INF1010

Programmation Orientée-Objet

Travail pratique #3

Héritage, Polymorphisme, Héritage multiple

Objectifs : Permettre à l'étudiant de se familiariser avec l'héritage, le polymorphisme, de l'héritage multiple, de conversion dynamique de type et de fonctions virtuelles

Remise du travail : Dimanche 07 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 !

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 troisième TP, on vous demande de généraliser le système de consultation en ligne en introduisant des classes abstraites. On vous demande de créer des nouvelles classes et de modifier le code des anciennes classes pour intégrer les notions d'héritage et de polymorphisme vues en cours

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.**

Classe Patient

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

La méthode suivante doit être retirée :

- L'opérateur << doit être retiré. Il sera remplacé par la méthode **virtuelle *afficher***

Les méthodes suivantes doivent être implémentées :

- La méthode ***virtuelle afficher(***`std::ostream&`***)*** doit être implémentée pour remplacer l'opérateur <<.

Classe Etudiant

Cette classe est une classe ***abstraite*** qui représente un étudiant.

Cette classe contient les attributs privés suivants :

- `nom_` (*`std::string`*) : le nom complet de l'étudiant
- `matricule _` (*`std::string`*) : le matricule de l'étudiant
- `dateDeNaissance_` (*`std::string`*) : la date de naissance de l'étudiant

Cette classe contient les attributs protégés suivants :

- `etablissement_` (*`std::string`*) : l'établissement de l'étudiant

Les méthodes suivantes doivent être implémentées :

- Un constructeur par paramètres qui initialise les attributs aux valeurs correspondantes.
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire.**
- La méthode ***`virtuelle pure afficher(`***`std::ostream&`***`)`*** : affiche le matricule et l'établissement de l'étudiant.

Classe PatientEtudiant

Cette classe est une classe qui représente un patient qui est étudiant. Elle **hérite** de la classe ***Patient*** et la classe ***Etudiant***

Cette classe contient les attributs privés suivants :

- `tauxRabais_` (*`double`*) : le taux de rabais de la consultation appliqué pour un patient étudiant. Il doit être initialisé au **TAUX_RABAIS.**

Les méthodes suivantes doivent être implémentées :

- Un constructeur par paramètres qui initialise les attributs aux valeurs correspondantes. Ce constructeur utilise les constructeurs des classes ***Patient*** et ***Etudiant***.
- La méthode ***`afficher(`*** `std::ostream&` ***`)`*** à surcharger pour afficher les informations du patient étudiant. Elle utilise les méthodes ***afficher*** des classes ***Patient*** et ***Etudiant***.
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire**.

Classe Personnel

Cette classe est une classe **abstraite** qui représente un personnel de l'hôpital.

Cette classe contient les attributs suivants :

- nom_ (*`std::string`*) : le nom complet du personnel
- id_ (*`std::string`*) : l'identifiant du personnel
- estActif_ (*`std::string`*) : l'état du personnel dans la clinique. Cet attribut sert à distinguer un personnel actif présentement actif dans l'hôpital, d'un personnel qui ne l'est pas. Il est initialisé à ***true***.

Les méthodes suivantes doivent être implémentées :

- La méthode ***`virtuelle afficher(`*** `std::ostream&)` pour afficher les informations du personnel.
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire**.

Les méthodes à ne pas implémenter

- La méthode ***`virtuelle pure getSalaireAnnuelle`*** qui est une méthode constante qui retourne un double.

Classe Medecin

Cette classe représente un médecin qui exerce à l'hôpital. Elle **hérite** de la classe ***Personnel***

Cette classe contient les attributs suivants :

- `specialite_` (*`Medecin::Specialite`*) : la spécialité du médecin.
- `patientsAssocies_` (*`std::vector<Patient*>`*) : la liste des patients associés au médecin.
- `nombreConsultations_` (*`size_t`*) : Le nombre de consultation effectuées par un médecin.

Les méthodes suivantes doivent être implémentées ou modifiées :

- Un constructeur par paramètres qui initialise les attributs aux valeurs correspondantes. Ce constructeur utilise le constructeur de la classe ***Personnel***
- La méthode **virtuelle** ***`afficher(`***`std::ostream&)` pour afficher les informations du médecin. Elle utilise la méthode ***afficher*** de la classe ***Personnel***.
- La méthode ***incrementNombreConsultation*** qui permet d'incrémenter le nombre de consultations effectuées par un médecin.
- La méthode ***virtuelle getSalaireAnnuel*** qui permet de calculer le salaire annuel du médecin.
- La méthode ***getPrixConsultation*** une méthode constante qui retourne le prix de la consultation suivant la spécialité du médecin. Pour les détails des prix de consultation, consultez la description de la méthode dans le fichier Medecin.cpp
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire.**

Classe MedecinResident

Cette classe représente un médecin résident qui exerce à l'hôpital. Elle **hérite** de la classe ***Medecin*** et de la classe ***Etudiant.***

Les méthodes suivantes doivent être implémentées ou modifiées :

- Un constructeur par paramètres qui initialise les attributs aux valeurs correspondantes. Ce constructeur utilise le constructeur de la classe ***Medecin*** et la classe ***Etudiant.***
- La méthode ***`afficher(`***`std::ostream&`) à surcharger pour afficher les informations du médecin résidant. Elle utilise la méthode ***afficher*** de la classe ***Medecin*** et la classe ***Etudiant.***
- La méthode ***getSalaireAnnuel*** à surcharger et qui permet de calculer le salaire annuel du médecin résident.

Classe GestionnairePatients

Cette classe permet de gérer les patients.

Les méthodes suivantes doivent être implémentées ou modifiées :

- L'opérateur += doit être modifié pour s'adapter aux types de patients à afficher.
- L'opérateur << doit être modifié pour afficher les informations des patients.
- La méthode ***getPatientsEtudiants*** permet de retourner un vecteur de ***PatienEtudiant**** qui contient les patients étudiants dans la liste ***patients***_.
- La méthode ***getNbPatientsEtudiants*** une méthode constante qui permet de retourner le nombre de patients étudiants dans le vecteur ***patients***_.
- La méthode ***LireLignePatient*** doit être modifier pour permet de lire les informations d'un patient normale et d'un patient étudiant puis utiliser += pour ajouter le patient au vecteur ***patients***_ selon son type.

Classe GestionnairePersonnels

Cette classe permet la gestion des personnels de l'hôpital. C'est une modification de la classe ***GestionnareMedecins***.

Cette classe contient les attributs privés suivants :

- `personnels_` (*`vector<std::shared_ptr<Personnel>>`*) : la liste du personnel de l'hôpital

Les méthodes suivantes doivent être implémentées ou modifiées :

- L'opérateur += qui permet d'ajouter un personnel dans le vecteur ***personnel***_ s'il n'existe pas déjà. Elle utilise la méthode ***chercherPersonnel***.
- L'opérateur << pour afficher les informations des personnels.
- 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.
- La méthode ***getNbPersonnels*** qui retourne le nombre de tout le personnel de l'hôpital.
- La méthode ***getNbMedecins*** qui retourne le nombre médecins parmi les personnels.
- La méthode ***getNbMedecinsResidents*** qui retourne le nombre médecins résidents parmi les personnels.
- La méthode ***LireLignePatient*** qui permet de lire les informations d'un personnel qu'il soit médecin ou médecin résident. Elle utilise += pour ajouter le personnel au vecteur ***personnels***_ selon son type.

Classe Consultation

Cette classe est une classe ***abstraite*** permet de gérer les consultations de l'hôpital.

Cette classe contient les attributs suivants :

- `medecin_` (*`Medecin*`*) : le médecin responsable de la consultation.
- `patient_` (*`Patient*`*) : le patient qui demande la consultation.
- `date_` (*`std::string`*) : la date de la consultation.
- `prix_` (*`double`*) : le prix de la consultation à initialiser à **PRIX_DE_BASE.**

Les méthodes suivantes doivent être modifiés ou implémentées :

- L'opérateur << doit être remplacé par la méthode ***afficher(***`std::ostream&`***)*** pour afficher les informations d'une consultation.

Les méthodes à ne pas implémenter

- La méthode ***virtuelle pure calculerPrix***.

Classe ConsultationEnligne

Cette classe est une classe qui ***hérite*** de la classe consultation.

Cette classe contient les attributs suivants :

- `diagnostique_` (*`std::string`*) : le diagnostique d'une consultation en ligne.
- `rabaisConsultationEnLigne_` (*`double`*) : le rabais appliqué à une consultation en ligne.

Les méthodes suivantes doivent être modifiés ou implémentées :

- Le constructeur par paramètres pour initialiser les attributs de la classe. Ce constructeur utilise le constructeur de la classe ***Consultation***.
- La méthode ***calculerPrix*** qui calculer le prix de la consultation en ligne. Cette méthode utiliser la méthode ***getPrixConsultation*** de ***Medecin***. Si le patient est un patient étudiant, on lui offre un rabais. On applique aussi un **rabais** puisqu'il s'agit d'une **consultation en ligne.**
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire.**

Classe ConsultationPhysique

Cette classe est une classe qui ***hérite*** de la classe consultation.

Cette classe contient les attributs suivants :

- `prescription_` (*`std::string`*) : la prescription d'une consultation physique.

Les méthodes suivantes doivent être modifiés ou implémentées :

- Le constructeur par paramètres pour initialiser les attributs de la classe. Ce constructeur utilise le constructeur de la classe ***Consultation***.
- La méthode ***calculerPrix*** qui calculer le prix de la consultation en ligne. Cette méthode utiliser la méthode ***getPrixConsultation*** de ***Medecin***. Si le patient est un patient étudiant, on lui offre un rabais.
- Les méthodes d'accès aux attributs (getters) **si nécessaire**.
- Les méthodes de modification des attributs (setters) **si nécessaire**.

Classe Hopital

Cette classe permet la gestion de l'hôpital.

Les attributs suivants devraient être modifiés :

- `gestionnaireMedecins_` (*`gestionnaireMdedecins`*) doit être modifiés en gestionnaire personnels.

Les méthodes suivantes doivent être implémentées :

- L'opérateur += qui ajoute une consultation au vecteur ***consultations_***. Il prend en paramètre une référence vers une consultation. Cette méthode utilise la méthode ***chercherPersonnel.*** La consultation est ajoutée si le médecin est actif et il existe dans le gestionnaire des médecins et si le patient dans le gestionnaire des patients. La méthode retourne ***true*** si l'opération de l'ajout est réussie.

Main.cpp

Le fichier main.cpp vous est fourni et ne devrait pas avoir à être modifié pour la remise. N'hésitez pas à commenter certaines parties si vous avez besoin de compiler votre code. Vous pouvez aussi utiliser ce fichier pour mieux comprendre les requis des fonctions.

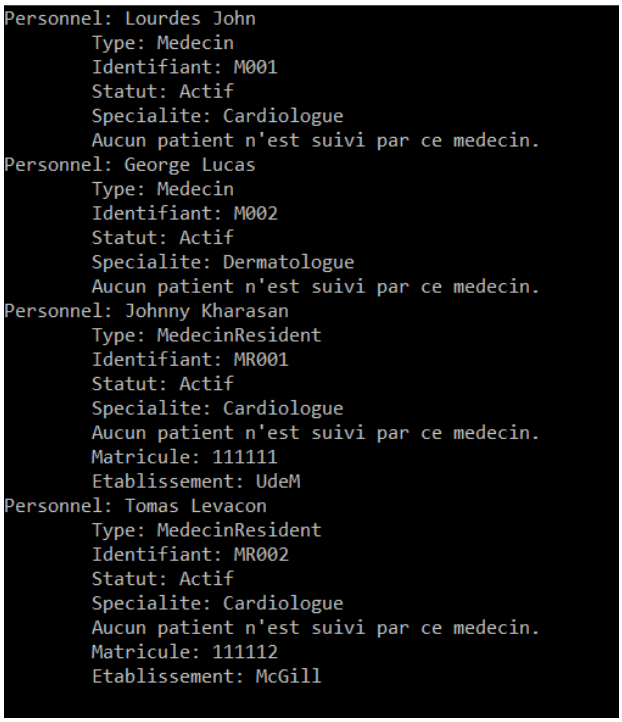
Une fois tous les fichiers complétés, tous les tests devraient passer.

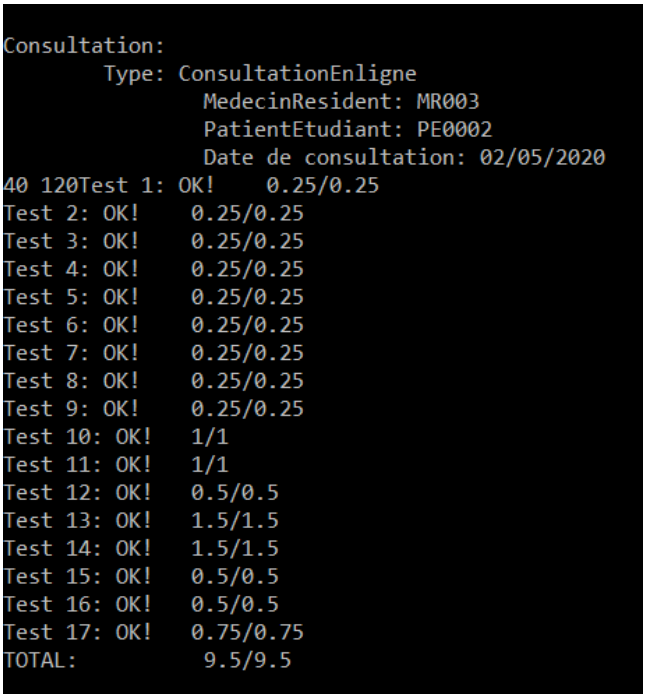
Fichiers complémentaires

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

```
Patient:
        Type: Patient
        Nom: Simon | Date de naissance: 25/10/92 | Numero d'assurance maladie: P003
Patient:
        Type: PatientEtudiant
        Nom: Timon | Date de naissance: 05/06/1998 | Numero d'assurance maladie: PE0002 | Matricule: 100002
        Etablissement: HEC Montreal | Taux de rabais: 0.2
Patient:
        Type: PatientEtudiant
        Nom: Timon | Date de naissance: 05/06/1998 | Numero d'assurance maladie: PE0002 | Matricule: 100002
        Etablissement: HEC Montreal | Taux de rabais: 0.2
Personnel: Tamard
        Type: Medecin
        Identifiant: M003
        Statut: Actif
        Specialite: Cardiologue
        Aucun patient n'est suivi par ce medecin.
Personnel: Tamard
        Type: Medecin
        Identifiant: M003
        Statut: Actif
        Specialite: Cardiologue
        Aucun patient n'est suivi par ce medecin.
Personnel: Tira
        Type: MedecinResident
        Identifiant: MR003
        Statut: Actif
        Specialite: Gynecologue
        Aucun patient n'est suivi par ce medecin.
        Matricule: 111113
        Etablissement: University Of Toronto
Personnel: Tira
        Type: MedecinResident
        Identifiant: MR003
        Statut: Actif
        Specialite: Gynecologue
        Aucun patient n'est suivi par ce medecin.
        Matricule: 111113
        Etablissement: University Of Toronto
Patient:
        Type: Patient
        Nom: John Lourdes | Date de naissance: 12/12/2001 | Numero d'assurance maladie: P001
Patient:
        Type: Patient
        Nom: George Lucas | Date de naissance: 01/04/1984 | Numero d'assurance maladie: P002
Patient:
        Type: PatientEtudiant
        Nom: Christopher Lavoie | Date de naissance: 01/04/1999 | Numero d'assurance maladie: PE001 | Matricule: 100001
        Etablissement: Polytechnique Montreal | Taux de rabais: 0.2
```





Spécifications générales

- Ajoutez un destructeur pour chaque classe lorsque nécessaire.
- Utilisez la liste d'initialisation pour l'implémentation de vos constructeurs.
- Documentez votre code source.
- Ne remettez que les fichiers .h et .cpp lors de votre remise.
- Bien lire le barème de correction ci-dessous.
- Ne modifier pas la signature des méthodes sauf c'est indiqué.

Correction

La correction du TP3 se fera sur 20 points.

Voici les détails de la correction :

- (2 points) Compilation du programme

- (2 points) Exécution du programme

- (9.5 points) Comportement exact des méthodes du programme

- (2 points) Documentation du code et bonne norme de codage

- (2.5 points) utilisation correcte de la conversion dynamique (dynamic_cast)

- (2 points) utilisation correcte de méthode virtuelle.