Cours 03 - Surcharge d’opérateurs

Constructeurs avancés et opérateurs

La dernière fois…

Généralités sur le C++.

Ce que nous avons vu :

- Classes et encapsulation
- Pointeurs et références
- Constructeurs basiques
- Destructeurs

… aujourd’hui

Programme de la séance

- Constructeur(s) avancés
- Surcharge d’opérateur(s)
- Règle de 3 et de 5
- Héritage (introduction)

Constructeur/Destructeur

Constructeur - Rappel

Initialisation des objets

Il est fortement conseillé d'initialiser un objet à l'aide d'un constructeur.

```
class Point {
private:
    double x, y, z;

public:
    // Constructeur
    Point(double x, double y, double z)
        : x(x), y(y), z(z) {
        std::cout << "Point créé" << std::endl;
    }
};
```

Caractéristiques

- Généralement déclaré `public`
- Initialise les membres
- Peut effectuer des validations

Constructeur - Exemple

```
class Vecteur {
private:
    double* coord;
    int dimension;

public:
    // Constructeur avec allocation dynamique
    Vecteur(int dim) : dimension(dim) {
        coord = new double[dimension];
        for(int i = 0; i < dimension; i++) {
            coord[i] = 0.0;
        }
    }

    // Destructeur nécessaire !
    ~Vecteur() {
        delete[] coord;
    }

    void afficher() const {
        for(int i = 0; i < dimension; i++) {
            std::cout << coord[i] << " ";
        }
        std::cout << std::endl;
```

Surcharge de constructeur

Plusieurs constructeurs pour différents usages

```
class Rectangle {
private:
    double largeur, hauteur;

public:
    // Constructeur par défaut
    Rectangle() : largeur(1.0), hauteur(1.0) {}

    // Constructeur carré
    Rectangle(double cote) : largeur(cote), hauteur(cote) {}

    // Constructeur rectangle
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}

    double aire() const { return largeur * hauteur; }
};

int main() {
    Rectangle r1;           // 1×1
    Rectangle r2(5.0);      // 5×5
    Rectangle r3(3.0, 4.0); // 3×4
}
```

Constructeur par copie

Copie profonde vs copie superficielle

Par défaut, un constructeur par copie est créé automatiquement, mais il se contente de copier les membres bit à bit.

Problème : copie superficielle

```
class Tableau {
    int* data;
    int taille;
public:
    Tableau(int n) : taille(n) {
        data = new int[n];
    }
};

Tableau t1(10);
Tableau t2 = t1;  // Copie superficielle
// Les deux partagent le même pointeur !
```

Solution : copie profonde

```
class Tableau {
    int* data;
    int taille;
public:
    // Constructeur par copie
    Tableau(const Tableau& autre)
        : taille(autre.taille) {
        data = new int[taille];
        for(int i = 0; i < taille; i++) {
            data[i] = autre.data[i];
        }
    }
};
```

Constructeur par copie - Quand est-il appelé ?

Invocation implicite

Le constructeur par copie est invoqué implicitement dans plusieurs situations :

```
class Point {
public:
    Point(const Point& autre) {
        std::cout << "Copie !" << std::endl;
    }
};

Point p1(10, 20);

// 1. Initialisation par copie
Point p2 = p1;        // Appelle le constructeur par copie

// 2. Passage par valeur
void fonction(Point p) { /* ... */ }
fonction(p1);         // Appelle le constructeur par copie

// 3. Retour par valeur
Point creer() {
    Point p(5, 5);
```

Appel de constructeur - Différentes formes

Formes d'initialisation en C++

```cpp
class Point {
    int x, y;
public:
    Point() : x(0), y(0) {}
    Point(int x, int y) : x(x), y(y) {}
};

int main() {
    // Différentes syntaxes d'initialisation
    Point p1;              // Défaut
    Point p2(10, 20);      // Directe
    Point p3 = Point(5,5); // Copie (souvent élidée)
    Point p4{15, 25};      // Liste (C++11) - PRÉFÉREZ CELLE-CI
    Point p5 = {1, 2};     // Liste avec =

    // Allocation dynamique
    Point* ptr = new Point(1, 2);
    delete ptr;
}
```

💡 **C++11 et plus :** Préférez l'initialisation par liste `{}` car elle évite les conversions implicites dangereuses.

Destructeur - Rappel

Libération des ressources

```
class GestionnaireMemoire {
private:
    int* buffer;
    size_t taille;

public:
    // Constructeur : allocation
    GestionnaireMemoire(size_t n) : taille(n) {
        buffer = new int[taille];
        std::cout << "Allocation de " << taille << " entiers" << std::endl;
    }

    // Destructeur : libération
    ~GestionnaireMemoire() {
        delete[] buffer;
        std::cout << "Libération de " << taille << " entiers" << std::endl;
    }
};

int main() {
    GestionnaireMemoire gm(1000);
```

Surcharge d’opérateur

Syntaxe de la surcharge

Principe général

On utilise le mot réservé `operator` avec la syntaxe suivante :

```
typeRetour operator symbole(arguments)
```

Règles

- Nombre d'arguments lié au type d'opérateur
  - Unaire : 0 ou 1 argument
  - Binaire : 1 ou 2 arguments
- Type de retour selon les objectifs

Exemple simple

```
class Compteur {
    int valeur;
public:
    // Opérateur ++
    Compteur& operator++() {
        valeur++;
        return *this;
    }
};
```

Exemple de surcharge

Surcharge d'opérateurs arithmétiques

```
class Point {
    double x, y;

public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}

    // Opérateur d'addition
    Point operator+(const Point& autre) const {
        return Point(x + autre.x, y + autre.y);
    }

    // Opérateur de multiplication par un scalaire
    Point operator*(double scalaire) const {
        return Point(x * scalaire, y * scalaire);
    }

    void afficher() const {
        std::cout << "(" << x << ", " << y << ")" << std::endl;
    }
};

int main() {
    Point p1(1, 2), p2(3, 4);
```

Opérateur d'affectation (=) - Étape 0

L'opérateur doit copier les membres

```
class Point {
    double x, y;

public:
    // Version de base (INCOMPLÈTE)
    void operator=(Point source) {
        x = source.x;
        y = source.y;
    }
};

int main() {
    Point p1(1, 2);
    Point p2(3, 4);
    p1 = p2;  // p1 devient (3, 4)
}
```

⚠️ Cette version a plusieurs problèmes que nous allons corriger étape par étape.

Opérateur d'affectation (=) - Étape 1

La source doit rester inchangée

```
class Point {
    double x, y;

public:
    // Passage par valeur → copie (inefficace)
    void operator=(Point source) {
        x = source.x;
        y = source.y;
        // source peut être modifiée localement
        // mais l'original n'est pas affecté
    }
};
```

Problème : On fait une copie complète de `source` à chaque affectation !

Opérateur d'affectation (=) - Étape 2

Passage par référence constante

```
class Point {
    double x, y;

public:
    // Passage par référence constante (efficace)
    void operator=(const Point& source) {
        x = source.x;
        y = source.y;
    }
};
```

Avantages :

- Pas de copie de l'objet source
- `const` garantit que source n'est pas modifiée
- Beaucoup plus efficace pour les gros objets

Opérateur d'affectation (=) - Étape 3

Type de retour approprié

```
class Point {
    double x, y;

public:
    // Retourne une référence sur Point
    Point& operator=(const Point& source) {
        x = source.x;
        y = source.y;
        return *this;  // Retourne l'objet lui-même
    }
};
```

Pourquoi retourner une référence ?

Permet le chaînage d'affectations :

```
Point p1, p2, p3;
p1 = p2 = p3;  // Équivalent à : p1 = (p2 = p3)
```

Opérateur d'affectation (=) - Version finale

Version complète et robuste

```
class Tableau {
    int* data;
    int taille;

public:
    Tableau& operator=(const Tableau& source) {
        // 1. Vérifier l'auto-affectation
        if (this == &source) {
            return *this;
        }

        // 2. Libérer l'ancienne mémoire
        delete[] data;

        // 3. Allouer et copier
        taille = source.taille;
        data = new int[taille];
        for(int i = 0; i < taille; i++) {
            data[i] = source.data[i];
        }

        return *this;
```

Surcharge externe

Opérateurs qui nécessitent une surcharge externe

Problème : `3 * Point`

```
class Point {
public:
    // Fonctionne : Point * 3
    Point operator*(double s) const {
        return Point(x*s, y*s);
    }
};

Point p(1, 2);
p * 3;    // OK
3 * p;    // ERREUR !
```

Solution : fonction externe

```
// Fonction externe (friend)
Point operator*(double s,
                const Point& p) {
    return Point(p.x*s, p.y*s);
}

// OU sans friend avec accesseurs
Point operator*(double s,
                const Point& p) {
    return p * s;  // Réutilise l'autre
}

Point p(1, 2);
3 * p;    // OK maintenant !
```

Surcharge externe (2)

Deux versions complémentaires

```
class Point {
    double x, y;

public:
    // Version membre : Point * double
    Point operator*(double scalaire) const {
        return Point(x * scalaire, y * scalaire);
    }

    // Version externe : double * Point
    friend Point operator*(double scalaire, const Point& p) {
        return Point(p.x * scalaire, p.y * scalaire);
    }
};

int main() {
    Point p(2, 3);
    Point p1 = p * 5;    // Appelle la version membre
    Point p2 = 3 * p;    // Appelle la version friend

    // Conversion automatique int → double
```

Choisir le type de surcharge

Interne vs Externe

Surcharge interne (membre)

Quand utiliser :

- L'opération modifie l'objet (`+=` , `*=` , etc.)
- Accès aux membres privés nécessaire
- Opérateur d'affectation

```
class Point {
    Point& operator+=(const Point& p) {
        x += p.x;
        y += p.y;
        return *this;
    }
};
```

Surcharge externe (friend)

Quand utiliser :

- Symétrie nécessaire (`a * b` et `b * a`)
- Premier argument n'est pas de la classe
- Meilleure visibilité

```
friend Point operator+(
    const Point& a,
    const Point& b) {
    return Point(a.x+b.x, a.y+b.y);
}
```

Surcharge fortement recommandée

Opérateurs composés (`+=` , `*=` , etc.)

```
class Point {
    double x, y;

public:
    // Opérateur += (INTERNE)
    Point& operator+=(const Point& p) {
        x += p.x;
        y += p.y;
        return *this;  // Retourne référence (évite copie)
    }

    // Opérateur + (EXTERNE) - utilise +=
    friend Point operator+(Point a, const Point& b) {
        a += b;  // Réutilise +=
        return a;
    }
};
```

Avantages :

- Une seule implémentation de la logique

Liste des opérateurs surchargeables

Opérateurs arithmétiques

```
+  -  *  /  %  ^  &  |  ~
+= -= *= /= %= ^= &= |=
++ --
```

Opérateurs de comparaison

```
== != < > <= >=
```

Opérateurs logiques

```
&& || !
```

Opérateurs d'accès

```
[] () -> ->*
```

Opérateurs de flux

```
<< >>
<<= >>=
```

Autres

```
= , (virgule)
new new[] delete delete[]
```

❌ **Ne peuvent PAS être surchargés :** `::` (résolution de portée), `.` (sélection de membre), `.*` (sélection via pointeur de fonction)

Surcharge rationnelle

Bonnes pratiques

❌ Mauvais exemple

```
class Point {
public:
    // Division de points ?!
    Point operator/(const Point& p) {
        return Point(x/p.x, y/p.y);
    }
};
```

Utiliser le sens usuel des opérateurs !

✅ Bon exemple

```
class Vecteur {
public:
    // () : accès mathématique (base 1)
    double& operator()(int i) {
        return data[i-1];
    }

    // [] : accès C++ (base 0)
    double& operator[](int i) {
        return data[i];
    }
};
```

💡 Retourner une référence permet la modification : `v[0] = 42;`

Opérateur de conversion

Transtypage personnalisé

```
class Vecteur3D {
    double x, y, z;

public:
    // Opérateur de conversion vers Point2D
    operator Point2D() const {
        return Point2D(x, y);  // Projection sur le plan xy
    }
};

// Utilisation
Vecteur3D v(1, 2, 3);
Point2D p = v;  // Conversion implicite !
```

⚠️ **Attention :** L'opérateur de conversion est utilisé implicitement par le compilateur. Cela peut créer des ambiguïtés !

Recommandation : Préférez un constructeur explicite dans la classe cible.

Opérateur de conversion - Alternative recommandée

Constructeur explicite (préférable)

Opérateur de conversion (implicite)

```
class Vecteur {
public:
    operator Point() const {
        return Point(x, y);
    }
};

Vecteur v(1, 2, 3);
Point p = v;  // Implicite
```

Constructeur (explicite)

```
class Point {
public:
    explicit Point(const Vecteur& v)
        : x(v.x), y(v.y) {}
};

Vecteur v(1, 2, 3);
Point p(v);   // Explicite
// Point p = v;  // ERREUR si explicit
```

💡 **Avantages du constructeur explicite :**

- Conversion contrôlée et intentionnelle
- Évite les conversions accidentelles
- Plus simple et plus sûr

Opérateurs de flux

Lecture et écriture dans les flux

Les opérateurs `<<` et `>>` sont réservés aux opérations d'entrée/sortie.

Flux standards

- `cin` : entrée clavier (classe `istream`)
- `cout` : sortie écran (classe `ostream`)

```
class Point {
    double x, y;

public:
    // Opérateur de sortie
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "Point(" << p.x << ", " << p.y << ")";
        return os;
    }

    // Opérateur d'entrée
    friend std::istream& operator>>(std::istream& is, Point& p) {
        is >> p.x >> p.y;
        return is;
```

Opérateur de flux - Exemple

```
class Point {
    double x, y;

public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}

    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }

    friend std::istream& operator>>(std::istream& is, Point& p) {
        is >> p.x >> p.y;
        return is;
    }
};

int main() {
    Point p(3.5, 2.1);
    std::cout << "Point: " << p << std::endl;  // Point: (3.5, 2.1)

    Point p2;
    std::cout << "Entrez un point (x y): ";
    std::cin >> p2;
```

Flux - Insertion dans une liste

Chaînage des opérateurs

```cpp
class Point {
    // ... (comme avant)
};

int main() {
    std::vector<Point> points;

    // Insertion de plusieurs points
    points.push_back(Point(1, 2));
    points.push_back(Point(3, 4));
    points.push_back(Point(5, 6));

    // Affichage chaîné
    for(const auto& p : points) {
        std::cout << p << " ";
    }
    // Affiche: (1, 2) (3, 4) (5, 6)

    // Chaînage possible car on retourne ostream&
    std::cout << points[0] << " et " << points[1] << std::endl;
}
```

Exemple complet de surcharge - Définition

```
class Point {
private:
    double x, y;

public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}

    // Opérateurs composés (membres)
    Point& operator+=(const Point& p) {
        x += p.x;
        y += p.y;
        return *this;
    }

    Point& operator*=(double s) {
        x *= s;
        y *= s;
        return *this;
    }

    // Opérateurs simples (friends)
    friend Point operator+(Point a, const Point& b);
    friend Point operator*(Point p, double s);
    friend Point operator*(double s, const Point& p);
```

Exemple complet de surcharge - Implémentation

```cpp
// Opérateur + utilise +=
Point operator+(Point a, const Point& b) {
    a += b;  // Réutilise l'opérateur +=
    return a;
}

// Opérateur * utilise *=
Point operator*(Point p, double s) {
    p *= s;  // Réutilise l'opérateur *=
    return p;
}

// Opérateur * symétrique
Point operator*(double s, const Point& p) {
    return p * s;  // Réutilise l'autre opérateur *
}

// Opérateur de sortie
std::ostream& operator<<(std::ostream& os, const Point& p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}
```

Exemple complet de surcharge - Commentaires

Avantages de cette approche

Principe

- L'opération `P + Q` appelle `+=`
- Une seule implémentation logique
- Plus facile à maintenir
- Plus robuste

```
Point p1(1, 2), p2(3, 4);
Point p3 = p1 + p2;  // Utilise +=
p1 += p2;            // Utilise += directement
```

Structure recommandée

1. Implémenter les opérateurs composés (`+=` , `*=` , etc.)
2. Implémenter les opérateurs simples en utilisant les composés
3. Éviter la duplication de code

```
// ✅ Bon
Point& operator+=(const Point& p) {
    x += p.x; y += p.y;
    return *this;
}
Point operator+(Point a, const Point& b) {
    return a += b;
}
```

Exemple évolué - Optimisation

Problème : matrices temporaires

Supposons une classe `Matrice` et le calcul : `2*A + 4*D - 5*C`

❌ Version naïve

```
// Crée 5 matrices temporaires !
Matrice T1 = 5*C;
Matrice T2 = 4*D;
Matrice T3 = T1 - T2;
Matrice T4 = 2*A;
Matrice result = T4 + T3;
```

Chaque opération crée une copie complète de la matrice.

✅ Version optimisée

```
// Utilise une structure intermédiaire
// qui décrit la combinaison
Combinaison C1 = 5*C;
Combinaison C2 = 4*D;
Combinaison C3 = C1 - C2;
Combinaison C4 = 2*A;
Matrice result = C4 + C3;
// Calcul effectué seulement ici
```

Une seule copie finale !

Exemple évolué - Implémentation

Structure intermédiaire

```
class Combinaison {
    std::vector<std::pair<double, const Matrice*>> termes;

public:
    Combinaison(double coef, const Matrice& m) {
        termes.push_back({coef, &m});
    }

    Combinaison& operator+(const Combinaison& autre) {
        for(const auto& terme : autre.termes) {
            termes.push_back(terme);
        }
        return *this;
    }
};

// Opérateurs pour Matrice
Combinaison operator*(double coef, const Matrice& m) {
    return Combinaison(coef, m);
}

Matrice& operator=(const Combinaison& comb) {
```

Méthodologie - Bonnes pratiques

Règles à suivre pour la surcharge d'opérateurs

1. **Sens naturel** : Ne jamais surcharger un opérateur avec un sens différent du sens commun
   - ❌ `+` pour faire une soustraction
   - ✅ `+` pour additionner des vecteurs
2. **Limiter les implémentations** : Pour des opérations similaires (`+` et `+=`)
   - Implémenter `+=`, puis `+` utilise `+=`
3. **Retour de valeurs** : Ne renvoyer des pointeurs qu'exceptionnellement
   - Si `A+B` retourne un pointeur, `(A+B)+C` est impossible
4. **Classes personnelles** : Ne surcharger que l'essentiel
   - Classes utilitaires : opérateurs de base
5. **Classes générales** : Prévoir toutes les surcharges possibles (complétude)

Ce que toutes les classes doivent avoir

Règle de 3 et de 5

Règle de 3 (C++98)

Si une classe définit l'une des 3 méthodes suivantes, alors les 3 doivent être définies :

1. **Destructeur** : `~Point()`
2. **Constructeur par copie** : `Point(const Point&)`
3. **Opérateur d'affectation** : `operator=(const Point&)`

Règle de 5 (C++11)

Avec C++11, il est recommandé de définir deux méthodes complémentaires :

4. **Constructeur par déplacement** : `Point(Point&&)`
5. **Opérateur d'affectation par déplacement** : `operator=(Point&&)`

Qualificateurs C++11 : `= default` ou `= delete`

Sémantique de copie

Copie profonde des ressources

```
class Tableau {
    int* data;
    int taille;

public:
    // 1. Constructeur
    Tableau(int n) : taille(n), data(new int[n]) {}

    // 2. Destructeur
    ~Tableau() {
        delete[] data;
    }

    // 3. Constructeur par copie
    Tableau(const Tableau& autre) : taille(autre.taille) {
        data = new int[taille];
        std::copy(autre.data, autre.data + taille, data);
    }

    // 4. Opérateur d'affectation
    Tableau& operator=(const Tableau& autre) {
        if (this != &autre) {
            delete[] data;
```

Sémantique de déplacement

Transfert de propriété (C++11)

```
class Tableau {
    int* data;
    int taille;

public:
    // Constructeur par déplacement
    Tableau(Tableau&& autre) noexcept
        : data(autre.data), taille(autre.taille) {
        autre.data = nullptr;   // Invalide la source
        autre.taille = 0;
    }

    // Opérateur d'affectation par déplacement
    Tableau& operator=(Tableau&& autre) noexcept {
        if (this ≠ &autre) {
            delete[] data;
            data = autre.data;
            taille = autre.taille;
            autre.data = nullptr;  // Invalide la source
            autre.taille = 0;
        }
        return *this;
```

Sémantique de déplacement - Avantages

Performance vs Copie

Copie (lente)

```
Tableau creer() {
    Tableau t(1000000);
    // ... remplir t ...
    return t;  // Copie complète !
}

Tableau t = creer();
// Allocation + copie de 1M éléments
```

Déplacement (rapide)

```
Tableau creer() {
    Tableau t(1000000);
    // ... remplir t ...
    return t;  // Déplacement !
}

Tableau t = creer();
// Juste transfert du pointeur
```

Quand utiliser le déplacement ?

- Retour de fonction
- Insertion dans conteneurs
- `std::move()` explicite

```
std::vector<Tableau> vec;
```

Utilisation de `= default` et `= delete`

Contrôle fin des méthodes spéciales

`= default` : générer par défaut

```
class Point {
public:
    Point() = default;  // OK

    Point(const Point&) = default;
    Point& operator=(const Point&)
        = default;

    ~Point() = default;
};
```

Utile quand on définit d'autres constructeurs.

`= delete` : interdire

```
class Singleton {
public:
    // Interdit la copie
    Singleton(const Singleton&)
        = delete;
    Singleton& operator=(
        const Singleton&) = delete;

    // OK : déplacement autorisé
    Singleton(Singleton&&) = default;
};

Singleton s1;
// Singleton s2 = s1;  // ERREUR
```

Héritage (aperçu)

Héritage - Introduction

Réutilisation et extension de classes

```
class Forme {
protected:
    std::string couleur;

public:
    Forme(const std::string& c) : couleur(c) {}

    virtual double aire() const = 0;  // Méthode virtuelle pure
    virtual void afficher() const {
        std::cout << "Forme de couleur " << couleur << std::endl;
    }
};

class Cercle : public Forme {
    double rayon;

public:
    Cercle(double r, const std::string& c)
        : Forme(c), rayon(r) {}

    double aire() const override {
        return 3.14159 * rayon * rayon;
```

Types d'héritage

Public, Protected, Private

Héritage public

```
class Derive
  : public Base {
  // Public → Public
  // Protected → Protected
};
```

Le plus courant.

Héritage protected

```
class Derive
  : protected Base {
  // Public → Protected
  // Protected → Protected
};
```

Rare en pratique.

Héritage private

```
class Derive
  : private Base {
  // Public → Private
  // Protected → Private
};
```

Implémentation cachée.

Règle générale : Utilisez l'héritage `public` sauf cas très spécifiques.

```
class Vehicule { /* ... */ };
class Voiture : public Vehicule { /* ... */ };  // "est un" Vehicule
```

Polymorphisme

Méthodes virtuelles

```
class Forme {
public:
    virtual void dessiner() const {
        std::cout << "Forme générique" << std::endl;
    }

    virtual ~Forme() {}  // Destructeur virtuel important !
};

class Cercle : public Forme {
public:
    void dessiner() const override {
        std::cout << "Cercle" << std::endl;
    }
};

int main() {
    Forme* f1 = new Forme();
    Forme* f2 = new Cercle();

    f1->dessiner();  // "Forme générique"
    f2->dessiner();  // "Cercle" - Polymorphisme !
```

Héritage multiple

Attention aux pièges

```
class A {
protected:
    int valeur;
};

class B : public A { };
class C : public A { };

class D : public B, public C {
    // D hérite deux fois de A !
    // Problème : deux copies de 'valeur'
};
```

⚠️ **Solution : Héritage virtuel**

```
class B : public virtual A { };
class C : public virtual A { };
```

Mais l'héritage virtuel a un coût en performance. À utiliser avec parcimonie.

Les amis (friends)

Accès privilégié entre classes

```
class Point {
private:
    double x, y;

public:
    Point(double x, double y) : x(x), y(y) {}

    // Fonction amie
    friend bool coincident(const Point& p1, const Point& p2);

    // Classe amie
    friend class Geometrie;
};

bool coincident(const Point& p1, const Point& p2) {
    return (p1.x == p2.x && p1.y == p2.y);  // Accès aux membres privés
}

class Geometrie {
public:
    double distance(const Point& p1, const Point& p2) {
        double dx = p1.x - p2.x;  // Accès aux membres privés
        double dy = p1.y - p2.y;
```

Les amis - Types

Différents niveaux d'amitié

1. **Fonction amie d'une classe**

```
friend void afficher(const Point& p);
```

2. **Fonction membre d'une classe, amie d'une autre**

```
friend void Couleur::colorer(Point& p);
```

3. **Fonction amie de plusieurs classes**

```
friend void comparer(const Point& p, const Vecteur& v);
```

4. **Toutes les fonctions d'une classe, amies d'une autre**

```
friend class Geometrie;
```

Les amis - Avantages et inconvénients

✅ Avantages

- Plus de flexibilité de conception
- Syntaxe parfois plus naturelle
  - Fonction amie : `f(objet)`
  - Fonction membre : `objet.f()`
- Utile pour opérateurs symétriques

❌ Inconvénients

- **L'amitié n'est pas héritée**
- **L'amitié n'est pas transitive**
- **L'amitié n'est pas réciproque**
- Ne comprend pas le polymorphisme
- Augmente risque de collisions
- Brise l'encapsulation

Règle d'or :
"Utilisez une fonction membre quand vous le pouvez,
utilisez une fonction amie quand vous le devez."

Conclusion

À retenir

Concepts clés du cours

Surcharge d'opérateurs

- Syntaxe : `operator symbole`
- Surcharge interne vs externe
- Opérateurs de flux (`<<` , `>>`)
- Méthodologie et bonnes pratiques

Règle de 3 et de 5

- Destructeur
- Constructeur par copie
- Opérateur d'affectation
- Constructeur par déplacement (C++11)

Sémantiques

- Copie profonde vs superficielle
- Déplacement (move semantics)
- `= default` et `= delete`

Héritage (aperçu)

- Polymorphisme
- Fonctions virtuelles
- Héritage multiple
- Les amis

Checkpoint - Ce que vous devez savoir

Compétences acquises

- Surcharger des opérateurs de base
- Implémenter la règle de 3
- Comprendre copie vs déplacement
- Créer une hiérarchie de classes simple
- Utiliser le polymorphisme de base

Auto-évaluation

- Puis-je surcharger `+` et `+=` correctement ?
- Ai-je compris la différence entre copie et déplacement ?
- Sais-je quand utiliser `virtual` ?

Exercice pratique

Créez une classe `Vecteur` avec :

```
class Vecteur {
    // Données membres
    // Règle de 5
    // Opérateurs : +, +=, *, <<
    // Méthode norme()
};
```

Puis créez une classe dérivée `Vecteur3D`.

La prochaine fois

Cours 4 : Patterns et Templates

Design Patterns

- Singleton
- Factory
- Observer
- Stratégie

Templates (Généricité)

- Templates de fonctions
- Templates de classes
- Spécialisation
- SFINAE

💡 **Préparation :** Révisez la surcharge d'opérateurs et l'héritage. Les templates vont généraliser ces concepts.

Questions ?

N'hésitez pas à poser vos questions sur la surcharge d'opérateurs, la règle de 3/5, ou l'héritage.