PROGRAMMATION ORIENTÉE OBJET EN C++

INTRODUCTION AUX CLASSES

DÉFINITION D'UNE CLASSE

Une **classe** est un modèle qui permet de créer des **objets** ayant des **attributs** et des **méthodes** communs.

```
class NomDeLaClasse {
  public:
    // Attributs
    int attribut1;

    // Méthodes
    void methode1() {
      // Code de la méthode
    }
};
```

SYNTAXE

```
class NomDeLaClasse {
public:
    // Attributs et méthodes
};
```

EXEMPLES D'UTILISATION

```
class Voiture {
    int vitesse;
    void accelerer();
};
```

Dans cet exemple, une classe **Voiture** est définie avec un attribut **vitesse** et une méthode **accelerer**.

ATTRIBUTS ET MÉTHODES

Les **attributs** sont des variables associées à chaque objet d'une classe. Les **méthodes** sont des fonctions applicables sur les objets.

```
class Exemple {
    // attributs
    int nombre;

    // méthodes
    void modifierNombre(int n) {
        nombre = n;
    }

    int obtenirNombre() {
        return nombre;
    }
};
```

Les attributs stockent des informations sur l'objet, tandis que les méthodes peuvent accéder ou modifier ces informations.

SYNTAXE

```
class NomDeLaClasse {
public:
    typeAttribut attribut;
    returnType nomMethode(arguments);
};
```

EXEMPLES D'UTILISATION

```
class Voiture {
public:
    int vitesse;
    void accelerer() {
        vitesse += 10;
    }
};
```

Cet exemple montre comment créer une **classe** Voiture avec un attribut `vitesse` et une méthode `accelerer` pour augmenter la vitesse de 10.

CONSTRUCTEUR

Un **constructeur** est une méthode spéciale qui s'exécute lors de la création d'un nouvel objet d'une **classe**.

```
class MyClass {
  public:
    MyClass() {
      // Code du constructeur
    }
};

int main() {
  MyClass obj; // Appel du constructeur
  return 0;
}
```

Les constructeurs peuvent prendre des paramètres pour initialiser les attributs de l'objet.

SYNTAXE

```
class NomDeLaClasse {
public:
    NomDeLaClasse(arguments) {
        // Initialisation des attributs
    }
};
```

Cette syntaxe montre comment définir une **classe** en C++ et comment déclarer un **constructeur** pour initialiser les attributs de la classe.

EXEMPLES D'UTILISATION

```
class Voiture {
public:
    Voiture() {
        vitesse = 0;
    }
    int vitesse;
    void accelerer();
};
```

Cette slide présente un exemple de déclaration d'une classe "Voiture" avec un constructeur, une variable membre "vitesse" et une fonction membre accelerer().

EXEMPLE 2

```
#include <iostream>
#include <string>// Définition de la classe "Personne"

class Personne {
public:
    // Constructeur de la classe
    Personne(std::string nom, int age) {
        // Initialisation des attributs de la classe
        this->nom = nom;
        this->age = age;
    }

    // Méthode pour afficher les détails de la personne
    void afficherDetails() {
        std::cout << "Nom : " << nom << std::endl;
```

Dans cet exemple, nous avons créé une classe appelée **`Personne`** avec un constructeur qui prend un nom et un âge en paramètres pour initialiser les attributs **`nom`** et **`age`** de la classe. La classe contient également deux

DESTRUCTEUR

Un **destructeur** est une méthode spéciale qui s'exécute lors de la **destruction** d'un objet d'une classe.

Syntaxe :

```
class NomDeLaClasse {
  public:
    // Constructeur
    NomDeLaClasse() {
      // Code du constructeur
    }

    // Destructeur
    ~NomDeLaClasse() {
      // Code du destructeur
    }
};
```

Le destructeur est automatiquement appelé à la fin du programme ou lorsqu'un objet ne peut plus être utilisé. Il permet de libérer les ressources, comme la mémoire allouée.

SYNTAXE

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

EXEMPLES D'UTILISATION

```
class Voiture {
public:
    Voiture();
    ~Voiture() {
        // Pas de ressources à libérer
    }
    int vitesse;
    void accelerer();
};
```

ENCAPSULATION

ENCAPSULATION

L'**encapsulation** est un mécanisme pour regrouper les données (attributs) et les opérations (méthodes) sur les données dans une seule unité appelée **classe**.

Il permet de contrôler l'accès aux attributs en les rendant "**privés**" et en fournissant des méthodes **publiques** pour les manipuler.

L'encapsulation facilite la maintenance et la réutilisation du code en séparant les détails d'implémentation des interfaces publiques.

PUBLIC, PRIVÉ ET PROTÉGÉ

SYNTAXE

```
class MaClasse {
public:
    // Attributs et méthodes publiques
protected:
    // Attributs et méthodes protégées
private:
    // Attributs et méthodes privées
};
```

EXEMPLES D'UTILISATION

```
class Animal {
public:
    void deplacer();

protected:
    int age;

private:
    double poids;
};
```

Les membres publics sont accessibles partout, tandis que les membres protégés sont accessibles uniquement par les classes dérivées (enfants) et les membres privés sont accessibles uniquement à l'intérieur de la classe.

MÉTHODES D'ACCÈS (GET, SET)

SYNTAXE

```
class MaClasse {
private:
    int monAttribut;
public:
    int getMonAttribut() const;
    void setMonAttribut(int valeur);
};
```

Ce code illustre la syntaxe de base pour créer une classe en C++ avec un attribut privé et des méthodes publiques pour accéder et modifier cet attribut.

EXEMPLES D'UTILISATION

```
class Animal {
private:
    double poids;

public:
    double getPoids() const {
        return poids;
    }

    void setPoids(double p) {
        if (p > 0) {
            poids = p;
        }
    }
};
```

Cet exemple montre comment créer une classe "Animal" avec des méthodes d'accès (getters) et de modification (setters) pour la propriété "poids".

HÉRITAGE

CLASSES DE BASE ET CLASSES DÉRIVÉES

L'**héritage** est un mécanisme qui permet à une classe d'hériter des attributs et méthodes d'une autre classe.

- **Classe de base** (parent) : la classe dont on hérite
- **Classe dérivée** (enfant) : la classe qui hérite

```
class Base {
public:
    int attribut_base;
    void methode_base() {
        // Code de la methode_base
    }
};

class Derivee : public Base {
public:
    int attribut_derivee;
    void methode_derivee() {
        // Code de la methode_derivee
    }
};
```

Pour déclarer qu'une classe est dérivée d'une autre, on utilise le mot-clé **public** suivi du nom de la classe de base.

SYNTAXE

```
class NomClasseDeBase {
    // Définition de la classe de base
};

class NomClasseDerivee : public NomClasseDeBase {
    // Définition de la classe dérivée
};
```

La classe dérivée hérite des membres et méthodes de la classe de base.

EXEMPLES D'UTILISATION

```
class Animal {
    int nbPattes;
    void seDeplacer();
};

class Chat : public Animal {
    void miauler();
};
```

Dans cet exemple, la classe **Chat** hérite de la classe **Animal**. Le mot-clé **public** indique que les membres publics de la classe de base restent publics dans la classe dérivée.

SURCHARGE ET SUBSTITUTION DE MÉTHODES

La **surcharge** est le processus de création de plusieurs méthodes avec le même nom mais avec des **paramètres différents**.

La **substitution** est le remplacement d'une méthode héritée d'une classe de base par une nouvelle implémentation dans la classe dérivée.

La surcharge permet d'utiliser la même méthode sous des formes différentes, tandis que la substitution permet de modifier une méthode d'une classe dérivée pour s'adapter à de nouveaux besoins.

SYNTAXE

```
class Base {
    void methode(); // Méthode à remplacer
};

class Derive : public Base {
    void methode(); // Nouvelle implémentation
};
```

EXEMPLES D'UTILISATION

```
class Animal {
    void seDeplacer() {
        // Implémentation générale
    }
};

class Poisson : public Animal {
    void seDeplacer() {
        // Implémentation spécifique aux poissons
    }
};
```

Cet exemple illustre l'utilisation de l'**héritage** en C++ et la **redéfinition** de méthodes dans les classes dérivées.

HÉRITAGE MULTIPLE

L'**héritage multiple** est un mécanisme qui permet à une classe d'hériter de **plusieurs classes de base**.

```
class ClasseDeBase1 {
  // Définition de la classe de base 1
};

class ClasseDeBase2 {
  // Définition de la classe de base 2
};

class ClasseDerivee : public ClasseDeBase1, public ClasseDeBase2 {
  // Définition de la classe dérivée, héritant des classes de base 1 et 2
};
```

SYNTAXE

```
class Derive : public Base1, public Base2 {
    // Définition de la classe dérivée
};
```

La classe dérivée hérite des propriétés et des méthodes de la classe Base1 et de la classe Base2.

EXEMPLES D'UTILISATION

```
class A { // Classe de base A
    void methodeA();
};

class B { // Classe de base B
    void methodeB();
};

class C : public A, public B { // Classe dérivée C héritant de A et B
    void methodeC();
};
```