Projet de C++ : Gestion de trajets

Vos trajets n'ont jamais été aussi simples !

Table des matières

Note : Tapez `*make dev*` pour compiler avec -DMAP et voir les traces lors de l'exécution.
Tapez `*make help*` pour plus d'information sur les différentes commandes.

I. Description des classes

Voici une description sommaire des différentes classes créées. Un diagramme UML de l'application est fournie en annexe.

Catalogue : Contient une liste de trajets. À sa construction, le catalogue est vide. Nous pouvons ajouter y ajouter des trajets simples ou composés, ces trajets seront ajoutés de façon que la liste soit triée alphabétiquement selon le nom de la ville d'arrivée des trajets. Nous pouvons afficher les trajets ou en rechercher grâce aux villes de départ et d'arrivée.

Trip : Classe abstraite représentant un trajet simple ou composé. La méthode virtuelle *Copy* permet de créer une copie d'un trajet simple ou d'un trajet composé.

SimpleTrip : Trajet simple constitué d'une ville de départ et d'arrivée ainsi que d'un moyen de transport. On peut afficher ce trajet. Hérite de la classe *Trip*.

CompoundTrip : Trajet composé constitué d'une liste de trajets simples. On peut afficher cette liste de trajets simples. Hérite de la classe *Trip*.

TripList : Implémentation d'une liste chainée pouvant contenir une série ordonnée de trajets. On peut ajouter des trajets en fin de liste (dans le cas d'un trajet composé) ou alphabétiquement triés selon le nom de la ville d'arrivée (dans le cas du catalogue). Cette liste permet de créer des itérateurs permettant de parcourir ses maillons.

ElementTripList : Maillon d'une liste chainée *TripList*. Contient un pointeur vers le trajet correspondant ainsi qu'un pointeur vers le maillon suivant.

Iterator : Itérateur permettant de parcourir les maillons d'une liste chainée *TripList* sans avoir besoin d'y accéder depuis des classes extérieures. La méthode *Next* renvoie le trajet correspondant au maillon courant et passe l'itérateur sur le maillon suivant.

Nous avons jugé pertinent de créer les classes *SimpleTrip* et *CompoundTrip* héritant d'une classe Trip abstraite car un trajet est inévitablement ou simple ou composé.

Les méthodes *Copy*, *Display*, *GetStartCity* et *GetFinishCity* sont communes aux trajets simples et composés et sont donc définies comme des méthodes virtuelles de la classe *Trip*.

Pour un trajet simple, les villes de départ et d'arrivées sont stockées en tant qu'attribut et pour un trajet composé, la ville de départ correspond à la ville de départ de son premier trajet tandis que la ville d'arrivée correspond à la ville d'arrivée de son dernier trajet.

Le graphe d'héritage est dessiné sur le diagramme UML disponible en annexe.

II. Structure de données

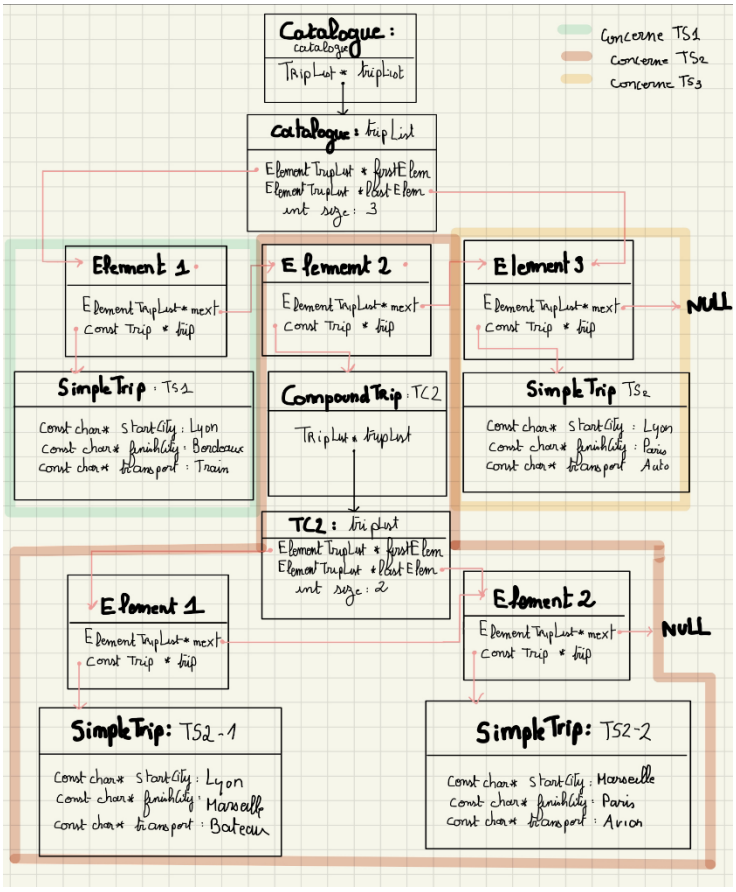
Nous avons décidé d'opter pour l'implémentation d'une liste chainée pour structurer les données car il faut gérer un nombre de trajets dynamique. L'avantage de la liste chainée par rapport à un tableau dynamique est sa complexité moindre lors de l'ajout de trajets en fin ou en milieu de liste (nous avons ici un pointeur de fin de liste). Un tableau dynamique est plus rapide lors de la lecture à un index précis, mais nous n'avons jamais besoin d'accéder à un index spécifique dans notre application. Le choix d'une liste chaînée nous a donc parut le plus pertinent.

La liste chainée est implémentée par la classe *TripList* qui a pour attribut son premier et son dernier maillon ainsi que sa taille. Cela permet de pouvoir rapidement accéder au premier et au dernier trajet ce qui s'avère utile pour les trajets composés qui nécessitent un accès aux villes de départ et d'arrivée. *TripList* est également utilisée pour stocker les trajets du catalogue.

Les maillons sont implémentés par la classe *ElementListTrip* et stockent un pointeur vers un trajet de type Trip, indépendamment qu'ils soient simples ou composés. Cependant, il ne faut pas pouvoir accéder aux maillons depuis des classes extérieures à *TripList*. La problématique du parcours de la liste s'est alors posée. Pour remédier à cela, nous avons crée une classe *Iterator* que *TripList* permet d'instancier en lui donnant en argument son premier maillon, et qui permet de parcourir les maillons de la liste avec la méthode *Next*. Lorsque l'itérateur arrive au bout de la liste, *Next()* renvoie *NULL*.

Ci-contre un schéma de l'organisation de notre structure de données dans le cas du jeu de données fournit dans l'énoncé (catalogue constitué de **TS1**, **TC2** composé de **TS2-1** et **TS2-2**, et **TS2**).

Un dessin de la mémoire après insertion de ce même jeu de données est également disponible en annexe.



III. Conclusion et axes d'améliorations

Nous avons rencontré plusieurs difficultés lors de la réalisation de ce TP. Tout d'abord, nous avons dû être très vigilants quant aux fuites de mémoires. L'utilisation de *valgrind* nous a permis de rapidement les repérer et les corriger.

Ce TP nous a également appris à être rigoureux en ce qui concerne la manipulation des *const* et nous a fait comprendre à quel point il était important de les mettre dès le départ lorsque c'est possible, pour ne pas devoir remodifier tout le code après coup.

Nous avons fait le choix de nous familiariser avec l'utilisation des passages par références et des constructeurs par copie au lieu des passages par pointeurs ce qui nous a confronté à certaines difficultés telles que la copie d'un trajet de type *Trip* (résolu avec la création d'une méthode virtuelle *Copy*).

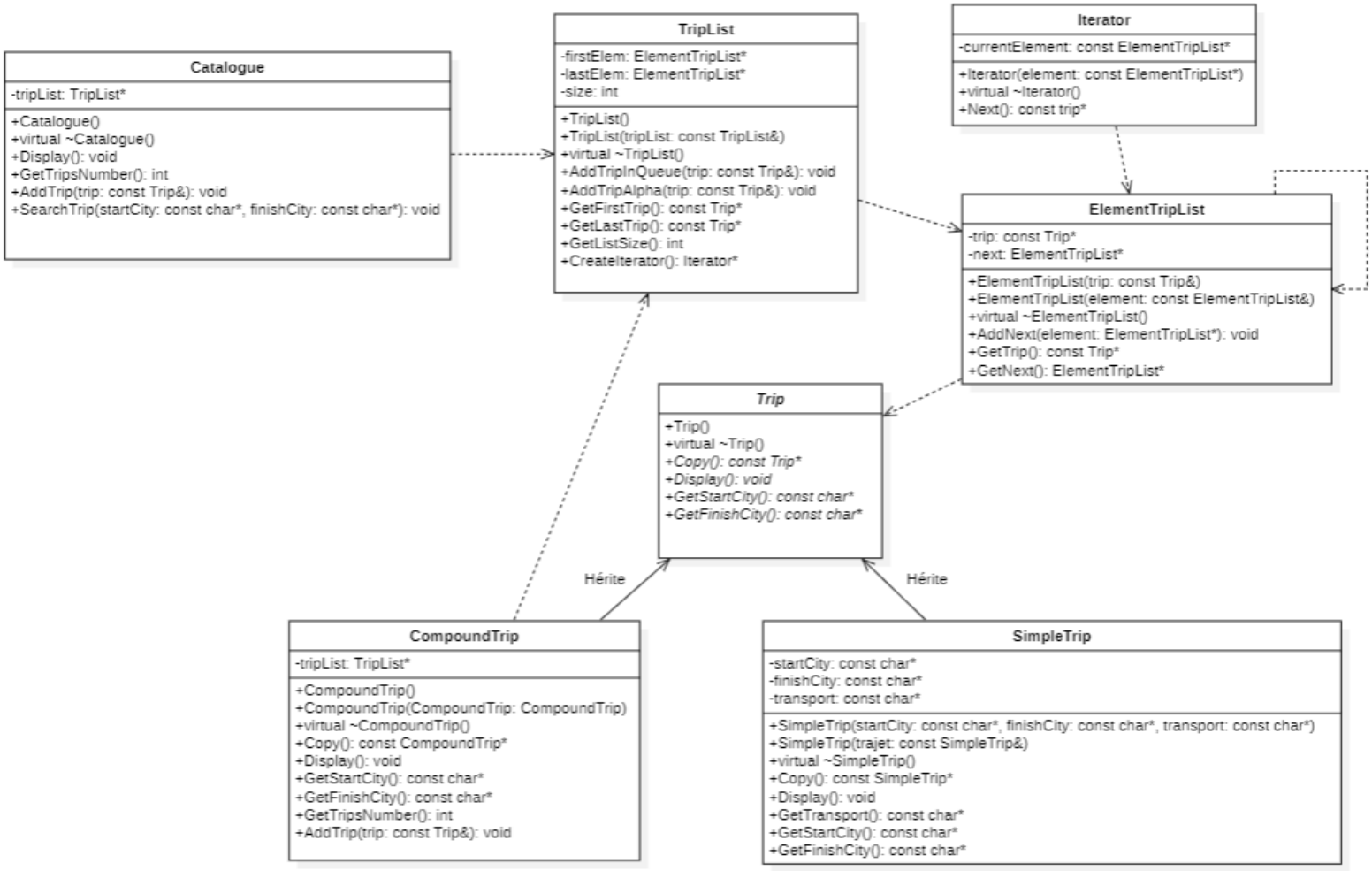
Nous avons découvert tout le potentiel et la flexibilité que *makefile* peut offrir (n'hésitez d'ailleurs pas à taper *`make help`* pour obtenir des informations sur les différentes commandes).

Afin de développer plus facilement en binôme, nous avons utilisé le gestionnaire de dépôt *GitHub*. Malgré quelques conflits à gérer lorsque nous travaillions en même temps sur un même fichier, *Git* nous a grandement simplifié la tâche.

Nous avons commencé à réfléchir à une manière d'implémenter la recherche de trajets avancée, récursivement avec un parcours d'arbre en profondeur et en veillant à éviter que les trajets ne bouclent à l'infini, mais nous n'avons pas eu le temps de l'implémenter.

IV. Annexes

1. Diagramme UML



2. Dessin de mémoire