Surcharge des opérateurs

Attributs de classe

Revenons à notre exemple précédent ou nous avons compté le nombre d'instances d'une classe :

```
long compteur(0); // variable globale
class Badenya {
    public:
        Badenya() {
            ++compteur;
            ...
```

Utiliser une variable globale est une **très mauvaise solution** !
(contraire au principe d'encapsulation, effets debord, mauvaise modularisation)

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

```
class Badenya {
    ...
    private:
        int numero;
        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 (« attribut statique »)
- Il existe même lorsqu'aucune instance de la classe n'est déclarée

Initialisation des attributs de classe

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

```
int Badenya::compteur = 0;
```

« :: » - Opérateur de résolution de portée

```
class Badenya {
    public:
        Badenya() {
            ++compteur;
            numero = compteur;
            cout << "Constructeur objet " << numero << endl;
        }
        ~Badenya() {
            --compteur;
            cout << "Destructeur objet " << numero << endl;
        }
        Badenya(Badenya const& r) {
            ++compteur;
            numero = compteur;
            cout << "Constructeur de copie " << numero << endl;
        }
    private:
        int numero;
        static int compteur;
};
int Badenya::compteur = 0;
```

```
Constructeur objet 1
Constructeur de copie 2
Constructeur objet 3
Destructeur objet 3
Destructeur objet 2
Destructeur objet 1

Process returned 0 (0x0)
Press any key to continue.
```

```
int main() {
    Badenya b1;
    {
        Badenya b2(b1);
        {
            Badenya b3;
        }
    }
    return 0;
}
```

Usage courant : constantes utiles pour toutes les instances de la classe

Méthodes de classe

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 << "Methode 1" << endl; }
        void methode2() { cout << "Methode 2" << endl; }
};
int main () {
    A::methode1(); // OK
    // A::methode2(); // ERREUR
    A x;
    x.methode1(); // OK
    x.methode2(); // OK
}
```

```
Methode 1
Methode 1
Methode 2

Process returned 0 (0x0)
Press any key to continue.
```

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.

Surcharge des opérateurs - contexte

```
class Complexe {
    private:
        double reelle;
        double imaginaire;
    public:
        Complexe(double a, double b) : reelle(a), imaginaire(b)
        {}
        Complexe() : Complexe(0.0, 0.0)
        {}
        Complexe add(Complexe c){
            Complexe r;
            r.reelle = reelle + c.reelle;
            r.imaginaire = imaginaire + c.imaginaire;
            return r;
        }
        void afficher(){
            cout << "nombre complexe - reelle = " << reelle
                 << " imaginaire " << imaginaire << endl;
        }
};
```

```
nombre complexe - reelle = 3.8 imaginaire 7.6
nombre complexe - reelle = 7 imaginaire 12

Process returned 0 (0x0)   execution time : 0.
Press any key to continue.
```

```
// plus naurel d'écrire
z4 = z1 + z2;

z5 = z1 + z2 + z3;
```

```
int main () {
    Complexe z1(2.2,4.8), z2(1.6,2.8), z3(3.2,4.4);
    Complexe z4(0.0, 0.0), z5(0.0, 0.0);
    z4 = z1.add(z2);
    z4.afficher();
    z5 = z1.add(z2.add(z3));
    z5.afficher();
}
```

De même, on préfèrera unifier l'affichage :

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

⇨ surcharge opérateur « << »

plutôt que d'écrire :

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

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

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

a *Op* b	⟶	operator*Op*(a, b)	ou	a.operator*Op*(b)
Op a	⟶	operator*Op*(a)	ou	a.operator*Op*()

Exemples d'appels d'opérateurs

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

Surcharge et opérateur

Rappel : surcharge de fonction

=> deux fonctions ayant le même nom mais pas les mêmes paramètres

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

De la même façon, on va pouvoir écrire plusieurs fonctions pour les opérateurs

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

La surcharge des opérateurs peut être réalisée

- **soit à l'extérieur** (par des **foncions**)

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

- **soit à l'intérieur** (par des **méthodes**)

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

de la classe à laquelle ils s'appliquent.

Exemple

```
class Complexe {
    private:
        double x;
        double y;
    public:
        Complexe(double a, double b) : x(a), y(b)
        {}
        Complexe() : Complexe(0.0, 0.0)
        {}
        void afficher(){
            cout << "nombre complexe - x = " << x
                << ", y = " << y << endl;
        }
        double get_x() const{ return x; }
        double get_y() const{ return y; }
};
```

```
nombre complexe - x = 3.8, y = 7.6

Process returned 0 (0x0)   execution
Press any key to continue.
```

```
Complexe operator+(Complexe z1, Complexe z2);
int main () {
    Complexe z1(2.2,4.8), z2(1.6,2.8), z3;
    z3 = z1 + z2;
    z3.afficher();

}
Complexe operator+(Complexe z1, Complexe z2)
{
    Complexe z3(z1.get_x() + z2.get_x(), z1.get_y() + z2.get_y() );
    return z3;
}
```

Autres possibilités

```
Complexe operator+(Complexe const& z1, Complexe const& z2)
{
    Complexe z3(z1.get_x() + z2.get_x(), z1.get_y() + z2.get_y());
    return z3;
}
```

```
Complexe& operator+(Complexe const& z1, Complexe const& z2)
{
    Complexe z3(z1.get_x() + z2.get_x(), z1.get_y() + z2.get_y());
    return z3;
}
```

```
const Complexe operator+(Complexe z1, Complexe const& z2)
{
    Complexe z3(z1.get_x() + z2.get_x(), z1.get_y() + z2.get_y());
    return z3;
}
```

C++11

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); n'a pas de sens.

=>

```
z2 = operator*(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;
}
```

2. écriture sur cout

```
cout << z1;
```

cout.operator<<(z1);n'a pas de sens.
On souhaite surcharger la classe Complexe.

=>

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

Prototype

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

- Le paramètre ostream est passé comme référence, nous le modifions
- Le type de retour, voire plus loin

Plusieurs solutions

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;
}
```

Surcharge externe / friend

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

Conseil : préférez passer par les accesseurs pour respecter l'encapsulation.

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

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

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)

Exemple complet

```
#include <iostream>
using namespace std;
class Complexe {
    friend ostream& operator<<(ostream&, Complexe const&);
    private:
        double x;
        double y;
    public:
        Complexe(double a, double b) : x(a), y(b)
        {}
};
```

```
ostream& operator<<(ostream&, Complexe const&);
int main () {
    Complexe z(2.2,4.8);
    cout << z;
    return 0;
}
ostream& operator<<(ostream& sortie, Complexe const& z) {
    sortie << '(' << z.x << ", " << z.y << ')';
    return sortie;
}
```

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 {
    ...
    // Définition méthode de l'opérateur Op
    type_retour operatorOp(type_parametre){

    }
    ...
};
```

```
class Complexe {
    private:
        double x;
        double y;
    public:
        Complexe(double a, double b) : x(a), y(b)
        {}
        void afficher(){
            cout << "(" << x << ", " << y << ")";
        }
        void operator+=(Complexe const& z2) {
            x += z2.x;
            y += z2.y;
        }
};
```

```
int main () {
    Complexe z1(2.2,4.8),z2(1.6,2.8);
    z1 += z2;
    z1.afficher();
    return 0;
}
```

```
(3.8, 7.6)
Process returned 0 (0x0)
Press any key to continue.
```

Lien entre opérateurs

L'on veut exprimer le lien sémantique des opérateurs + et += (dans les deux cas, la même opération somme)
L'opérateur += est par nature un opérateur demandant moins de traitement car il ne créé pas de nouvel objet Complexe
=> définir le plus lourd en fonction du plus léger

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

Remarque : Le paramètre z1 étant passé par valeur, ici z1 est locale à la fonction.

```
const Complexe operator+(Complexe z1, Complexe const& z2);
int main () {
    Complexe z1(2.2,4.8),z2(1.6,2.8),z3;
    z3 = z1 + z2;
    z3.afficher();
    return 0;
}
const Complexe operator+(Complexe z1, Complexe const& z2) {
    z1 += z2; // utilise l'opérateur += redéfini précédemment
    return z1;
}
```

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

Ces deux opérateur ne doivent pas être découplés.

Surcharge interne ou surcharge externe ?

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

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

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

ou

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

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 des méthodes de l'interface la classe

Préférez la surcharge interne si l'opérateur est « proche de la classe »,

c.-à-d. nécessite des accès internes (modifier l'objet) ou des copies supplémentaires inutiles (typiquement operator+=)

Exemples de surcharges de quelques opérateurs usuels

```
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);          // pas de sens!
z1 + z2 = f(x);       // idem
```

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 :

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

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

Avertissement

Les performances du programme peuvent être gravement affectées par des opérateurs surchargés mal écrits, ces opérations sont souvent appelées.
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 !

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,
- 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;
};
```

```
a = b = c; // valide en c++
```

Surcharge de l'opérateur d'affectation

Si l'on doit redéfinir l'opérateur d'affectation on choisira, depuis C++ 2011, le schéma suivant :

C++11

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

- on utilisera la fonction swap() de la bibliothèque utility (#include <utility>)
- ou si nécéssaire on en définira une, pour échanger 2 objets de la classe