Braun Gwendal
Pataille Arthur

Compte Rendu des TP de C++

Lors de ces différents TP, nous avons commencé par faire le TP4. Nous sommes ensuite passé au TP5 car c'est la continuation logique du 4°. Par la suite, nous avons débuté TP6 (qui n'est pas fini) et le TP2. Nous avons choisi de présenté dans ce compte rendu le TP2 et le TP5.

Comme nous avions un niveau en C++ différents, nous avons décidé de faire le TP4 qui était un TP de niveau intermédiaire, pour ce faire nous avons principalement suivie les consignes du sujet. Une fois fini, nous sommes passé au TP5 et par la suite nous avons fait le TP2 et Gwendal a commencé le TP6.

Pour tous les TP, nous avons utilisé GitHub pour le partage du code et du Makefile. Chacun de notre côté nous avons utilisé Visual Studio pour programmer. Comme nous avions des problèmes de comptabilité entre nos deux versions de Visual Studio nous n'avons pas inclue les fichiers spécifiques à Visual Studio à GitHub. Nous avons également utilisé WSL pour tester la compilation avec le Makefile.

En ce qui concerne la répartition de travail, Gwendal a davantage travaillé sur les TP 5 et 6 alors qu'Arthur c'est plus occupé du TP2. Aussi Gwendal s'occupait plus de la partie planification du travail et organisation des classes et Arthur de la création des classes et de l'implémentation de leurs fonctionnalités.

TP2 :

Nous avons choisi ce TP puisqu'il permet de travailler avec différentes classes et utilise la surcharge d'opérateur. De plus, l'objectif de faire une réservation d'hôtel est intéressant et ce TP permet de voir une grande partie des notions vu en cours.

Pour commencer, Arthur a fait la partie 1 pendant que Gwendal débutait le TP6. Une fois finis, Gwendal a relu le code et soulevé deux trois points à changer. Puis, nous avons fait la partie 2 ensemble.

Dans ce TP, il y a différentes classes dont nous allons détailler les méthodes et attributs par la suite.

Concernant les classes, il y en a 5 qui sont Date, Client, Room(chambre), Hotel et Reservation.

La classe Date possède 3 attributs privés qui sont des entiers (le jour, le mois et l'année). Pour cette classe, le code du cours a été repris et nous l'avons modifié pour rajouter l'année. Nous avons dû créer un nouveau getter et setter pour l'année et nous avons modifié les méthodes next() ainsi que back() pour que l'année soit prise en compte (par exemple, on peut désormais passé du 31/12/2022 au 1/1/2023 sans problème avec la classe next() et revenir au jour précédent avec la classe back(). Pour cela, nous avons pris en compte les années bissextiles en créant une fonction helper qui vérifie si l'année est une année bissextile. Si c'est le cas, alors nous avons le 29 février si ce n'est pas le cas, alors il y a que 28 jours en février. La fonction helper getDaysInMonth() a aussi été modifié pour prendre en compte le cas des années bissextiles et la fonction helper toString a été modifié pour rajouter l'année. Nous avons également ajouté des surcharges d'opérateur (+, -, <, >, <= et >=) qui

ont été rajouté plus tard dans le TP pour faciliter les opérations avec les Date où nous avons besoin de savoir si deux intervalles de date se chevauche.

La classe Client possède 3 attributs privés dont 1 entier pour l'identifiant et 2 string pour le nom et le prénom. Nous avons 3 getters pour pouvoir accéder à tous les attributs et une surcharge d'opérateur << pour afficher l'ensemble des informations d'un client.

La classe Room (Chambre) possède 7 attributs. Il y a 3 entiers pour le numéro de la chambre, le nombre de lits dans la chambre et le prix de celle-ci. Pour le type de la chambre nous avons utilisé une énumération (RoomType) car on considère qu'il n'y a que 3 types de chambre possibles (Simple, Double et Suite). Il y a un bool pour savoir si la chambre est réservée (reserved). De plus, il y a une structure Reservation qui contient la date de début de réservations et le nombre de nuit pour permettre de connaître à partir de quand et combien de temps la chambre va être réservée et donc indisponible. Nous avons également un vector contenant plusieurs réservations car il peut y avoir plusieurs réservations pour une chambre (pas au même moment). Au niveau des méthodes, nous avons des getters ainsi qu'une surcharge d'opérateur pour afficher les informations de la chambre. De plus, il y a deux autres méthodes permettant d'ajouter une réservation (addReservation()) et de retirer une réservation (removeReservation()). La première vérifie si le nombre de lit demandé est inférieur à 1 ou s'il est supérieur au nombre de lit disponible mais également si le nombre de nuit demandé est inférieur à 1 et si la chambre est déjà réservée au moment voulu. Dans ces cas-là, ont averti l'utilisateur et on n'ajoute pas la réservation. Ensuite, on ajoute la réservation dans le vector donné précédemment. La méthode isReserved() permet de vérifier dans le vector si il y a une réservation dans la entre le début et la fin de réservation demandée.

La classe Hotel possède 4 attributs privés dont 1 entier qui est l'identifiant, 2 string pour le nom de l'hôtel et la ville où il se situe ainsi que un vector de Room qui contient toutes les chambres de l'hôtel. Il y a 4 getters et 2 constructeurs. Un constructeur dans le cas où on n'en met pas de chambre et un où on en met. Il y a une méthode addChambre qui permet d'ajouter une chambre. Dans cette méthode, on vérifie s'il y a déjà une chambre possédant un numéro identique. Dans ce cas, on avertit l'utilisateur et la chambre n'est pas ajoutée. Il y a une méthode pour enlever une chambre. Dans celle-ci, on vérifie si la chambre existe avant de l'enlever. Nous avons également une méthode permettant d'afficher les informations de toutes les chambres présentent dans l'hôtel s'appelant displayAllRoom(). Il y a aussi une surcharge d'opérateur << pour afficher les informations de l'hôtel excepté les chambres.

La classe Réservation est constituée de 6 attributs privés. Il y a 2 dates qui correspondent à la date de début de réservation et à la date de fin de réservations. Il y a 2 entiers correspondant au numéro de chambre et au prix total du séjour. Et nous avons un Hotel et un Client. Nous avons un constructeur, un destructeur pour supprimer une réservation et 7 getters. Dans le constructeur, on utilise la surcharge de l'opérateur + de la classe Date pour définir la date de fin de la réservation. Pour calculer le prix total du séjour, on accède au prix de la chambre via le getter correspondant de la classe Room. Le destructeur utilise la fonction removeReservation() de la classe Room pour enlever la réservation de la chambre et la rendre de nouveau disponible.

Dans le main, nous avons créé 3 chambres simples, 5 doubles et 2 suites que nous avons insérer dans un vector chambreHotel1. Ensuite, on affiche toutes les chambres contenues dans ce vector et on créé un Hotel hotel1 en mettant en paramètre le vector des chambres. Puis, on affiche les infos de l'hotel1. Par la suite, on crée des clients qu'on met dans un vector client_list et on affiche tous les clients avec leurs informations. Pour continuer, on crée un vector de Réservation nommé

reservation_list. On demande le nom d'un client à l'utilisateur. Si le nom n'existe pas dans la client_list, on l'ajoute dedans et on affiche de nouveau les clients présents dans la liste. Ensuite, on demande à l'utilisateur de donner la date de réservation et le nombre de nuits et on vérifie si la date rentrée et le nombre de nuit sont valides avec un do while. Si la date est valide, on créée une date et si le nombre de nuit est valide on le met dans une variable. Après, on demande le type de chambre que souhaite l'utilisateur. On utilise toujours un do while pour la vérification du type rentré par l'utilisateur. Puis, on affiche les chambres de l'hôtel qui sont disponible et dont le type correspond à celui demandé et on demande à l'utilisateur de choisir la chambre qu'il souhaite réserver. Il lui suffit de donner le numéro du choix et la réservation est créée avec toutes les informations données précédemment. Enfin, on affiche le prix total du séjour qui est le prix de la chambre choisie * le nombre de nuit.

En conclusion, ce TP nous aura permis de renforcer notre connaissance sur l'utilisation des classes. De plus, la notion d'énumération a été éclaircit et comprise. Nous avons acquis davantage d'expérience concernant la surcharge d'opérateur. La connaissance sur les vector a été également améliorer. Nous avons également eu l'occasion d'utiliser un destructeur. Pour notre futur métier, je pense que nous pourrions davantage commenter le code et écrire tout en anglais pour plus de cohérence. Bien évidemment, il nous manque encore plus de pratique pour qu'on soit de plus en plus à l'aise avec toutes les fonctionnalités offertes par le C++.

TP5 :

Nous avons choisi ce TP car nous avions déjà fait le TP4 et celui-ci en découle. De plus, il permet de faire de l'héritage et d'implémenter des algorithmes de chiffrement.

Pour commencer, nous nous sommes renseignés sur le fonctionnement de la machine Enigma. Après avoir lu le sujet, le choix de faire la « vraie » machine Enigma nous est venus. Nous avons ensuite décidé de séparer la machine en 3 classes : une pour les 3 rotors (Rotor), une 2° pour le réflecteur, et une troisième pour une planche à fiche, qui sert à échanger 10 paires de lettre. Cependant, comme la 2° et la 3° sont très similaire, nous les avons regroupées dans une unique class PairSwitch.

De plus voulant laisser la trace de la demande de l'énoncé, nous avons construit 3 classes Enigma : la première avec un unique rotor issue du TP4, la deuxième étant celle demande dans l'énoncé, et la dernière étant le modèle M3 d'Enigma. Respectivement Enigma, Enigma2 et EnigmaM3

Nous avons donc un total de 6 classes.

La classe Encrypt est la classe modèle d'une machine d'encryptage, elle possède 2 attributs protected pour qu'ils puissent être accessible par les prochaines classes qui seront hérité de celle-ci. Ces attributs sont des strings. Il y a le _plain qui est la variable qui contiendra la phrase non chiffrée et le _cipher qui contiendra celle chiffrée. Il y a un getter pour chacune de ces variables. Il y a aussi deux méthodes virtuelles permettant de chiffrer et déchiffrer (encode et decode) et qui seront utilisé dans une prochaine classe. Il y a deux autres méthodes permettant de lire un fichier et d'écrire dessus.

Pour ces deux dernières méthodes, on utilise en arguments le nom du fichier et un char qui permet de spécifier si ce qu'on écrit ou lit est le plain ou le cipher.

Le fichier EncryptHelper permet de mettre les fonctions helpers qui vont nous être utile pour la suite. Il y a la fonction isMixAlphabet() qui permet de vérifier qu'il y ait bien les 26 lettres. La fonction isPairSwitch() permet de vérifier la validité d'une chaine de caractères comme étant utilisable par la classe PairSwitch, cela étant plus strict que pour les rotors.

Les fonctions isMinuscule() et isMajuscule() permettent de vérifier qu'une chaine de caractères ne contient bien que des minuscules ou des majuscules.

La classe Rotor possède 4 attributs privés dont 2 strings pour la clé et la clé inverse, 1 entier non signé qui est l'offset et un vector d'entiers non signés d'encoches. Ces encoches permettent de choisir quand le rotor fait tourner le suivant (envoie l'information qu'il faut tourner). Nous avons un constructeur par défaut et un paramétré avec comme paramètres la clé, l'offset et les encoches. Le rôle du constructeur par default est de pouvoir générer les arrays de rotors vide. Dans le constructeur paramétré, on met l'offset modulo 26 pour qu'il ait une valeur maximale à 26. On utilise la fonction isMixAlphabet(key) pour vérifier que la clé soit bien composée de caractères alphabétiques et on ajoute toutes les encoches en mettant un modulo 26 dans le vector notches pour les mêmes raisons que l'offset. Ensuite, on a la méthode rotate() qui permet d'incrémenter l'offset de 1 (ou de revenir à 0 si besoin) puis de regarder s'il y a besoin de faire tourner le rotors suivant en retournant true. On a un setter pour l'offset qui permet de mettre le rotor sur une position choisit. Enfin il y a les méthodes forward et backward qui font passer une lettre dans le rotor, soit dans le sens normal soit dans le sens inverse en utilisant les clefs.

La classe PairSwitch est très similaire à la classe Rotor, à la différence qu'elle ne différencie pas le sens normal de l'inverse. Ainsi elle ne possède pas de clef inverse et les méthodes forward et backward sont combiné dans pass_through. Cela pose cependant plus de condition sur la chaine qui sert de clef, c'est pour cela que la fonction helper isPairSwitch est utilisée pour vérifier la validité de l'objet.

Les classes Enigma et Enigma2 ont toutes deux un constructeur qui prend les paramètres pour construire les rotors, et un autre qui prend les rotors directement, elles ont aussi un setter pour redéfinir la rotation des rotors. Dans le cas d'Enigma2 les rotors sont stockés comme une array.

En ce qui concerne EnigmaM3, comme Enigma2, elles a les 3 rotors stockés dans une array mais elle a en plus un réflecteur et fiche à trou (Plugboard). Une différence est cependant présente quant à l'override d'encode et decode, dans le cas des machines précédentes ces deux fonctions avaient un comportement distinct alors que la machine Enigma M3 est symétrique, dans le sens où l'encodage et le décodage ne sont différent que de notre point de vue de lisibilité du message. Elles utilisent donc toutes deux la même méthode pass_through (méthode qui permet de « faire passer » un texte dans la machine.

De plus pour un souci de facilité d'utilisation, des headers définissent les différents rotors ou réflecteurs qui existent.

En conclusion, ce TP nous a permis d'approfondir nos connaissances et d'acquérir de l'expérience sur l'héritage ainsi que sur des fonctions de bases de chiffrements. De plus, nous avons bien remarqué que faire plusieurs classes est bénéfique et permet de simplifier le code lorsque nous utilisons des mêmes fonctionnalités dans plusieurs autres.