INTRODUCTION AUX CLASSES EN C++

DEFINITION

Un constructeur par défaut est une fonction membre spéciale d'une classe. Il est appelé lorsqu'un objet de la classe est créé sans arguments. Le constructeur par défaut initialise les membres de données de l'objet. Il peut être défini explicitement par le programmeur. S'il n'est pas défini, le compilateur génère un constructeur par défaut implicite. Le constructeur par défaut n'a pas de paramètres.

SYNTAXE

La syntaxe d'un constructeur par défaut est :

```
class MaClasse {
public:
    MaClasse(); // Déclaration du constructeur par défaut
};
```

L'implémentation peut être faite dans le fichier source :

```
MaClasse::MaClasse() {
    // Initialisation des membres de données
}
```

EXEMPLE

Voici un exemple de constructeur par défaut :

```
class Voiture {
public:
    Voiture() {
        marque = "Inconnue";
        annee = 0;
    }
private:
    std::string marque;
    int annee;
};
```

Dans cet exemple, le constructeur par défaut initialise `marque` à "Inconnue" et `annee` à 0.

ROLE DU CONSTRUCTEUR PAR DEFAUT

- Initialiser les membres de données avec des valeurs par défaut.
- Garantir que l'objet est dans un état cohérent dès sa création.
- Permettre la création d'objets sans fournir d'arguments.
- Faciliter l'utilisation des classes dans des conteneurs et autres structures.

AVANTAGES ET INCONVENIENTS

AVANTAGES

- Simplicité d'utilisation.
- Assure l'initialisation des objets.
- Facilite la gestion de la mémoire.

INCONVÉNIENTS

- Peut entraîner des valeurs par défaut non souhaitées.
- Moins de contrôle sur l'initialisation des objets.
- Peut masquer des erreurs de conception.

CONSTRUCTEUR PAR DEFAUT IMPLICITE VS EXPLICITE

IMPLICITE

- Généré automatiquement par le compilateur.
- N'initialise pas les membres de données de type fondamental.

EXPLICITE

- Défini par le programmeur.
- Permet d'initialiser explicitement les membres de données.
- Offre plus de contrôle sur l'initialisation.

MEILLEURES PRATIQUES

- Toujours définir un constructeur par défaut explicite si des valeurs spécifiques sont nécessaires.
- Utiliser des listes d'initialisation pour les membres de données.
- Éviter les initialisations coûteuses dans le constructeur par défaut.
- Documenter le comportement du constructeur par défaut.
- Tester l'initialisation par défaut pour s'assurer de la cohérence de l'objet.

CONSTRUCTEUR PAR DÉFAUT

DÉFINITION

Un constructeur par défaut est un constructeur qui ne prend aucun argument. Il est appelé automatiquement lors de la création d'un objet. Il initialise les membres de la classe avec des valeurs par défaut.

QUAND L'UTILISER

Utilisez un constructeur par défaut lorsque vous souhaitez initialiser un objet avec des valeurs par défaut. Il est utile pour créer des objets sans spécifier de valeurs initiales. Il permet de simplifier la création d'objets.

SYNTAXE

La syntaxe d'un constructeur par défaut en C++ est la suivante :

```cpp
class MaClasse {
public:
    MaClasse() {
        // Initialisation par défaut
    }
};
```

EXEMPLE

```
class Personne {
public:
    Personne() {
        nom = "Inconnu";
        age = 0;
    }
private:
    std::string nom;
    int age;
};

Personne p; // Utilise le constructeur par défaut
```

AVANTAGES

- Simplifie la création d'objets.
- Fournit des valeurs initiales par défaut.
- Facilite la maintenance du code.

LIMITATIONS

- Ne permet pas de spécifier des valeurs initiales différentes.
- Peut nécessiter des modifications si les exigences de l'initialisation changent.
- Moins flexible que les constructeurs avec paramètres.

COMPARAISON AVEC AUTRES CONSTRUCTEURS

Type de constructeur	Description
Par défaut	Aucun argument, valeurs par défaut
Avec paramètres	Prend des arguments, plus flexible
De copie	Crée un objet à partir d'un autre
De déplacement	Optimise les performances

CONSTRUCTEUR AVEC PARAMÈTRES

DÉFINITION

Un constructeur avec paramètres initialise un objet avec des valeurs spécifiques. Il permet de passer des arguments lors de la création de l'objet. Il est utilisé pour initialiser les membres de la classe avec des valeurs fournies par l'utilisateur.

QUAND L'UTILISER

- Lorsque des valeurs spécifiques sont nécessaires pour initialiser un objet.
- Pour éviter des appels de fonctions supplémentaires après la création de l'objet.
- Pour garantir que l'objet est toujours dans un état valide dès sa création.

SYNTAXE

La syntaxe d'un constructeur avec paramètres en C++ est la suivante :

```
class NomDeClasse {
public:
    NomDeClasse(Type param1, Type param2);
};
```

EXEMPLE

```
class Point {
public:
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
private:
    int x, y;
};
```

INITIALISATION DES MEMBRES

Il est possible d'initialiser les membres directement dans la liste d'initialisation :

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

CONSTRUCTEUR AVEC PARAMÈTRES PAR DÉFAUT

Un constructeur peut avoir des paramètres par défaut :

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

AVANTAGES ET INCONVÉNIENTS

Avantages :

- Initialisation en une seule étape.
- Garantie de l'état valide de l'objet dès sa création.

Inconvénients :

- Complexité accrue si plusieurs constructeurs avec différents paramètres.
- Peut rendre le code moins lisible si trop de paramètres par défaut.

DESTRUCTEUR

DÉFINITION

Un destructeur est une fonction membre spéciale d'une classe. Il est appelé automatiquement lorsqu'un objet de cette classe est détruit. Le destructeur a le même nom que la classe, précédé d'un tilde (~). Il ne prend aucun paramètre et ne retourne rien.

QUAND L'UTILISER

Le destructeur est utilisé pour libérer les ressources allouées par l'objet. Il est particulièrement utile pour gérer la mémoire dynamique. Il est également utilisé pour fermer des fichiers ou libérer des connexions réseau. Le destructeur est appelé automatiquement à la fin de la durée de vie d'un objet.

SYNTAXE

La syntaxe d'un destructeur en C++ est la suivante :

```
class NomDeClasse {
public:
    ~NomDeClasse();
};
```

EXEMPLE

Voici un exemple de destructeur dans une classe :

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

RÔLE DU DESTRUCTEUR

Le destructeur nettoie les ressources utilisées par l'objet. Il libère la mémoire allouée dynamiquement. Il ferme les fichiers ouverts par l'objet. Il libère les connexions réseau établies par l'objet.

DESTRUCTEUR PAR DÉFAUT

Si aucun destructeur n'est défini, le compilateur en génère un par défaut. Le destructeur par défaut ne fait rien de spécifique. Il est suffisant si la classe n'utilise pas de ressources dynamiques.

DESTRUCTEUR PERSONNALISÉ

Un destructeur personnalisé est nécessaire si la classe alloue des ressources dynamiques. Il permet de libérer explicitement ces ressources. Il peut également inclure du code de nettoyage spécifique à l'application.

GESTION DE LA MÉMOIRE

Le destructeur est crucial pour éviter les fuites de mémoire. Il doit libérer toute mémoire allouée dynamiquement par l'objet. Utilisez `delete` ou `delete[]` pour libérer la mémoire.

BONNES PRATIQUES

Toujours définir un destructeur si la classe gère des ressources dynamiques. Assurez-vous que le destructeur est public. Évitez les exceptions dans le destructeur. Utilisez des smart pointers pour une gestion automatique de la mémoire.

OPÉRATEUR D'ASSIGNATION

DÉFINITION

L'opérateur d'assignation (`operator=`) permet de copier les valeurs d'un objet à un autre. Il est utilisé pour assigner les valeurs d'un objet existant à un autre objet déjà initialisé. Il est souvent redéfini pour gérer correctement les ressources dynamiques.

QUAND L'UTILISER

L'opérateur d'assignation est utilisé quand :

- Vous devez copier les valeurs d'un objet à un autre déjà existant.
- Vous gérez des ressources dynamiques comme la mémoire.
- Vous devez éviter les fuites de mémoire ou les doubles libérations.

SYNTAXE

La syntaxe de l'opérateur d'assignation est la suivante :

```
ClassName& operator=(const ClassName& other) {
    // logique d'assignation
    return *this;
}
```

EXEMPLE

```cpp
class MyClass {
public:
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            // libérer les ressources actuelles
            // copier les ressources de 'other'
        }
        return *this;
    }
};
```

DIFFÉRENCE AVEC L'OPÉRATEUR DE COPIE

Opérateur	Fonction
Opérateur de copie	Initialise un nouvel objet avec les valeurs d'un autre.
Opérateur d'assignation	Copie les valeurs d'un objet à un autre déjà existant.

GESTION DE LA MÉMOIRE

Lors de la redéfinition de l'opérateur d'assignation :

- Libérez les ressources actuelles de l'objet.
- Allouez et copiez les nouvelles ressources de l'objet source.
- Évitez les fuites de mémoire et les doubles libérations.

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- Contrôle précis sur la copie des ressources.
- Évite les fuites de mémoire.

INCONVÉNIENTS

- Complexité accrue.
- Risque de bugs si mal implémenté.

OPÉRATEUR DE COPIE

DÉFINITION

L'opérateur de copie est une fonction membre spéciale. Il permet de créer une nouvelle instance d'une classe. Cette nouvelle instance est une copie d'une instance existante. Il est souvent appelé constructeur de copie. Il est utilisé pour copier les attributs d'un objet à un autre.

QUAND L'UTILISER

L'opérateur de copie est utilisé quand :

- On souhaite dupliquer un objet existant.
- On passe un objet par valeur à une fonction.
- On retourne un objet par valeur depuis une fonction.
- On veut éviter les effets de bord dus au partage de pointeurs.

SYNTAXE

La syntaxe de l'opérateur de copie est la suivante :

```
ClassName(const ClassName &other);
```

EXEMPLE

Voici un exemple d'opérateur de copie :

```
class MyClass {
public:
    MyClass(const MyClass &other) {
        // Copier les attributs de 'other' ici
    }
};
```

DIFFÉRENCE AVEC L'OPÉRATEUR D'ASSIGNATION

Aspect	Opérateur de copie	Opérateur d'assignation
Utilisation	Lors de la création d'un nouvel objet	Lors de la réaffectation d'un objet
Syntaxe	`ClassName(const ClassName &other)`	`ClassName& operator=(const ClassName &other)`
Moment d'appel	À l'initialisation	Après l'initialisation

CAS D'UTILISATION COURANTS

- Copie d'un objet pour éviter les effets de bord.
- Passage d'un objet par valeur à une fonction.
- Retour d'un objet par valeur depuis une fonction.
- Création d'objets temporaires.

BONNES PRATIQUES

- Toujours définir un opérateur de copie si la classe gère des ressources dynamiques.
- Utiliser `const` pour le paramètre de l'opérateur de copie.
- Vérifier l'auto-copie (`if (this != &other)`).
- Libérer les ressources existantes avant de copier.

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- Facilite la duplication d'objets.
- Évite les effets de bord dus au partage de ressources.

INCONVÉNIENTS

- Peut être coûteux en termes de performance.
- Nécessite une gestion manuelle des ressources pour éviter les fuites de mémoire.

INITIALISATION DES MEMBRES

DÉFINITION

L'initialisation des membres d'une classe en C++ se réfère à la manière dont les variables membres sont assignées à des valeurs initiales. Elle peut se faire de différentes manières, notamment dans le constructeur, via une liste d'initialisation, ou par des méthodes spécifiques comme la valeur, la référence, le pointeur, etc.

SYNTAXE

La syntaxe de base pour l'initialisation des membres dans le constructeur est :

```
class MaClasse {
    int a;
public:
    MaClasse(int val) : a(val) {}
};
```

INITIALISATION DANS LE CONSTRUCTEUR

L'initialisation des membres peut se faire directement dans le corps du constructeur :

```
class MaClasse {
    int a;
public:
    MaClasse(int val) {
        a = val;
    }
};
```

LISTE D'INITIALISATION DES MEMBRES

La liste d'initialisation des membres permet d'initialiser les membres avant l'exécution du corps du constructeur :

```
class MaClasse {
    int a;
public:
    MaClasse(int val) : a(val) {}
};
```

INITIALISATION PAR VALEUR

L'initialisation par valeur assigne une copie de la valeur fournie au membre :

```
class MaClasse {
    int a;
public:
    MaClasse(int val) : a(val) {}
};
```

INITIALISATION PAR RÉFÉRENCE

L'initialisation par référence utilise une référence à une variable existante :

```
class MaClasse {
    int& ref;
public:
    MaClasse(int& val) : ref(val) {}
};
```

INITIALISATION PAR POINTEUR

L'initialisation par pointeur assigne une adresse mémoire au membre pointeur :

```
class MaClasse {
    int* ptr;
public:
    MaClasse(int* val) : ptr(val) {}
};
```

INITIALISATION PAR DÉFAUT

L'initialisation par défaut assigne une valeur par défaut au membre :

```
class MaClasse {
    int a = 0;
public:
    MaClasse() {}
};
```

INITIALISATION CONSTANTE

Les membres constants doivent être initialisés dans la liste d'initialisation :

```
class MaClasse {
    const int a;
public:
    MaClasse(int val) : a(val) {}
};
```

EXEMPLES

```
class MaClasse {
    int a;
    int& ref;
    const int c;
public:
    MaClasse(int val, int& r) : a(val), ref(r), c(100) {}
};
```

GESTION DE LA MÉMOIRE

DEFINITION

Les classes canoniques en C++ sont des classes qui implémentent les quatre fonctions spéciales suivantes :

- Constructeur
- Destructeur
- Opérateur de copie
- Opérateur d'assignation

CONSTRUCTEUR

Un constructeur initialise les objets d'une classe. Il a le même nom que la classe et n'a pas de type de retour.

```
class MyClass {
public:
    MyClass() {
        // Initialisation
    }
};
```

DESTRUCTEUR

Un destructeur libère les ressources lorsqu'un objet est détruit. Il a le même nom que la classe précédé d'un tilde (~) et n'a pas de type de retour.

```
class MyClass {
public:
    ~MyClass() {
        // Libération des ressources
    }
};
```

OPERATEUR DE COPIE

L'opérateur de copie permet de créer une copie d'un objet existant. Il prend un objet de la même classe en paramètre.

```
class MyClass {
public:
    MyClass(const MyClass& other) {
        // Copie des membres
    }
};
```

OPERATEUR D'ASSIGNATION

L'opérateur d'assignation permet d'assigner un objet à un autre. Il retourne une référence à l'objet courant.

```
class MyClass {
public:
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            // Assignation des membres
        }
        return *this;
    }
};
```

GESTION DES RESSOURCES

La gestion des ressources dans une classe canonique inclut :

- Allocation dynamique de mémoire
- Libération de mémoire dans le destructeur
- Copie et assignation sécurisées des ressources

EXEMPLE DE CLASSE CANONIQUE

```cpp
class MyClass {
private:
    int* data;
public:
    MyClass() : data(new int[10]) {}
    ~MyClass() { delete[] data; }
    MyClass(const MyClass& other) : data(new int[10]) {
        std::copy(other.data, other.data + 10, data);
    }
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {
            delete[] data;
            data = new int[10];
            std::copy(other.data, other.data + 10, data);
        }
        return *this;
```

AVANTAGES ET INCONVENIENTS

Avantages :

- Contrôle total sur la gestion des ressources
- Prévention des fuites mémoire

Inconvénients :

- Code plus complexe et verbeux
- Risque de duplication de code

ENCAPSULATION

DÉFINITION

L'encapsulation est un concept de programmation orientée objet. Elle consiste à regrouper les données et les méthodes qui agissent sur ces données au sein d'une même classe. Elle permet de cacher les détails d'implémentation et de ne montrer que les fonctionnalités essentielles.

OBJECTIFS DE L'ENCAPSULATION

- Protéger les données des accès non autorisés.
- Faciliter la maintenance et l'évolution du code.
- Améliorer la modularité du programme.
- Réduire les risques d'erreurs en limitant les interactions directes avec les données.

ACCÈS AUX MEMBRES

Les membres d'une classe peuvent être :

- Public : accessibles depuis l'extérieur de la classe.
- Private : accessibles uniquement depuis l'intérieur de la classe.
- Protected : accessibles depuis l'intérieur de la classe et des classes dérivées.

MODIFICATEURS D'ACCÈS (PUBLIC, PRIVATE, PROTECTED)

- **Public** :

```
public:
    int publicVar;
```

- **Private** :

```
private:
    int privateVar;
```

- **Protected** :

```
protected:
    int protectedVar;
```

GETTERS ET SETTERS

Les getters et setters sont des méthodes pour accéder et modifier les données privées. Ils permettent de contrôler l'accès aux données.

```
class MyClass {
private:
    int value;
public:
    int getValue() { return value; }
    void setValue(int v) { value = v; }
};
```

AVANTAGES DE L'ENCAPSULATION

- **Sécurité** : protège les données des modifications non autorisées.
- **Modularité** : facilite la division du code en modules indépendants.
- **Maintenance** : simplifie la mise à jour et la correction du code.
- **Réutilisabilité** : permet de réutiliser des classes sans connaître leur implémentation interne.

EXEMPLES D'ENCAPSULATION EN C++

```cpp
class Person {
private:
    std::string name;
    int age;
public:
    void setName(std::string n) { name = n; }
    std::string getName() { return name; }
    void setAge(int a) { age = a; }
    int getAge() { return age; }
};
```

BONNES PRATIQUES D'ENCAPSULATION

- Toujours utiliser des modificateurs d'accès appropriés.
- Utiliser des getters et setters pour accéder aux données privées.
- Éviter de rendre les membres de données publics.
- Garder les méthodes qui ne sont pas destinées à être utilisées en dehors de la classe en privé.
- Documenter les interfaces publiques pour faciliter leur utilisation.