TP 3 – Héritage Simple et Multiple

Durée : 1h30

Concepts clés

— **Héritage simple** : une classe dérivée hérite d'une seule classe de base
— **Héritage multiple** : une classe dérivée hérite de plusieurs classes de base
— **Classe de base / parent** : classe dont on hérite les membres
— **Classe dérivée / enfant** : classe qui hérite et étend les fonctionnalités
— **protected** : accès pour la classe et ses dérivées, mais pas de l'extérieur
— **Syntaxe** : `class Derivee : public Base1, public Base2 { ... }`
— **Constructeur dérivé** : doit initialiser toutes les classes de base

Exemple de référence

Exemple complet : Héritage simple avec Animal et Chien

Voici un exemple complet montrant comment créer une classe de base et une classe dérivée :

```
// Classe de base (parent)
class Animal {
protected:                    // Accessible dans les classes derivees
    string nom;
    int age;
public:
    // Constructeur de la classe de base
    Animal(string n, int a) : nom(n), age(a) {
        cout << "Constructeur Animal" << endl;
    }

    void afficher() {
        cout << "Nom: " << nom << ", Age: " << age << endl;
    }
};

// Classe derivee (enfant) - herite de Animal
class Chien : public Animal {
private:
    string race;              // Nouvel attribut specifique
public:
    // Le constructeur DOIT appeler le constructeur de la base
    Chien(string n, int a, string r) : Animal(n, a), race(r) {
        cout << "Constructeur Chien" << endl;
    }

    // Redefinition de la methode afficher()
    void afficher() {
        Animal::afficher();   // Appel de la methode de la classe parent
        cout << "Race: " << race << endl;
    }

    void aboyer() {           // Methode specifique a Chien
        cout << nom << " aboie: Woof!" << endl;
    }
};

// Utilisation dans main()
int main() {
    Chien rex("Rex", 3, "Berger Allemand");
    rex.afficher();           // Affiche nom, age ET race
    rex.aboyer();             // Methode specifique
    return 0;
}
```

Points essentiels à retenir :

— `protected` : les attributs sont accessibles dans les classes dérivées mais pas de l'extérieur
— `: public Animal` : syntaxe d'héritage, Chien hérite publiquement de Animal
— `: Animal(n, a)` : dans la liste d'initialisation, on appelle le constructeur de la base
— `Animal::afficher()` : pour appeler la méthode de la classe parent depuis la classe dérivée

Exercice 1 : Héritage simple - Personne et Étudiant

Exercice

Durée estimée : 30 minutes **Difficulté :** ⋆⋆
Contexte : Créer une hiérarchie où un Étudiant hérite d'une Personne pour illustrer l'héritage simple.
À faire :

1. Créer une classe `Personne` avec :
   - Attributs **protected** : `nom` (string), `age` (int)
   - Constructeur avec nom et âge
   - Méthode `afficher()` qui affiche nom et âge
   - Méthode `vieillir()` qui augmente l'âge de 1
2. Créer une classe `Etudiant` qui hérite de `Personne` :
   - Attribut privé : `numero_etudiant` (string)
   - Attribut privé : `moyenne` (float)
   - Constructeur avec nom, âge et numéro étudiant
   - Méthode `afficher()` qui affiche toutes les informations (redéfinition)
   - Méthode `setMoyenne(float m)` pour définir la moyenne
   - Méthode `est_admis()` qui retourne true si moyenne $>= 10$
3. Dans `main()` :
   - Créer une Personne "Ali" 30 ans
   - Créer un Étudiant "Fatima" 20 ans, numéro "E2024001"
   - Afficher les deux et fêter l'anniversaire de l'étudiant
   - Donner une moyenne de 14.5 à l'étudiant et vérifier s'il est admis

Résultat attendu :

```
Personne: Ali, 30 ans
Etudiant: Fatima, 20 ans, N°E2024001, Moyenne: 0.00
Après anniversaire: Fatima, 21 ans
Moyenne: 14.50 - Statut: Admis
```

Note

- Les attributs `protected` sont accessibles dans les classes dérivées
- Le constructeur dérivé initialise la base via : `Etudiant(...) : Personne(nom, age) { ... }`
- La redéfinition de `afficher()` permet d'ajouter des informations spécifiques

Exercice 2 : Héritage simple - Véhicule et Voiture

Exercice

Durée estimée : 25 minutes **Difficulté :** ⋆⋆
Contexte : Modéliser une hiérarchie véhicule → voiture pour réutiliser les propriétés communes.
À faire :

1. Créer une classe `Vehicule` avec :
   - Attributs **protected** : `marque` (string), `annee` (int), `prix` (float)
   - Constructeur avec marque, année et prix
   - Méthode `afficher_details()` qui affiche marque, année, prix
   - Méthode `calculer_age()` qui retourne l'âge (2025 - annee)
2. Créer une classe `Voiture` qui hérite de `Vehicule` :
   - Attribut privé : `nombre_portes` (int)
   - Attribut privé : `kilometrage` (int)
   - Constructeur avec marque, année, prix et nombre de portes
   - Méthode `afficher_details()` redéfinie avec toutes les infos
   - Méthode `ajouter_kilometres(int km)` qui ajoute des km
   - Méthode `calculer_valeur_residuelle()` : prix - (age × 2000) - (km × 0.1)
3. Dans `main()` :
   - Créer un Véhicule "Toyota" 2020, 25000€
   - Créer une Voiture "BMW" 2018, 45000€, 5 portes
   - Afficher les détails des deux
   - Ajouter 50000 km à la voiture et calculer sa valeur résiduelle

Résultat attendu (extrait) :

```
Véhicule: Toyota (2020), Prix: 25000€, Âge: 5 ans
Voiture: BMW (2018), Prix: 45000€, 5 portes, 50000 km
Valeur résiduelle: 30000€
```

Note

- L'héritage évite de dupliquer `marque`, `annee`, `prix`
- Les méthodes de la classe de base restent accessibles dans la dérivée
- La redéfinition permet d'adapter l'affichage aux besoins spécifiques

Exercice 3 : Héritage multiple - Étudiant-Salarié

Exercice

Durée estimée : 35 minutes **Difficulté :** ⋆⋆⋆
Contexte : Un étudiant-salarié combine les propriétés d'un étudiant ET d'un salarié (héritage multiple).
À faire :

1. Créer une classe `Etudiant` avec :
   - Attributs **protected** : `nom` (string), `numero_etudiant` (string)
   - Constructeur avec nom et numéro
   - Méthode `afficher_etudiant()` qui affiche nom et numéro
2. Créer une classe `Salarie` avec :
   - Attributs **protected** : `entreprise` (string), `salaire` (float)
   - Constructeur avec entreprise et salaire
   - Méthode `afficher_salarie()` qui affiche entreprise et salaire
3. Créer une classe `EtudiantSalarie` qui hérite de `Etudiant` ET `Salarie` :
   - Constructeur avec nom, numéro, entreprise et salaire
   - Méthode `afficher_complet()` qui affiche toutes les informations
   - Méthode `peut_se_loger()` qui retourne true si salaire >= 1000
4. Dans `main()` :
   - Créer un EtudiantSalarie "Omar" E2024002, "TechCorp", 1200€
   - Afficher ses informations complètes
   - Vérifier s'il peut se loger

Résultat attendu :

```
=== Étudiant-Salarié ===
Nom: Omar
Numéro étudiant: E2024002
Entreprise: TechCorp
Salaire: 1200€
Peut se loger: Oui
```

Note

- Syntaxe d'héritage multiple : `class EtudiantSalarie : public Etudiant, public Salarie`
- Le constructeur dérivé initialise TOUTES les bases : `EtudiantSalarie(...) : Etudiant(...), Salarie(...) { ... }`
- L'héritage multiple combine les fonctionnalités de plusieurs classes
- Attention aux conflits de noms (utiliser le préfixe de classe si nécessaire)

Bonus - Exercices à la Maison

Exercice

Durée estimée : 45 minutes **Difficulté :** ⋆⋆⋆

Exercice Bonus 1 : Héritage simple Animal → Chat/Chien
Créer une classe `Animal` avec `nom`, `age`, `poids`. Créer deux classes dérivées `Chat` et `Chien` qui ajoutent respectivement `couleur` et `race`. Ajouter des méthodes spécifiques : `miauler()`, `aboyer()`.

Exercice Bonus 2 : Héritage multiple Appareil
Créer `Ecran` (taille, resolution) et `Ordinateur` (processeur, RAM). Créer `OrdinateurPortable` qui hérite des deux et ajoute `autonomie_batterie`.

Exercice Bonus 3 : Système de gestion universitaire
Créer `Personne` (nom, age), puis `Etudiant` (numero, moyenne) et `Enseignant` (matiere, salaire) qui héritent de `Personne`. Créer enfin `EtudiantEnseignant` avec héritage multiple.

Note

— Les bonus consolident les concepts d'héritage simple et multiple
— Testez toujours l'initialisation correcte des classes de base
— Documentez vos choix de `protected` vs `private`

Concepts à Retenir

Concepts clés

— **Héritage simple** : `class Derivee : public Base { ... };`
— **Héritage multiple** : `class Derivee : public Base1, public Base2 { ... };`
— **Constructeur dérivé** : `Derivee(...) : Base1(...), Base2(...) { ... }`
— **protected** : accessible dans la classe et ses dérivées
— **Redéfinition** : réécrire une méthode de la base pour adapter son comportement