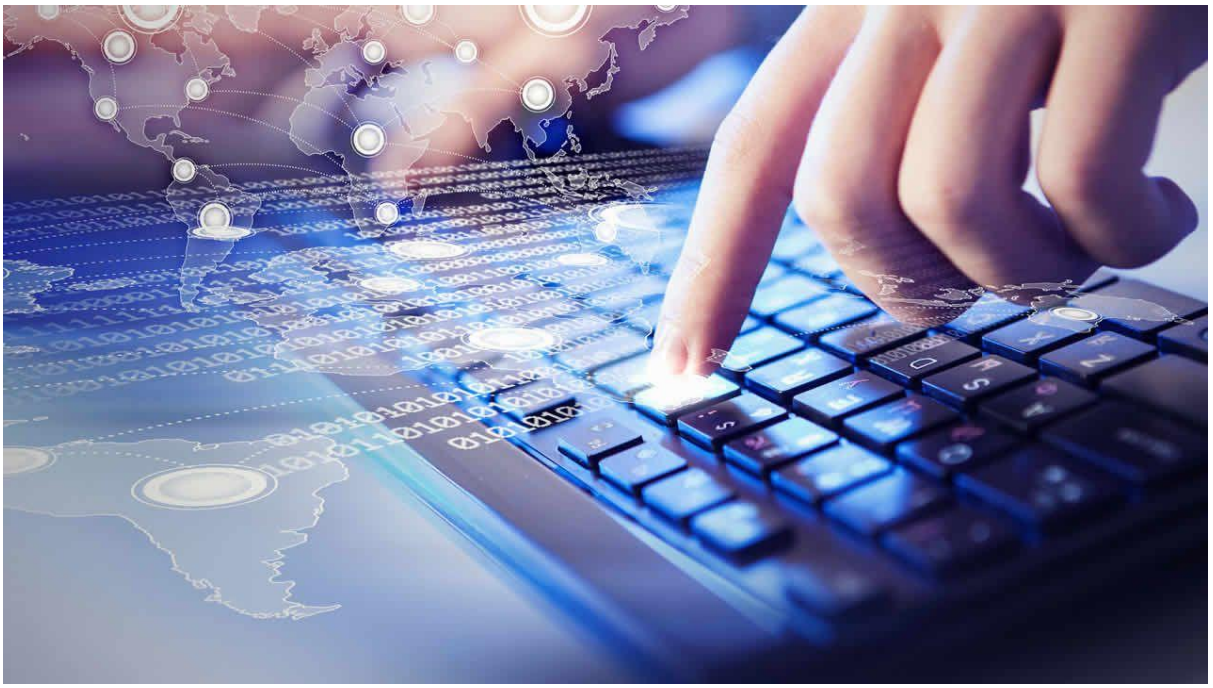
Compte Rendu TP de C++



M. Ginhac Dominique

ROB 3A
Elie Bienvenu
Baptiste Lesquerré

Table des matière :

TP1

Le choix du TP

Notre choix c'est porter sur le TP 1 pour plusieurs raisons. Tout d'abord, nous sommes deux débutants en C++, et le choix d'un TP accessible permet nous a permis d'ajuster notre charge de travail et de mieux nous répartir les tâches durant le codage.

De plus le TP 1 fait appel à des connaissances fondamentales qu'il est important d'écrire avant de se lancer dans des projets de plus grandes ampleurs.

Enfin le TP 1 est ludique, car il traite de la réalisation d'un gestionnaire de bibliothèque, un cas concret que l'on peut croiser en dehors du cadre scolaire. Cet aspect permet de mieux s'investir et de plus s'impliquer dans la réalisation de ce TP.

Répartition du travaille

Nous avons avancé question par question, en nous répartissant les questions qui étaient dépendantes afin que chacun puisse avancer à son rythme. Cela a mené à la rencontre de problèmes plus tard dans le code, notamment lorsque nous nous sommes rendu compte que l'un utilise systématiquement les namespace alors que l'autre non. Nous avons finalement opté pour les enlever, facilitant ainsi la lecture des méthodes et des classes et nous faisant gagner un temps précieux lors de l'écriture du code.

Organisation de la programmation a deux

Sur le TP 1, nous avons décidé des normes suivantes :

- Les noms des classes commencent par une majuscule
- les noms de fichier sont entièrement en minuscule, avec des tirets du 8 '_' pour les espaces
- Les noms de variables, des fichiers et des classes doivent être explicités.
- L'utilisation des méthodes prime sur l'utilisation des fonctions helpers

Outils utilisé

Si l'utilisation de github a pour nous deux était une évidence, nous n'utilisons en revanche pas le même IDE. Baptiste code sur Visual Studio Code et Élie code sur CLion.
L'utilisation du "Github : petit guide, no deep shit" nous a particulièrement aidés dans l'appréhension du nouveau outil que représente Github pour nous.

Organisation des classes C++ & Choix technique retenues

Class Date

La classe Date a pour variable privée jour, mois et année.

Les méthodes de la classe Date sont ;

- Son constructeur, qui prend en paramètre 3 int, un pour le jour, un pour le mois et une l'année. Le constructeur de Date vérifie que les paramètres rentrés sont conformes, autrement dit que :
    - le mois est bien compris 1 et 12
    - le jour est compris entre 1 et 31 pour les mois impairs
    - le jour est compris entre 1 et 10 pour les mois pairs
    - le jour est compris entre 1 et 28 pour le mois de février, sauf les années bissextiles ou le jour est compris entre 1 et 29 pour février
- Ses getters pour les variables jour, mois, années.
- Ses setters pour les variables jours, mois, années.
- Une surcharge de l'opérateur <<..
-

Class Auteur

La classe auteur a pour variables privées un identifiant pour chaque auteur, le nom de l'auteur, son prénom et sa date de naissance. La date de naissance fait bien sûr appel à la classe Date dont nous avons parlé ci-dessus.

Les méthodes de la classe auteur sont :

- Son constructeur, qui prend en paramètre une Date, un string identifiant, un string nom et un string prenom. À la création d'un auteur, on assignera simplement chaque paramètre à sa variable privée correspondante.
- Les getters de _identifiant, _ nom, _ prenom et _date_de_naissance.
- Une surcharge de l'opérateur <<.

Class Livre

La classe Livre a pour variables privé un titre string _titre, un auteur Auteur auteur, une langue string auteur, un genre string genre, une date Date date, un numéro ISBN int _ISBN, une liste des identifiants des lecteurs ayant déjà emprunté ce livre vector<string> _id_list, et une variable qui permet de savoir si le livre est disponible à l'heure actuel bool _is_free.

Les méthodes de la classe lecteur sont :

- Son constructeur qui prend en paramètre un titre, une langue, un genre, une date, un ISBN, un Auteur, et un bool permettant de créer un livre existant mais pas forcément disponible par défaut (si ce livre vient d'être acheté mais qu'il n'a pas encore été reçu par exemple)
- Ses getters pour les variables _titre, _langue, _genre, _ISBN, _is_free, _id_list et _auteur
- Un add qui permet d'ajouter un element à _id_list
- Un setter pour changer l'état de la variable _is_free
- Une méthode qui permet d'afficher la date reste d'un test mais qui fait malgré tout partie du programme 🙂
- Une surcharge de l'opérateur <<.

Class Emprunt

La classe emprunt a pour variables privé une date Date _date, un ISBN du livre emprunter int _ISBN et l'identifiant du lecteur qui emprunte string _identifiant.

Les méthodes de la classe Lecteur sont ;

- Sont constructeurs qui prend en paramètre une date Date date, un lecteur Lecteur& lecteur et une livre Livre& livre. Le constructeur vérifie que le livre que l'on essaie d'emprunter est bien disponible avant de réaliser l'emprunt.
- Ses getters pour les variables _dates, _ISBN et _identifiant.

La classe Emprunt a également une fonction helper Resituer, qui permet de restituer un livre. Restituer vérifie que le lecteur qui essaie de restituer le livre emprunté est bien le même lecteur qui l'a emprunté.

Class Lecteur

La classe Lecteur a pour variables privé un identifiant _identifiant, le nom du lecteur _nom, son prénom _prenom, une liste des ISBN des livres que le lecteur a emprunté _ISBN_list et une liste des livres a en cours d'emprunt.

Les méthodes de la classe Lecteur sont :

- Son constructeur, qui prend en paramètre un string identifiant, un string nom et un string prenom.
- Ses getters pour les variables _identifiant, _ nom, _ prenom, _ISBN_list et _emprunt_list
- Un add pour ajouter des éléments à la variable _ISBN_list
- Un remove pour supprimer des éléments à la variable _ISBN_list
- Une méthode Emprunter pour emprunter un livre avec un lecteur. Emprunter fait appel a la classe Emprunt.
- Une méthode Restituer pour restituer un livre à partir d'un lecteur. Restituer fait appel à la fonction helper Restitution.
- Une surcharge de l'opérateur <<.

Conclusion

Le TP 1 nous a appris à travailler en collaboration en utilisant des outils tels que github. Nous avons donc non seulement transformé nos bases théoriques en expérience de programmation mais également appris à utiliser des outils qui permettent une meilleure productivité lors d'un travail de groupe mais aussi lors d'un travail individuel.

TP2

Nous avons choisi ce TP principalement car il était de difficulté facile et nous ne voulions pas nous engager dans un TP trop complexe que nous n'aurions pu finir.
Pour la répartition du travail nous nous sommes réparties les différentes questions du TP et avons tâché de travailler en parallèle. Nous nous concertons régulièrement à propos d'une question ou d'une autre sur différents choix à réaliser ou pour résoudre différents problèmes.
Le compte rendu de ce TP2 à été rédigé par Baptiste Lesquerré.
Au cours de ce TP l'objectif était de réaliser plusieurs classes et de les compléter au fil de celui-ci en créant les méthodes et les fonctions nécessaires de manière à répondre aux attentes de l'énoncé. Pour celà il a été nécessaire de créer des getters, setters, helpers, surcharges d'opérateurs et itérateurs tout en manipulant des vecteurs.
Le but de ce TP était de créer des classes et des programmes permettant la gestion des réservations d'un hôtel.

Création des classes (parti 1):

Les premières questions du TP ont conduit à la création de plusieurs classes : Client, Hotel, Chambre, Date, Reservation. La classe Date a été créée en utilisant la classe date précédemment créé comme exemple dans le cours. Les classes créées à ce point sont les plus minimales possibles. Chaque classe comporte au moins un constructeur, des variables et des getters pour chaque variable. On crée dans certaines classes des setters pour certaines variables s'il seront utiles pour les futures questions(aurions nous besoin de changer ces variables dans le futur?).
Au cours de la création de chaque classe nous avons commencé par créer chacune dans son propre namespace, seulement pour réaliser la difficulté apportée notamment lors de l'appel des méthodes. Nous avons donc décidé de créer un unique namespace "hotel" contenant les classes hotel et chambre, celà facilitant l'interaction entre les deux classes. En effet un objet hotel comporte un tableau de chambre.
Dans différentes classes des variables identifiants sont créées. On les définit comme des chaînes de caractère pour que l'utilisateur puisse écrire ceux-ci comme bon lui semble.
On note que pour la classe Reservation on rajoute la variable "_id_reservation" pour l'identifiant de la réservation. Cette variable sera utile ensuite pour identifier les différentes réservations et les distinguer sachant qu'un client peut réserver plus d'une fois le même type de chambre sur le même créneau.
Certaines classes sont appelé dans d'autre classes (#include "date.h" dans reservation.cpp par exemple). On écrit donc les lignes #ifndef … , #define …, #endif … dans le fichier .h pour que chaque classe ne soit appelée qu'une seule fois lors de l'exécution du programme.
Chaque classe est créée avec un fichier .ccp et un fichier .h. On crée également un fichier main.cpp dans lequel l'utilisateur pourra interagir avec les différents objets créés.
Après avoir écrit chaque classe, on teste le bon fonctionnement de celle-ci (de ces différentes méthodes) grâce à un programme test écrit dans le fichier main.cpp. Nous nous heurtons cependant à certains problèmes notamment au niveau des classes Hotel Chambre et Client . En effet, pour vérifier le bon fonctionnement des setters on se rend compte qu'il serait important de pouvoir afficher les objets clients mais également chambre (un hôtel possédant un vecteur de chambre) pour ainsi mieux vérifier les modifications apportées à

ces objets. Ceci entame la seconde partie du TP.

Création de nouveaux helpers et setters (partie 2):

La question 6 débute par la création d'un hôtel (h1) qui servira pour toute la suite du TP. Cet hôtel contient dix chambres réparties en trois types différents (single, double, suite). On utilise une méthode "cch" qui permet d'ajouter plusieurs chambre du même type (même prix) avec des identifiants différents, a la variable "_tabchambre" de l'hôtel crée. L'utilisateur interagira par la suite avec cet hôtel.

Comme remarqué précédemment il est nécessaire d'afficher les objets créés. L'énoncé nous indique alors de réaliser une surcharge d'opérateur "<<" pour les classes Client, Hotel et Chambre. Pour chacune on commence par créer une méthode "tostring" créant une chaine de caractère figurant les informations d'un objet. La surcharge d'opérateur utilise ensuite cette fonction et affiche ainsi la chaîne de caractère créée. Ceci est très utile rendant visible les informations d'un objet lorsque l'on utilise l'opérateur <<. On réalise ainsi un meilleur contrôle des modifications. On crée également une fonction display pour Client et Chambre. Ces fonctions prennent en paramètre un vecteur (de clients/ de chambre) et utilisent une fonction "tostring" retournant une chaîne de caractère correspondant aux éléments du tableau. On peut ensuite afficher les éléments de ces vecteur de manière ordonnée. On peut ainsi régulièrement surveiller les changements dans ces vecteurs notamment dans le vecteur de chambre l'un des éléments d'un objet hotel.

Un premier exemple d'utilisation de ces fonctions est avec un tableau de client initialisé dans le fichier main.cpp. On crée quatre clients et on les ajoute dans le tableau. On utilise ensuite la surcharge d'opérateur ou la fonction display (couplé avec un "cout") pour afficher les informations de chaque client. On vérifie ensuite s'il y a eu des changements dans les informations saisies.

Dans la question 7, on débute par la création d'une code permettant d'afficher les informations d'un client et/ou d'ajouter celui-ci au tableau de client. Le programme demande à l'utilisateur de saisir le nom du client. On parcourt ensuite le tableau comparant le nom à celui de chaque client enregistré. S'il figure dans le tableau, on affiche les informations du client au moyen de la surcharge d'opérateur et on affecte à un objet string "id" l'identifiant du client (sera nécessaire pour la suite). Si le client n'est pas dans la liste, l'utilisateur remplit les informations du client. On affiche ensuite celle-ci et "id" est affecté du nouvel identifiant. Après l'écriture du code on transforme celui-ci en une fonction helper "saisi_client" de la classe client retournant un objet type string (l'identifiant) et agissant sur un vecteur d'objet Clients. Celà facilite la lisibilité et permet d'exécuter toutes ces différentes étapes en une seule ligne dans le fichier main.cpp.

Au vu des questions suivantes on décide d'écrire un code qui sera également transformé en fonction helper (de la classe Reservation) permettant de saisir une réservation tout en vérifiant la validité de celle-ci : "saisi_reservation". Pour que la réservation soit valide il faut que le créneau (date) soit disponible et que le type de chambre souhaité soit libre pour ce créneau. On commence par initialiser un tableau de Chambre "chambre_dispo" qui contiendra les chambres disponibles. L'utilisateur saisi la date et le nombre de nuit souhaité, la fonction fait appelle a la fonction "creneau_valide" qui vérifie l'existence de créneau déjà réservé chevauchant le créneau désiré. S'il n'y a pas chevauchement on ajoute la chambre concernée dans le tableau "chambre_dispo" qui sera ensuite complété par les chambres sans réservations sur le créneau (fonction "chambre_libre"). Si le créneau est non

disponible, l'utilisateur est invité à en saisir un nouveau. Après cela, une fonction “search_chambre” va vérifier si le type de chambre désiré est présent dans “chambre_dispo”. Si ce n’est pas le cas, l'utilisateur est invité à choisir un nouveau type de chambre. En débordant sur la question 8 on crée une fonction qui calcule le montant de la réservation (dans “search_chambre” si le type de chambre est disponible) et l’affecte à la réservation. Initialement dans “saisi_reservation”, une réservation par défaut a été créée et sera modifiée au fur et à mesure. Lorsque la réservation est terminée, on affiche celle-ci et le montant pour confirmer sa bonne saisie. Pour cela nous avons donc créé une fonction “tostring” et une surcharge d’opérateur “<<” dans la classe Reservation. Finalement on ajoute cette réservation a un tableau de réservation pour conserver celle-ci. Ce tableau est initialisé dans le fichier main.cpp grâce au clients précédemment créé. On remarque cependant qu’il aurait été plus judicieux lors d’une impossibilité dans la réservation, de demander à l'utilisateur s’il souhaite saisir celle-ci ou abandonner la saisie.

Une fois cette étape terminée nous testons différents cas de figure de réservation, différents problèmes de réservation pour ainsi voir si le programme réagit correctement. Malheureusement ce n’est pas le cas et nous nous heurtons a beaucoup de problèmes différents notamment au niveau de la vérification de la validité du créneau et de la disponibilité du type de chambre. En effet plusieurs chambres peuvent être réservées sur le même créneau, certaines chambres semble ne pas pouvoir être disponible, les créneau se chevauchant renvoi une indisponibilité alors que d’autres chambres sont libres…
Pour résoudre ces problèmes, on réalise des débogages manuels et nous utilisons un débogueur dans le but d’identifier les sources de ces problèmes. Certains problèmes sont rapidement résolus, étant dû à une faute de frappe ou une étourderie. D’autres erreurs sont en revanche plus complexes à résoudre et nécessitent une bonne compréhension du problème. On peut citer par exemple pour la fonction “creneau_valide” qu’il était essentiel de bien comprendre ce qui définissait un “créneau valide” par rapport aux autres créneaux sélectionnés dans les réservations. Certains problèmes plus subtiles ne sont découverts que lors d’une manipulation spécifique du programme (problème de deux réservations réalisées sur des créneaux proches par exemple).

On aborde ensuite la question 9. Pour commencer il faut créer une fonction permettant d’afficher toutes les réservations. On procède alors comme avec les classes Client et Chambre en utilisant la fonction “to_string” dans une fonction “display” qui prend en argument un vecteur de réservation et affiche ensuite toutes les réservations en utilisant l’opérateur “<<”. On crée ensuite deux autres fonctions qui vont permettre d’afficher une réservation selon les informations mis en paramètre. “display_reservation” affiche une réservation selon le numero de reservation donné. “display_reservation_client” prend en paramètre le tableau des réservations et celui des clients, et demande à l’utilisateur soit un identifiant soit le nom d’un client. La fonction affichera ensuite les réservations réalisées par ce client. Cette fonction est plus difficile à réaliser et celle-ci utilise une autre sous fonction “display_reservation_clientid” qui détecte si l’objet passé en paramètre est bien un identifiant et affiche dans ce cas la réservation. Cette fonction renvoie un booléen a la fonction “display_reservation_client” lui confirmant s’il s’agit bien d’un identifiant ou pas. Si un nom à été saisi, la fonction recherche l’identifiant correspondant et affiche les réservations (“display_reservation_clientid”).

Avant de continuer dans la progression du TP, on se penche de nouveau sur la génération de réservation. On décide pour simplifier et mieux arranger le code, de réaliser la création de réservation sous forme d'une fonction helper de la classe réservation : "reservation_creator". Celle-ci prend en paramètre un hôtel et un vecteur de client. Elle initialise un vecteur de réservations, puis avec une boucle while elle enregistre les clients et leurs réservations (dans les deux vecteurs). On utilise "saisi_client" et "saisi_reservation". La boucle while demande a chaque itération si l'utilisateur veut continuer à saisir des réservations. S'il refuse (toute autre réponse que "yes") la boucle s'arrête. Après chaque saisie de réservation, le tableau est affiché.
Suite à certains problèmes liés à un tableau de réservation initialement vide, on initialise celui-ci par une réservation 0 d'une salle ne pouvant être réservée par les clients et n'étant pas affichée à chaque affichage du tableau des réservations.

On crée ensuite les fonctions "modif_reservation" et "delete_reservation". On a précédemment divisé la fonction "saisi_réservation" en "saisi_reservation1" et "saisi_reservation2" pour réutiliser "saisi_reservation1" dans "modif_reservation" après la saisie de l'identifiant de la réservation à modifier. Pour la fonction "delete_reservation" on va utiliser un itérateur. Le tableau des réservations est placé en paramètre et après avoir récupéré l'identifiant de la réservation à supprimer, on parcourt le tableau grâce à l'itérateur. Celui-ci étant un pointeur sur les cases mémoires du tableau (successivement) on peut ainsi (utilisant "erase") facilement supprimer la case mémoire de la réservation désirée.
On modifie plusieurs fois le fichier main.cpp pour tester les différentes fonctions récemment implémenter (certaines sont ajoutées à "reservation_creator"). Ce sont les fonctions "display_reservation_client", "delete_reservation" et "modif_reservation" qui posent problème. Pour "modif_reservation" il faut pouvoir modifier la réservation sans créer une seconde réservation (conservation des versions modifiées et non modifiées). Pour "display_reservation_client" le programme doit fonctionner également lorsqu'on saisit le nom d'un client. Pour "delete_reservation" la bonne compréhension de la manipulation d'un itérateur est crucial pour bien parcourir le tableau et supprimer la bonne réservation.
Après avoir résolu ces différents problèmes, on réalise une dernière modification dans "reservation_creator". On sort la création du tableau de réservation de la fonction (ajoutée dans le fichier main.cpp). Dans le cas contraire, le tableau n'existerait qu'au sein de la fonction rendant sa manipulation difficile en dehors de celle-ci.

Conclusion et remarques sur le TP

Ce TP a consisté en une expérience constructive de programmation c++. La réalisation des classes et leurs évolutions au cours du TP a permis de mieux comprendre comment choisir et privilégier certaines méthodes et helper pour parvenir aux fins désirés. Il est important également de réaliser quelles parties du code peuvent être réutilisées pour ainsi éviter d'écrire du code redondant. Ce TP a permis également de bien prendre conscience pourquoi il est plus avisé de rédiger quelques notes sur une feuille que de s'acharner sur le code face à un problème. Ce TP à apporté certains savoir-faire, comme la meilleure compréhension d'un itérateur, d'une surcharge d'opérateur "<<" et de l'écriture des fonctions type "tostring" pour afficher des objets créés. Au niveau des choses à changer ou améliorer, il serait important de mieux ranger et arranger le code des différents fichiers pour qu'il soit présentable et plus lisible (à faire dès le début du TP). Il faut aussi faire attention aux pertes

de temps. En effet certains problèmes était très long à résoudre, et il serait donc intéressant de s’améliorer sur ce front pour ainsi gagner en efficacité et en vitesse. Pour continuer à progresser il faut donc continuer à s'entraîner tout en faisant attention à ces détails.