

INTRODUCTION À L'HÉRITAGE

DÉFINITION

L'héritage est un concept clé de la programmation orientée objet. Il permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe, appelée classe dérivée, hérite des attributs et méthodes de la classe existante, appelée classe de base. L'héritage favorise la réutilisation du code et la modularité.

POURQUOI UTILISER L'HÉRITAGE

- Réutilisation du code existant.
- Facilite la maintenance et les mises à jour.
- Permet de créer des hiérarchies de classes.
- Encourage la modularité et la lisibilité du code.
- Simplifie la gestion des relations entre objets.

CLASSES DE BASE ET CLASSES DÉRIVÉES

- **Classe de base** : La classe existante dont les attributs et méthodes sont hérités.
- **Classe dérivée** : La nouvelle classe qui hérite de la classe de base.
- La classe dérivée peut ajouter de nouveaux attributs et méthodes.
- La classe dérivée peut également redéfinir les méthodes de la classe de base.

SYNTAXE DE L'HÉRITAGE

La syntaxe de l'héritage en C++ est la suivante :

```
class Base {  
    // Membres de la classe de base  
};  
  
class Dérivée : public Base {  
    // Membres de la classe dérivée  
};
```

EXEMPLE SIMPLE

Voici un exemple simple d'héritage en C++ :

```
class Animal {
public:
    void manger() {
        cout << "Manger" << endl;
    }
};

class Chien : public Animal {
public:
    void aboyer() {
        cout << "Aboyer" << endl;
    }
};
```

ACCÈS AUX MEMBRES DE LA CLASSE DE BASE

Dans la classe dérivée, on peut accéder aux membres de la classe de base :

```
Chien monChien;  
monChien.manger(); // Appel de la méthode de la classe de base  
monChien.aboyer(); // Appel de la méthode de la classe dérivée
```

HÉRITAGE PUBLIC, PRIVÉ ET PROTÉGÉ

- **Public** : Les membres publics de la classe de base restent publics dans la classe dérivée.
- **Privé** : Les membres publics de la classe de base deviennent privés dans la classe dérivée.
- **Protégé** : Les membres publics de la classe de base deviennent protégés dans la classe dérivée.

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Réutilisation du code.
- Facilite la maintenance.
- Crée des hiérarchies de classes.

Inconvénients :

- Peut compliquer la conception.
- Risque de dépendances fortes entre classes.
- Peut introduire des problèmes de performance.

CONCEPTS DE BASE DE L'HÉRITAGE

DÉFINITION DE L'HÉRITAGE

L'héritage en C++ permet de créer une nouvelle classe à partir d'une classe existante. La nouvelle classe est appelée classe dérivée. La classe existante est appelée classe de base. La classe dérivée hérite des membres (attributs et méthodes) de la classe de base. L'héritage favorise la réutilisation du code. Il permet également d'étendre les fonctionnalités des classes existantes.

POURQUOI UTILISER L'HÉRITAGE

Réutilisation du code existant. Facilite la maintenance et l'extension du code. Permet de modéliser des relations "est-un" entre les objets. Encapsule les comportements communs dans des classes de base. Facilite la gestion des modifications. Permet le polymorphisme, augmentant la flexibilité du code.

TYPES D'HÉRITAGE

Héritage simple Héritage multiple Héritage en diamant Héritage hiérarchique Héritage multiniveau

HÉRITAGE SIMPLE

L'héritage simple implique une seule classe de base et une seule classe dérivée. Exemple :

```
class Base {};
class Derived : public Base {};
```

La classe **Derived** hérite des membres de la classe **Base**.

HÉRITAGE MULTIPLE

L'héritage multiple permet à une classe dérivée d'hériter de plusieurs classes de base. Exemple :

```
class Base1 {};
class Base2 {};
class Derived : public Base1, public Base2 {};
```

La classe `Derived` hérite des membres de `Base1` et `Base2`.

HÉRITAGE EN DIAMANT

L'héritage en diamant se produit lorsque deux classes dérivées héritent d'une même classe de base et qu'une autre classe hérite de ces deux classes. Exemple :

```
class A {};
class B : public A {};
class C : public A {};
class D : public B, public C {};
```

Peut causer des ambiguïtés, résolues par l'héritage virtuel.

CLASSES DE BASE ET CLASSES DÉRIVÉES

La classe de base est la classe dont les membres sont hérités. La classe dérivée est la classe qui hérite des membres. Les classes dérivées peuvent ajouter de nouveaux membres. Les classes dérivées peuvent redéfinir les méthodes de la classe de base. Les classes de base peuvent être abstraites ou concrètes.

ACCÈS AUX MEMBRES DE LA CLASSE DE BASE

Les membres publics de la classe de base sont accessibles par la classe dérivée. Les membres protégés de la classe de base sont accessibles par la classe dérivée. Les membres privés de la classe de base ne sont pas accessibles directement par la classe dérivée. Exemple :

```
class Base {
    public: int pub;
    protected: int prot;
    private: int priv;
};

class Derived : public Base {
    void func() {
        pub = 1; // OK
        prot = 2; // OK
        // priv = 3; // Erreur
    }
};
```

POLYMORPHISME ET HÉRITAGE

Le polymorphisme permet d'utiliser une classe dérivée comme si c'était une classe de base. Les méthodes virtuelles sont utilisées pour le polymorphisme. Permet d'écrire du code générique et extensible. Exemple :

```
class Base {  
    public: virtual void show() { cout << "Base"; }  
};  
class Derived : public Base {  
    public: void show() override { cout << "Derived"; }  
};  
Base* b = new Derived();  
b->show(); // Affiche "Derived"
```

UTILISATION DU MOT-CLÉ 'PUBLIC'

Le mot-clé `public` rend tous les membres publics de la classe de base publics dans la classe dérivée. Les membres protégés restent protégés. Les membres privés restent inaccessibles. Exemple :

```
class Base {  
    public: int pub;  
};  
class Derived : public Base {};  
Derived d;  
d.pub = 1; // OK
```

UTILISATION DU MOT-CLÉ 'PROTECTED'

Le mot-clé `protected` rend tous les membres publics et protégés de la classe de base protégés dans la classe dérivée. Les membres privés restent inaccessibles. Exemple :

```
class Base {  
    public: int pub;  
    protected: int prot;  
};  
class Derived : protected Base {};  
Derived d;  
// d.pub = 1; // Erreur
```

UTILISATION DU MOT-CLÉ 'PRIVATE'

Le mot-clé `private` rend tous les membres publics et protégés de la classe de base privés dans la classe dérivée. Les membres privés restent inaccessibles. Exemple :

```
class Base {  
    public: int pub;  
    protected: int prot;  
};  
class Derived : private Base {};  
Derived d;  
// d.pub = 1; // Erreur
```

MÉCANISMES DE L'HÉRITAGE EN C++

DÉFINITION

L'héritage en C++ permet de créer une nouvelle classe à partir d'une classe existante. La classe existante est appelée "classe de base" ou "classe parente". La nouvelle classe est appelée "classe dérivée" ou "classe enfant". L'héritage favorise la réutilisation du code et la modularité.

SYNTAXE

La syntaxe de base pour déclarer une classe dérivée est :

```
class ClasseDérivée : access_specifier ClasseBase {  
    // Membres de la classe dérivée  
};
```

HÉRITAGE PUBLIC

Avec l'héritage public, les membres publics de la classe de base restent publics dans la classe dérivée. Les membres protégés de la classe de base restent protégés dans la classe dérivée. Syntaxe :

```
class ClasseDérivée : public ClasseBase {  
    // Membres de la classe dérivée  
};
```

HÉRITAGE PRIVÉ

Avec l'héritage privé, les membres publics et protégés de la classe de base deviennent privés dans la classe dérivée. Syntaxe :

```
class ClasseDérivée : private ClasseBase {  
    // Membres de la classe dérivée  
};
```

HÉRITAGE PROTÉGÉ

Avec l'héritage protégé, les membres publics et protégés de la classe de base deviennent protégés dans la classe dérivée. Syntaxe :

```
class ClasseDérivée : protected ClasseBase {  
    // Membres de la classe dérivée  
};
```

CONSTRUCTEURS ET DESTRUCTEURS

Les constructeurs de la classe de base sont appelés avant ceux de la classe dérivée. Les destructeurs de la classe dérivée sont appelés avant ceux de la classe de base. Syntaxe :

```
ClasseDérivée::ClasseDérivée() : ClasseBase() {  
    // Initialisation spécifique à la classe dérivée  
}
```

REDÉFINITION DE MÉTHODES

Une classe dérivée peut redéfinir les méthodes de la classe de base. Utilisez le mot-clé `override` pour indiquer qu'une méthode est redéfinie. Syntaxe :

```
class ClasseDérivée : public ClasseBase {  
public:  
    void Méthode() override {  
        // Nouvelle implémentation de la méthode  
    }  
};
```

UTILISATION DU MOT-CLÉ "VIRTUAL"

Le mot-clé `virtual` permet de créer des méthodes virtuelles. Les méthodes virtuelles peuvent être redéfinies dans les classes dérivées. Syntaxe :

```
class ClasseBase {
public:
    virtual void Methode() {
        // Implémentation de la méthode
    }
};
```

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Réutilisation du code
- Modélisation naturelle des relations hiérarchiques
- Facilité de maintenance

Inconvénients :

- Complexité accrue
- Risque de mauvaise utilisation
- Problèmes de performance (méthodes virtuelles)

CLASSES DE BASE ET CLASSES DÉRIVÉES

DÉFINITION

L'héritage en C++ permet de créer une nouvelle classe à partir d'une classe existante. La classe existante est appelée "classe de base". La nouvelle classe est appelée "classe dérivée". La classe dérivée hérite des membres (attributs et méthodes) de la classe de base.

RELATION ENTRE CLASSES DE BASE ET CLASSES DÉRIVÉES

La classe dérivée hérite des propriétés et comportements de la classe de base. Elle peut ajouter de nouveaux membres ou redéfinir des membres existants. La relation entre les deux classes est souvent décrite comme une relation "est-un".

ACCÈS AUX MEMBRES DE LA CLASSE DE BASE

Les membres publics et protégés de la classe de base sont accessibles dans la classe dérivée. Les membres privés de la classe de base ne sont pas accessibles directement dans la classe dérivée. Les membres protégés sont accessibles par les classes dérivées mais pas par d'autres parties du code.

REDÉFINITION DE MÉTHODES

Une classe dérivée peut redéfinir les méthodes de la classe de base. Cela permet à la classe dérivée de fournir une implémentation spécifique de la méthode. La redéfinition se fait en déclarant une méthode avec le même nom et la même signature.

UTILISATION DU MOT-CLÉ "PUBLIC"

Le mot-clé "public" rend les membres accessibles à l'extérieur de la classe. En utilisant "public" lors de l'héritage, les membres publics de la classe de base restent publics dans la classe dérivée. Syntaxe :

```
class Derived : public Base { ... };
```

UTILISATION DU MOT-CLÉ "PROTECTED"

Le mot-clé "protected" rend les membres accessibles uniquement aux classes dérivées et à la classe elle-même. En utilisant "protected" lors de l'héritage, les membres publics de la classe de base deviennent protégés dans la classe dérivée. Syntaxe : `class Derived : protected Base { ... };`

UTILISATION DU MOT-CLÉ "PRIVATE"

Le mot-clé "private" rend les membres accessibles uniquement à la classe elle-même. En utilisant "private" lors de l'héritage, les membres publics et protégés de la classe de base deviennent privés dans la classe dérivée. Syntaxe : `class Derived : private Base { ... };`

EXEMPLE DE CLASSES DE BASE ET DÉRIVÉES

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

AVANTAGES DE L'HÉRITAGE

- Réutilisation du code existant.
- Facilite la maintenance et l'extension du code.
- Favorise la modularité et la compréhension du code.

LIMITATIONS DE L'HÉRITAGE

- Peut introduire une complexité accrue.
- Risque de créer des dépendances étroites entre classes.
- Peut entraîner des problèmes de performance si mal utilisé.

CONSTRUCTEURS ET DESTRUCTEURS EN HÉRITAGE

DÉFINITION

En C++, les constructeurs et destructeurs sont des fonctions spéciales. Les constructeurs initialisent les objets lorsqu'ils sont créés. Les destructeurs libèrent les ressources lorsque les objets sont détruits. En héritage, les classes dérivées héritent des constructeurs et destructeurs de la classe de base. Ils permettent une gestion correcte des ressources et de l'initialisation.

CONSTRUCTEURS DE LA CLASSE DE BASE

Un constructeur de classe de base initialise les membres de cette classe. Syntaxe :

```
class Base {  
public:  
    Base() {  
        // Code d'initialisation  
    }  
};
```

Les classes dérivées peuvent appeler ces constructeurs.

CONSTRUCTEURS DE LA CLASSE DÉRIVÉE

Les constructeurs de la classe dérivée initialisent les membres de la classe dérivée. Ils peuvent aussi appeler les constructeurs de la classe de base. Syntaxe :

```
class Derived : public Base {  
public:  
    Derived() : Base() {  
        // Code d'initialisation supplémentaire  
    }  
};
```

APPEL DES CONSTRUCTEURS DE LA CLASSE DE BASE

Les constructeurs de la classe de base sont appelés avant ceux de la classe dérivée. Cela garantit que la classe de base est correctement initialisée. Exemple :

```
Derived::Derived() : Base() {  
    // Initialisation de Derived  
}
```

DESTRUCTEURS DE LA CLASSE DE BASE

Un destructeur de classe de base libère les ressources de la classe de base. Syntaxe :

```
class Base {  
public:  
    ~Base() {  
        // Code de nettoyage  
    }  
};
```

Les destructeurs de la classe dérivée peuvent appeler ceux de la classe de base.

DESTRUCTEURS DE LA CLASSE DÉRIVÉE

Les destructeurs de la classe dérivée libèrent les ressources de cette classe. Ils peuvent aussi appeler les destructeurs de la classe de base. Syntaxe :

```
class Derived : public Base {  
public:  
    ~Derived() {  
        // Code de nettoyage supplémentaire  
    }  
};
```

ORDRE D'APPEL DES DESTRUCTEURS

Les destructeurs sont appelés dans l'ordre inverse des constructeurs. Le destructeur de la classe dérivée est appelé avant celui de la classe de base. Cela garantit que les ressources de la classe dérivée sont libérées en premier. Exemple :

```
Derived::~Derived() {  
    // Nettoyage de Derived  
}
```

EXEMPLE DE CODE

```
class Base {
public:
    Base() { /* Initialisation */ }
    ~Base() { /* Nettoyage */ }
};

class Derived : public Base {
public:
    Derived() : Base() { /* Initialisation supplémentaire */ }
    ~Derived() { /* Nettoyage supplémentaire */ }
};
```


AVANTAGES

- Facilite la réutilisation du code.
- Assure une initialisation et un nettoyage corrects.

INCONVÉNIENTS

- Peut compliquer la hiérarchie des classes.
- Peut introduire des dépendances non souhaitées.

ACCÈS AUX MEMBRES DE LA CLASSE DE BASE

DÉFINITION

L'accès aux membres de la classe de base en C++ dépend du type d'héritage. Les membres peuvent être publics, privés ou protégés. Le type d'héritage détermine comment les membres de la base sont accessibles dans la classe dérivée.

SYNTAXE

La syntaxe de base pour l'héritage est :

```
class Derived : access_specifier Base {  
    // Corps de la classe dérivée  
};
```

`access_specifier` peut être `public`, `private` ou `protected`.

MEMBRES PUBLICS

Les membres publics de la classe de base sont accessibles partout. Ils peuvent être accédés directement par les instances de la classe dérivée. Exemple :

```
class Base {  
public:  
    int publicMember;  
};
```

MEMBRES PRIVÉS

Les membres privés de la classe de base ne sont accessibles que dans la classe de base. Ils ne peuvent pas être accédés directement par la classe dérivée. Exemple :

```
class Base {  
private:  
    int privateMember;  
};
```

MEMBRES PROTÉGÉS

Les membres protégés de la classe de base sont accessibles dans la classe de base et dans les classes dérivées. Ils ne sont pas accessibles en dehors de ces classes. Exemple :

```
class Base {  
protected:  
    int protectedMember;  
};
```

ACCÈS AVEC "PUBLIC"

Lors de l'héritage public, les membres publics restent publics dans la classe dérivée. Les membres protégés restent protégés. Les membres privés restent inaccessibles.

ACCÈS AVEC "PRIVATE"

Lors de l'héritage privé, tous les membres de la classe de base deviennent privés dans la classe dérivée. Les membres publics et protégés ne sont plus accessibles directement.

ACCÈS AVEC "PROTECTED"

Lors de l'héritage protégé, les membres publics et protégés de la classe de base deviennent protégés dans la classe dérivée. Les membres privés restent inaccessibles.

EXEMPLE D'ACCÈS

```
class Base {
public:
    int publicMember;
protected:
    int protectedMember;
private:
    int privateMember;
};

class Derived : public Base {
public:
    void accessMembers() {
        publicMember = 1;    // Accessible
        protectedMember = 2; // Accessible
        // privateMember = 3; // Inaccessible
    }
}
```

MEILLEURES PRATIQUES

- Utilisez l'héritage public pour un comportement "est un".
- Utilisez l'héritage privé pour un comportement "implémenté en termes de".
- Utilisez l'héritage protégé pour partager des implémentations sans exposer l'interface.
- Préférez la composition à l'héritage lorsque cela est possible.

HÉRITAGE PUBLIC, PRIVÉ, ET PROTÉGÉ

HÉRITAGE PUBLIC

En héritage public, les membres publics de la classe de base restent publics. Les membres protégés de la classe de base restent protégés. Les membres privés ne sont pas accessibles directement. Syntaxe :

```
class Derived : public Base {  
    // Membres et méthodes  
};
```

HÉRITAGE PRIVÉ

En héritage privé, les membres publics de la classe de base deviennent privés. Les membres protégés de la classe de base deviennent privés. Les membres privés ne sont pas accessibles directement. Syntaxe :

```
class Derived : private Base {  
    // Membres et méthodes  
};
```

HÉRITAGE PROTÉGÉ

En héritage protégé, les membres publics de la classe de base deviennent protégés. Les membres protégés de la classe de base restent protégés. Les membres privés ne sont pas accessibles directement. Syntaxe :

```
class Derived : protected Base {  
    // Membres et méthodes  
};
```

DIFFÉRENCES ENTRE LES TYPES D'HÉRITAGE

Type d'héritage	Membres publics	Membres protégés	Membres privés
Public	Public	Protégé	Inaccessible
Privé	Privé	Privé	Inaccessible
Protégé	Protégé	Protégé	Inaccessible

ACCÈS AUX MEMBRES SELON LE TYPE D'HÉRITAGE

Type d'héritage	Accès aux membres publics	Accès aux membres protégés	Accès aux membres privés
Public	Oui	Oui	Non
Privé	Non	Non	Non
Protégé	Non	Oui	Non

UTILISATION PRATIQUE DE L'HÉRITAGE PUBLIC

L'héritage public est utilisé lorsque la relation "est un" s'applique. Exemple :

```
class Animal {  
public:  
    void breathe();  
};  
  
class Dog : public Animal {  
public:  
    void bark();  
};
```

UTILISATION PRATIQUE DE L'HÉRITAGE PRIVÉ

L'héritage privé est utilisé pour la réutilisation de code sans exposer l'interface de base. Exemple :

```
class Engine {
public:
    void start();
};

class Car : private Engine {
public:
    void drive();
};
```

UTILISATION PRATIQUE DE L'HÉRITAGE PROTÉGÉ

L'héritage protégé est utilisé lorsqu'on veut protéger l'accès aux membres de base. Exemple :

```
class Person {
protected:
    void sleep();
};

class Employee : protected Person {
public:
    void work();
};
```

REDÉFINITION DE MÉTHODES

DÉFINITION

La redéfinition de méthodes en C++ permet à une classe dérivée de fournir une implémentation spécifique d'une méthode déjà définie dans sa classe de base. Cela permet de modifier ou d'étendre le comportement hérité.

POURQUOI REDÉFINIR UNE MÉTHODE

- Adapter le comportement hérité aux besoins spécifiques de la classe dérivée.
- Ajouter des fonctionnalités supplémentaires.
- Modifier l'implémentation pour optimiser les performances.
- Personnaliser le comportement pour des cas d'utilisation spécifiques.

SYNTAXE DE REDÉFINITION

Pour redéfinir une méthode, la méthode dans la classe dérivée doit avoir la même signature que celle de la classe de base.

```
class Base {  
public:  
    virtual void display();  
};  
  
class Derived : public Base {  
public:  
    void display() override;  
};
```

EXAMPLE SIMPLE

```
class Base {
public:
    virtual void greet() {
        std::cout << "Hello from Base" << std::endl;
    }
};

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

int main() {
```

REDÉFINITION DANS UNE CLASSE DÉRIVÉE

La classe dérivée peut redéfinir une méthode de la classe de base en utilisant le mot-clé `override` pour assurer que la méthode est correctement redéfinie.

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

ACCÈS AUX MÉTHODES REDÉFINIES

Pour appeler la méthode redéfinie de la classe de base à partir de la classe dérivée, utilisez l'opérateur de portée ::.

```
class Derived : public Base {  
public:  
    void display() override {  
        Base::display(); // Appelle la méthode de la classe de base  
        std::cout << "Derived display" << std::endl;  
    }  
};
```

REDÉFINITION VS SURCHARGE

- **Redéfinition** : Modifier une méthode héritée avec la même signature.
- **Surcharge** : Déclarer plusieurs méthodes avec le même nom mais des signatures différentes dans la même classe.

CONSÉQUENCES DE LA REDÉFINITION

- La méthode redéfinie dans la classe dérivée remplace celle de la classe de base.
- Les objets de type base utilisant des pointeurs ou des références peuvent appeler la méthode redéfinie si elle est virtuelle.
- Permet une meilleure personnalisation et extension des classes de base.

UTILISATION DE LA CLÉ DE VOÛTE "VIRTUAL"

DÉFINITION DE "VIRTUAL"

La clé de voûte "virtual" en C++ est utilisée pour permettre la redéfinition de méthodes dans les classes dérivées. Elle permet d'assurer que la méthode redéfinie dans une classe dérivée est appelée, même si l'objet est manipulé par un pointeur ou une référence de la classe de base. Une méthode déclarée avec le mot-clé "virtual" dans une classe de base est appelée méthode virtuelle.

UTILITÉ DE "VIRTUAL"

"Virtual" permet l'implémentation du polymorphisme en C++. Elle permet aux classes dérivées de redéfinir les méthodes de la classe de base. Cela permet d'écrire du code plus flexible et extensible. Elle est essentielle pour la création de hiérarchies de classes polymorphiques.

SYNTAXE DE "VIRTUAL"

La syntaxe pour déclarer une méthode virtuelle est la suivante :

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

La méthode "afficher" est maintenant virtuelle et peut être redéfinie dans les classes dérivées.

EXEMPLE SIMPLE AVEC "VIRTUAL"

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

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

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

"VIRTUAL" ET POLYMORPHISME

Le polymorphisme permet d'utiliser une interface commune pour différents types dérivés. Les méthodes virtuelles permettent de réaliser ce polymorphisme. En utilisant des pointeurs ou des références de la classe de base, les méthodes appropriées des classes dérivées sont appelées. Cela permet de traiter les objets de manière uniforme tout en utilisant leurs comportements spécifiques.

MÉTHODES VIRTUELLES PURES

Une méthode virtuelle pure est déclarée en ajoutant "= 0" à la déclaration de la méthode. Elle rend la classe abstraite, ce qui signifie qu'on ne peut pas instancier directement cette classe. Exemple :

```
class Base {  
public:  
    virtual void afficher() = 0;  
};
```

Les classes dérivées doivent implémenter cette méthode.

AVANTAGES DE "VIRTUAL"

- Permet le polymorphisme.
- Facilite la gestion des hiérarchies de classes complexes.
- Rend le code plus flexible et extensible.
- Permet la réutilisation du code.

INCONVÉNIENTS DE "VIRTUAL"

- Peut introduire une légère surcharge en termes de performance.
- Peut rendre le code plus difficile à comprendre et à maintenir.
- Les erreurs liées aux méthodes virtuelles peuvent être difficiles à diagnostiquer.
- Nécessite une compréhension approfondie de l'héritage et du polymorphisme.

SURCHARGE DES OPÉRATEURS EN HÉRITAGE

DÉFINITION

La surcharge d'opérateurs permet de redéfinir le comportement des opérateurs pour des objets de classes définies par l'utilisateur. Elle permet d'utiliser des opérateurs comme +, -, *, etc., avec des objets de vos classes.

POURQUOI SURCHARGER DES OPÉRATEURS

- Simplifie l'utilisation des objets.
- Rend le code plus lisible et intuitif.
- Permet des opérations naturelles sur des objets complexes.
- Facilite la maintenance et l'extension du code.

SYNTAXE DE SURCHARGE D'OPÉRATEURS

La syntaxe de surcharge d'opérateurs en C++ est la suivante :

```
class Classe {  
public:  
    Type operator+(const Classe& autre) {  
        // Implémentation  
    }  
};
```

EXEMPLE DE SURCHARGE D'OPÉRATEURS

```
class Complex {
public:
    int real, imag;
    Complex operator+(const Complex& autre) {
        Complex temp;
        temp.real = real + autre.real;
        temp.imag = imag + autre.imag;
        return temp;
    }
};
```

SURCHARGE D'OPÉRATEURS ET HÉRITAGE

- Les opérateurs surchargés peuvent être hérités.
- Les classes dérivées peuvent redéfinir les opérateurs surchargés.
- Utiliser `virtual` pour permettre la surcharge polymorphe.

OPÉRATEURS COURAMMENT SURCHARGÉS

- Opérateurs arithmétiques : +, -, *, /
- Opérateurs de comparaison : ==, !=, <, >
- Opérateurs d'affectation : =, +=, -=
- Opérateurs d'accès : [], ->

BONNES PRATIQUES

- Surcharger les opérateurs seulement si cela a du sens.
- Maintenir la sémantique des opérateurs.
- Utiliser des fonctions membres pour les opérateurs d'affectation.
- Documenter clairement la surcharge.

AVANTAGES

- Rend le code plus naturel et intuitif.
- Simplifie les opérations sur des objets complexes.

INCONVÉNIENTS

- Peut rendre le code difficile à comprendre.
- Peut introduire des erreurs si mal utilisé.
- Peut compliquer la maintenance.

HÉRITAGE MULTIPLE EN C++

DÉFINITION

L'héritage multiple en C++ permet à une classe de dériver de plusieurs classes de base. Cela signifie qu'une classe dérivée peut hériter des membres de plusieurs classes. L'héritage multiple est utile pour combiner les fonctionnalités de plusieurs classes. Cependant, il peut aussi introduire des complexités supplémentaires.

QUAND L'UTILISER

Utilisez l'héritage multiple lorsque :

- Vous avez besoin de combiner les fonctionnalités de plusieurs classes.
- Vous souhaitez éviter la duplication de code.
- Les classes de base sont indépendantes les unes des autres.
- Vous avez une conception claire pour gérer les conflits potentiels.

SYNTAXE

La syntaxe pour l'héritage multiple en C++ est la suivante :

```
class Derived : public Base1, public Base2 {  
    // Membres de la classe dérivée  
};
```

EXEMPLE SIMPLE

```
class Basel {
public:
    void function1() {}
};

class Base2 {
public:
    void function2() {}
};

class Derived : public Basel, public Base2 {
    // Utilise les membres de Basel et Base2
};
```

HÉRITAGE MULTIPLE AVEC CLASSES DE BASE

Lorsque vous utilisez l'héritage multiple, chaque classe de base doit être spécifiée :

```
class Derived : public Base1, public Base2 {  
    // Membres de la classe dérivée  
};
```

La classe dérivée peut accéder aux membres publics et protégés des classes de base.

GESTION DES CONFLITS DE NOM

Les conflits de nom peuvent survenir si les classes de base ont des membres avec le même nom. Pour résoudre ces conflits, utilisez l'opérateur de résolution de portée.

```
Derived d;  
d.Base1::function();  
d.Base2::function();
```

UTILISATION DE VIRTUAL POUR ÉVITER LES CONFLITS

Le mot-clé `virtual` peut être utilisé pour éviter les conflits dans l'héritage multiple. Il permet de spécifier une relation d'héritage virtuelle.

```
class Basel {
public:
    virtual void function() {}
};

class Derived : public virtual Basel, public virtual Base2 {
    // Membres de la classe dérivée
};
```


AVANTAGES

- Combinaison de fonctionnalités.
- Réduction de la duplication de code.
- Flexibilité dans la conception.

INCONVÉNIENTS

- Complexité accrue.
- Risque de conflits de nom.
- Difficulté de maintenance et de débogage.

PROBLÈMES DE DIAMANT ET SOLUTIONS

DÉFINITION DU PROBLÈME DE DIAMANT

Le problème de diamant survient lors de l'héritage multiple. Il se produit lorsque deux classes dérivent d'une même classe de base. Une quatrième classe hérite de ces deux classes dérivées. Cela forme une structure en diamant. Les ambiguïtés apparaissent lors de l'accès aux membres de la classe de base.

CONSÉQUENCES DU PROBLÈME DE DIAMANT

Ambiguïté lors de l'appel des méthodes de la classe de base. Redondance des membres de la classe de base. Difficulté de maintenance du code. Problèmes de performance et de mémoire.

SOLUTION : UTILISATION DE LA CLASSE VIRTUELLE

Utiliser l'héritage virtuel pour résoudre le problème de diamant. L'héritage virtuel permet de partager une seule instance de la classe de base. Réduit les ambiguïtés et les redondances. Facilite la maintenance et améliore les performances.

EXEMPLE DE SOLUTION AVEC CLASSE VIRTUELLE

```
class A {
public:
    void show() { cout << "A"; }
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D obj;
    obj.show(); // Affiche "A"
}
```

AVANTAGES DE LA CLASSE VIRTUELLE

Évite les ambiguïtés lors de l'héritage multiple. Réduit la duplication de la classe de base. Améliore la lisibilité et la maintenance du code. Optimise l'utilisation de la mémoire.

LIMITATIONS DE LA CLASSE VIRTUELLE

Peut augmenter la complexité du code. Peut introduire une légère surcharge de performance. Nécessite une compréhension approfondie de l'héritage en C++. Peut ne pas être nécessaire pour des hiérarchies simples.

CLASSES ABSTRAITES ET MÉTHODES VIRTUELLES PURES

DÉFINITION

Une classe abstraite est une classe qui ne peut pas être instanciée. Elle sert de modèle pour d'autres classes. Elle peut contenir des méthodes virtuelles pures. Elle peut aussi contenir des méthodes avec une implémentation.

UTILITÉ DES CLASSES ABSTRAITES

Permet de définir une interface commune pour un groupe de classes. Facilite la gestion des relations entre classes. Encourage la réutilisation de code. Aide à organiser le code de manière structurée.

MÉTHODES VIRTUELLES PURES : DÉFINITION

Une méthode virtuelle pure est une méthode sans implémentation. Elle est déclarée dans une classe abstraite. Elle doit être redéfinie dans les classes dérivées. Elle force les classes dérivées à implémenter cette méthode.

SYNTAXE DES CLASSES ABSTRAITES

```
class AbstractClass {  
public:  
    virtual void pureVirtualMethod() = 0;  
};
```

SYNTAXE DES MÉTHODES VIRTUELLES PURES

```
class AbstractClass {  
public:  
    virtual void pureVirtualMethod() = 0;  
};
```


AVANTAGES DES CLASSES ABSTRAITES

Encouragent la conception orientée objet. Facilitent la maintenance du code. Permettent de définir des interfaces claires. Favorisent la réutilisation du code.

LIMITATIONS DES CLASSES ABSTRAITES

Ne peuvent pas être instanciées. Peuvent rendre le code plus complexe. Nécessitent une bonne compréhension des concepts OOP. Peuvent compliquer le débogage.