INF1010

*Programmation Orientée-Objet*

Travail pratique #4

Fonctions et classes géneriques, librairie standard STL et foncteurs

**Objectifs :** Permettre à l'étudiant de se familiariser avec les fonctions et les classes génériques, la librairie standard STL et foncteurs.

**Remise du travail :** Dimanche 14 juin 2020, 23h55.

**Références :** Notes de cours sur Moodle & livre Big C++ 2e éd.

**Documents à remettre :**
- Les fichiers .cpp et .h **uniquement**, complétés et réunis sous la forme d'une archive au format **.zip**. Le nom du fichier devrait être : matricule1_matricule2_X.zip avec X = L1 ou L2 selon votre groupe.

**Directives :**
- Directives de remise des Travaux pratiques sur Moodle
- Les entêtes (fichiers, fonctions) et les commentaires sont obligatoires.
- Les travaux dirigés s'effectuent obligatoirement en équipe de deux personnes faisant partie du même groupe.
- **Pas de remise possible sans être dans un groupe.**
- **Veuillez suivre le guide de codage**

**Conseils** :
- Lisez tout le document avant de commencer !
- Ayez lu vos notes de cours !
- Profitez de la présence virtuelle des chargés de lab pendant les séances de TP pour avoir des réponses à la plupart de vos questions !

## Mise en contexte

Pour faire face à la crise sanitaire actuelle, le Centre hospitalier de l'Université de Montréal a besoin de mettre en place un système de consultation en ligne afin d'évaluer l'état des patients avant leur arrivée sur place et d'appliquer davantage les mesures de la distanciation sociale et limiter les risques de contamination entre les personnes.

Pour ce faire, le CHUM fait appel aux étudiants de Polytechnique Montréal pour implémenter un système capable de répondre à leurs besoins. Ayant une grande confiance au niveau des étudiants du cours INF1010, l'école a décidé de vous confier ce projet qui s'étalerait sur 5 travaux pratiques.

Pour ce quatrième TP, on vous demande de rendre votre système plus robuste et efficace en utilisant la librairie standard STL, des fonctions génériques et des foncteurs. On vous demander de modifier certaines classes, d'implémenter des foncteurs et des méthodes ou classes génériques.

## Spécifications générales

- Toutes les classes, fonctions et entêtes de fichiers doivent être documentées.
- Ne modifiez pas la signature des méthodes sauf indication contraire.
- Toutes les méthodes doivent être définies dans le fichier d'implémentation (.cpp) **dans le même ordre** que leur déclaration dans le fichier d'en-tête (.h).
- Utilisez la liste d'initialisation pour l'implémentation de vos constructeurs.
- Tout warning à la compilation sera pénalisé. Si vous utilisez Visual Studio, vous devez activer l'option /**W4**.
- Utilisez le plus possible la liste d'initialisation des constructeurs.
- N'utilisez aucun `new` ou `delete`.
- N'utilisez aucune boucle **for**, **while** ou **do while** sauf indication contraire.
- L'utilisation la plus efficace possible de la libraire standard est exigée pour toutes les méthodes. **Assurez-vous donc de bien respecter les contraintes quant au nombre maximal de lignes de code lorsque demandé**. Notez que le nombre de lignes de code indiqué ne compte pas les lignes vides, les commentaires et les accolades seules sur leur ligne. Si un appel de fonction est trop long et que ses arguments se retrouvent sur des lignes individuelles, ceci compte toujours comme une seule ligne.
- Suivez le guide de codage sur Moodle.
- Le corps de tous les énoncés de contrôle comme **if**, **for**, **while** et **switch** doit toujours être borné par des accolades, même s'il ne fait qu'une ligne.
- **Conteneurs associatifs** : Ce TP utilise les conteneurs associatifs **std::unordered_map** plutôt que std::map. Les versions non ordonnées sont hashées et donc beaucoup plus performantes et devraient être favorisées.

- **Méthodes des conteneurs** : Une grande partie du TP consiste à trouver la façon la plus élégante de satisfaire le comportement demandé, ce qui importe d'utiliser le moins de lignes de code pour utiliser la librairie standard à son plein potentiel. En programmant, ayez toujours la documentation de cppreference sur les conteneurs et algorithmes avec vous pour trouver la meilleure façon de résoudre le problème! N'oubliez pas de regarder si certaines méthodes des conteneurs retournent des valeurs pertinentes pour simplifier encore plus votre code!
- **Débogage** : Si vos tests ne passent pas, allez jeter un coup d'œil à leur code, et pensez à utiliser votre débogueur et des points d'arrêt(breakpoints) dans les tests! Les tests sont très détaillés vous trouverez sans doute le problème en inspectant les variables.
- **Documentation :** Ayez les pages https://en.cppreference.com/w/cpp/container et https://en.cppreference.com/w/cpp/algorithm ouvertes avec vous lorsque vous travaillez, elles vous seront indispensables!.
- Ne remettez que les fichiers .h et .cpp lors de votre remise.
- Bien lire le barème de correction ci-dessous.

## Travail à **réaliser**

On vous demande de compléter les fichiers qui vous sont fournis pour pouvoir implémenter le système décrit ci-dessus.

**Notes importantes :**

- **Attention à bien lire les spécifications générales plus bas.**
- **On ne vous demande pas le contrôle des entrées (par exemple l'unicité des ID, la validité des dates …)**
- **Les fonctions doivent être implémentées dans le même ordre que la définition de la classe**
- **Utilisez la liste d'initialisation pour l'implémentation de vos constructeurs. L'ordre dans la liste doit être le même ordre que la définition des attributs de la classe.**
- **N'utilisez pas "using namespace std". Exemple d'usage accepté : std ::string.**
- **L'utilisation du mot-clef auto est permise.**

## Tests

Le fichier ***Tests.cpp*** vous est fourni et contient les tests des classes et méthodes à modifier ou à implémenter. Il y a 4 macros dans le fichier ***Tests.h***, initialement mis à la valeur de false. Aussitôt que vous avez terminé d'implémenter une classe, vous pouvez activer ses tests. Ce fichier ne devrait pas être modifié pour la remise.

Une fois tous les fichiers complétés, tous les tests devraient passer.

## Foncteurs

Les foncteurs utilisés tout au long de ce TP doivent être crées dans le fichier **foncteur.h**. Les foncteurs à créer sont les suivants :

**ComparateurSecondElementPaire :**

- Foncteur prédicat binaire comparant les seconds éléments de paires pour déterminer si elles sont en ordre.
- On veut que le foncteur fonctionne avec des paires contenant n'importe quels éléments, et c'est pourquoi il doit être **générique** (un ***template).***
- Le ***template*** prend deux paramètres, soit les types T1 et T2 tenus par une paire std::pair<T1, T2>.
- L'opérateur() doit prendre en paramètre deux **références constantes** (on ne veut pas copier ni modifier les paramètres) vers des paires dont les éléments sont de type T1 et T2. Il retourne ***true*** si le second élément de la première paire est strictement inférieur au second élément de la deuxième paire, ***false*** sinon. Référence recommandée : https://en.cppreference.com/w/cpp/utility/pair.

**ComparateurEstEgalAvecId :**

- Foncteur prédicat unaire servant à comparer un objet avec un ***id*** de type ***string***.
- On veut que le foncteur compare l'***id*** avec n'importe quel objet (Patient ou Personnel), et c'est pourquoi il doit être générique (un ***template***). La classe de l'objet comparé doit implémenter l'opérateur == pour que la comparaison fonctionne.
- Le ***template*** prend un seul paramètre de type T.
- L'opérateur() doit prendre en paramètre un pointeur ***shared_ptr*** tenant un élément de type T. Il retourne ***true*** si l'objet est égal à ***l'id*** et ***false*** si non.

**AccumulateurPeriodePersonnel :**

- Foncteur qui se charge de l'ajout de la période passée par les personnels à l'hôpital à une somme.
- L'opérateur() prend en paramètre une variable ***somme*** de type ***double***, et une paire de type ***std::pair< const std::string, std::shared_ptr<Personnel>>.*** Il retourne la somme de la variable et la ***somme*** de l'ancienneté du personnel passé en paramètre. **Indice** : utiliser les fonctions ***getDateCourante*** et

***getDateAdhesion*** pour calculer l'ancienneté en prenant en compte juste les années.

**ComparateurTypePatient :**

- Foncteur prédicat unaire servant à comparer un objet de type Patient à n'importe quel autre objet.
- On veut que le foncteur compare n'importe quel objet avec Patient, et c'est pourquoi il doit être **générique**.
- Le ***template*** de classe prend un seul paramètre de type T.
- L'opérateur() doit prendre en paramètre un pointeur ***shared_ptr*** tenant un élément de type ***Patient***. Il retourne ***true*** si l'objet est de type ***Patient*** et ***false*** si non.

**EstDansIntervalleDatesConsultation :**

- Foncteur prédicat unaire servant à déterminer si une consultation est réalisée entre deux dates.
- Constructeur par paramètres initialisant l'intervalle de dates. Il prend deux références constantes vers deux objets de type ***tm***.
- L'opérateur() prend une référence constante vers un ***shared_ptr*** de ***Consultation***. Il utilise les deux fonctions définies dans **utils.h** ***convertirStringDate*** pour convertir une date de ***string*** vers ***tm*** et ***comparerDate*** pour comparer deux dates.

## Classe Patient

Cette classe représente un patient avec son nom, sa date de naissance et son numéro d'assurance maladie.

**Les attributs suivants doivent être ajoutés :**

- `dateAdhesion_ (tm):` La date de la première consultation du patient.

**Les méthodes suivantes doivent être modifiées :**

- Le constructeur par paramètres doit être adapté aux changements effectués aux attributs de la classe.

## Classe Personnel

**Les attributs suivants doivent être ajoutés :**

- `dateAdhesion_ (tm):` La date de recrutement du personnel.

**Les méthodes suivantes doivent être modifiées :**

- Le constructeur par paramètres doit être adapté aux changements effectués aux attributs de la classe.

## Classe GestionnairePatients

Cette classe permet de gérer les patients.

**Les méthodes suivantes doivent être implémentées ou modifiées :**

- La méthode ***chercherPatient*** permet de chercher un patient avec son numéro d'assurance maladie. Elle utilise la fonction ***find_if*** et le foncteur ***ComparateurEstEgalAvecId.*** **Indice** : Conservez l'itérateur retourné par la fonction ***find_if*** dans une variable locale pour le tester et pour retourner le pointeur vers le patient facilement si jamais l'itérateur indique qu'il a été trouvé.
- L'opérateur += doit être remplacé par la méthode ***générique ajouterPatient*** qui permet d'ajouter un patient au vecteur ***patients_*** s'il n'y existe pas déjà. Elle utilise la méthode ***chercherPatient***. Elle prend comme paramètre une référence vers le patient à ajouter.
- La méthode ***supprimerPatient*** qui permet de supprimer un patient. Elle utilise les fonctions ***erase***, ***remove_if*** et le foncteur ***ComparateurEstEgalAvecId.*** Elle prend comme paramètre l'identificateur de patient à supprimer.
- L'opérateur << utilise une boucle **range-based** plutôt que d'utiliser une boucle à itérateurs et une boucle à index pour afficher les informations des patients. ***ComparateurTypePatient***
- La méthode ***getNbPatientsEtudiants*** une méthode constante qui permet de retourner le nombre de patients étudiants dans le vecteur ***patients_***.
- La méthode ***LireLignePatient*** doit être modifiée pour permettre de lire les informations d'un patient et d'un patient étudiant puis utiliser ***ajouterPatient*** pour ajouter le patient au vecteur ***patients_*** selon son type. Elle utilise la méthode ***convertirStringDate*** pour convertir la date du ***string*** au format ***tm.***

## Classe GestionnairePersonnels

Cette classe permet la gestion des membres du personnel de l'hôpital.

**Cette classe contient les attributs privés suivants :**

- `personnels_ std::unordered_map<std::string, std::shared_ptr<Personnel>>` la liste du personnel de l'hôpital

**Les méthodes suivantes doivent être supprimées :**

- La méthode ***getMedecins*** qui retourne un vecteur de ***Medecin**** qui contient les personnels qui sont des médecins.
- La méthode ***getMedecinsResidents*** qui retourne un vecteur de ***MedecinResident**** qui contient les personnels qui sont des médecins résidents.

**Les méthodes suivantes doivent être implémentées ou modifiées :**

- La méthode ***chercherPersonnel*** qui prend ***Id*** du personnel et utilise la fonction ***find*** pour trouver le personnel correspond. **Indice** : Conservez l'itérateur retourné par la fonction ***find*** de la ***map*** dans une variable locale pour le tester et pour retourner le pointeur vers le personnel facilement si jamais l'itérateur indique qu'il a été trouvé.
- L'opérateur += doit être remplacer par la méthode **générique *ajouterPersonnel*** qui permet d'ajouter un personnel au ***personnels_*** s'il n'y existe pas déjà. Elle utilise la méthode ***chercherPersonnel***. Elle prend comme paramètre une référence vers le personnel à ajouter.
- L'opérateur -= à remplacer par la méthode générique ***supprimerPersonnel*** qui permet de changer l'attribut ***estActif*** à ***false*** pour un personnel existant. Elle utilise la méthode ***chercherPersonnel***. Elle prend comme paramètre l'identificateur du personnel à supprimer.
- L'opérateur << utilise une boucle **range-based** plutôt que d'utiliser une boucle à itérateurs et une boucle à index pour afficher les informations des personnels.
- La méthode ***getPersonnels*** à modifier pour qu'elle s'adapte au type de conteneur ***std::unordered_map***
- La méthode **générique *getPersonnelsAvecType*** qui retourne une ***std::unordered_map***. Chaque élément du conteneur une paire dont le premier élément est ***l'id*** du personnel et le second est un pointeur vers le personnel.
- La méthode ***getPersonnelsTriesSuivantSalaireAnnuel*** trie le personnel de l'hôpital suivant le salaire annuel. On doit tout d'abord copier les éléments de la map ***personnels_*** dans un vecteur de ***std::pair<std::string, std::shared_ptr<Personnel>>*** . On utilise un algorithme de tri pour trier les éléments du vecteur. Cette méthode utilise le foncteur ***ComparateurSecondElementPaire.***
- La méthode ***getNbMedecins*** à adapter au changement effectué sur la classe ***GestionnairePersonnels***.
- La méthode ***getNbMedecinsResidents*** à adapter au changement effectué sur la classe ***GestionnairePersonnels***.
- La méthode ***LireLignePatient*** qui permet de lire les informations d'un personnel qu'il soit médecin ou médecin résident. Elle utilise ***ajouterPesonnel*** pour ajouter le personnel à ***personnels_*** selon son type. Elle doit être modifiée pour s'adapter aux changements apportés à la classe ***GestionnairePersonnels***. Elle utilise la méthode ***convertirStringDate*** pour convertir la date du ***string*** au format ***tm***

## Classe Hopital

Cette classe permet la gestion de l'hôpital.

**Les méthodes suivantes doivent être implémentées ou modifiées :**

- La méthode ***chargerBaseDeDonnees*** doit être modifiée pour qu'elle prenne un troisième paramètre qui est le nom du fichier qui contient les consultations. Elle utilise la méthode ***chargerDepuisFichierConsultation*** pour charger les consultations depuis le fichier.
- La méthode ***getAncienneteMoyenneDesPersonnels*** qui retourne la moyenne de l'ancienneté du personnel de l'hôpital. Elle utilise la fonction ***accumulate*** et le foncteur ***AccumulateurPeriodePersonnel***
- La méthode ***getConsultationsEntreDate*** retourne les consultations qui ont été réalisées entre deux dates. Elle prend en paramètre deux références constantes vers les dates. Ces dates sont de type ***tm***. Elle utilise la fonction ***copy_if, back_inserter*** et le foncteur ***EstDansIntervalleDatesConsultation.***
- Elle retourne un vecteur de ***shared_ptr*** vers ***Consultation***
- L'opérateur += qui ajoute un personnel au ***gestionnairePersonnel_*** doit être remplacé par la méthode **générique *ajouterPersonnel***. Elle prend comme paramètre une référence vers le personnel à ajouter.
- L'opérateur += qui ajoute un patient au ***gestionnairePatient_*** doit être remplacé par la méthode **générique *ajouterPatient***. Elle prend comme paramètre une référence vers le patient à ajouter.
- L'opérateur += qui ajoute un une consultation au vecteur ***consultations_*** doit être remplacé par la méthode **générique *ajouterConsultation***. Elle prend comme paramètre une référence vers la consultation à ajouter.

**Main.cpp**

Le fichier ***main.cpp*** vous est fourni et fait appel aux tests implémenter dans le fichier ***Tests.cpp*** et ne devrait pas être modifié pour la remise.

**Fichiers complémentaires**

En plus des fichiers ci-dessus, l'énoncé contient le fichier debogageMemoire.hpp. Ce dernier est présent pour vous aider à vérifier s'il y a des fuites de mémoire dans votre code. Exécutez la solution en débogage pour qu'il fonctionne. Si une fuite de mémoire est détectée, un message sera affiché dans la fenêtre Sortie (Output) de Visual studio.

## Exemple d’Affichage

```
Tests pour GestionnairePatients:
--------
Test  1: GestionnairePatients::chercherPatient            : OK
Test  2: GestionnairePatients::ajouterPatient             : OK
Test  3: GestionnairePatients::supprimerPatient           : OK
Test  4: GestionnairePatients::getPatientsEtudiants       : OK
Test  5: GestionnairePatients::operator<<                 : OK
--------
Total pour la section: 2.5/2.5

Tests pour GestionnairePersonnels:
--------
Test  1: GestionnairePersonnels::chercherPersonnel        : OK
Test  2: GestionnairePersonnels::ajouterPersonnel         : OK
Test  3: GestionnairePersonnels::getPersonnelsAvecType    : OK
Test  4: GestionnairePersonnels::getNbMedecins            : OK
Test  5: GestionnairePersonnels::getNbMedecinsResidents   : OK
Test  6: GestionnairePersonnels::getPersonnelsTriesSuivantSalaireAnnuel: OK
Test  7: GestionnairePersonnels::operator<<               : OK
--------
Total pour la section: 2.5/2.5

Tests pour Hopital:
--------
Test  1: Hopital::chargerBaseDeDonnees                    : OK
Test  2: Hopital::ajouterPersonnel                        : OK
Test  3: Hopital::ajouterPatient                          : OK
Test  4: Hopital::ajouterConsultation                     : OK
Test  5: Hopital::getAncienneteMoyenneDesPersonnels       : OK
Test  6: Hopital::getCosultationsEntreDates               : OK
--------
Total pour la section: 3/3

Total pour tous les tests: 8/8

---------------
Question Bonus:
---------------

Expliquez pourquoi il y a une différence remarquable entre la recherche
de Patient dans GestionnairePatients et la recherche de Personnel dans GestionnairePersonnels
pourtant les deux ont le même npmbres d'élements = 50 elements
Remarquez bien que dans le test ci-dessous, on est entrain de faire le pire cas:
la recherche d'un element inexistant donc un parcours de tout le conteneur.
Repondez à cette question dans un fichier txt nommé bonus.txt que vous remetteriez avec vos autres fichiers

La recherche d'un personnel inexistant a durée : 0.0045 ms
La recherche d'un patient inexistant a durée : 0.0216 ms
```

**Correction**

La correction du TP4 se fera sur 20 points.

Voici les détails de la correction :

- (2 points) Compilation du programme.
- (2 points) Exécution du programme.
- (8 points) Comportement exact des méthodes du programme.
- (1 points) Documentation du code et bonne norme de codage.
- (2 points) Utilisation des fonctions et méthode génériques.
- (3 points) Implémentation et utilisation des foncteurs.
- (2 points) Utilisation des fonctions STL.
- (+ 1 point) Question bonus.