INTRODUCTION AU POLYMORPHISME

DÉFINITION

Le polymorphisme est un concept de programmation orientée objet. Il permet à une fonction ou à un objet de se comporter différemment selon le contexte. Le terme vient du grec "poly" (plusieurs) et "morph" (formes). Il permet d'utiliser une interface commune pour des actions différentes. C++ supporte le polymorphisme statique et dynamique.

IMPORTANCE DU POLYMORPHISME

Facilite la réutilisation du code. Permet d'écrire du code plus flexible et extensible. Encourage l'utilisation d'interfaces communes. Simplifie la maintenance du code. Facilite l'ajout de nouvelles fonctionnalités sans modifier le code existant.

CONCEPTS CLÉS

- Classes et objets
- Héritage
- Surcharge de fonction
- Surcharge d'opérateur
- Interfaces (classes abstraites)
- Méthodes virtuelles

TYPES DE POLYMORPHISME

- Polymorphisme statique (ou de compilation)
  - Surcharge de fonction
  - Surcharge d'opérateur
- Polymorphisme dynamique (ou d'exécution)
  - Héritage
  - Méthodes virtuelles

EXEMPLE SIMPLE

```cpp
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void sound() {
        cout << "Some sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Bark" << endl;
    }
```

AVANTAGES DU POLYMORPHISME

- Réduction du couplage entre les classes
- Flexibilité accrue dans le design du logiciel
- Facilite l'implémentation des patterns de conception
- Améliore la lisibilité du code
- Permet l'extension du code sans modification majeure

LIMITATIONS DU POLYMORPHISME

- Peut compliquer le débogage
- Peut introduire des problèmes de performance
- Nécessite une bonne compréhension des concepts OO
- Peut rendre le code moins lisible pour les débutants
- Peut introduire des erreurs difficiles à détecter

POLYMORPHISME STATIQUE VS DYNAMIQUE

DÉFINITION

Le polymorphisme en C++ permet à une fonction ou un objet de prendre plusieurs formes. Il se divise en deux types : statique et dynamique. Le polymorphisme statique est résolu à la compilation. Le polymorphisme dynamique est résolu à l'exécution.

POLYMORPHISME STATIQUE

Le polymorphisme statique est aussi appelé polymorphisme de compilation. Il est réalisé via la surcharge de fonctions et les templates. Les décisions sont prises par le compilateur lors de la compilation.

POLYMORPHISME DYNAMIQUE

Le polymorphisme dynamique est aussi appelé polymorphisme d'exécution. Il est réalisé via l'héritage et les fonctions virtuelles. Les décisions sont prises lors de l'exécution du programme.

DIFFÉRENCES ENTRE POLYMORPHISME STATIQUE ET DYNAMIQUE

Caractéristique	Polymorphisme statique	Polymorphisme dynamique
Moment de résolution	Compilation	Exécution
Techniques utilisées	Surcharge, Templates	Héritage, Fonctions virtuelles
Performance	Plus rapide	Plus lent

AVANTAGES DU POLYMORPHISME STATIQUE

- Résolution à la compilation, donc plus rapide.
- Moins de surcharge mémoire.
- Vérification des erreurs à la compilation.
- Meilleure optimisation par le compilateur.

AVANTAGES DU POLYMORPHISME DYNAMIQUE

- Permet une plus grande flexibilité.
- Supporte l'héritage et les interfaces complexes.
- Permet de créer des systèmes extensibles.
- Utile pour les architectures orientées objets.

INCONVÉNIENTS DU POLYMORPHISME STATIQUE

- Moins flexible comparé au polymorphisme dynamique.
- Les erreurs sont plus difficiles à corriger après compilation.
- Limité aux types connus à la compilation.

INCONVÉNIENTS DU POLYMORPHISME DYNAMIQUE

- Plus lent en raison de la résolution à l'exécution.
- Consomme plus de mémoire.
- Plus complexe à déboguer.
- Peut introduire des erreurs d'exécution difficiles à détecter.

EXEMPLES DE POLYMORPHISME STATIQUE

```cpp
#include <iostream>

// Surcharge de fonction
void print(int i) {
    std::cout << "Int: " << i << std::endl;
}

void print(double f) {
    std::cout << "Double: " << f << std::endl;
}

// Template
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

EXEMPLES DE POLYMORPHISME DYNAMIQUE

```cpp
#include <iostream>

class Base {
public:
    virtual void show() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class" << std::endl;
    }
};
```

REDÉFINITION DES MÉTHODES

DÉFINITION

La redéfinition des méthodes en C++ permet à une classe dérivée de fournir une implémentation spécifique d'une méthode déjà définie dans sa classe de base. Elle est utilisée pour personnaliser ou étendre le comportement des méthodes héritées.

OBJECTIF

L'objectif de la redéfinition des méthodes est de permettre une personnalisation du comportement des classes dérivées. Elle permet de fournir des implémentations spécifiques pour des méthodes déjà existantes dans la classe de base.

SYNTAXE DE LA REDÉFINITION

Pour redéfinir une méthode dans une classe dérivée, utilisez le même nom de méthode et la même signature que dans la classe de base.

```
class Base {
public:
    virtual void afficher();
};

class Derivee : public Base {
public:
    void afficher() override;
};
```

EXEMPLE DE REDÉFINITION

```cpp
#include <iostream>

class Base {
public:
    virtual void afficher() {
        std::cout << "Affichage de Base" << std::endl;
    }
};

class Derivee : public Base {
public:
    void afficher() override {
        std::cout << "Affichage de Derivee" << std::endl;
    }
};
```

UTILISATION AVEC CLASSES DÉRIVÉES

La redéfinition est utilisée pour fournir des implémentations spécifiques dans les classes dérivées. Elle permet de modifier le comportement des méthodes héritées sans changer la classe de base.

REDÉFINITION ET HÉRITAGE

La redéfinition des méthodes fonctionne en conjonction avec l'héritage. Elle permet à une classe dérivée de remplacer ou d'étendre les fonctionnalités d'une méthode de la classe de base.

AVANTAGES

- Permet la personnalisation des méthodes héritées.
- Facilite la réutilisation du code.
- Améliore la flexibilité et l'extensibilité des programmes.
- Supporte le polymorphisme dynamique.

LIMITATIONS

- Peut entraîner des erreurs si mal utilisée.
- Nécessite une compréhension claire de l'héritage et du polymorphisme.
- Peut compliquer la maintenance du code.

BONNES PRATIQUES

- Utilisez le mot-clé `override` pour indiquer explicitement une redéfinition.
- Assurez-vous que les signatures des méthodes redéfinies correspondent exactement.
- Documentez les redéfinitions pour améliorer la lisibilité du code.
- Testez soigneusement les méthodes redéfinies pour éviter les erreurs.

POINTEURS ET RÉFÉRENCES DE BASE

DÉFINITION

Les pointeurs et références permettent de manipuler des objets de manière indirecte. Un pointeur stocke l'adresse mémoire d'un objet. Une référence est un alias pour un objet existant. Ils sont utilisés pour le polymorphisme et la gestion dynamique de la mémoire.

DIFFÉRENCE ENTRE POINTEUR ET RÉFÉRENCE

- Un pointeur peut être nul, une référence ne peut pas.
- Un pointeur peut être réassigné, une référence ne peut pas.
- Les pointeurs nécessitent une gestion explicite de la mémoire.
- Les références sont plus sûres mais moins flexibles.

SYNTAXE DES POINTEURS

```cpp
int* ptr = nullptr; // Déclaration d'un pointeur
ptr = &variable;    // Assignation de l'adresse de la variable
```

SYNTAXE DES RÉFÉRENCES

```cpp
int variable = 10;
int& ref = variable; // Déclaration et initialisation d'une référence
```

UTILISATION DE POINTEURS

Les pointeurs sont utilisés pour :

- Allouer de la mémoire dynamiquement
- Accéder aux membres d'objets dérivés via des pointeurs de base
- Passer des arguments par adresse aux fonctions

UTILISATION DE RÉFÉRENCES

Les références sont utilisées pour :

- Passer des arguments par référence aux fonctions
- Retourner des objets par référence
- Accéder à des objets dérivés via des références de base

AVANTAGES DES POINTEURS

- Flexibilité dans la gestion de la mémoire.
- Permet de manipuler directement les adresses mémoire.
- Utile pour les structures de données dynamiques comme les listes chaînées.

AVANTAGES DES RÉFÉRENCES

- Plus sûres car elles ne peuvent pas être nulles.
- Syntaxe plus simple et lisible.
- Pas de gestion explicite de la mémoire nécessaire.

RISQUES ET PRÉCAUTIONS

- Les pointeurs peuvent causer des fuites de mémoire.
- Les pointeurs non initialisés peuvent provoquer des erreurs.
- Les références doivent être initialisées lors de leur déclaration.
- Utiliser des smart pointers pour une gestion automatique de la mémoire.

EXEMPLES PRATIQUES

```cpp
class Base {
public:
    virtual void afficher() { std::cout << "Base" << std::endl; }
};

class Derive : public Base {
public:
    void afficher() override { std::cout << "Dérivé" << std::endl; }
};

Base* ptr = new Derive();
ptr->afficher(); // Affiche "Dérivé"

Base& ref = *ptr;
ref.afficher(); // Affiche "Dérivé"
```

UTILISATION DES POINTEURS DE BASE POUR ACCÉDER AUX OBJETS DÉRIVÉS

DÉFINITION

Le polymorphisme permet de traiter des objets de types dérivés comme s'ils étaient des objets de type de base. Il est souvent utilisé avec des pointeurs ou des références de la classe de base. Cela permet d'écrire du code plus flexible et extensible.

EXEMPLE SIMPLE

```cpp
class Base {
public:
    virtual void afficher() { cout << "Base" << endl; }
};

class Derive : public Base {
public:
    void afficher() override { cout << "Derive" << endl; }
};

Base* ptr = new Derive();
ptr->afficher();  // Affiche "Derive"
```

SYNTAXE

Déclaration d'un pointeur de base pointant vers un objet dérivé :

```
Base* ptr = new Derive();
```

Appel d'une méthode virtuelle :

```
ptr->afficher();
```

CONVERSION DE POINTEURS DE BASE

Utilisation de `dynamic_cast` pour convertir un pointeur de base en pointeur dérivé :

```
Base* basePtr = new Derive();
Derive* derivePtr = dynamic_cast<Derive*>(basePtr);
```

ACCÈS AUX MEMBRES DÉRIVÉS

Pour accéder aux membres spécifiques de la classe dérivée :

```
Derive* derivePtr = dynamic_cast<Derive*>(basePtr);
if (derivePtr) {
    derivePtr->methodeSpecifique();
}
```

UTILISATION AVEC FONCTIONS VIRTUELLES

Les fonctions virtuelles permettent de définir une interface dans la classe de base et de fournir des implémentations spécifiques dans les classes dérivées :

```
class Base {
public:
    virtual void afficher() = 0; // Fonction virtuelle pure
};
```

AVANTAGES

- Flexibilité accrue du code
- Extensibilité facilitée
- Réduction du couplage entre les classes
- Facilite l'utilisation des collections hétérogènes

LIMITATIONS

- Peut entraîner une surcharge de performance
- Nécessite une gestion prudente des ressources
- Peut compliquer le débogage et la maintenance

AVANTAGES DU POLYMORPHISME

RÉUTILISABILITÉ DU CODE

Le polymorphisme permet de réutiliser le code existant. Les classes dérivées peuvent utiliser et étendre le comportement des classes de base. Cela réduit le besoin de réécrire du code similaire. Les méthodes polymorphiques peuvent être appelées sur des objets de différentes classes. Cela permet d'écrire des algorithmes génériques applicables à plusieurs classes. En utilisant des interfaces communes, le code devient plus modulaire et adaptable.

EXTENSIBILITÉ

Le polymorphisme facilite l'extension des programmes. Il permet d'ajouter de nouvelles fonctionnalités sans modifier le code existant. Les nouvelles classes peuvent hériter des classes existantes et ajouter de nouvelles méthodes. Cela rend les systèmes plus évolutifs et adaptables aux changements. Les méthodes polymorphiques permettent d'intégrer facilement de nouvelles classes. Cela encourage une conception orientée objet plus flexible et évolutive.

FLEXIBILITÉ

Le polymorphisme offre une grande flexibilité dans la conception des logiciels. Il permet de traiter des objets de différentes classes de manière uniforme. Les objets peuvent être substitués sans affecter le reste du code. Cela permet de créer des systèmes plus souples et adaptables. Les méthodes polymorphiques facilitent l'interaction entre différents objets. Cela simplifie la gestion de comportements variés dans un programme.

MAINTENANCE FACILITÉE

Le polymorphisme simplifie la maintenance du code. Les modifications peuvent être apportées dans les classes de base sans affecter les classes dérivées. Les classes dérivées peuvent être mises à jour indépendamment. Cela réduit les risques d'introduire des erreurs lors des modifications. Les interfaces communes facilitent la compréhension et la gestion du code. Cela améliore la lisibilité et la maintenabilité du logiciel.

RÉDUCTION DU COUPLAGE

Le polymorphisme permet de réduire le couplage entre les classes. Les classes peuvent interagir via des interfaces ou des classes de base communes. Cela diminue les dépendances directes entre les classes. Les modifications dans une classe n'affectent pas les autres classes. Cela facilite l'évolution et la maintenance du code. Les systèmes deviennent plus modulaires et indépendants.

SIMPLIFICATION DU CODE

Le polymorphisme simplifie le code en réduisant les structures conditionnelles. Les méthodes polymorphiques remplacent les instructions `if-else` ou `switch`. Cela rend le code plus lisible et moins sujet aux erreurs. Les algorithmes peuvent être appliqués de manière uniforme à différents objets. Cela réduit la duplication de code et améliore la clarté. Le code devient plus facile à comprendre et à maintenir.

GESTION DES COLLECTIONS D'OBJETS

Le polymorphisme facilite la gestion des collections d'objets. Les collections peuvent contenir des objets de différentes classes dérivées. Les méthodes polymorphiques permettent de traiter ces objets de manière uniforme. Cela simplifie les opérations sur les collections, comme le tri ou la recherche. Les algorithmes peuvent être appliqués à des collections hétérogènes. Cela améliore la flexibilité et la robustesse des programmes.

IMPLÉMENTATION DE STRUCTURES DE DONNÉES GÉNÉRIQUES

Le polymorphisme permet d'implémenter des structures de données génériques. Les structures comme les listes, les piles ou les files peuvent contenir des objets polymorphiques. Cela permet de créer des structures de données réutilisables et flexibles. Les algorithmes peuvent être appliqués à des structures contenant différents types d'objets. Cela réduit la duplication de code et améliore la modularité. Les programmes deviennent plus robustes et adaptables.

EXEMPLES PRATIQUES DE POLYMORPHISME

EXEMPLE AVEC CLASSES DE BASE ET DÉRIVÉES

```cpp
class Base {
public:
    virtual void show() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class" << std::endl;
    }
};

int main() {
    Base* b
```

POLYMORPHISME AVEC POINTEURS

```cpp
class Animal {
public:
    virtual void sound() {
        std::cout << "Some sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        std::cout << "Bark" << std::endl;
    }
};

int main() {
    Animal* a = new Dog();
```

POLYMORPHISME AVEC RÉFÉRENCES

```cpp
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing shape" << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing circle" << std::endl;
    }
};

void display(Shape& s) {
```

UTILISATION DE FONCTIONS VIRTUELLES

```cpp
class Base {
public:
    virtual void print() {
        std::cout << "Base function" << std::endl;
    }
};

class Derived : public Base {
public:
    void print() override {
        std::cout << "Derived function" << std::endl;
    }
};

int main() {
    Base* b = new Derived();
```

POLYMORPHISME AVEC DES COLLECTIONS

```cpp
#include <vector>

class Animal {
public:
    virtual void sound() const {
        std::cout << "Some sound" << std::endl;
    }
};

class Cat : public Animal {
public:
    void sound() const override {
        std::cout << "Meow" << std::endl;
    }
};
```

EXEMPLE DE POLYMORPHISME AVEC DES INTERFACES

```cpp
class IShape {
public:
    virtual void draw() const = 0; // Pure virtual function
};

class Square : public IShape {
public:
    void draw() const override {
        std::cout << "Drawing square" << std::endl;
    }
};

int main() {
    IShape* shape = new Square();
    shape->draw(); // Output: Drawing square
    delete shape;
```

POLYMORPHISME AVEC DES TEMPLATES

```cpp
template <typename T>
class Printer {
public:
    void print(const T& obj) {
        obj.display();
    }
};

class Document {
public:
    void display() const {
        std::cout << "Displaying document" << std::endl;
    }
};

int main() {
```

EXEMPLE DE POLYMORPHISME AVEC DES OPÉRATEURS

```cpp
class Number {
public:
    virtual int value() const = 0;
};

class Integer : public Number {
    int val;
public:
    Integer(int v) : val(v) {}
    int value() const override {
        return val;
    }
};

void printValue(const Number& num) {
```