POO & C++

Rappels

Rappels

https://wandbox.org/permlink/r7UoPoO6t9itfgaJ

```cpp
// Programme "Points" (version 1)
#include <iostream>
#include <cstdlib>
#include <cmath>

class Point
{
    // Attribut de classe
    static int compteur ;

// Implémentation
private :
    double x,y ;
//Interface
public :
    //Constructeur
    Point(double X, double Y)
    : x(X), y(Y)
    {
        ++compteur ;
        std::cout << "compteur = " << compteur << std::endl;
    }
    // Destructeur
    ~Point()
    {
        std::cout << "Bye " << std::endl;
    }
    // Méthode affichage
    void affiche()
    {
        std::cout << "Point " << Point::compteur << " ( " << x << " , " << y << " )" << std::endl;
    }

    // Méthode distance
    static void distance(const Point &p1, const Point &p2)
    // Le passage d'arguments par référence évite une copie sur la pile
    // static void distance(Point p1, Point p2) marche aussi (passage d'arguments par valeur)
    {
      double dist ;
      dist = sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y)) ;
      std::cout << "Distance = " << dist << std::endl ;
    }

    // Méthode milieu
    static void milieu(const Point &p1, const Point &p2)
    {
        double X,Y ;
        X = (p1.x + p2.x) / 2 ;
        Y = (p1.y + p2.y) / 2 ;
        std::cout << "Milieu " << " ( " << X << " , " << Y << " )" << std::endl;
    }

};
//initialisation de l'attribut compteur à 0
int Point::compteur(0) ;

//Programme principal
int main()
{
    Point point1(0.0,0.0) ;
    point1.affiche() ;
    Point point2(1.0,1.0) ;
    point2.affiche() ;
    Point::distance(point1,point2) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point2) ;

    Point point3(10.0,10.0) ;
    point3.affiche() ;
    Point::distance(point1,point3) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point3) ;

    return 0 ;
}
```

https://wandbox.org/permlink/r7UoPoO6t9itfgaJ

Rappels

```
compteur = 1
Point 1 ( 0 , 0 )
compteur = 2
Point 2 ( 1 , 1 )
Distance = 1.41421
Milieu  ( 0.5 , 0.5 )
compteur = 3
Point 3 ( 10 , 10 )
Distance = 14.1421
Milieu  ( 5 , 5 )
Bye
Bye
Bye
```

```cpp
// Programme "Points" (version 1)
#include <iostream>
#include <cstdlib>
#include <cmath>

class Point
{
    // Attribut de classe
    static int compteur ;

// Implémentation
private :
    double x,y ;
//Interface
public :
    //Constructeur
    Point(double X, double Y)
    : x(X), y(Y)
    {
        ++compteur ;
        std::cout << "compteur = " << compteur << std::endl;
    }
    // Destructeur
    ~Point()
    {
        std::cout << "Bye " << std::endl;
    }
    // Méthode affichage
    void affiche()
    {
        std::cout << "Point " << Point::compteur << " ( " << x << " , " << y << " )" << std::endl;
    }

    // Méthode distance
    static void distance(const Point &p1, const Point &p2)
    // Le passage d'arguments par référence évite une copie sur la pile
    // static void distance(Point p1, Point p2) marche aussi (passage d'arguments par valeur)
    {
      double dist ;
      dist = sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y)) ;
      std::cout << "Distance = " << dist << std::endl ;
    }

    // Méthode milieu
    static void milieu(const Point &p1, const Point &p2)
    {
        double X,Y ;
        X = (p1.x + p2.x) / 2 ;
        Y = (p1.y + p2.y) / 2 ;
        std::cout << "Milieu " << " ( " << X << " , " << Y << " )" << std::endl;
    }

};
//initialisation de l'attribut compteur à 0
int Point::compteur(0) ;

//Programme principal
int main()
{
    Point point1(0.0,0.0) ;
    point1.affiche() ;
    Point point2(1.0,1.0) ;
    point2.affiche() ;
    Point::distance(point1,point2) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point2) ;

    Point point3(10.0,10.0) ;
    point3.affiche() ;
    Point::distance(point1,point3) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point3) ;

    return 0 ;
}
```

Rappels

https://wandbox.org/permlink/r7UoPoO6t9itfgaJ

Encapsulation
(private / public)

Méthodes
(fonctions membres)

```
compteur = 1
Point 1 ( 0 , 0 )
compteur = 2
Point 2 ( 1 , 1 )
Distance = 1.41421
Milieu  ( 0.5 , 0.5 )
compteur = 3
Point 3 ( 10 , 10 )
Distance = 14.1421
Milieu  ( 5 , 5 )
Bye
Bye
Bye
```

```cpp
// Programme "Points" (version 1)
#include <iostream>
#include <cstdlib>
#include <cmath>

class Point
{
    // Attribut de classe
    static int compteur ;

// Implémentation
private :
    double x,y ;
//Interface
public :
    //Constructeur
    Point(double X, double Y)
    : x(X), y(Y)
    {
        ++compteur ;
        std::cout << "compteur = " << compteur << std::endl;
    }
    // Destructeur
    ~Point()
    {
        std::cout << "Bye " << std::endl;
    }
    // Méthode affichage
    void affiche()
    {
        std::cout << "Point " << Point::compteur << " ( " << x << " , " << y << " )" << std::endl;
    }

    // Méthode distance
    static void distance(const Point &p1, const Point &p2)
    // Le passage d'arguments par référence évite une copie sur la pile
    // static void distance(Point p1, Point p2) marche aussi (passage d'arguments par valeur)
    {
      double dist ;
      dist = sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y)) ;
      std::cout << "Distance = " << dist << std::endl ;
    }

    // Méthode milieu
    static void milieu(const Point &p1, const Point &p2)
    {
        double X,Y ;
        X = (p1.x + p2.x) / 2 ;
        Y = (p1.y + p2.y) / 2 ;
        std::cout << "Milieu " << " ( " << X << " , " << Y << " )" << std::endl;
    }

};
//initialisation de l'attribut compteur à 0
int Point::compteur(0) ;

//Programme principal
int main()
{
    Point point1(0.0,0.0) ;
    point1.affiche() ;
    Point point2(1.0,1.0) ;
    point2.affiche() ;
    Point::distance(point1,point2) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point2) ;

    Point point3(10.0,10.0) ;
    point3.affiche() ;
    Point::distance(point1,point3) ; // Appel d'une méthode statique d'une classe
    Point::milieu(point1,point3) ;

    return 0 ;
}
```

Bilan

- Définition d’une classe
  → class Point { } ;
- Instanciation d’une classe en 3 objets
  → point1, point2, point3
- Passage d’argument par valeur
  → static void distance(Point p1, Point p2)
- Passage d’argument par référence
  → static void distance(const Point &p1, const Point &p2)
- Attribut statique et méthode statique
  → static int compteur ; → static void distance ()
- Les constructeurs et destructeurs
  → Point(double X, double Y): x(X), y(Y) → ~Point()

Le pointeur **this**

Le pointeur **this**

Le mot clé **this** dans une classe désigne un pointeur qui est initialisé à la création de chaque objet avec son adresse mémoire.
Lors de l'écriture d'une classe, il est utilisé comme un pointeur de l'objet en cours :

- **this** vaut l'adresse mémoire de l'instance courant de l'objet ;
- ***this** désigne le contenu des valeurs de l'instance courante de l'objet
- **this->** permet d'accéder depuis la classe à n'importe quel élément constitutif de la classe.

Le pointeur **this**

Le mot clé **this** dans une classe désigne un pointeur qui est initialisé à la création de chaque objet avec son adresse mémoire.
Lors de l'écriture d'une classe, il est utilisé comme un pointeur de l'objet en cours :

- **this** vaut l'adresse mémoire de l'instance courant de l'objet ;
- ***this** désigne le contenu des valeurs de l'instance courante de l'objet
- **this->** permet d'accéder depuis la classe à n'importe quel élément constitutif de la classe.

```cpp
// Programme "Personne" (version 1)
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>

class Personne
{
private:
    std::string _nom = "";
    int _age = 0;

public:
void SetPersonne(std::string nom, int age){
    this -> _nom = nom;
    this -> _age = age;
}
void affiche(){
    std::cout << "Nom : " << _nom << std::endl;
    std::cout << "Age : " << _age << std::endl;
}
};
```

```cpp
int main(){
    // Création d'une instance de Personne
    Personne p1;

    // Initialisation des valeurs
    p1.SetPersonne("Alice", 30);

    // Affichage des informations
    p1.affiche();

    return 0;
    }
```

https://wandbox.org/permlink/T6lp30jhFiFRvBeU

Le pointeur **this**

Le mot clé **this** dans une classe désigne un pointeur qui est initialisé à la création de chaque objet avec son adresse mémoire.
Lors de l'écriture d'une classe, il est utilisé comme un pointeur de l'objet en cours :

- **this** vaut l'adresse mémoire de l'instance courant de l'objet ;
- ***this** désigne le contenu des valeurs de l'instance courante de l'objet
- **this->** permet d'accéder depuis la classe à n'importe quel élément constitutif de la classe.

```cpp
// Programme "Personne" (version 2)
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>

class Personne
{
private:
    std::string nom = "";
    int age = 0;

public:
void SetPersonne(std::string nom, int age){
    this -> nom = nom;
    this -> age = age;
}
void affiche(){
    std::cout << "Nom : " << nom << std::endl;
    std::cout << "Age : " << age << std::endl;
}
};
```

```cpp
int main(){
    // Création d'une instance de Personne
    Personne p1;

    // Initialisation des valeurs
    p1.SetPersonne("Alice", 30);

    // Affichage des informations
    p1.affiche();

    return 0;
    }
```

https://wandbox.org/permlink/ElxRGZHGCvGYMoVL

Le pointeur **this** est un élément clé de la programmation orientée objet en C++. Il fait référence à l'objet courant sur lequel une méthode est appelée.

1. Définition du pointeur **this** :

En C++, **this** est un pointeur spécial accessible uniquement à l'intérieur des méthodes d'une classe.

Il pointe automatiquement vers l'objet courant (l'instance en cours d'utilisation).

```
this->_nom;  // accède à l'attribut _nom de l'objet actuel
```

2. Quand utiliser le pointeur **this** ?

a. Pour résoudre les conflits de noms :

Par exemple, dans un **constructeur** ou un **setter**, si les paramètres portent le même nom que les attributs :

```
1 class Personne {
2 private:
3     std::string nom;
4
5 public:
6     void setNom(std::string nom) {
7         this->nom = nom; // ici, this est nécessaire pour distinguer l'attribut de la variable locale
8     }
9 };
```

Sans **this**, le compilateur ne pourrait pas différencier clairement :

```
nom = nom; // incorrect (ambiguïté)
```

b. Pour retourner une référence à l'objet courant :

Permet d'enchaîner des appels de fonctions (fluent interface) :

```cpp
class Personne {
public:
    Personne& setNom(const std::string& nom) {
        this->nom = nom;
        return *this;
    }

    Personne& setAge(int age) {
        this->age = age;
        return *this;
    }
private:
    std::string nom;
    int age;
};
```

```cpp
// Permet une écriture concise :
Personne p;
p.setNom("Alice").setAge(30);
```

Ici, le pointeur **this** permet d'implémenter des méthodes chaînables (fluent interface)

Note : signifie que la méthode **setNom** retourne une référence (**&**) à un objet de type **Personne**.
→ L'intérêt principal est d'implémenter ce qu'on appelle le « fluent interface », ou interface fluide.
Cela permet d'enchaîner plusieurs appels de méthodes sur le même objet.
Pour que cela soit possible, les méthodes doivent :

- retourner une référence (**&**) à l'objet appelant (c'est-à-dire ***this**) ;
- et non pas retourner une copie (qui créerait inutilement un nouvel objet différent).

Sans référence (Personne) :

```
Personne setNom(const std::string& nom) {
    this->nom = nom;
    return *this;  // Retourne une COPIE de l'objet
}

Personne p;
p.setNom("Alice").setAge(30);  // Erreur, car setAge est appelé sur une copie temporaire, pas sur p directement
```

Problème : On modifie une copie temporaire qui disparaît immédiatement après l'appel.

Avec référence (Personne&) :

```
Personne& setNom(const std::string& nom) {
    this->nom = nom;
    return *this;  // Retourne une référence à l'objet lui-même
}

Personne p;
p.setNom("Alice").setAge(30);  // Fonctionne parfaitement
```

Correct : On modifie directement le même objet (**p**) à chaque appel.

c. Pour passer l'objet courant en argument :

Par exemple, quand une méthode prend un objet de la même classe en paramètre :

```
void comparerAvec(const Personne& autre) {
    if(this->age > autre.age)
        std::cout << this->nom << " est plus âgé(e)." << std::endl;
}
```

On peut l'utiliser ainsi :

```
Personne p1("Alice", 30);
Personne p2("Bob", 25);

p1.comparerAvec(p2); // 'this' représente ici p1
```

3. Cas où le pointeur **this** est facultatif :

Lorsque le contexte est clair, l'usage explicite du pointeur **this** est optionnel.

```
class Personne {
private:
    int age;

public:
    void vieillir() {
        age++; // utilisation implicite de this->age
    }
};
```

Toutefois, certains programmeurs utilisent explicitement **this** par souci de lisibilité.

Bilan

Le pointeur **this** est utile principalement pour :

- Éviter les ambiguïtés en cas de conflits de noms.
  → *this->nom = nom;*
- Implémenter des méthodes chaînables (fluent interface).
  → *return *this;*
- Passer l’objet lui-même à d'autres méthodes ou fonctions.
  → *autre.fonction(*this);*
- Clarifier le code et améliorer sa lisibilité.
  → *this->methode();*

Il est intrinsèquement lié à la notion d'instance d'objet et à l'orienté objet en C++.

Les « getters » et les « setters »

Les **getters** et les **setters**

```cpp
#include <iostream>
#include <string>

class Personne
{
private:
    std::string _nom;
    int _age;
public:
    // Constructeur
    Personne(const std::string& nom = "", int age = 0)
        : _nom(nom), _age(age) {}

    // Setters
    void setNom(const std::string& nom) { _nom = nom; }
    void setAge(int age) { _age = age; }

    // Getters
    std::string getNom() const { return _nom; }
    int getAge() const { return _age; }

    // Affichage
    void affiche() const {
        std::cout << "Nom : " << _nom << std::endl;
        std::cout << "Age : " << _age << std::endl;
    }
};
```

```cpp
int main() {
    Personne p1("Alice", 30);
    p1.affiche();

    // Utilisation des setters
    p1.setNom("Bob");
    p1.setAge(25);

    // Affichage avec les getters
    std::cout << "Nouveau nom : " << p1.getNom() << std::endl;
    std::cout << "Nouvel âge : " << p1.getAge() << std::endl;

    return 0;
}
```

https://wandbox.org/permlink/n8rOmz9ixgayECnR

Encapsulation :
Utiliser des setters et getters améliore la gestion des données privées, facilite le contrôle de l'intégrité des données et augmente la sécurité du code.

Constructeurs :
Ils permettent de simplifier la création d'objets en une seule instruction claire.

Utiliser const sur les méthodes non-modifiantes garantit la sécurité de ton code.

Un exemple

https://wandbox.org/permlink/k3LDYwrI2tUsnJvd

```cpp
#include <iostream>

class Rectangle
{
private:
    double largeur;
    double hauteur;

public:
    // Constructeur
    Rectangle(double largeur, double hauteur)
    {
        // Utilisation explicite du pointeur this pour différencier les attributs
        this->largeur = largeur;
        this->hauteur = hauteur;
    }

    // Setter pour la largeur (renvoie une référence au Rectangle pour permettre l'enchaînement)
    Rectangle& setLargeur(double largeur)
    {
        this->largeur = largeur;
        return *this;
    }

    // Setter pour la hauteur (renvoie une référence pour l'enchaînement)
    Rectangle& setHauteur(double hauteur)
    {
        this->hauteur = hauteur;
        return *this;
    }

    // Méthode pour calculer l'aire
    double aire() const
    {
        return this->largeur * this->hauteur; // this-> facultatif ici, mais clarifie l'origine des données
    }

    // Affichage des dimensions
    void affiche() const
    {
        std::cout << "Rectangle [largeur: " << this->largeur
                  << ", hauteur: " << this->hauteur
                  << ", aire: " << this->aire() << "]" << std::endl;
    }
};

int main()
{
    // Création d'un objet Rectangle
    Rectangle rect(5.0, 3.0);

    // Affichage initial
    rect.affiche();

    // Changement des valeurs en utilisant les setters chaînés
    rect.setLargeur(10.0).setHauteur(4.0);

    // Affichage après modification
    rect.affiche();

    return 0;
}
```

https://wandbox.org/permlink/k3LDYwrI2tUsnJvd

```cpp
#include <iostream>

class Rectangle
{
private:
    double largeur;
    double hauteur;

public:
    // Constructeur
    Rectangle(double largeur, double hauteur)
    {
        // Utilisation explicite du pointeur this pour différencier les attributs
        this->largeur = largeur;
        this->hauteur = hauteur;
    }

    // Setter pour la largeur (renvoie une référence au Rectangle pour permettre l'enchaînement)
    Rectangle& setLargeur(double largeur)
    {
        this->largeur = largeur;
        return *this;
    }

    // Setter pour la hauteur (renvoie une référence pour l'enchaînement)
    Rectangle& setHauteur(double hauteur)
    {
        this->hauteur = hauteur;
        return *this;
    }

    // Méthode pour calculer l'aire
    double aire() const
    {
        return this->largeur * this->hauteur; // this-> facultatif ici, mais clarifie l'origine des données
    }

    // Affichage des dimensions
    void affiche() const
    {
        std::cout << "Rectangle [largeur: " << this->largeur
                  << ", hauteur: " << this->hauteur
                  << ", aire: " << this->aire() << "]" << std::endl;
    }
};

int main()
{
    // Création d'un objet Rectangle
    Rectangle rect(5.0, 3.0);

    // Affichage initial
    rect.affiche();

    // Changement des valeurs en utilisant les setters chaînés
    rect.setLargeur(10.0).setHauteur(4.0);

    // Affichage après modification
    rect.affiche();

    return 0;
}
```

Rectangle [largeur: 5, hauteur: 3, aire: 15]
Rectangle [largeur: 10, hauteur: 4, aire: 40]

Pédagogie par projets

- 8 heures
- 1 projet pour un groupe de 4/5 étudiants
- 1 concept à présenter à la classe par oral (diaporama) avec une fiche de cours
- Diagramme SysML : diagramme des exigences

=> Vidéo de démonstration (3 minutes) + soutenance (10 minutes)

1/ Station météo connectée
- Classes : Capteur (abstraite), Température, Humidité, Luminosité, AfficheurLCD, gaz EnregistreurSD.
- Extensions : Communication série ou WiFi (ESP8266/ESP32).

2/ Robot suiveur de ligne
- Classes : Moteur, CapteurIR, ContrôleurPID, LCD I2C, Robot.
- Objectif : Appliquer la POO au contrôle du robot.

3/ Feux de circulation intelligents
- Classes : Feu, CapteurPrésence, Intersection, Minuteur, ESP32 OLED.
- Approche : Coordination entre objets simulant une intersection.

4/ Alarme avec capteur de mouvement
- Classes : CapteurPIR, Sirène, ÉcranOLED, clavier matriciel, RFID, GestionnaireAlarme.
- But : Intégrer des événements matériels dans une architecture objet.

5/ Système d’arrosage automatique
- Classes : CapteurHumidité, Température, Pompe (relais + pompe), Horloge, clavier matriciel GestionnaireArrosage.
- Extension : Gestion autonome avec horaires, modes auto/manu.