

TP2 C++ : Héritage - Polymorphisme

Introduction

Ce TP a pour but de nous familiariser avec les concepts et mécanismes de base des langages à objets (classes, héritage, polymorphisme...) et avec leur expression en langage C++.

Pour atteindre cet objectif, nous nous sommes appuyés sur une application simple visant à proposer des parcours pour un voyage défini par une ville de départ et une ville d'arrivée, et tout cela à partir d'un catalogue de trajets à implémenter par nos soins. Ce problème donne lieu à la manipulation d'un objet correspondant à une collection d'objets hétérogènes qu'il faut gérer sans distinction de leur nature en exploitant le polymorphisme et la liaison dynamique.

NB : Tout le long de ce rapport, vous pouvez cliquer sur les images afin de les ouvrir dans une application dédiée à cela. Vous pouvez aussi accéder aux images manuellement dans le dossier *Annexe* fourni dans le livrable.

I. Description simple des classes

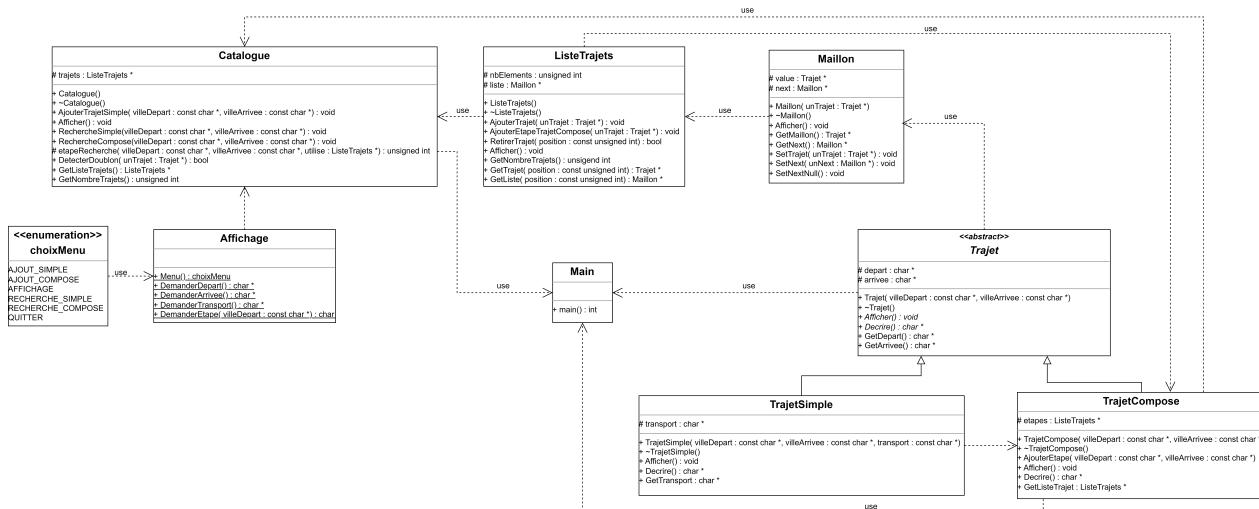


Figure 1: Diagramme UML de notre application

- **Trajet** est une classe abstraite définissant ce qu'est un trajet à partir d'un point de départ et un point d'arrivée.
- **TrajetSimple** est une classe héritant de la classe abstraite **Trajet**. Elle définit un trajet simple, à savoir un **Trajet** auquel on ajoute un moyen de transport.
- **TrajetCompose** est une classe héritant également de la classe abstraite **Trajet**. Elle définit un **Trajet** muni d'une liste d'une ou plusieurs étapes représentées par des **TrajetSimple**s.
- **Maillon** est une classe représentant un élément venant composé la liste chaînée **ListeTrajets**. **Maillon** possède un **Trajet**, simple ou composé, et un pointeur sur le **Maillon** qui suit dans la liste chaînée.
- **ListeTrajets** est classe définissant une liste chaînée composée de pointeur de **Maillon**, et d'un nombre d'éléments.
- **Catalogue** est une classe permettant de stocker tous types de trajets sans doublon, d'en ajouter, de les afficher ou d'effectuer une recherche simple ou complexe. Elle est notamment munie d'un pointeur sur une **ListeTrajets**.
- **Affichage** est une classe dédiée à la gestion des interactions possibles entre le **Catalogue** et l'utilisateur.

II. Description détaillée de la structure de données utilisée pour gérer la collection ordonnée de trajets

Pour parvenir à réaliser cette description détaillée, nous nous sommes appuyés sur le jeu d'essai proposé par l'énoncé de ce travaux pratique (cf. énoncé). La légende des dessins de la mémoire est située [ici](#).

II.1. Création du catalogue C

1) Crédit de la mémoire c :



Figure 2: Dessin de la mémoire lors de la création du catalogue

II.2. Ajout de TS1 au catalogue

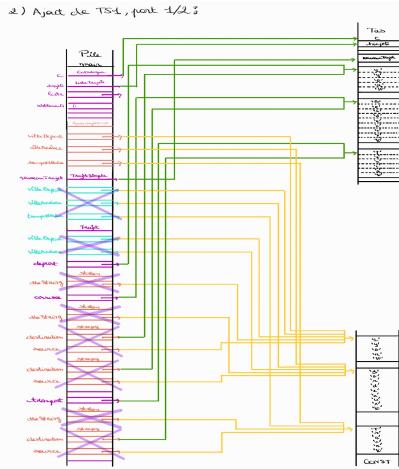


Figure 3: Dessin de la mémoire lors de l'ajout de TS1, partie 1/2

On fait ensuite appel à la méthode AjouterTrajetSimple de **Catalogue** pour pouvoir ajouter le trajet TS1 au catalogue sur lequel pointe C . Cette méthode vient construire dynamiquement un pointeur *nouveauTrajet* pointant sur un **TrajetSimple** qui vient initialisé dynamiquement les attributs de ce **TrajetSimple**, à savoir les pointeurs *depart*, *arrivee*, et *transport* à partir des paramètres constants de AjouterTrajetSimple et des fonctions strcpy et strlen provenant de la bibliothèque *cstring*.

2) Ajout de TS1, point 2/2 :

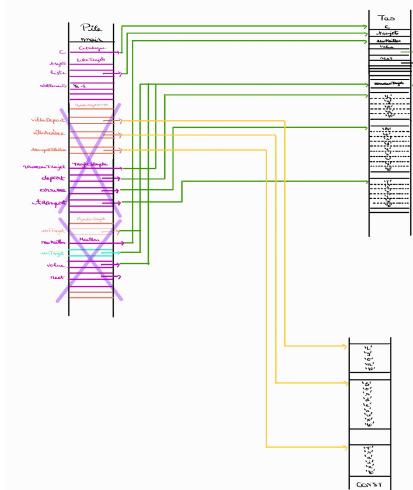


Figure 4: Dessin de la mémoire lors de l'ajout de TS1, partie 2/2

Une fois créée, on vient appeler la méthode AjouterTrajet de **ListeTrajets**. Cette méthode construit dynamiquement un pointeur *newMaillon* pointant sur un **Maillon** auquel on fait pointer son attribut *value* sur le *nouveauTrajet*, et son attribut *next* sur **NULL**. *newMaillon* est par la suite chaîné à la **ListeTrajets** *trajets* du **Catalogue** sur lequel pointe C en faisant pointer son attribut *liste* sur le **Maillon** sur lequel pointe *newMaillon*, et en mettant à jour son attribut *nbElements* en l'incrémentant de 1. Le chaînage suit la mécanique d'une liste chaînée basée sur des pointeurs.

II.3. Ajout de TC2 au catalogue

On commence par créer dynamiquement un pointeur *nouveauTrajet* pointant sur un **TrajetCompose** auquel on initialise dynamiquement ses pointeurs *depart* et *arrivee* à partir d'une copie en profondeur employant les fonctions *strlen* et *strcpy*. Son attribut *etapes* pointant sur une **ListeTrajet** est lui initialisé de sorte à ce que son sous attribut *liste* pointe sur NULL et que son autre sous attribut *nbElements* soit initialisé à 0. Une fois mise en place, on fait appel deux fois à la méthode *AjouterEtape* de **TrajetCompose** pour chaîner les 2 étapes de TC2 à son attribut *etapes*. Pour y parvenir, on construit dynamiquement un pointeur *nouveauTrajet* pointant sur un trajet simple à partir des paramètres constants de la méthode (cf. II.2.). Ensuite, on emploie la méthode *AjouterEtapeTrajetCompose* de **ListeTrajets** en créant un pointeur *newMaillon* pointant sur un **Maillon** auquel on fait pointer son attribut *value* sur le *nouveauTrajet*, et son attribut *next* sur NULL. Ce maillon est par la suite chainé à la **ListeTrajets etapes** du **TrajetCompose** *nouveauTrajet*, en faisant pointer son attribut *liste* sur le **Maillon** sur lequel pointe *newMaillon*, et en affectant la valeur 1 à son attribut *nbElements*. Le chaînage des étapes du trajet composé suit la mécanique d'une liste chaînée basée sur des pointeurs, avec un ordre FIFO.

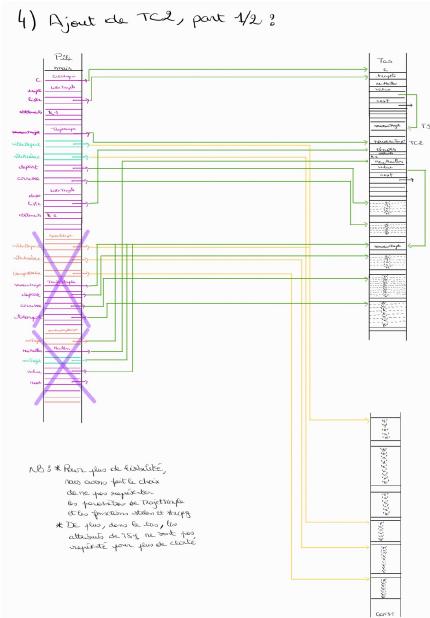


Figure 5: Dessin de la mémoire lors de l'ajout de TC2, partie 1/2

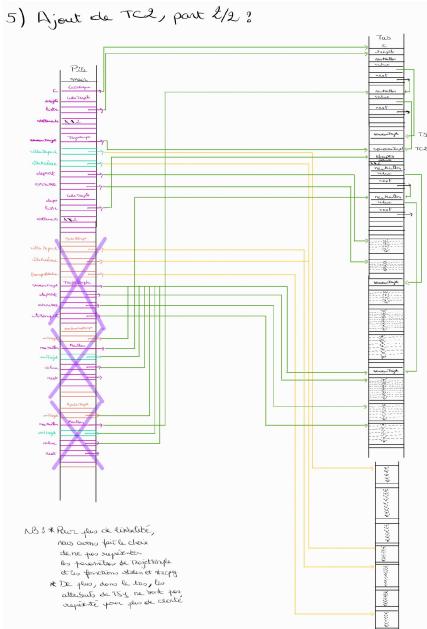


Figure 6: Dessin de la mémoire lors de l'ajout de TC2, partie 2/2

Une fois que le **TrajetComposé** *nouveauTrajet* est initialisé correctement, on refait appel à la méthode *AjouterTrajet* de **ListeTrajets**. Cette méthode construit dynamiquement un pointeur *newMaillon* pointant sur un **Maillon** auquel on fait pointer son attribut *value* sur le *nouveauTrajet*, et son attribut *next* sur NULL. Il est intéressant de noter que la **ListeTrajets** *trajets* peut tout autant prendre un **TrajetSimple** qu'un **TrajetCompose** grâce au principe de polymorphisme et d'héritage qui sied bien à notre structure de donnée gérant les trajets du catalogue. *newMaillon* est par la suite chaîné à dans la **ListeTrajets** *trajets* du catalogue en faisant pointer son attribut *liste* sur le **Maillon** sur lequel pointe *newMaillon*, et en mettant à jour son attribut *nbElement* en l'incrémentant de 1. Ici, le chaînage employé dans la méthode *AjouterTrajet* de **ListeTrajets** suit la mécanique d'une liste chaînée ordonnancée dans l'ordre lexicographique des **Trajets**. En effet, les comparaisons entre les différents trajets (simple ou composé) se font à partir de la méthode *Décrire* de **TrajetSimple** et **TrajetCompose** qui octroie une chaîne de caractères décrivant un **TrajetSimple** ou un **TrajetCompose**. À partir de cette description, on peut aisément comparer les **Trajets**, sur lesquelles *value pointe*, de chaque maillon grâce à la fonction *strcmp* tirant son origine de la bibliothèque *cstring*.

II.4. Ajout de TS3 au catalogue

Pour finir, on s'appuie sur la méthode `AjouterTrajetSimple` de `ListeTrajets` pour pouvoir ajouter le `TrajetSimple` TS3 au catalogue sur lequel pointe C. Cette méthode vient construire dynamiquement un pointeur `nouveauTrajet` pointant sur un `TrajetSimple` qui vient initialisé dynamiquement les attributs de ce `TrajetSimple` (cf. II.2.). Une fois établi, on vient appeler la méthode `AjouterTrajet` de `ListeTrajets`. Cette méthode construit dynamiquement un pointeur `newMaillon` pointant sur un `Maillon` auquel on fait pointer son attribut `value` sur le `nouveauTrajet`, et son attribut `next` sur `NULL`. `newMaillon` est ensuite chaîné à la `ListeTrajets` `trajets` du `Catalogue` sur lequel pointe C en faisant pointer son attribut `liste` sur le `Maillon` sur lequel pointe `newMaillon`, et en mettant à jour son attribut `nbElements` en l'incrémentant de 1. Comme cité précédemment, le chaînage employé dans la méthode `AjouterTrajet` suit la mécanique d'une liste chaînée ordonnancée par le biais de l'ordre lexicographique des `Trajets`. (cf. II.3.).

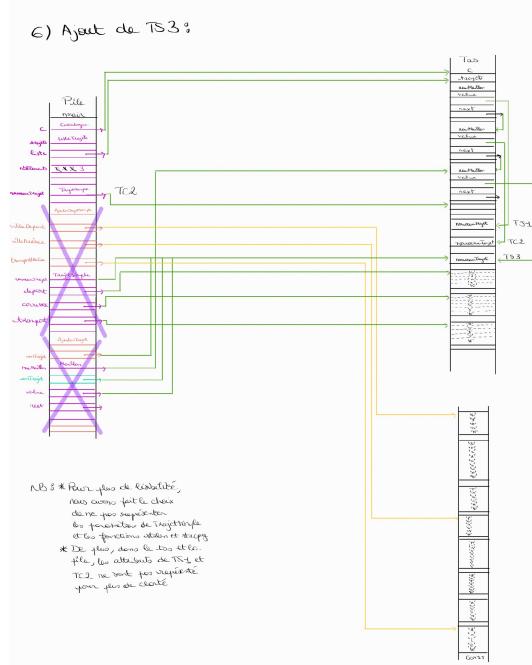


Figure 7: Dessin de la mémoire lors de l'ajout de TS3 au catalogue

NB : L'allocation dynamique de la mémoire s'appuie sur la notion de pointeurs, utilisée à outrance dans notre projet. Cela implique alors qu'à chaque ligne de code, nous avons pris soin d'être sûr de toujours libérer la mémoire à partir du moment où on l'a alloué, de manière à ne pas perdre de données, ou pire, à en faire fuiter.

III. Conclusion

Pour conclure, notre travail a fini par mener ses fruits malgré un chemin semé d'embûches. En effet, la manipulation de pointeurs nous a souvent fait perdre la tête, notamment à cause des oubli de libération de la mémoire, mais aussi des pointeurs qui ne pointaient pas là où l'on pensait. À cela s'ajoute également toute la complexité de l'allocation dynamique de la mémoire et l'amoncellement des lignes de codes qui demande une grande rigueur dans le codage, ce que nous n'avions pas au début de ce TP. Mais bien heureusement, nous avons su surmonter ces épreuves grâce à un travail graduel, des tests à chaque étape du codage, et bien évidemment, grâce aux commentaires avisés de valgrind, qui, bien que parfois décourageant, nous ont toujours mené à bon port.

Il convient de noter que notre réalisation finale répond certes en grande partie au cahier des charges imposé, mais pour autant, elle n'est pas parfaite. Effectivement, cette dernière se voit posséder de multiples axes d'évolutions et d'améliorations qui auraient pu prendre place si le temps nous était bien plus clément. D'une part, le menu aurait pu être beaucoup plus peaufiné de sorte à ce que l'utilisateur soit ravi tant par la beauté de celle-ci, que par son utilisation. Par exemple, faire ce menu par le biais d'une interface graphique aurait été la bienvenue. D'autre part, nous aurions pu essayer de réaliser une recherche avancée un tant soit peu plus complexe en permettant de rechercher des trajets avec un moyen de transport spécifique, ou encore de rechercher des compositions de trajet au sein même des étapes d'un trajet composé. La tâche est certainement plus ardue, mais le jeu en vaut la chandelle si tant est qu'on aime les défis algorithmiques. Par ailleurs, appliquer certaines notions vues dans le chapitre C++ avancée (tel que la surcharge d'opérateur d'entrée/sortie, ou encore le principe de généricité) fait partie des pistes d'évolution à penser et à peut-être mettre en place sur ce projet, dans un avenir proche où le chapitre C++ avancée nous sera claire comme de l'eau de roche.