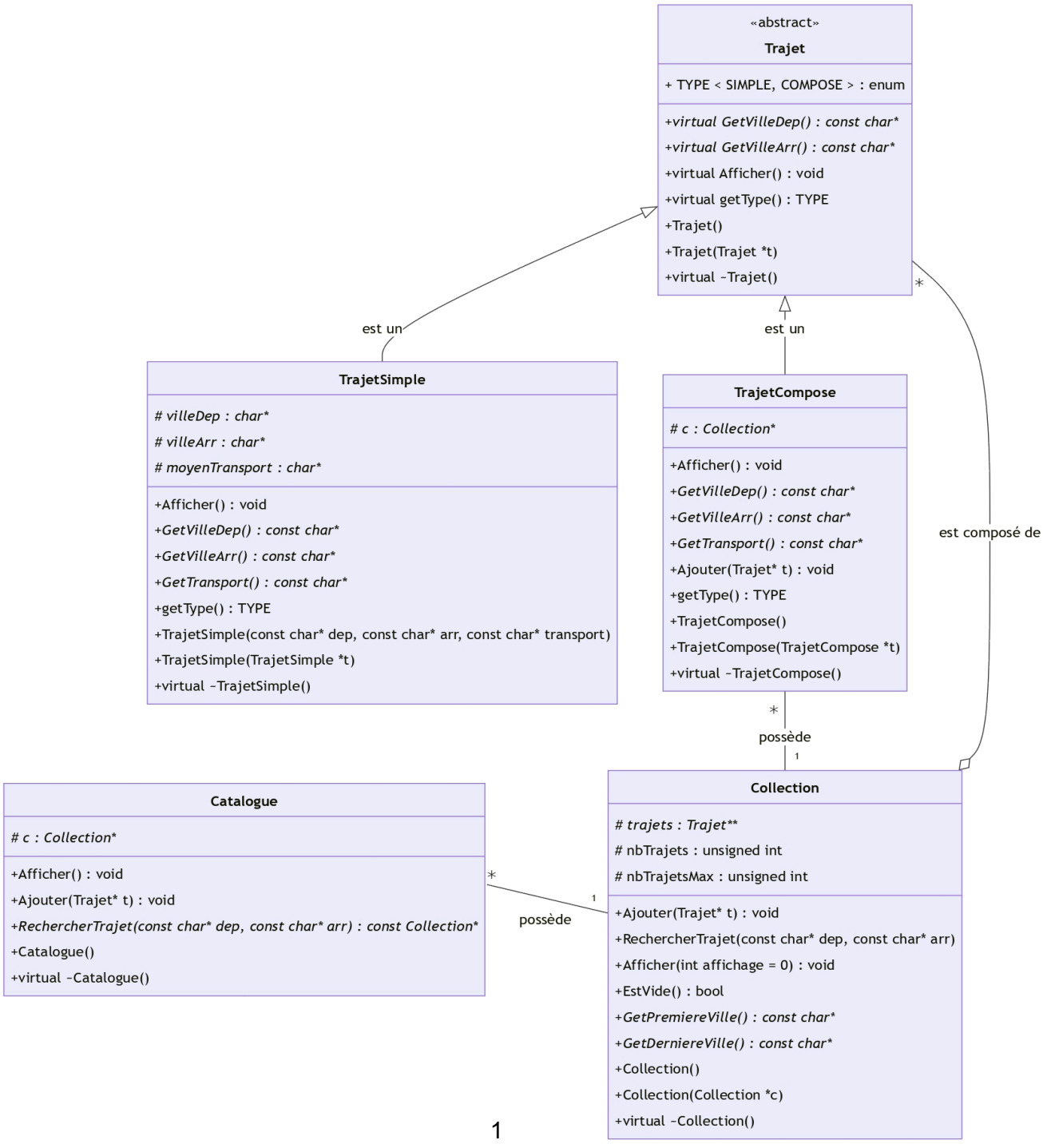
Compte rendu TP C++ n°2 : Héritage – Polymorphisme

Diagramme de classes



Description détaillée des classes

1. Classe `Catalogue`

- **Responsabilité :**
  La classe `Catalogue` gère une collection de trajets (simples ou composés) et offre des fonctionnalités pour ajouter un trajet et rechercher des trajets selon une ville de départ et d'arrivée.
- **Attributs :**
  - `Collection *c` : Une collection de trajets.
- **Méthodes :**
  - `void Afficher()` : Affiche tous les trajets du catalogue.
  - `void Ajouter(Trajet* t)` : Ajoute un trajet à la collection.
  - `const Collection* RechercherTrajet(const char* dep, const char* arr)` : Recherche des trajets dans la collection en fonction de la ville de départ et d'arrivée. Retourne un tableau de pointeurs de collections.
  - Constructeur et destructeur virtuel.

2. Classe `Collection`

- **Responsabilité :**
  La classe `Collection` est responsable de stocker plusieurs trajets (simples ou composés). Elle assure la gestion des trajets avec des méthodes pour les ajouter et les rechercher.
- **Attributs :**
  - `unsigned int nbTrajets` : Nombre de trajets actuellement dans la collection.
  - `unsigned int nbTrajetsMax` : Capacité maximale de la collection (permet d'anticiper l'ajout de trajets).
  - `Trajet **trajets` : Tous les trajets stockés dans la collection (tableau de pointeurs sur des trajets).
- **Méthodes :**
  - `void Ajouter(Trajet* t)` : Ajoute un trajet à la collection.
  - `const Collection* RechercherTrajet(const char* dep, const char* arr) const` : Recherche un trajet selon la ville de départ et d'arrivée.
  - `void Afficher(int affichage = 0) const` : Affiche les trajets dans la collection, avec une option pour un affichage personnalisé.
  - `bool EstVide() const` : Indique si la collection est vide.

- `const char* GetPremiereVille()` et `const char* GetDerniereVille()` : Renvoient respectivement la première et la dernière ville de la collection. Cela est utile pour les trajets composés qui utilisent cette classe. Cela nous permet d'avoir la première et la dernière ville de celui-ci pour connaître le départ et l'arrivée du trajet composé.
- Constructeur, Constructeur par recopie et destructeur virtuel.

3. Classe abstraite `Trajet`

- **Responsabilité :**
  La classe `Trajet` est une classe abstraite qui définit une interface commune pour tous les types de trajets. Les classes dérivées implémentent des méthodes virtuelles pour fournir un comportement spécifique aux trajets simples et composées.
- **Attributs :**
  - `enum TYPE {SIMPLE, COMPOSE}` : Un énumérateur indiquant le type de trajet (simple ou composé).
- **Méthodes abstraites :**
  - `virtual const char* GetVilleDep()` : Renvoie la ville de départ du trajet.
  - `virtual const char* GetVilleArr()` : Renvoie la ville d'arrivée du trajet.
  - `virtual void Afficher()` : Affiche les détails du trajet.
  - Constructeur, Constructeur par recopie et destructeur virtuel.

4. Classe `TrajetSimple` héritant de la classe abstraite `Trajet`

- **Responsabilité :**
  La classe `TrajetSimple` est une classe héritant de la classe Trajet. Elle représente un voyage d'un point A à un point B sans escale avec un moyen de transport.
- **Attributs :**
  - `char* villeDep` : Chaîne de caractères représentant la ville de départ du trajet (exemple : "Strasbourg")
  - `char* villeArr` : Chaîne de caractères représentant la ville d'arrivée du trajet (exemple : "Lyon")
  - `char* moyenTransport`: Chaîne de caractères représentant le moyen de transport du trajet (exemple : "Avion")
- **Méthodes :**

- `void Afficher ()` : Affiche le trajet à l'écran
- `const char* GetVilleDep ()` : Renvoie la ville de départ du trajet
- `const char* GetVilleArr ()` : Renvoie la ville d'arrivée du trajet
- `const char* GetTransport ()` : Renvoie le moyen de transport du trajet
- `TYPE GetType()` : Retourne le type de trajet (ici simple).
- Constructeur, Constructeur par recopie et destructeur virtuel.

5. Classe `TrajetCompose` héritant de la classe abstraite `Trajet`

- **Responsabilité :**
  La classe `TrajetCompose` est une classe héritant de la classe Trajet. Elle représente un voyage d'un point A à un point B avec une ou plusieurs escales. C'est un enchaînement de trajets simples ou composés. Il peut donc y avoir plusieurs moyens de transport dans un trajet composé.
- **Attributs :**
  - `enum TYPE {SIMPLE, COMPOSE}` : Un énumérateur indiquant le type de trajet (simple ou composé).
- **Méthodes :**
  - `void Afficher ()` : Affiche le trajet à l'écran
  - `const char* GetVilleDep ()` : Renvoie la ville de départ du trajet
  - `const char* GetVilleArr ()` : Renvoie la ville d'arrivée du trajet
  - `void Ajouter (Trajet *t)` : Ajoute un trajet au trajet (donc une ou plusieurs étapes suivant si le trajet ajouté est simple ou composé)
  - `TYPE GetType()` : Retourne le type de trajet (ici simple).
  - Constructeur, Constructeur par recopie et destructeur virtuel.

Description détaillée de la structure de données utilisée

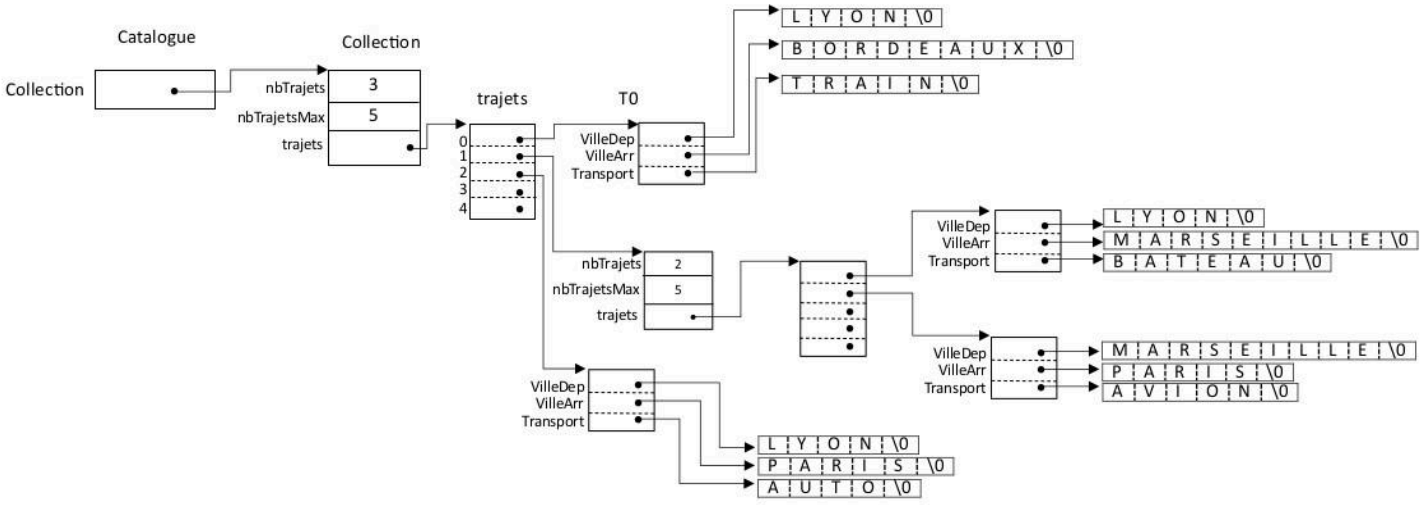
Pour réaliser cette petite application, nous avons utilisé, comme structure de donnée, un tableau de pointeurs vers des "Trajet", un nouveau type que nous avons déclaré à travers la classe "Trajet".

Ce tableau de pointeurs n'est pas directement accessible, il est géré à travers la classe "Collection" qui est elle-même gérée par la classe "Catalogue". (ou dans certains cas par la classe "TrajetCompose")

L'utilisateur crée donc, en ouvrant le programme, un catalogue qui permettra d'ajouter des trajets dans la collection reliée à ce catalogue. Ces trajets peuvent être des trajets simples ou des trajets composés.
Les trajets simples sont uniquement trois chaînes de caractères, ville de départ, d'arrivée et moyen de transport. Les trajets composés sont, eux, des agrégations de trajets simples ou composés. Ils ont donc aussi une collection de trajets.
Au niveau de la mémoire, si on reprend l'exemple proposé :

- 1 trajet simple de Lyon à Bordeaux en train
- 1 trajet compose de Lyon à Marseille en bateau puis de Marseille à Paris en avion
- 1 trajet simple de Lyon à Paris en auto

Notre mémoire ressemblera typiquement au schéma ci-dessous:



Conclusion

Lors de notre travail, nous avons rencontré plusieurs problèmes.

Conception de l'application

Le premier a été la modélisation de l'application. En effet, nous n'avions pas remarqué que le trajet composé utilisait la même structure de données que le catalogue. Nous allions donc doubler la quantité de code pour rien. Finalement, nous avons créé la classe "Collection" qui nous a permis de réduire la quantité de code. Cela permet d'être plus rapide au développement (2x moins de tests) et permet aussi une meilleure maintenance du code s'il y en avait eu une à

faire. Le schéma de mémoire n'a pas été simple non plus, mais avec plusieurs réflexions, nous avons réussi à faire le bon schéma. Il a été très utile lors des développements pour savoir exactement quels types choisir et savoir à quelles données nous accédions.

Recherche de Trajets

Nous avons également rencontré une difficulté que nous n'avions pas notée tout de suite au niveau de la recherche de trajet. En effet, notre recherche parcours le tableau de trajets dans la collection et nous renvoie une nouvelle collection de trajets correspondants aux critères. Cette collection est affichée puis est directement supprimée. Or nous n'avions pas fait une copie en profondeur mais en surface des trajets présents dans cette collection de résultat, ce qui fait que nous supprimions les trajets du catalogue en supprimant les résultats de la recherche. Pour ce faire, nous avons créé 4 constructeurs par copie (Dans les classes Trajet, TrajetSimple, TrajetCompose et Collection) ce qui nous a permis de recopier les trajets dans les résultats en profondeur et de ne plus supprimer les trajets du catalogue.

Fuites de mémoire

Enfin, nous avions aussi quelques problèmes de fuite de mémoire, détectés par valgrind. Le problème venait principalement de la destruction des objets de la classe "Collection". En effet, nous détruisions le tableau de pointeur mais ne détruisions pas ce qui était pointé par ces pointeurs. Nous avons donc rajouté une boucle dans le destructeur pour détruire chaque trajet avant de détruire le tableau de pointeurs.

Propositions d'évolutions

Au niveau des évolutions possibles, nous pourrions déjà réaliser la recherche avancée.
Ensuite, nous pourrions créer une interface utilisateur plus utilisable et plus agréable. Nous pourrions également penser à ajouter de nouveaux attributs aux trajets, comme le prix d'un trajet, la durée d'un trajet simple, les horaires de départ et d'arrivée ainsi que le temps entre chaque trajet pour les trajets composés. Ensuite, nous pourrions donc affiner la recherche pour rechercher sur ces nouveaux critères.
Pour aller encore plus loin, il serait intéressant de connecter tout cela à une API pour récupérer une base de données de trajets pour pouvoir les intégrer et faire de vraies recherches.