Atelier 3 : Programmation Orientée Objet en C++

Exercice 1 : Pile

L'exercice demande de créer une classe Pile capable d'empiler (push) et de dépiler (pop) des éléments de type int.

Concepts C++ requis

- **Classe et Objets**
- **Constructeur et Destructeur**
- **Encapsulation** (membres privés et méthodes publiques)

Code C++ (Pile.cpp)

```cpp
#include <iostream>
```

```cpp
#include <vector>
#include <stdexcept>

// Définition de la classe Pile
class Pile {
private:
  // Utilisation d'un std::vector pour simplifier la gestion de la pile
  std::vector<int> elements;

public:
  // Constructeur
  Pile() {
    std::cout << "Pile créée." << std::endl;
  }

  // Destructeur
  ~Pile() {
    std::cout << "Pile détruite. Nombre d'éléments restants: " << elements.size() << std::endl;
  }

  // Méthode pour empiler un élément (push)
  void empiler(int element) {
    elements.push_back(element);
    std::cout << "Empilé: " << element << std::endl;
  }
```

```
    // Méthode pour dépiler un élément (pop)
    int depiler() {
        if (elements.empty()) {
            throw std::out_of_range("La pile est vide.");
        }
        int top_element = elements.back();
        elements.pop_back();
        std::cout << "Dépilé: " << top_element << std::endl;
        return top_element;
    }

    // Méthode pour vérifier si la pile est vide
    bool estVide() const {
        return elements.empty();
    }
};

// Programme principal
int main() {
    // Création de deux piles
    Pile p1;
    Pile p2;

    // Remplissage de p1
    p1.empiler(10);
    p1.empiler(20);
    p1.empiler(30);

    // Remplissage de p2
    p2.empiler(100);
    p2.empiler(200);

    std::cout << "\nDébut du dépilement de p1:" << std::endl;
    // Dépilement de p1
    while (!p1.estVide()) {
        try {
            p1.depiler();
        } catch (const std::out_of_range& e) {
            std::cerr << "Erreur: " << e.what() << std::endl;
        }
    }

    std::cout << "\nDébut du dépilement de p2:" << std::endl;
    // Dépilement de p2
    while (!p2.estVide()) {
        try {
            p2.depiler();
        } catch (const std::out_of_range& e) {
            std::cerr << "Erreur: " << e.what() << std::endl;
        }
```

```
    }

    // Les destructeurs de p1 et p2 seront appelés automatiquement à la fin de main()
    std::cout << "\nFin du programme principal." << std::endl;

    return 0;
}
```

Exercice 2 : Fichier

la **gestion de la mémoire dynamique** et la distinction entre copie superficielle et **copie profonde**.

Concepts C++ requis

- **Allocation dynamique de mémoire** (new[] et delete[])
- **Constructeur de copie** (pour la copie profonde)
- **Destructeur** (pour libérer la mémoire)

Code C++ (Fichier.cpp)

```
#include <iostream>
#include <cstring> // Pour memcpy
#include <algorithm> // Pour std::min

// Définition de la classe Fichier
class Fichier {
private:
    char* pointeurP; // Pointeur vers le bloc mémoire
    int taille;      // Taille du bloc mémoire
    std::string nom; // Nom du fichier

public:
    // 1. Méthode "Creation" (Constructeur)
    Fichier(const std::string& nomFichier, int tailleFichier) : nom(nomFichier), taille(tailleFichier) {
        std::cout << "Constructeur (Creation) appelé pour " << nom << std::endl;
        if (taille > 0) {
            pointeurP = new char[taille];
            std::memset(pointeurP, 0, taille);
        } else {
```

```
            pointeurP = nullptr;
            this->taille = 0;
        }
    }

    // 2. Méthode "Rempli" (Constructeur de copie - Copie Profonde)
    Fichier(const Fichier& autre) : nom(autre.nom + "_copie"), taille(autre.taille) {
        std::cout << "Constructeur de copie (Rempli - Copie Profonde) appelé pour " << nom << std::endl;
        if (taille > 0) {
            // Allouer une NOUVELLE zone mémoire
            pointeurP = new char[taille];
            // Copier le contenu de l'ancienne zone vers la nouvelle
            std::memcpy(pointeurP, autre.pointeurP, taille);
        } else {
            pointeurP = nullptr;
        }
    }

    // 3. Destructeur
    ~Fichier() {
        std::cout << "Destructeur appelé pour " << nom << std::endl;
        if (pointeurP != nullptr) {
            delete[] pointeurP;
            pointeurP = nullptr;
            std::cout << "Mémoire libérée pour " << nom << std::endl;
        }
    }

    // 4. Méthode "Afficher" (Affiche l'adresse et la taille)
    void Afficher() const {
        std::cout << "Fichier: " << nom
                  << ", Taille: " << taille << " octets"
                  << ", Adresse mémoire: " << static_cast<void*>(pointeurP)
                  << std::endl;
    }
};

// Programme principal
int main() {
    // Création d'un objet Fichier (méthode "Creation")
    Fichier f1("mon_document.txt", 1024);
    f1.Afficher();

    std::cout << "\n--- Test de la copie profonde ---\n";
    // Création d'un deuxième objet par copie (méthode "Rempli")
    Fichier f2 = f1; // Appel du constructeur de copie
    f2.Afficher();

    // Vérification que les adresses sont différentes (copie profonde)
    std::cout << "Adresses de f1 et f2 sont différentes (copie profonde): "
              << (&f1 != &f2) << std::endl;
```

```
    std::cout << "\n--- Fin du programme ---\n";
    return 0;
}
```

Exercice 3 : Liste Chaînée

créer une classe ListeChainee simple, en utilisant une structure pour les éléments et en gérant l'ajout et la suppression au début.

Concepts C++ requis

- **Structures (struct)**
- **Pointeurs** (* et ->)
- **Gestion de la mémoire dynamique** (new et delete)

Code C++ (ListeChainee.cpp)

```
#include <iostream>
#include <string>

// Structure pour un élément de la liste
struct Element {
    std::string donnee;
    Element* suivant;

    // Constructeur
    Element(const std::string& d) : donnee(d), suivant(nullptr) {}
};

// Définition de la classe ListeChainee
class ListeChainee {
private:
    Element* tete; // Pointeur vers le premier élément de la liste

public:
    // Constructeur
    ListeChainee() : tete(nullptr) {
        std::cout << "Liste Chaînée créée." << std::endl;
    }
```

```
// Destructeur (pour libérer toute la mémoire)
~ListeChainee() {
    std::cout << "Destructeur de Liste Chaînée appelé." << std::endl;
    Element* courant = tete;
    while (courant != nullptr) {
        Element* aSupprimer = courant;
        courant = courant->suivant;
        delete aSupprimer;
    }
    tete = nullptr;
    std::cout << "Toute la mémoire de la liste a été libérée." << std::endl;
}

// Méthode pour ajouter un élément au début de la liste
void ajouterAuDebut(const std::string& donnee) {
    Element* nouvelElement = new Element(donnee);
    nouvelElement->suivant = tete;
    tete = nouvelElement;
    std::cout << "Ajouté: " << donnee << std::endl;
}

// Méthode pour supprimer un élément du début de la liste
void supprimerAuDebut() {
    if (tete == nullptr) {
        std::cout << "La liste est vide, rien à supprimer." << std::endl;
        return;
    }
    Element* aSupprimer = tete;
    tete = tete->suivant;
    std::cout << "Supprimé: " << aSupprimer->donnee << std::endl;
    delete aSupprimer;
}

// Méthode pour afficher tous les éléments de la liste
void afficherListe() const {
    if (tete == nullptr) {
        std::cout << "La liste est vide." << std::endl;
        return;
    }
    Element* courant = tete;
    std::cout << "Contenu de la liste: ";
    while (courant != nullptr) {
        std::cout << "[" << courant->donnee << "]";
        if (courant->suivant != nullptr) {
            std::cout << " -> ";
        }
        courant = courant->suivant;
    }
    std::cout << std::endl;
}
```

```
};

// Programme principal
int main() {
    ListeChainee maListe;

    // Ajouter des éléments
    maListe.ajouterAuDebut("Trois");
    maListe.ajouterAuDebut("Deux");
    maListe.ajouterAuDebut("Un");

    // Afficher la liste
    maListe.afficherListe();

    // Supprimer un élément
    maListe.supprimerAuDebut();

    // Afficher la liste après suppression
    maListe.afficherListe();

    // Le destructeur sera appelé automatiquement à la fin de main()
    return 0;
}
```

Exercice 4 : Programme de Gestion Simplifiée de Comptes Bancaires

concepts de POO, notamment la composition, les membres statiques et les fonctions utilitaires.

Concepts C++ requis

- **Composition** (la classe Compte contient un objet Client)
- **Membres Statiques** (variable et méthode pour le total des comptes)
- **Fonction utilitaire** (fonction globale calculInteret)

Code C++ (Banque.cpp)

```
#include <iostream>
#include <string>
```

```cpp
#include <vector>
#include <algorithm>

// --- 1. Classe Client ---
class Client {
private:
    int identifiant;
    std::string nom;
    std::string prenom;
    std::string stringNom; // Nom long (nom + prenom)

public:
    // 1. Constructeur
    Client(int id, const std::string& n, const std::string& p)
        : identifiant(id), nom(n), prenom(p), stringNom(n + " " + p) {}

    // 2. Constructeur par défaut (vide)
    Client() : identifiant(0), nom(""), prenom(""), stringNom("") {}

    // 3. Méthode pour afficher les informations d'un client
    void afficher() const {
        std::cout << "  [Client ID: " << identifiant << ", Nom: " << stringNom << "]" << std::endl;
    }

    // 4. Méthode pour obtenir le nom complet
    std::string getStringNom() const {
        return stringNom;
    }
};

// --- 2. Classe Compte ---
class Compte {
private:
    int numero;
    float solde;
    float floatSolde; // Solde flottant
    Client titulaire; // Composition: un Compte a un Client
    static int totalComptes; // Variable statique pour compter les comptes

public:
    // 1. Constructeur avec paramètres
    Compte(int num, float s, float fs, const Client& c)
        : numero(num), solde(s), floatSolde(fs), titulaire(c) {
        totalComptes++;
    }

    // 2. Constructeur sans paramètre
    Compte() : numero(0), solde(0.0f), floatSolde(0.0f), titulaire() {
        totalComptes++;
    }
```

```
    // 3. Destructeur
    ~Compte() {
        totalComptes--; // Décrémenter le compteur
        // std::cout << "Compte détruit: " << numero << std::endl;
    }

    // 4. Méthode d'affichage
    void afficher() const {
        std::cout << "Compte N°" << numero << ", Solde: " << solde << "€ (Float: " << floatSolde << "€)"
<< std::endl;
        titulaire.afficher();
    }

    // 5. Méthode statique pour obtenir le nombre total de comptes
    static int getTotalComptes() {
        return totalComptes;
    }

    // 6. Méthode pour obtenir le solde
    float getSolde() const {
        return solde;
    }

    // 7. Méthode pour obtenir le numéro de compte
    int getNumero() const {
        return numero;
    }
};

// Initialisation de la variable statique
int Compte::totalComptes = 0;

// --- 3. Fonction utilitaire ---
// Fonction globale pour calculer l'intérêt (non-inline)
float calculInteret(float solde, float taux) {
    return solde * taux;
}

// --- 4. Programme principal (main) ---
int main() {
    // 1. Créer plusieurs clients et comptes
    Client client1(1, "Dupont", "Jean");
    Client client2(2, "Martin", "Sophie");

    Compte compte1(101, 1000.0f, 1000.0f, client1);
    Compte compte2(102, 500.50f, 500.50f, client2);
    Compte compte3(103, 250.0f, 250.0f, client1);

    std::cout << "\n--- 1. Affichage des comptes initiaux ---" << std::endl;
    compte1.afficher();
    compte2.afficher();
```

```
    compte3.afficher();

    std::cout << "\n--- 2. Nombre total de comptes créés ---" << std::endl;
    std::cout << "Total comptes: " << Compte::getTotalComptes() << std::endl;

    // 3. Copier certains comptes pour tester le constructeur de copie
    std::cout << "\n--- 3. Test du constructeur de copie ---" << std::endl;
    Compte compte4 = compte1; // Copie
    compte4.afficher();
    std::cout << "Total comptes après copie: " << Compte::getTotalComptes() << std::endl;

    // 4. Supprimer certains comptes et observer le comportement du destructeur
    {
        std::cout << "\n--- 4. Test du destructeur (dans un bloc) ---" << std::endl;
        Compte compteTemporaire(999, 10.0f, 10.0f, client2);
        std::cout << "Total comptes avant sortie du bloc: " << Compte::getTotalComptes() << std::endl;
    } // Le destructeur de compteTemporaire est appelé ici
    std::cout << "Total comptes après sortie du bloc: " << Compte::getTotalComptes() << std::endl;

    // 6. Appliquer des intérêts
    float tauxInteret = 0.05f; // 5%
    std::cout << "\n--- 6. Application de la fonction calculInteret ---" << std::endl;

    float interet1 = calculInteret(compte1.getSolde(), tauxInteret);
    std::cout << "Intérêt pour compte " << compte1.getNumero() << ": " << interet1 << "€" << std::endl;

    float interet2 = calculInteret(compte2.getSolde(), tauxInteret);
    std::cout << "Intérêt pour compte " << compte2.getNumero() << ": " << interet2 << "€" << std::endl;

    std::cout << "\n--- Fin du programme principal ---" << std::endl;

    return 0;
}
```

Questions de Réflexion

1. Quelle est la différence entre une copie superficielle et une copie profonde dans ce contexte ?

La différence entre une **copie superficielle** (shallow copy) et une **copie profonde** (deep copy) est fondamentale dans la gestion de la mémoire, en particulier lorsqu'une classe contient des pointeurs vers des ressources allouées dynamiquement (comme dans l'Exercice 2 avec pointeurP).

Caractéristique	Copie Superficielle (Shallow Copy)	Copie Profonde (Deep Copy)
Pointeurs	Seul le pointeur est copié.	Une nouvelle zone mémoire est allouée, et le contenu est copié.
Mémoire	Les deux objets pointent vers la **même** zone mémoire.	Chaque objet possède sa **propre** zone mémoire.
Destructeur	Problème du "double free" : le destructeur du premier objet libère la mémoire, et le destructeur du second tente de libérer la même zone, provoquant une erreur.	Chaque destructeur libère sa propre zone mémoire sans conflit.
Utilisation	Suffisante si la classe ne gère pas de ressources dynamiques (pas de pointeurs).	**Nécessaire** si la classe gère des ressources dynamiques (pointeurs).

2. Pourquoi le compteur du nombre de comptes doit-il être statique ?

Le compteur du nombre de comptes (totalComptes dans la classe Compte de l'Exercice 4) doit être déclaré **statique** pour les raisons suivantes :

1. **Partage de l'information :** Une variable membre statique est partagée par **toutes les instances** de la classe. Il n'y a qu'une seule copie de cette variable, quel que soit le nombre d'objets Compte créés.
2. **Indépendance de l'objet :** Le compteur représente une propriété de la classe dans son ensemble (le nombre total de comptes dans la banque), et non une propriété spécifique à un objet Compte individuel.

3. Quelle est la différence entre une méthode statique et une méthode normale ?

Caractéristique	Méthode Statique	Méthode Normale (ou d'instance)
Appel	Appelée via le nom de la classe (Classe::methode()).	Appelée uniquement via un objet de la classe (objet.methode()).
Accès aux membres	Ne peut accéder qu'aux **membres statiques** de la classe.	Peut accéder à **tous les membres** (statiques et non statiques) de la classe.
Pointeur this	N'a pas accès au pointeur this.	A accès au pointeur this.

4. Dans quel cas est-il pertinent de rendre une fonction inline ?

Rendre une fonction **inline** est une *suggestion* faite au compilateur d'insérer le corps de la fonction directement à l'endroit où elle est appelée. C'est pertinent pour les **fonctions très courtes** (comme les accesseurs simples) afin de **réduire la surcharge** liée à l'appel de fonction. Pour les fonctions longues ou complexes, l'inlining est généralement déconseillé car il augmente la taille du code exécutable.

5. Que se passe-t-il si on oublie de libérer la mémoire dans le destructeur ?

Si l'on oublie de libérer la mémoire allouée dynamiquement (avec new ou new[]) dans le destructeur, il en résulte une **fuite de mémoire** (memory leak). La zone mémoire allouée reste réservée et inaccessible, ce qui peut entraîner l'épuisement de la mémoire disponible du système si le phénomène se répète.