

Chapitre 9

Surcharge (redéfinition) des opérateurs

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 9 : Surcharge (redéfinition) des opérateurs

159

1. Généralités

En langage C, l'opérateur / est défini comme suit:

$3 / 2 \rightarrow$ division de deux valeurs entières

$3.0 / 2.0 \rightarrow$ division de deux valeurs réelles

L'opérateur / a été redéfini afin de réaliser deux types de division: une entière et une réelle.

En langage C++, il est possible de redéfinir un (des) opérateur(s) pour les classes.

Pourquoi?

Afin d'assurer une meilleure intégration des classes au programme.

Comment?

- notation: `operator/`
Désigne la fonction à 2 opérandes associée à l'opérateur /
- au moins un opérande de type classe,
=> Impossible de redéfinir / dans une opération 3 / 2
- paramètres de défaut interdits.
- priorité et associativité des opérateurs ne changent pas (standard).
- opérateurs membres ou non membres d'une classe.

Qui peut être redéfini?

On peut redéfinir une quarantaine d'opérateurs:

+ - * / % ^ & | (voir une liste plus ou moins récente dans Deitel&Deitel page 466)

-> opérateur de sélection de membre via pointeur.
[] opérateur d'indexation.
() opérateur d'appel de fonction.
new opérateur d'allocation de mémoire dynamique.
delete opérateur de désaffectation de mémoire dynamique.
etc.

Qui ne peut pas être redéfini?

Ne peuvent être surchargés les opérateurs suivants (une liste non exhaustive):

- opérateur de sélection de membre via objet.
- * opérateur pointeur vers un membre via objet.
- :: opérateur de résolution de portée.
- ? : opérateur conditionnel ternaire.
- sizeof** opérateur déterminant la taille en octets.

2. Surcharge des opérateurs dans le cadre de classes

Soit la classe compte:

```
class compte {  
    double actif;  
public:  
    // etc.  
};
```

Si `C` est du type `compte`, on désire réaliser les deux opérations suivantes:

Ajouter directement à l'actif d'un compte une valeur donnée (dans cet exemple 123.45),

```
C += 123.45;
```

Afficher les données membres d'un compte sur la sortie standard,

```
cout << C;
```

Pour ce faire, nous devons surcharger les opérateurs `+=` et `<<` de la classe `compte`.

3. Mécanisme de surcharge

Un opérateur est une fonction en C++,

Une écriture équivalente à `C+=123.45` est la fonction: `operator+=(C, 123.45);`

Une écriture équivalente à `cout << C` est la fonction: `operator<<(cout, C);`

Un opérateur peut être défini:

- comme une fonction non membre de la classe, à un ou deux arguments:

```
operator+=(C, 123.45)
```

- comme une fonction membre de la classe, avec un argument de moins:

```
C.operator+=(123.45); où operator+=(double) est une fonction membre de la classe compte.
```

4. Étude du cas de l'opérateur +=

4.1. Comme fonction non membre de la classe

```
#include <iostream>

using namespace std;

class compte {
    double actif;
public:
    compte(int i=0):actif(i) {} // constructeur
    void affiche() {cout << "actif: " << actif << endl;} // affiche l'actif
    friend void operator+=(compte&,double); // surcharge de l'opérateur +=
};

// L'opérateur += a été déclaré friend afin de lui permettre l'accès aux données private de la classe compte.

void operator+=(compte& c, double d) {
    c.actif += d;
}

int main() {
    compte X(200.89);
    X+=10.50;
    X.affiche(); // actif: 211.39
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 9 : Surcharge (redéfinition) des opérateurs

4.2. Comme fonction membre de la classe

```
#include <iostream>

using namespace std;

class compte {
    double actif;
public:
    compte(double i=0):actif(i) {} // constructeur.

    void affiche() {cout << "actif: " << actif << endl;} // affiche l'actif.

    void operator+=(double); // surcharge de l'opérateur +=
};

// opérateur += est une fonction membre de la classe compte.

void compte::operator+=(double d){
    actif += d;
}

int main() {
    compte X(200.89);
    X+=10.50;
    X.affiche(); // actif: 211.39
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

5. Étude du cas de l'opérateur <<

Afin de remplacer la fonction affiche membre de la classe compte, nous pouvons surcharger l'opérateur de sortie <<

5.1. Comme fonction non membre de la classe

```
#include <iostream>

using namespace std ;

class compte {
    double actif;
public:
    compte(double i=0):actif(i) {} // constructeur
    friend ostream& operator<<(ostream& out, const compte& c) { // surcharge de l'opérateur <<
    friend void operator+=(compte& compte, double) ; // surcharge de l'opérateur +=
    };
    // Opérateurs << et += ont été déclarés friend afin de leur permettre l'accès aux données private de la classe
    // compte. cout est définie dans la classe ostream (output stream ou flux de sortie), d'où la déclaration du type de
    // out. On passe l'objet c par constante, vu qu'on ne fait qu'afficher le contenu de c. On retourne un ostream pour
    // permettre l'enchaînement des opérations sur la sortie standard, exemple:
    // cout << "1er opération vers cout" << "2e opération vers cout" << "etc.." ;

    ostream& operator<<(ostream& out, const compte& c){
        out << "actif: " << c.actif << endl;
        return out;
    }
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 9 : Surcharge (redéfinition) des opérateurs

```
// On pouvait inclure operator+= comme fonction membre de la classe compte.
void operator+=(compte& c, double d){
    c.actif += d;
}

int main() {
    compte X(200.89);
    X+=10.50;
    cout << X; // actif: 211.39
    return 0;
}
```

5.2. Comme fonction membre de la classe

Si on doit écrire en C++ l'opération a+=b, et si l'opérateur est membre de la classe a, cela se traduit comme suit:

```
a.operator+(b);
```

Il sera de même, sous les mêmes conditions, pour décrire l'opération a << b:

```
a.operator<<(b);
```

De ce fait, il faut que l'opérateur << soit défini dans la classe a pour que cette opération puisse avoir lieu.

Prenons l'exemple suivant:

Si a est un objet de la sortie standard cout et b est une variable du type double,

```
cout << b; revient à écrire cout.operator<<(b);
```

Si cette opération est autorisée en C++, c'est parce que dans la classe ostream (le type de cout), il a été inclus la déclaration suivante:

```
classe ostream {  
    /* etc. */  
public:  
    ostream& operator<<(double);  
    /* etc. */  
};
```

Nous avons montré dans le chapitre 2 "Entrées/Sorties C++", que les types manipulés par le flux en sortie cout et cerr sont:

```
char short int long float double char*
```

Ainsi dans la classe ostream, l'opérateur << a été surchargé afin de supporter les types char short etc.

```
classe ostream {  
    // etc.  
public:  
    ostream& operator<<(double);  
    ostream& operator<<(int);  
    // etc.  
};
```

Or rien n'a été prévu dans la classe ostream pour supporter le type compte!

Que faire alors?

On ne peut pas intervenir sur la classe ostream, pour inclure le type compte. La classe ostream ne peut pas être modifiée.

Puisque c'est ainsi, nous allons redéfinir l'opérateur << comme membre de la classe compte, puisque, après tout, c'est son contenu que nous voulons afficher sur la sortie standard.

En se basant sur l'exemple de l'opérateur +=, pour que l'opérateur << soit membre de la classe compte, il faudra écrire alors:

```
class compte {  
    double actif;  
public:  
    // etc.  
    ostream& operator <<(ostream& out);  
};
```

Cette écriture est traduite en C++ comme suit ...

```
compte.operator<<(out);  
Puis,  
compte << out;
```

Cette écriture, même si elle est correcte, viole le style de C++, puisque la forme d'écriture voulue est:

```
out << compte; et non pas compte << out;
```

Conclusion: On ne peut pas redéfinir l'opérateur << comme membre de la classe (il en est d'ailleurs de même pour l'opérateur >>).

Pour plus de détails voir : C++ Effective Object-Oriented Software Construction, par K. Dattatri, Prentice Hall, chapitre 7 (page 336 et +).

6. Comment le compilateur fonctionne lors d'une surcharge de l'opérateur << ?

```
cout << compte;
```

Le compilateur procède comme suit:

-1- il cherche d'abord une fonction membre de la classe `ostream` qui peut accepter le type `compte`.

-2- il cherche ensuite:

- une fonction à deux arguments qui accepte l'appel suivant:

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

- sinon, une fonction template (voir le cours suivant) qui va générer une fonction acceptant l'appel en -2-.

-3- s'il ne trouve rien, il signale une erreur de compilation.

7. Limitation de la surcharge des opérateurs

* on ne peut surcharger que les opérateurs qui existent déjà. On ne peut pas en inventer !

L'opérateur `??` n'existe pas en C++, donc impossible de le redéfinir.

* respecter la parité binaire et unaire.

`a+b` (parité binaire) `a++` (parité unaire)

on ne peut pas utiliser l'opérateur `++` pour réaliser l'opération:

`a++b` on doit écrire `(a++)+b` ou `a+(++b)`

* priorité et associativité

`a+b*c`

La multiplication (`*`) est exécutée avant l'addition (`+`) même lorsque les opérateurs sont redéfinis.

* pas de liens sémantiques implicites entre opérateurs redéfinis (parce que c'est vous qui donnez la sémantique)

`a+=b` n'est pas forcément la même opération que `a=a+b`

`a==b` n'implique pas que `a!=b` soit faux !

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 9 : Surcharge (redéfinition) des opérateurs

173

* pas de commutativité automatique

```
double operator+(compte& c, double d) {}  
compte x(...);  
double d=1234.76;
```

```
double z = x+d; // ok les arguments sont: compte, double
```

```
z = d+x; // erreur car operator+ n'a été défini pour supporter cet ordre des arguments: double, compte
```

8. Opérateur ami ou membre de la classe?

Déclarez un opérateur ami, lorsque l'opération est symétrique, exemple:

`a+b` `a-b` `a*b` `a/b` `a==b` etc.

Déclarez un opérateur membre de la classe, lorsque l'opération est asymétrique:

`a+=b` `a-=b` `a*=b` `a/=b` etc.

9. Classe canonique

On appelle une classe canonique toute classe qui contient au moins les éléments suivants:

- 1- au moins un constructeur régulier
- 2- un constructeur de copie
- 3- un destructeur
- 4- un opérateur d'affectation

```
class bidon {  
    // membres privés  
public:  
    bidon(...); // un constructeur régulier  
    bidon(const bidon&); //un constructeur de copie  
    ~bidon(); // un destructeur  
    bidon &operator=(const bidon&); // un opérateur d'affectation  
};
```

Il est conseillé de concevoir des classes canoniques afin de prévoir les cas où la classe dispose de pointeurs sur des parties dynamiques. Nous avons déjà étudié ces cas là dans le chapitre "Construction & destruction des objets", dans ce qui suit, nous allons examiner ces mêmes cas pour l'opérateur d'affectation et l'intérêt d'en définir un dans une classe.

10. Opérateur d'affectation =

10.1. Généralités

Un constructeur de copie est appelé dans 3 situations:

- 1- lors de la création en même temps que l'initialisation de l'objet créé;
- 2- passage de paramètres;
- 3- retour d'un objet comme résultat de fonction.

Chaque classe possède un opérateur d'affectation par défaut. L'affectation est superficielle comme la copie et consiste en une affectation (de surface) membre à membre, ce qui pose les mêmes problèmes que ceux posés par le constructeur de copie par défaut (voir chapitre "Construction & destruction des objets").

Un opérateur d'affectation sert donc à l'affectation dans une expression, et retourne une référence. La signature de cet opérateur est comme suit:

```
x& operator=(x) équivalent d'écrire y=x ou bien y.operator=(x)
```

Le résultat de l'affectation est dans l'objet appelant.

L'opérateur d'affectation doit être une fonction membre.

10.2. Exemple d'une classe tableau

```
#include <iostream>
using namespace std;
class tableau {
    int n; // la taille du tableau tab
    double *tab; // pointeur vers un tableau de double
    void alloc_test() { // test d'allocation mémoire
        if (tab==NULL) {
            cerr << "allocation de la mémoire a échoué !\n";
            exit(1);
        }
    }
    void index_test(int taille_entree){ // test la taille du tableau
        if (taille_entree<1) {
            cerr << "taille du tableau à allouer est inférieure ou égale à zéro! \n";
            exit(1);
        }
    }
    void recopie_elt(const tableau& T) { // recopie élément par élément
        for (int i=0;i<n;i++) tab[i] = T.tab[i];
    }
public:
    tableau(int x) { // constructeur
        index_test(x);
        tab = new double[n=x]; // allocation
        alloc_test();
    }
```

Chapitre 9 : Surcharge (redéfinition) des opérateurs

```
tableau(const tableau& T) { // constructeur de recopie
    tab = new double[n=T.n];
    alloc_test();
    recopie_elt(T);
}
tableaux operator=(const tableaux T) { // opérateur d'affectation
    if (this != &T) { // on teste si on n'affecte pas l'objet à lui-même.
        delete [] tab; // Si ce n'est pas le cas, on détruit l'objet existant.
        tab = new double[n=T.n]; // on alloue de la mémoire à nouveau.
        alloc_test(); // test d'allocation.
        recopie_elt(T); // recopie des éléments de T. tab à tab.
    }
    return (*this); // on retourne l'objet appelant.
}

void init_tab (double nbre){ // initialise tous les éléments de tab à nbre.
    for (int i=0;i<n;i++) tab[i] = nbre;
}
~tableau(){
    delete [] tab; // libération.
}
friend ostream& operator<< (ostream& out,const tableaux T); // sortie ...
};

// affichage vers la sortie de tab.
ostream& operator<< (ostream& out,const tableaux T) {
    for (int i=0;i<T.n;i++) out << "tab[" << i << " ] " << T.tab[i] << endl;
    return out;
}
```

```
int main() {  
    tableau a(3); // on crée un tableau de 3 éléments.  
    a.init_tab(10.5); // on initialise tous ses éléments à 10.5  
  
    cout << "on affiche a ...\n";  
    cout << a; // on affiche le contenu de a.  
    tableau b(2); // on crée un second tableau b de taille 2 (éléments)  
    b = a; // affectation de a vers b. Suite à l'utilisation de operator= la taille de b devient égale à 3.  
    cout << "le tour de b=a ...\n";  
    cout << b; // on affiche b.  
  
    return 0;  
}
```

En sortie:

```
on affiche a ...  
tab[0] 10.5  
tab[1] 10.5  
tab[2] 10.5  
le tour de b=a ...  
tab[0] 10.5  
tab[1] 10.5  
tab[2] 10.5
```

11. Opérateur d'affectation [] (indexation)

11.1. Généralités

Si a est un objet alors :

a[expr] est équivalente à : a.operator[](expr)

L'opérateur [] ne peut être redéfini qu'avec une fonction membre.

11.2. Exemple d'une classe tableau

Pour la classe tableau précédente (10.2), nous aurons:

```
class tableau {  
    // etc.  
public:  
    // etc.  
    double& operator[](int i){  
        if ((i>=0) && (i<n)) return tab[i];  
        else {  
            cerr << "problème avec l'index!\n";  
            exit(1);  
        }  
    }  
};
```

```
int main() {  
    tableau a(3);  
    a.init_tab(10.5);  
  
    // Accès direct aux membres données private de la classe tableau grâce à l'opérateur d'indexation [], on  
    // peut accéder directement aux éléments du tableau tab de l'objet a, dans ce cas, on change la valeur  
    // de a[0].  
  
    // C'est l'équivalent aussi de a.operator[](0) ou bien a.tab[0].  
  
    a[0] = 38.5;  
  
    cout << "on affiche a ...\n";  
    cout << a;  
    tableau b(2);  
    b = a;  
  
    // Même remarque que pour a[0].  
  
    b[1] = 44.44;  
  
    cout << "le tour de b=a ...\n";  
    cout << b;  
  
    return 0;  
}
```

11.3. Bug généré par l'opérateur []

Prenons l'exemple suivant:

```
const tableau s(10); // un tableau de 10 éléments.
```

Nous avons montré dans le chapitre 6 "Les Propriétés des fonctions membres", que les objets constants doivent être manipulés par des fonctions membres constantes, de ce fait:

```
class tableau {  
    // etc.  
public:  
    double& operator[](int i) const;  
    // etc.  
};  
  
double& tableau::operator[](int i) const {  
    if ((i>=0) && (i<n)) return tab[i];  
    else {  
        cerr << "problème avec l'index!\n";  
        exit(1);  
    }  
}
```

On constate que pour une instruction de la sorte ...

```
s[5] = 1.234;
```

Même si le tableau s est constant, il y aura quand même affectation dans s[5] de la valeur 1.234!

Pour remédier à ce problème nous devons redéfinir **deux** opérateurs d'indexation:

```
class tableau {
    // etc.
public:
    double operator[](int i) const; // constant
    double& operator[](int i); // non constant
    // etc.
};

double& tableau::operator[](int i) {
    if ((i>=0) && (i<n)) return tab[i];
    else {
        cerr << "problème avec l'index!\n";
        exit(1);
    }
}

double tableau::operator[](int i) const {
    if ((i>=0) && (i<n)) return tab[i];
    else {
        cerr << "problème avec l'index!\n";
        exit(1);
    }
}
```

```
int main() {
    // un tableau quelconque
    tableau a(3);

    // appel de: double& operator[](int i);
    a[0] = 23.5;

    // un tableau constant
    const tableau s(10);

    // Appel de: double operator[](int i) const; retourne uniquement une valeur et ne touche pas donc
    // au contenu de ce fait, nous allons obtenir, si dans s[3] il y avait la valeur 210.23 : 210.23 = 2345.89
    // erreur de compilation, car il ne peut pas y avoir ce type d'affectation.
    s[3] = 2345.89;

    return 0;
}
```

12. Opérateur () (appel de fonction)

12.1. Généralités

- doit être une fonction membre,
- peut prendre autant d'arguments qu'on veut du type qu'on veut,
- peut être redéfini plusieurs fois,
- peut prendre des arguments par défaut (ce qui n'est pas le cas des autres opérateurs).

12.2. Exemple d'une classe tableau

Pour la classe tableau définie en 10.2,

```
class tableau {
// etc.
public:
    double operator()(int i, char* f) {
        // fait quelque chose ...
    }
    // redéfini ...
    int operator()(double j, int k) {
        // fait quelque chose ...
    }
    // etc.
};

int main() {
    tableau t(3);
    double x = t(3, "Fred"); // x=t.operator()(3, "Fred");
    return 0;
}
```

13. Opérateurs ++ et --

Nous allons décrire le cas de l'opérateur ++, les mêmes remarques s'appliquent pour l'opérateur --.

Il y a deux formes; préfixe et suffixe.

Préfixe → ++a : incrémente a et retourne le **nouveau** a
Suffixe → a++ : incrémente a et retourne l'**ancien** a

	opérateur++ préfixe	opérateur++ suffixe
global	type opérateur++(type)	type opérateur++(type, int)
classe	type opertaor++()	type opérateur++(int)

a++ signifie: opérateur++(a, 0) ou bien a.opérateur++(0) (idem que pour a--)

```
#include <iostream>

using namespace std;

class compte {
    float solde;
public:
    compte(float m):solde(m) {}
    compte opérateur++();
    compte opérateur++(int);
    void afficher();
};
```

```
    compte compte::operator++() {
        solde++;
        return (*this);
    }
    compte compte::operator++(int n) {
        compte temp = *this;
        solde++;
        return temp;
    }
    void compte::afficher() {
        cout << "solde: " << solde << endl;
    }

    int main() {
        compte C1(100);
        compte C2(200);
        compte C3(300);

        C3 = C1++; // <-- postfixe

        C1.afficher(); // solde: 101
        C3.afficher(); // solde: 100

        C3 = ++C2; // <-- préfixe

        C2.afficher(); // solde: 201
        C3.afficher(); // solde: 201
        return 0;
    }
}
```