Programmation en C++

83

### Surcharges d'opérateurs

- Objectif : permettre l'utilisation plus intuitive d'une classe à partir d'opérateurs connus
- Exemple (Vector3)

cette expression est bien plus compacte et plus claire que la séquence d'instructions précédente

```
Vector3 v, v1(1.1, 2.2, 3.3), v2(0.3, 0.4, 0.5);
Matrix3 mat;

// sans les opérateurs
v2.mult(2.0);
v1.add(v2);
mat.rotate(0z, M_PI/2.0);
v = mat.mult(v1);

// avec les opérateurs
mat.rotate(0z, M_PI/2.0);
v = mat * (v1 + 2.0 * v2);
```





Surcharge d'opérateurs

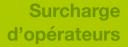
84

# Surcharge des opérateurs > Considérations générales

- Tous les opérateurs du C++ peuvent être surchargés sauf :
  - ::,.,\*,?:,sizeof,typeid, static\_cast, dynamic\_cast, const cast,reinterpret cast
- Il est impossible de changer la priorité, l'associativité et l'arité des opérateurs (ex : ++ unaire, + binaire, ?: ternaire et () n-aire)
- Il est impossible de créer de nouveaux opérateurs
- Il est préférable de respecter la sémantique habituelle de l'opérateur pour que son utilisation reste intuitive







# Surcharge des opérateurs > 2 approches différentes

### Surcharge des opérateurs interne

- on considère ces opérateurs comme des méthodes normales de la classe (déclarés à l'intérieur de la classe)
- syntaxe: Type operatorOp(paramètres);
- alors: a Op b <=> a.operatorOp(b)

### Surcharge des opérateurs externe

- les opérateurs sont en fait des fonctions définies en dehors (mais souvent amies) de la classe concernée
- syntaxe: friend Type operatorOp(paramètres);
- alors: a Op b <=> operatorOp(a,b)





### Surcharge des opérateurs interne > Affectations et arithmétiques

- Opérateurs : =, +=, ++, -=, /=, \*=, +, -, \*, /
- Avec cette syntaxe le premier opérande est l'objet sur lequel s'applique l'opérateur
  - les opérantes suivantes sont les paramètres de la fonction opérateur
  - ex:v1 += v2 <=> v1.operator+=(v2) //v1 modifié
  - ex: v = v1 \* v2 <=> v.operator=(v1.operator\*(v2))
- Les opérateurs =, +=, ++, -=, /= et \*= doivent retourner par référence l'objet sur lequel il travaillent par : return \*this;
- Les opérateurs +, -, \* et /, doivent retourner une copie du résultat calculé



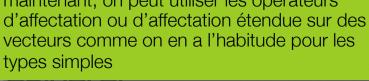


```
// fichier vector3.h
class Vector3 {
   double _coeff[3];
public:
   Vector3& operator= (const Vector3& v);
   Vector3& add(const Vector3& v):
   Vector3& operator+= (const Vector3& v);
   Vector3& operator+= (const double s); ✓
};
// fichier main.cc
                                             les opérateurs d'affectation ou d'affectation
Vector3 v1, v2, v3;
                                             étendue retournent une référence sur
                                             l'objet : on donc peut les chaîner!
v1 = v2 = (v3 += 2.0);
v1 *= 5.0;
               maintenant, on peut utiliser les opérateurs
```

Surcharge d'opérateurs > Exemple

on peut même surcharger avec un autre type pour des opérations hétérogènes (a-extentions...)





```
// fichier vector3.cc
  Vector3& Vector3::operator= (const Vector3& v)
                                                     on ne réalise l'affectation que si la
                                                     source est une instance différente
    if (&v != this) {
        _coeff[0] = v._coeff[0]; _coeff[1] = v._coeff[1]; ...
    return *this:
  Vector3& Vector3::add(const Vector3& v)
     coeff[0] += v. coeff[0]; coeff[1] += v. coeff[1]; ...
     return *this:
  Vector3& Vector3::operator+= (const Vector3& v)
     return this->add(v);
  Vector3& Vector3::operator+= (const double s)
     _coeff[0] += s; _coeff[1] += s; _coeff[2] += s;
     return *this:
```



il ne faut pas oublier de retourner une référence sur l'objet courant pour assurer l'associativité de l'opérateur







```
class Car {
protected:
    char *_name;
public:
    Car(const char* desc);
    Car(const Car& exp);
    Car& operator=(const Car& c);

    const char* getName() const;
};

    on ne traite l'affectation que si les
        deux objets ne sont pas un seul
```

## Surcharge des opérateurs internes > Remarques sur l'opérateur =

- Deux problèmes à traiter
  - cas de l'auto-affectation : obj = obj
  - l'affectation de deux objets quand ils contiennent des pointeurs peut nécessiter une allocation mémoire

```
Car& Car::operator= (const Car& c) {
   if (&c != this) {
     if (_name != NULL) free(_name);
        _name = strdup(c._name);
   }
   return *this;
}
```

comme dans le cas du constructeur de copie, on doit recréer une copie de la chaîne de caractères \_name





et même objet : gain de temps!



## Surcharge des opérateurs internes > Opérateurs ++ et --

- Opérateurs doubles : préfixes (++i, --j) ou suffixes (i++, j--)
  - pas de paramètre s'ils sont préfixes, un paramètre fictif s'ils sont suffixes

```
class Vector3 {
protected:
    double _coeff[3];
public:
    Vector3(double x, double y, double z, double w=1.0);

    Vector3& operator++() {
        ++_coeff[0]; ++_coeff[1]; ++_coeff[2];
        return *this;
    }

    Vector3 operator++(int i) {
        return Vector3(_coeff[0]++, _coeff[1]++, _coeff[2]++);
    }
};
```

v2 = ++v; v2.print(); v2 = v++;

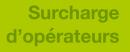
Vector3 v(1.0), v2;



v2.print();



on retourne une copie : pouvez-vous dire pourquoi ?



# Surcharge des opérateurs internes > Opérateur [ ]

Deux formes (une const, l'autre non)

```
class Vector3 {
   double _coeff[3];

public:

   double& operator[](const int i) { return _coeff[i]; } // (1)
   double operator[](const int i) const { return _coeff[i]; } // (2)
};

void printVector(const Vector3& v) {
   cout << "["<<v[0]<<","<<v[1]<<","<<v[2]<<"]" << endl; // appels de (2)
}

void initVector(Vector3& v) {
   v[0] = v[1] = v[2] = 0; // appels de (1)
}

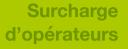
là, on modifie le contenu de v, c'est donc l'opérateur non</pre>
```

ici, on ne peut pas modifier le contenu de v, c'est donc l'opérateur const (2) qui est utilisé





const (1) qui est utilisé



### Surcharge des opérateurs internes > Opérateur ()

Deux formes (une const, l'autre non)

```
class Matrix3 {
  Vector3 line[3];
public:
  Vector3& operator[](const int i) { return _line[i]; }
  double& operator()(const int i, const int j) { return line[i][j]; }
  double operator()(const int i, const int j) const { return line[i][j]; }//(2)
};
int main() {
  Matrix4 m;
  m(2,2) = 1.0; // appel de (1) car on modifie l'état de la matrice
  cout << m(1,0) << endl; // appel de (2)
  m[1][2] = 0.0; // quel(s) opérateur(s) ???
```





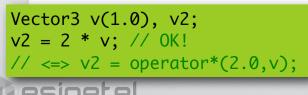
//(1)

## Surcharge des opérateurs externes > Opérateurs arithmétiques

- Problème : la surcharge d'opérateurs en interne impose que l'objet concerné soit la première opérande
  - v\*2.0 (avec v de type Vector3) est possible mais pas 2.0\*v
  - mais on peut utiliser une fonction operator amie (friend)

```
// fichier Vector3.h
class Vector3{
public:
    friend Vector3 operator*(const double s, const Vector3& v);
};

//fichier Vector3.cc
Vector3 operator*(const double s, const Vector3& v) {
    return Vector3(s*v._coeff[0], s*v._coeff[1], s*v._coeff[2]);
}
```



### Surcharge des opérateurs d'E/S > Opérateurs << et >>

- On a vu qu'on peut utiliser l'opérateur << pour écrire sur un flot de sortie standard cout (cout << x;) et l'opérateur >> pour lire depuis un flot d'entrée standard cin (cin >> x;)
- Ces opérateurs assurent le transfert d'information ainsi que son éventuel formatage
- Les flots prédéfinis cout et cin sont en fait des objets (instances) de la classe ostream et istream respectivement
- Ces deux classes surchargent les opérateurs << et >> pour les principaux types de base (int, float, double etc.)
- Ces opérateurs retournent une référence au flot d'E/S, ce qui permet de l'appliquer plusieurs fois de suite : cout << "a=" << x;





Surcharge d'opérateurs

95

### Surcharge des opérateurs d'E/S > Opérateurs << et >>

- On peut surcharger ces opérateurs pour n'importe quelle classe Classe créée par l'utilisateur à condition de noter que :
  - le premier argument de ces opérateurs est un flot (istream ou ostream), il n'est donc pas possible d'en faire une fonction membre de la classe utilisateur. On pourra en revanche utiliser des fonctions indépendantes (le plus souvent amies)
  - la valeur de retour doit obligatoirement être une référence au flot concerné

```
[friend] ostream& operator<< (ostream& os, const Classe& obj);
[friend] istream& operator>> (istream& is, Classe& obj);
```





```
Surcharge des
//fichier car.h
                                     I il faut inclure les définitions des types
                                                                                           opérateurs E/S
                                      ostream et istream
#include <iostream> ←
                                                                                            > Exemple
class Car {
                                                                        les opérateurs d'E/S sont
public:
                                                                         déclarés comme fonctions
    //méthodes publiques
                                                                         amies et définis dans le
    Car(const string& name="Voiture");
                                                                        fichier source de la classe
    friend ostream& operator<<(ostream& os, const Car& aCar);</pre>
     friend istream& operator>>(istream& is, Car& aCar);
                                                                       lici exp est 'const' car il ne sera pas
};
                                                                       modifié par l'opérateur de sortie
//fichier car.cc
ostream& operator<<(ostream& os, const Car& aCar) {</pre>
     os << aCar.getName();</pre>
                                                                  l ici exp n'est pas 'con<mark>st' car il sera</mark>
     return os;
                                                                  modifié par l'opérateur d'entrée
}
istream& operator>>(istream& is, Car& aCar) {
     is >> aCar. name;
     return is:
```







une **Car** est désormais transmissible à un flux d'E/S (ostream ou istream) comme tout autre type standard (int, float, string...)



```
> Twingo

< Clio

> aCar = Clio

> *aCarPtr = Picasso
```

97

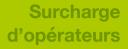


}

return 0:



cout << "\*aCarPtr = " << \*aCarPtr;</pre>

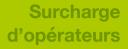


## Surcharge d'opérateurs > Opérateurs de conversion

- Conversions explicites (opérateur cast) comme en C
  - ex:int n; double z; ... z=double(n); ... n=int(z)
- Conversions implicites mises en place par le compilateur selon le contexte, à plusieurs niveaux :
  - dans les affectations : conversion forcée dans le type de la variable réceptrice
  - dans les paramètres de fonction : conversion forcée dans le type du paramètre déclaré dans le prototype
  - dans les expressions : pour chaque opérateur, il existe des règles de conversion précises entre les différents arguments







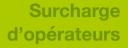


# Surcharge d'opérateurs > Opérateurs de conversion

- On peut définir des conversions pour des classes créées par l'utilisateur
- Deux sortes de conversions "utilisateur" en C++ :
  - la conversion d'un type prédéfini vers le type classe en question. On va créer pour cela un **constructeur de conversion**
  - la conversion du type classe vers un type prédéfini (ou défini préalablement). Ce type de conversion sera effectuée par des **fonctions de conversion**







### Opérateurs de conversion > Constructeur de conversion

Défini dans une classe A, un constructeur avec un seul argument de type T permet de réaliser une conversion d'une variable de type T vers un objet de la classe A

```
class Vector3 {
   double _coeff[3];

public:
    Vector3(double s=0.0);
...
};

void f(Vector3 v) { ... }

int main() {
   Vector3 v2, v1 = 0.5;
    v2 = 2.7;
   f(2.7);
} // OK ! <=> Vector3(2.7);
} // OK ! <=> f(Vector3(2.7));
}
```







### Opérateurs de conversion > Fonction de conversion

- Une fonction de conversion est une méthode qui effectue une conversion vers un type T
- Elle est nommée operator T() et n'a pas de type de retour
- ե Elle doit (quand même !) retourner une valeur du type т





```
#include <iostream>
using namespace std;
class Vector3 {
  double coeff[3];
public:
  Vector3(double s=0.0) {
      \_coeff[0] = \_coeff[1] = \_coeff[2] = 0.0;
  Vector3(double x, double y, double z) {
     coeff[0] = x; coeff[1] = y; coeff[2] = z;
  operator double() { //opérateur de conversion vers un double
      cout << "Cast double() pour un vecteur" << endl;</pre>
     return coeff[0]; // retourne le premier coefficient
};
void f(double d) { cout << "appel de fonction avec arg = " << d << endl; }</pre>
int main() {
  Vector3 v(2.1,3.7,1.8);
                                     ici, un transtypage du vecteur v vers le
                                      type double est réalisé implicitement par
  f(7.2);
                                     le compilateur grâce à la méthode operator
  f(v);
  return 0;
                                     double() de la classe Vector3
```



en sortie

> appel de fonction avec arg = 7.2

> Cast double() pour un vecteur

> appel de fonction avec arg = 2.1



