Atelier 4 : Concepts Avancés de POO en C++

Exercice 1 : Gestion d'une Médiathèque

Cet exercice met en œuvre l'héritage multiple et le polymorphisme pour gérer différents types de ressources.

Concepts C++ requis

- **Héritage** (simple et multiple)
- **Polymorphisme** (méthodes virtuelles et classes abstraites)
- **Surcharge d'opérateurs** (operator==)
- **Surcharge de fonctions** (méthode rechercher())

Structure des Classes

Classe	Rôle	Concepts Clés
Ressource	Classe de base abstraite. Contient les attributs communs et les méthodes virtuelles pures (afficherInfos).	Polymorphisme, Surcharge d'opérateur (==).
Telechargeable	Classe de base pour l'héritage multiple. Contient la méthode telecharger() (virtuelle pure) et afficherMessage().	Héritage Multiple.
Livre, **Magazine**, **Video**	Classes dérivées. Héritent de Ressource et de Telechargeable. Redéfinissent les méthodes virtuelles.	Héritage Multiple, Redéfinition de méthodes.
Mediatheque	Classe de gestion. Contient un vecteur de pointeurs de Ressource*. Implémente la surcharge de rechercher().	Polymorphisme, Surcharge de fonction.

Code C++ (Exercice1_Mediatheque.cpp)

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

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

// ----------------------------------------------------------------
// 1. Classe de base: Ressource
// ----------------------------------------------------------------
class Ressource {
```

```
protected:
    int id;
    std::string titre;
    std::string auteur;
    int annee;

public:
    Ressource(int i, const std::string& t, const std::string& a, int y)
        : id(i), titre(t), auteur(a), annee(y) {}

    virtual ~Ressource() = default;

    // Méthode virtuelle pure pour afficher les informations (rend Ressource abstraite)
    virtual void afficherInfos() const = 0;

    // Méthode virtuelle pour télécharger (implémentation par défaut)
    virtual void telecharger() const {
        std::cout << "Téléchargement de la ressource " << titre << " (ID: " << id << ") en cours..." <<
std::endl;
    }

    // Surcharge de l'opérateur == pour comparer deux ressources selon leur identifiant
    bool operator==(const Ressource& autre) const {
        return this->id == autre.id;
    }

    // Accesseurs pour la recherche
    int getId() const { return id; }
    const std::string& getTitre() const { return titre; }
    const std::string& getAuteur() const { return auteur; }
    int getAnnee() const { return annee; }
};

// ----------------------------------------------------------------
// 3. Classe pour l'héritage multiple: Telechargeable
// ----------------------------------------------------------------
class Telechargeable {
public:
    virtual ~Telechargeable() = default;

    // Méthode virtuelle pure pour le téléchargement
    virtual void telecharger() const = 0;

    // Méthode spécifique pour afficher un message de téléchargement
    void afficherMessage() const {
        std::cout << "Message de Telechargeable: Le téléchargement est soumis à nos conditions
d'utilisation." << std::endl;
    }
};

// ----------------------------------------------------------------
```

```cpp
// 2. Classes dérivées de Ressource et Telechargeable (Héritage Multiple)
// ----------------------------------------------------------------------

// Classe Livre
class Livre : public Ressource, public Telechargeable {
private:
  std::string isbn; // Attribut spécifique

public:
  Livre(int i, const std::string& t, const std::string& a, int y, const std::string& is)
    : Ressource(i, t, a, y), isbn(is) {}

  // Redéfinition de afficherInfos()
  void afficherInfos() const override {
    std::cout << "Livre - ID: " << id << ", Titre: " << titre << ", Auteur: " << auteur
          << ", Année: " << annee << ", ISBN: " << isbn << std::endl;
  }

  // Implémentation de telecharger() de Telechargeable
  void telecharger() const override {
    afficherMessage(); // Utilisation de la méthode de Telechargeable
    std::cout << "Téléchargement du Livre '" << titre << "' en format PDF/ePub..." << std::endl;
  }
};

// Classe Magazine
class Magazine : public Ressource, public Telechargeable {
private:
  int numero; // Attribut spécifique

public:
  Magazine(int i, const std::string& t, const std::string& a, int y, int n)
    : Ressource(i, t, a, y), numero(n) {}

  // Redéfinition de afficherInfos()
  void afficherInfos() const override {
    std::cout << "Magazine - ID: " << id << ", Titre: " << titre << ", Auteur: " << auteur
          << ", Année: " << annee << ", Numéro: " << numero << std::endl;
  }

  // Implémentation de telecharger() de Telechargeable
  void telecharger() const override {
    afficherMessage();
    std::cout << "Téléchargement du Magazine N°" << numero << " en format numérique..." <<
std::endl;
  }
};

// Classe Video
class Video : public Ressource, public Telechargeable {
private:
```

```
    int duree; // Attribut spécifique (en minutes)

public:
    Video(int i, const std::string& t, const std::string& a, int y, int d)
        : Ressource(i, t, a, y), duree(d) {}

    // Redéfinition de afficherInfos()
    void afficherInfos() const override {
        std::cout << "Vidéo - ID: " << id << ", Titre: " << titre << ", Auteur: " << auteur
                  << ", Année: " << annee << ", Durée: " << duree << " min" << std::endl;
    }

    // Implémentation de telecharger() de Telechargeable
    void telecharger() const override {
        afficherMessage();
        std::cout << "Téléchargement de la Vidéo '" << titre << "' en format MP4..." << std::endl;
    }
};

// ----------------------------------------------------------------
// 4. Classe Mediatheque
// ----------------------------------------------------------------
class Mediatheque {
private:
    std::vector<Ressource*> ressources;

public:
    // Destructeur pour libérer la mémoire allouée dynamiquement
    ~Mediatheque() {
        for (Ressource* r : ressources) {
            delete r;
        }
    }

    void ajouterRessource(Ressource* r) {
        ressources.push_back(r);
        std::cout << "Ressource ajoutée (ID: " << r->getId() << ")" << std::endl;
    }

    void afficherRessources() const {
        std::cout << "\n--- Contenu de la Médiathèque ---\n";
        for (const Ressource* r : ressources) {
            r->afficherInfos();
        }
        std::cout << "---------------------------------\n";
    }

    // Surcharge de la méthode rechercher() par Titre
    std::vector<Ressource*> rechercher(const std::string& titre) const {
        std::vector<Ressource*> resultats;
        std::cout << "\nRecherche par Titre: '" << titre << "'" << std::endl;
```

```
        for (Ressource* r : ressources) {
            if (r->getTitre().find(titre) != std::string::npos) {
                resultats.push_back(r);
            }
        }
        return resultats;
    }

    // Surcharge de la méthode rechercher() par Année
    std::vector<Ressource*> rechercher(int annee) const {
        std::vector<Ressource*> resultats;
        std::cout << "\nRecherche par Année: " << annee << std::endl;
        for (Ressource* r : ressources) {
            if (r->getAnnee() == annee) {
                resultats.push_back(r);
            }
        }
        return resultats;
    }

    // Surcharge de la méthode rechercher() par Auteur
    std::vector<Ressource*> rechercher(const std::string& auteur, bool parAuteur) const {
        std::vector<Ressource*> resultats;
        std::cout << "\nRecherche par Auteur: '" << auteur << "'" << std::endl;
        for (Ressource* r : ressources) {
            if (r->getAuteur().find(auteur) != std::string::npos) {
                resultats.push_back(r);
            }
        }
        return resultats;
    }
};

// ----------------------------------------------------------------
// 6. Fonction main()
// ----------------------------------------------------------------
int main() {
    Mediatheque maMediatheque;

    // Création de plusieurs objets
    maMediatheque.ajouterRessource(new Livre(1, "Le Seigneur des Anneaux", "Tolkien", 1954, "978-2070612889"));
    maMediatheque.ajouterRessource(new Magazine(2, "Science & Vie", "Divers", 2024, 1250));
    maMediatheque.ajouterRessource(new Video(3, "Inception", "Nolan", 2010, 148));
    maMediatheque.ajouterRessource(new Livre(4, "Le Hobbit", "Tolkien", 1937, "978-2070612896"));

    maMediatheque.afficherRessources();

    // Tester les méthodes afficherInfos() et telecharger() (Polymorphisme)
    std::cout << "\n--- Test des méthodes polymorphes ---\n";
    std::vector<Ressource*> resultatsSeigneur = maMediatheque.rechercher("Seigneur");
```

```
    if (!resultatsSeigneur.empty()) {
        resultatsSeigneur[0]->afficherInfos();
        resultatsSeigneur[0]->telecharger();
    }

    // Tester la surcharge de rechercher()
    std::vector<Ressource*> resultatsTolkien = maMediatheque.rechercher("Tolkien", true);
    std::cout << "\nRésultats trouvés par Auteur: " << resultatsTolkien.size() << std::endl;
    for (Ressource* r : resultatsTolkien) {
        r->afficherInfos();
    }

    std::vector<Ressource*> resultats2010 = maMediatheque.rechercher(2010);
    std::cout << "\nRésultats trouvés par Année: " << resultats2010.size() << std::endl;
    for (Ressource* r : resultats2010) {
        r->afficherInfos();
    }

    // Tester l'opérateur == (point 5)
    std::cout << "\n--- Test de l'opérateur == ---\n";
    Livre livre1(1, "Test", "A", 2000, "1");
    Livre livre2(4, "Test2", "B", 2001, "2");
    std::cout << "Livre 1 == Livre 1 (même ID): " << (livre1 == livre1) << std::endl;
    std::cout << "Livre 1 == Livre 2 (ID différents): " << (livre1 == livre2) << std::endl;

    return 0;
}
```

Exercice 2 : Système de Gestion Bancaire

Cet exercice se concentre sur l'**encapsulation avancée** et l'utilisation des **classes amies** (friend) pour permettre un accès sélectif aux données sensibles.

Concepts C++ requis

- **Encapsulation** (membres privés)
- **Classes Amies** (friend class)
- **Accesseurs** (get/set)
- **Composition** (la classe CompteBancaire référence un Client)

Problématique et Solution

La contrainte principale est que les données sensibles (codeSecret) ne doivent pas être accessibles publiquement, mais doivent l'être par l'**AgentBancaire** et la **Banque** (pour l'audit). La solution idéale en C++ est d'utiliser le mot-clé friend.

En déclarant friend class AgentBancaire; et friend class Banque; dans la classe CompteBancaire, nous accordons à ces deux classes un accès direct à tous les membres privés et protégés de CompteBancaire, sans compromettre l'encapsulation pour le reste du programme.

Code C++ (Exercice2_Banque.cpp)

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

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

// Déclarations anticipées
class CompteBancaire;
class AgentBancaire;
class Banque;

// ----------------------------------------------------------------
// 1. Classe Client
// ----------------------------------------------------------------
class Client {
private:
    std::string nom;
    std::string cin;
    int idClient;

public:
    Client(const std::string& n, const std::string& c, int id)
        : nom(n), cin(c), idClient(id) {}

    // Accesseurs classiques (get/set)
    const std::string& getNom() const { return nom; }
    const std::string& getCin() const { return cin; }
    int getIdClient() const { return idClient; }

    void afficher() const {
        std::cout << "Client ID: " << idClient << ", Nom: " << nom << ", CIN: " << cin << std::endl;
    }
};

// ----------------------------------------------------------------
// 2. Classe CompteBancaire
```

```cpp
// ----------------------------------------------------------------
class CompteBancaire {
private:
    int numeroCompte;
    double solde;
    // Information sensible (non accessible publiquement)
    std::string codeSecret;
    // Référence vers le client propriétaire
    Client* proprietaire;

    // Déclaration des classes amies pour un accès privilégié
    friend class AgentBancaire;
    friend class Banque;

public:
    CompteBancaire(int num, double s, const std::string& code, Client* prop)
        : numeroCompte(num), solde(s), codeSecret(code), proprietaire(prop) {}

    // Accesseurs classiques (get/set) - sans code secret
    int getNumeroCompte() const { return numeroCompte; }
    double getSolde() const { return solde; }
    Client* getProprietaire() const { return proprietaire; }

    void afficherSolde() const {
        std::cout << "Compte N°" << numeroCompte << " - Solde: " << solde << " DH" << std::endl;
    }

    // Méthodes internes pour les opérations
    bool deposer(double montant) {
        if (montant > 0) {
            solde += montant;
            return true;
        }
        return false;
    }

    bool retirer(double montant) {
        if (montant > 0 && solde >= montant) {
            solde -= montant;
            return true;
        }
        return false;
    }
};

// ----------------------------------------------------------------
// 3. Classe AgentBancaire
// ----------------------------------------------------------------
class AgentBancaire {
private:
    std::string nomAgent;
```

```
public:
    AgentBancaire(const std::string& nom) : nomAgent(nom) {}

    // Accès direct au membre privé 'codeSecret' grâce à la déclaration 'friend'
    void consulterCodeSecret(const CompteBancaire& compte) const {
        std::cout << "Agent " << nomAgent << " consulte le code secret du compte N°"
                  << compte.numeroCompte << ": " << compte.codeSecret << std::endl;
    }

    // Accès aux informations privées du client (via le pointeur proprietaire)
    void afficherInfosPrivees(const CompteBancaire& compte) const {
        std::cout << "Agent " << nomAgent << " consulte les infos privées du client: "
                  << compte.proprietaire->getNom() << ", CIN: " << compte.proprietaire->getCin() <<
std::endl;
    }

    // Méthode pour effectuer un transfert
    bool effectuerTransfert(CompteBancaire& source, CompteBancaire& destination, double montant)
const {
        if (source.retirer(montant)) {
            destination.deposer(montant);
            std::cout << "Transfert de " << montant << " DH effectué de "
                      << source.getNumeroCompte() << " vers " << destination.getNumeroCompte() <<
std::endl;
            return true;
        }
        std::cout << "Échec du transfert: Solde insuffisant dans le compte source." << std::endl;
        return false;
    }
};

// ----------------------------------------------------------------------
// 4. Classe Banque
// ----------------------------------------------------------------------
class Banque {
private:
    std::vector<Client> clients;
    std::vector<CompteBancaire> comptes;

public:
    void ajouterClient(const Client& c) {
        clients.push_back(c);
    }

    void ajouterCompte(const CompteBancaire& c) {
        comptes.push_back(c);
    }

    // Méthode pour afficher les détails internes d'un compte (pour audit)
    // Accès direct aux membres privés grâce à la déclaration 'friend'
```

```
    void rapportAudit(int numeroCompte) const {
        std::cout << "\n--- Rapport d'Audit Interne (Banque) ---" << std::endl;
        for (const auto& compte : comptes) {
            if (compte.numeroCompte == numeroCompte) {
                std::cout << "Compte N°" << compte.numeroCompte << std::endl;
                std::cout << "  Solde: " << compte.solde << " DH" << std::endl;
                std::cout << "  Code Secret: " << compte.codeSecret << " (Information sensible)" << std::endl;
                std::cout << "  Propriétaire ID: " << compte.proprietaire->getIdClient() << std::endl;
                std::cout << "-------------------------------------" << std::endl;
                return;
            }
        }
        std::cout << "Compte N°" << numeroCompte << " non trouvé." << std::endl;
    }

    // Méthode pour trouver un compte par numéro
    CompteBancaire* trouverCompte(int num) {
        for (auto& compte : comptes) {
            if (compte.getNumeroCompte() == num) {
                return &compte;
            }
        }
        return nullptr;
    }
};

// ----------------------------------------------------------------------
// main()
// ----------------------------------------------------------------------
int main() {
    Banque maBanque;

    // Créer plusieurs clients
    Client client1("Dupont", "AB12345", 1);
    Client client2("Martin", "CD67890", 2);

    maBanque.ajouterClient(client1);
    maBanque.ajouterClient(client2);

    // Créer plusieurs comptes
    CompteBancaire compte1(1001, 5000.0, "1234", &client1);
    CompteBancaire compte2(1002, 1500.0, "5678", &client2);

    maBanque.ajouterCompte(compte1);
    maBanque.ajouterCompte(compte2);

    // Simuler un agent bancaire
    AgentBancaire agent("M. Smith");

    std::cout << "\n--- 1. Simulation des opérations ---" << std::endl;
```

```
    // Dépôt
    CompteBancaire* c1 = maBanque.trouverCompte(1001);
    if (c1) {
        c1->deposer(1000.0);
        c1->afficherSolde();
    }

    // Retrait
    CompteBancaire* c2 = maBanque.trouverCompte(1002);
    if (c2) {
        c2->retirer(500.0);
        c2->afficherSolde();
    }

    // Transfert entre comptes (via l'agent)
    if (c1 && c2) {
        agent.effectuerTransfert(*c1, *c2, 200.0);
        c1->afficherSolde();
        c2->afficherSolde();
    }

    std::cout << "\n--- 2. Accès privilégié de l'Agent (via 'friend') ---" << std::endl;
    if (c1) {
        // L'agent accède directement au code secret (membre privé)
        agent.consulterCodeSecret(*c1);
        agent.afficherInfosPrivees(*c1);
    }

    // Afficher un rapport d'audit via la classe Banque
    // La classe Banque accède directement au code secret (membre privé)
    maBanque.rapportAudit(1001);
    maBanque.rapportAudit(1002);

    return 0;
}
```

Conclusion

L'Atelier 4 a permis d'explorer des concepts avancés de la POO en C++ :

1. **Héritage Multiple et Polymorphisme** : L'Exercice 1 a montré comment une classe peut hériter de plusieurs classes de base (Ressource et Telechargeable) et comment le polymorphisme permet de traiter différents types d'objets de manière uniforme.

2 **Encapsulation et Amis** : L'Exercice 2 a illustré l'importance de l'encapsulation pour les données sensibles et comment le mécanisme des classes amies (friend) permet de créer des exceptions contrôlées à cette encapsulation pour des besoins spécifiques (comme l'audit ou les opérations d'un agent bancaire).