes ou des copies supplémentaires inutiles (typiquement `operator+=`), utilisez la *surcharge interne*

En pratique, quelle surcharge des opérateurs ?

Dans votre pratique du C++, vous pouvez, en fonction de votre niveau :

- ① ne pas faire de surcharge des opérateurs ;
- ② surcharger simplement les opérateurs arithmétiques de base (+, -, ...) sans leur version « auto-affectation » (+=, -=, ...); surcharger l'opérateur d'affichage (<<);
- ③ surcharger les opérateurs en utilisant leur version « auto-affectation », mais sans valeur de retour pour celles-ci :

```
void operator+=(Classe const&);
```

- ④ faire la surcharge avec valeur de retour des opérateurs d'« auto-affectation » :

```
Classe& operator+=(Classe const&);
```

Exemples de surcharges de quelques opérateurs usuels

(au niveau ④ du tr. précédent)

```
bool operator==(Classe const&) const; // ex: p == q
bool operator<(Classe const&) const;  // ex: p < q
```

```
Classe& operator+=(Classe const&); // ex: p += q
Classe& operator-=(Classe const&); // ex: p -= q
```

```
Classe& operator*=(autre_type const); // ex: p *= x;
```

```
Classe& operator++(); // ex: ++p
Classe& operator++(int inutile); // ex: p++
```

```
const Classe operator-() const; // ex: r = -p;
```

```
// ===== surcharges externes =====
```

```
const Classe operator+(Classe, Classe const&); // r = p + q
```

```
const Classe operator-(Classe, Classe const&); // r = p - q
```

```
ostream& operator<<(ostream&, Classe const&); // ex: cout << p;
```

```
const Classe operator*(autre_type, Classe const&); // ex: q = x * p;
```

Pourquoi const en type de retour ?

```
const Complexe operator+(Complexe, Complexe const&);
```

```
z3 = z1 + z2;

++(z1 + z2);
z1 + z2 = f(x);
```

Pourquoi operator<< retourne-t-il un ostream& ?

```
ostream& operator<<(ostream& sortie, Complexe const& z);
```

```
cout << z1 << endl;
operator<<(cout << z1, endl);
operator<<(operator<<(cout, z1), endl);
```

Quel type de retour pour `operator+=` ?

```
z1 += z2;

void Complexe::operator+=(Complexe const&);
```

En C++, chaque expression **fait** quelque chose et **vaut** quelque chose :

`x = Expression;` `z3 = (z1 += z2);`

```
class Complexe {
    // ...
    Complexe& operator+=(Complexe const& z2);
    // ...
};

Complexe& Complexe::operator+=(Complexe const& z2)
{
    x += z2.x;
    y += z2.y;
    return *this;
}
```

Avertissement

Attention de ne pas utiliser la surcharge des opérateurs à **mauvais escient** et à veiller à les **écrire avec un soin particulier**.

Les performances du programme peuvent en être gravement affectées par des opérateurs surchargés mal écrits.

En effet, l'utilisation inconsidérée des opérateurs peut conduire à un grand nombre de copies d'objets :

Utiliser des références dès que cela est approprié !

Avertissement (2)

Exemple : comparez le code suivant qui fait de 1 à 3 copies inutiles :

```
Complexe Complexe::operator+=(Complexe z2)
{
    Complexe z3;
    x += z2.x;
    y += z2.y;
    z3 = *this;
    return z3;
}
```

avec le code suivant qui n'en fait pas :

```
Complexe& Complexe::operator+=(Complexe const& z2)
{
    x += z2.x;
    y += z2.y;
    return *this;
}
```

Opérateur d'affectation

L'opérateur d'affectation `=` (utilisé par exemple dans `a = b`) :

- ▶ est le seul opérateur universel (il est fourni de toutes façons par défaut pour toute classe)
- ▶ est très lié au constructeur de copie, sauf que le premier s'appelle lors d'une **affectation** et le second lors d'une **initialisation**
- ▶ la version par défaut, qui fait une copie de surface, est suffisante dans la très grande majorité des cas
- ▶ si nécessaire, on peut supprimer l'opérateur d'affectation :

```
class EnormeClasse {
    // ...
private:
    EnormeClasse& operator=(EnormeClasse const&) = delete;
};
```

Surcharge de l'opérateur d'affectation

Si l'on doit redéfinir l'opérateur d'affectation on choisira, depuis **C++11**, le schéma suivant :

on commencera par définir une fonction `swap()` pour échanger 2 objets de la classe, (sûrement en utilisant celle de la bibliothèque `utility` (`#include <utility>`) sur les attributs)

puis on définira l'opérateur d'affectation comme suit :

```
Classe& Classe::operator=(Classe source)
// Notez le passage par VALEUR
{
    swap(*this, source);
    return *this;
}
```