Programmation C / C++

#5 POO en C++

par David Albert

GEMA
GRANDES ÉCOLES DES MÉTIERS D'AVENIR

2024

Table des matières

01

Qu'est-ce que la POO ?

Rappel : La programmation procédurale

Définition

La **programmation procédurale** décrit les opérations d'un programme comme des séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.

Concept associés

- assignation
- condition
- boucle
- branchement
- séquence d'instructions

Langages : C, Python, PHP, Javascript

Exemple

```
int addition(int a, int b) {
    return a + b;
}

int pTomatoes = 2.80;
int pPotatoes = 3.68;
int pTotal = addition(pTomatoes, pPotatoes);

if (pTotal > 10)
    cout << "C'est cher" << endl;
else
    cout << "C'est pas cher" << endl;
```

La programmation orientée objet

Définition

La **programmation orienté objet (POO)** décrit les opérations d'un programme grâce à la définition et l'interaction de briques logicielles appelées *objets*. En POO, on cherche à représenter ces objets et leurs relations.

Concept associés

- classes et instances
- encapsulation
- abstraction
- héritage
- polymorphisme

Langages : Java, Python, C++

Intérêts

- Modularité et réutilisabilité
- Facilité de compréhension
  - plus proche du langage parlé
- Code flexible et extensible

La programmation orientée objet

Exemple

1. Instanciation et appel de méthodes

```
ShoppingCart cart(10);
cart.addItem(Item("tomatoes", 2.80));
cart.addItem(Item("potatoes", 3.68));

if (cart.isExpensive())
    cout << "C'est cher" << endl;
else:
    cout << "C'est pas cher" << endl;
```

1. Déclaration de la classe `Item`

```
class Item {
    private:
        string name;
        float price;

    public:
        Item(string name, float price): name(name), price(price) {}

        float getPrice() {
            return this->price;
        }
};
```

1. Déclaration de la classe `ShoppingCart`

```
class ShoppingCart {
    private:
        std::vector<Item> cart = {};
        float maxPrice;

    public:
        ShoppingCart(float max) : maxPrice(max) {}

        void addItem(Item item) {
            this->cart.append(item);
        }

        float getPrice() {
            float tot = 0;
            for (int i=0; i< this->cart.size(); i++) {
                tot += this->cart[i].getPrice();
            }
            return tot
        }

        bool isExpensive() {
            return (this->getPrice() > this->maxPrice)
        }
};
```

Encore plus de paradigmes

Programmation fonctionnelle
Décrit les opérations d'un programme exclusivement à l'aide de *fonctions*. Les données du programme sont ainsi immutables (elles ne sont pas modifiées après leur création).
Langages: OCaml, LISP

Programmation logique
Décrit les opérations d'un programme à l'aide de faits et les règles logiques. Cela permet de déduire des conclusions à partir d'un ensemble de faits et de requêtes.
Langages: Prolog

Programmation descriptive
Décrit une application et/ou des structures de données sans état interne.
Langages: HTML, LaTex

i

La plupart des langages sont **multi-paradigmes**, c'est-à-dire qu'ils supportent simultanément plusieurs paradigmes: impératif, orienté objet, fonctionnel, etc.

C'est notamment le cas du **C++**.

Comparaison paradigmes de programmation

	prog. impérative	prog. orientée objet	prog. fonctionnelle
Popularité	le plus courant et le plus vieux	le plus flexible et lisible	le plus sûre
Le développeur	décrit de façon séquentielle comment un programme doit travailler	décrit des concepts et les relations entre ces concepts	décrit le programme exclusivement à l'aide de fonctions
Mots clés	assignation, condition, boucle, branchement, séquence d'instructions	classe, instance, constructeur, encapsulation, abstraction, héritage, polymorphisme	immuable, fonctions pures, fonctions d'ordres supérieur
Langages	C, Python, PHP, Javascript	Java, Python, C++	OCaml, LISP, ML

02

Concepts fondamentaux

Les 5 concepts clés en POO

- les classes
- les objets
- l'encapsulation
- l'héritage
- le polymorphisme

Classes et Objets

Les classes

Une **classe** est un **type de données composite** constitué:

- de données que l'on appelle **attributs** (des variables primitives ou des objets)
- de **méthodes** permettant de traiter ces données et des données extérieures à la classe

Les objets

Un **objet** est une **variable** dont le type est une classe particulière
→ on parle d'une **instance de classe**.

Exemple

```
class Voiture {
  private:
    string marque;
    int annee;

  public:
    Voiture(string marque, int annee)
        : marque(marque), annee(annee) {
    }

    void demarrer() {
        cout << "La voiture démarre." << endl;
    }
};

// Création d'un objet voiture
Voiture maVoiture("Toyota", 2022);
maVoiture.demarrer();
```

Encapsulation

Encapsuler les données

Les données d'une classe sont cachées et ne peuvent être accédées directement.

Accesseurs (getters) et Mutateurs (setters)

Contrôlent l'accès aux données d'une classe.

Exemple

```
class CompteBancaire {
private:
  double solde;

public:
  double getSolde() {
      return solde;
  }

  void setSolde(double montant) {
      solde = montant;
  }
};
```

Héritage

En POO, l'héritage est le concept qui permet de créer une nouvelle classe à partir d'une classe existante.

Visibilité (public, protected, private)

Contrôle l'accès aux membres de la classe de base.

Exemple

```
class Animal {
public:
    void manger() {
        cout << "L'animal mange." << endl;
    }
};

class Chien : public Animal {
public:
    void aboyer() {
        cout << "Le chien aboie." << endl;
    }
};
```

Polymorphisme

En POO, le polymorphisme est le concept qui permet à plusieurs classes d'implémenter une même interface de manière spécifique.

Fonctions virtuelles

Permettent le polymorphisme.

Exemple

```
class Forme {
public:
    virtual void dessiner() {
        cout << "Dessine une forme." << endl;
    }
};

class Cercle : public Forme {
public:
    void dessiner() override {
        cout << "Dessine un cercle." << endl;
    }
};
```

Abstraction

En POO, le concept d'abstraction sert à définir un modèle pour d'autres classes.

Classe abstraite

Une classe abstraite est une classe qui comprend **au moins** une méthode **non implémentée**.

Interface

Une interface est une classe abstraite particulière. Elle ne contient **aucun attribut** et ses méthodes ne sont **pas implémentées**.

Exemple

```
class Interface {
public:
    virtual void operation() = 0;
};

class ConcreteClass : public Interface {
public:
    void operation() override {
        cout << "Implémentation de l'opération." << endl;
    }
};
```

03

Gestion de la Mémoire en POO

Constructeurs et Destructeurs

Constructeurs
Initialisent les objets lors de leur création.

Destructeurs
Libèrent la mémoire utilisée par un objet lors de sa destruction.

Exemple

```
class Personne {
private:
    string nom;

public:
    Personne(string nom) : nom(nom) {
        cout << "Personne créée avec le nom : " << nom << endl;
    }
    ~Personne() {
        cout << "Personne détruite" << endl;
    }
};
```

Allocation dynamique d'objets

Allocation d'Objets

La syntaxe pour allouer dynamiquement un objet en C++ est d'utiliser l'opérateur `new`. Cela renvoie un pointeur vers l'objet nouvellement alloué.

```
class MaClasse {
public:
// ... (attributs et méthodes de la classe)
};

int main() {
    // Allocation dynamique d'un objet MaClasse
    MaClasse* monObjet = new MaClasse();

    // Utilisation de monObjet

    // Libération de la mémoire allouée
    delete monObjet;
}
```

Pointeurs intelligents

unique_ptr

L'utilisation des pointeurs intelligents (`std::unique_ptr` et `std::shared_ptr`) peut simplifier la gestion de la mémoire en C++.

Exemple `std::unique_ptr`

```cpp
#include <memory>

class MaClasse {
public:
// ... (membres et méthodes de la classe)
};

int main() {
    // Utilisation de std::unique_ptr pour allouer dynamiquement un objet
    std::unique_ptr<MaClasse> monObjet = std::make_unique<MaClasse>();

    // Pas besoin de delete, la mémoire est automatiquement libérée lorsque monObjet sort de la portée.

    return 0;

}
```

Pointeurs intelligents

shared_ptr

L'utilisation des pointeurs intelligents (`std::unique_ptr` et `std::shared_ptr`) peut simplifier la gestion de la mémoire en C++.

Exemple `std::shared_ptr`

```cpp
#include <memory>

class MaClasse {
public:
// ... (membres et méthodes de la classe)
};

int main() {
    // Utilisation de std::shared_ptr pour partager la propriété d'un objet entre plusieurs pointeurs
    std::shared_ptr<MaClasse> monObjet = std::make_shared<MaClasse>();

    // Pas besoin de delete, la mémoire est automatiquement libérée
    // lorsque le dernier pointeur partageant la propriété est détruit.
}
```