

Cours C++

Surcharge d'opérateur

Surcharge d'opérateur

- ▶ On voudrait pouvoir faire

```
Vecteur2D vec1 { .x=2, .y=3 };  
Vecteur2D vec2 { .x=4, .y=5 };  
vec1 += vec2;                                // addition de vec2 à vec1  
Vecteur2D vec3 = (vec1 - vec2) * 0.7;        // différence puis mult. scalaire  
...
```

- ▶ Fondamentalement, l'appel à un opérateur tel que +, =, +=, ++, <<, [], (), !... est identique à l'appel d'une fonction/méthode
- ▶ Par défaut, ces opérations ne sont pas définies sur les classes que l'on crée
- ▶ Mais possibilité de (sur)définir ces opérateurs comme n'importe quelle autre méthode

Surcharge d'opérateur

- ▶ On voudrait pouvoir faire

```
Vecteur2D vec1 { .x=2, .y=3 };  
Vecteur2D vec2 { .x=4, .y=5 };  
vec1 += vec2;                                // addition de vec2 à vec1  
Vecteur2D vec3 = (vec1 - vec2) * 0.7;        // différence puis mult. scalaire  
...
```

- ▶ Fondamentalement, l'appel à un opérateur tel que +, =, +=, ++, <<, [], (), !... est identique à l'appel d'une fonction/méthode
- ▶ Par défaut, ces opérations ne sont pas définies sur les classes que l'on crée
- ▶ Mais possibilité de (sur)définir ces opérateurs comme n'importe quelle autre méthode

Surcharge d'opérateur

- ▶ On voudrait pouvoir faire

```
Vecteur2D vec1 { .x=2, .y=3 };  
Vecteur2D vec2 { .x=4, .y=5 };  
vec1 += vec2;                                // addition de vec2 à vec1  
Vecteur2D vec3 = (vec1 - vec2) * 0.7;        // différence puis mult. scalaire  
...
```

- ▶ Fondamentalement, l'appel à un opérateur tel que +, =, +=, ++, <<, [], (), !... est identique à l'appel d'une fonction/méthode
- ▶ Par défaut, ces opérations ne sont pas définies sur les classes que l'on crée
- ▶ Mais possibilité de (sur)définir ces opérateurs comme n'importe quelle autre méthode

Surcharge d'opérateur

- ▶ On voudrait pouvoir faire

```
Vecteur2D vec1 { .x=2, .y=3 };  
Vecteur2D vec2 { .x=4, .y=5 };  
vec1 += vec2;                                // addition de vec2 à vec1  
Vecteur2D vec3 = (vec1 - vec2) * 0.7;        // différence puis mult. scalaire  
...
```

- ▶ Fondamentalement, l'appel à un opérateur tel que +, =, +=, ++, <<, [], (), !... est identique à l'appel d'une fonction/méthode
- ▶ Par défaut, ces opérations ne sont pas définies sur les classes que l'on crée
- ▶ Mais possibilité de (sur)définir ces opérateurs comme n'importe quelle autre méthode

Exemple de surcharge de l'opérateur unaire +=

```
// Déclaration
```

```
class Vecteur2D {  
public:  
    double x;  
    double y;  
};
```

```
// Définition
```

```
void Vecteur2D::operator+= (const Vecteur2D& right) {  
    x += right.x;  
    y += right.y;  
}
```

Exemple de surcharge de l'opérateur unaire +=

```
// Déclaration
class Vecteur2D {
public:
    double x;
    double y;

    void operator+= (const Vecteur2D& right);
};

// Définition
void Vecteur2D::operator+= (const Vecteur2D& right) {
    x += right.x;
    y += right.y;
}
```

Exemple de surcharge de l'opérateur unaire +=

```
// Définition
void Vecteur2D::operator+= (const Vecteur2D& right) {
    x += right.x;
    y += right.y;
}

// Utilisation
void main () {
    Vecteur2D vec1 {2, 3};
    Vecteur2D vec2 {4, 5};
    vec1 += vec2;
}
```

→ équivalent à l'usage d'une méthode ajoute s'utilisant de la façon suivante `vec1.ajoute(vec2);`

Opérateurs unaires

Pratique usuelle avec les opérations unaires : renvoyer le résultat de l'opération.

```
// Déclaration
class Vecteur2D {
    ...
    Vecteur2D& operator+= (const Vecteur2D& right);
};

// Définition
Vecteur2D& Vecteur2D::operator+= (const Vecteur2D& right) {
    x += right.x;
    y += right.y;
    return *this;
}
```

→ utilisation du pointeur `this` qui retourne l'adresse de l'objet courant

Surcharge de l'opérateur binaire + : Méthode constante

Objectif :

```
Vecteur2D vec1 {2, 3};  
Vecteur2D vec2 {4, 5};  
Vecteur2D vec3 = vec1 + vec2;
```

Implémentation :

```
// Déclaration  
class Vecteur2D {  
    ...  
    Vecteur2D operator+ (const Vecteur2D& right) const;  
};  
  
// Définition  
Vecteur2D Vecteur2D::operator+ (const Vecteur2D& right) const {  
    return Vecteur2D {  
        x + right.x,  
        y + right.y  
    };  
}
```

Surcharge de l'opérateur binaire + : Fonction globale

Objectif :

```
Vecteur2D vec1 {2, 3};  
Vecteur2D vec2 {4, 5};  
Vecteur2D vec3 = vec1 + vec2;
```

Implémentation :

```
// Déclaration en dehors de la classe  
Vecteur2D operator+ (const Vecteur2D& left, const Vecteur2D& right);  
  
// Définition  
Vecteur2D operator+ (const Vecteur2D& left, const Vecteur2D& right) {  
    return Vecteur2D {  
        left.x + right.x,  
        left.y + right.y  
    };  
}
```

Accès d'un élément de matrice

Imaginons une classe `Matrice` qui stocke une matrice sous forme de tableau de **double** linéaire :

$$(x_{ij})_{i,j} = \begin{pmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \\ x_{20} & x_{21} \end{pmatrix}$$



$$[x_k]_k = [x_{00}, x_{01}, x_{10}, x_{11}, x_{20}, x_{21}]$$

$$k = i + n j$$

(cf. suite du cours)

Accès d'un élément de matrice

Une méthode accès() permet de récupérer l'élément (row, col) :

```
class Matrice {
    double* data;
    unsigned int cols, rows;

public:
    ...
    double accès (int row, int col); // Accès à un élément de la matrice
};

double Matrice::accès (int row, int col) {
    if (row < 0 or row >= rows or col < 0 or col >= cols)
        throw std::out_of_range();
    return data[ row*cols + col ];
}

cout << mat1.accès(1, 2) << endl;
```

Accès d'un élément de matrice

On peut la remplacer par un opérateur plus pratique :

```
class Matrice {
    double* data;
    unsigned int cols, rows;

public:
    ...
    double operator[] (int row, int col); // Accès à un élément de la matrice
};

double Matrice::operator[] (int row, int col) {
    if (row < 0 or row >= rows or col < 0 or col >= cols)
        throw std::out_of_range();
    return data[ row*cols + col ];
}

cout << mat1[1,2] << endl;
```