Correction TP de programmation n°6

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Les classes

Dans cette séance de travaux pratiques, nous allons apprendre à structurer un code en utilisant les `classes`. Il est indispensable de respecter la structure et les conventions vues dans le cours Objets et classes.

▸ Exercice 1. (État civil)

Dans cet exercice nous allons reprendre le code de l'exercice de l'état civil de la semaine 5 et le restructurer en utilisant une classe `EtatCivil`. Le fichier à compléter est le fichier `ClasseEtatCivil.cpp` fourni. Pour gagner du temps et vous concentrer sur les notions de classes et méthodes, vous pouvez consulter votre fichier de la semaine dernière ou le fichier `EtatCivil_correction.cpp` fourni (ne pas modifier ce fichier). Nous gardons la structure Personne et l'énumeration Genre comme la semaine dernière et nous allons modifier uniquement la structure `EtatCivil`.

1. Déclarer dans la classe `EtatCivil` la méthode `void initialise(string titreEtatCivil);` qui initialise un état civil, puis la définir hors de la classe en utilisant la syntaxe

```
void EtatCivil::initialise(string titreEtatCivil) {
  code de la fonction....
}
```

✂ ----------

```
119 void EtatCivil::initialise(string titreEtatCivil) {
120   //
121   titre = titreEtatCivil;
122   table.clear();  // autre possibilité: table = vector<Personne>(0);
123   //
124 }
```

---------- ✂

2. Consulter le `main`, mettre en commentaire les 3 appels à des méthodes non encore écrites (il faudra penser à les décommenter plus tard), exécuter et vérifier que l'affichage obtenu est bien le bon.
3. Déclarer dans la classe EtatCivil la méthode `int cherche(string nom)` qui recherche le nom d'une personne et retourne l'indice de la personne dans l'état civil si elle la trouve ou -1 sinon. Puis définir cette méthode. N'oubliez pas d'ajouter le mot clé `const` après la liste des paramètres pour indiquer que la méthode ne modifie pas l'objet.

✂ ----------

```
155 int EtatCivil::cherche(string nom) const {
156   //
157   for (size_t i = 0; i < table.size(); i++) {
158     if (nom == table[i].nom)
```

```
159         return i;
160     }
161     return -1;
162     //
163 }
```

---

4. Tester le bon fonctionnement de la méthode `cherche`. Utiliser la fonction `creeEtatCivildeTest` pour génerer un exemple d'un état civil pour les tests.

```
167 TEST_CASE("cherche une personne") {
168     EtatCivil a;
169     a = creeEtatCivildeTest();
170     CHECK(a.cherche("Noemie") == 2);
171     // Ajouter d'autre exemples ici
172     //
173     CHECK(a.cherche("Armand") == 6);
174     CHECK(a.cherche("Antoine") == -1);
175     //
176 }
```

---

5. Déclarer et définir les méthodes `imprimePersonne` et `imprimeEtatCivil`. La méthode `imprimePersonne` affiche dans le terminal une personne dont le nom est passé en paramètre. La méthode `imprimeEtatCivil` affiche dans le terminal un état civil (son nom et sa table). Notez que les deux fonctions membres auront le mot clé `const` dans leurs déclarations puisqu'elles ne modifient pas l'objet EtatCivil.

   ```
   void imprimePersonne(string nom) const;
   void imprimeEtatCivil() const;
   ```

   Facultatif : Si c'est plus simple pour vous, on pourra utiliser une fonction auxilliaire `imprimeIndPersonne` qui prend l'indice d'une personne dans l'état civil et affiche les informations de cette personne.

   ```
   void imprimeIndPersonne(int ind) const;
   ```

   Sinon, commenter l'appel de `imprimeIndPersonne` dans le main.

```
180 void EtatCivil::imprimePersonne(string nom) const {
181     //
182     int ind = cherche(nom);
183     if (ind < 0) {
184       throw invalid_argument("Nom incorrect dans imprimePersonne: ");
185     } else {
186       imprimeIndPersonne(ind);
187     }
188     //
189 }
193 void EtatCivil::imprimeEtatCivil() const {
194     //
195     if (table.size() == 0) {
```

```
196       cout << "L'état civil est vide" << endl;
197     } else {
198       for (size_t i = 0; i < table.size(); i += 1) {
199         imprimeIndPersonne(i);
200       }
201     }
202     //
203   }
```

---

6. Déclarer, définir et tester la méthode `int personne(string sonNom, Genre s)` qui ajoute une nouvelle personne à l'état civil. Cette méthode modifie l'objet EtatCivil donc elle ne prend pas la mention `const`. La méthode lève une exception au cas où le nom serait vide, ou la personne existe déjà dans l'état civil. Sinon, elle renvoie l'indice de la personne dans l'état civil.
   Pour tester la méthode, on pourra utiliser la fonction de `doctest`

   ```
   CHECK_THROWS_WITH_AS(codeATester, "message de l'exception", type_exception);
   ```

   Cette fonction vérifie à la fois que l'exception levé dans le méthode est la même que celle indiqué à la place de `exception` mais également le message de l'exception.

```
207   int EtatCivil::personne(string sonNom, Genre s) {
208     //
209     if (sonNom.size() == 0) {
210       throw invalid_argument("Le nom de la personne est vide !");
211     }; /* pas de nom vide */
212     if (cherche(sonNom) != -1) {
213       throw invalid_argument("La personne existe déjà dans l'etat civil !");
214     }
215     vector<int> enfants;  // vector vide.
216     Personne p = {sonNom, s, -1, -1, -1, enfants};
217     table.push_back(p);
218     return table.size() - 1;
219     //
220   }
221
222   TEST_CASE("Ajout d'une personne") {
223     //
224     EtatCivil a;
225     a = creeEtatCivildeTest();
226     CHECK(a.personne("Lamia", Genre::Fem) == 8);
227     CHECK_THROWS_WITH_AS(a.personne("Lamia", Genre::Fem), "La personne existe déjà dans l'etat civil !", invalid_argumen
228     CHECK(a.personne("Antoine", Genre::Masc) == 9);
229     CHECK_THROWS_WITH(a.personne("Antoine", Genre::Masc), "La personne existe déjà dans l'etat civil !");
230     CHECK_THROWS_WITH_AS(a.personne("Noemie", Genre::Masc), "La personne existe déjà dans l'etat civil !", invalid_argum
231     CHECK_THROWS_WITH(a.personne("", Genre::Masc), "Le nom de la personne est vide !");
232     CHECK_THROWS_WITH_AS(a.personne("", Genre::Fem), "Le nom de la personne est vide !", invalid_argument);
233     //
234   }
```

▸ **Exercice 2. (Autres méthodes)** Cet exercice est à faire dans le même fichier que le précédent. Il n'est pas indispensable, donc si vous êtes plutôt lents, passez directement à l'exercice suivant (qui lui est indispensable et devra être terminé d'ici le prochain TP).

Rédiger la déclaration et la définition des méthodes `mariage`, `naissance`, `ascendantI` et `ascendantR` vues dans le TP de la semaine dernière et faire les modifications nécessaires pour les tests.

7. mariage : méthode qui enregistre le mariage de deux personnes dont on passe les noms en paramètre. La fonction renvoie `true` si le mariage est possible et `false` sinon. On impose que les deux personnes soient enregistrées et ne soient pas déjà mariées.

   ```
   bool EtatCivil::mariage(string lun, string lautre);
   ```

8. naissance : méthode qui enregistre la naissance d'une personne. Son en-tête est :

   ```
   bool EtatCivil::naissance(string qui, Genre s, string p1, string p2);
   ```

   Les paramètres sont le nom de l'enfant, son genre, les noms des parents ; les parents doivent être enregistrés et être conjoints. Si les conditions ne sont pas remplies l'enfant n'est pas enregistré. La fonction renvoie true ou false selon que la filiation a pu être enregistrée ou non.

9. ascendantI : méthode (version itérative) qui retourne si une personne est un ancêtre au sens large d'une personne. On pourra se servir d'un vecteur auxiliaire (géré en pile) dans lequel on stockera les indices des personnes dont on doit vérifier si elles sont ou non des ancêtres de la personne concernée, en commençant par ses parents. On rappelle que la méthode `pop_back()` permet de supprimer le dernier élément d'un vecteur. En-tête :

   ```
   bool EtatCivil::ascendantI(string qui, string ancetre);
   ```

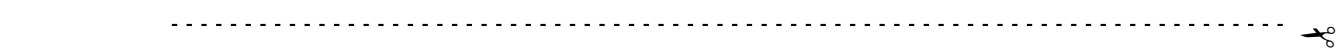
10. ascendantR : version récursive de ascendantI

```
239 bool EtatCivil::mariage(string lun, string lautre) {
240   //
241   int ilun = cherche(lun), ilautre = cherche(lautre);
242   if (ilun < 0 or ilautre < 0 or ilun == ilautre) {
243     return false;
244   }
245   if (table[ilun].indConjoint != -1 or table[ilautre].indConjoint != -1) {
246     return false;
247   }
248   table[ilun].indConjoint = ilautre;
249   table[ilautre].indConjoint = ilun;
250   return true;
251   //
252 }
256 TEST_CASE("Mariage de deux personnes") {
257   //
258   EtatCivil a;
259   a = creeEtatCivildeTest();
260   a.personne("Lamia", Genre::Fem);
261   CHECK(a.mariage("Lamia", "Yuri"));
262   CHECK_FALSE(a.mariage("Guillaume", "Yuri"));
263   //
264 }
268 bool EtatCivil::naissance(string qui, Genre s, string p1, string p2) {
269   //
```

```
270   int iqui, ip1, ip2;
271
272   ip1 = cherche(p1);
273   ip2 = cherche(p2);
274   // ATTENTION: ici ne pas utiliser la fonction mariage de la question
275   // precedente car celle-ci ne teste pas si deux personnes sont mariées
276   // mais elle les marient s'ils étaient célibataires ! On doit donc
277   // refaire les tests, mais ici plus simples.
278   if (ip1 == -1 or ip2 == -1 or
279       table[ip1].indConjoint != ip2
280       // ce dernier test est inutile si la fonction mariage est correcte:
281       // si l'un est conjoint de l'autre, la reciproque doit etre vraie aussi
282       // !
283       or table[ip2].indConjoint != ip1) {
284     return false;
285   }
286
287   // essayer d'ajouter le nouveau-né
288   iqui = personne(qui, s);
289   if (iqui < 0) {
290     return false;
291   }
292
293   // Enregistrer la filiation dans les deux sens: de l'enfant vers les
294   // parents et l'inverse.
295   table[iqui].indParent1 = ip1;
296   table[ip1].enfants.push_back(iqui);
297
298   table[iqui].indParent2 = ip2;
299   table[ip2].enfants.push_back(iqui);
300
301   return true;
302   //
303 }
307 TEST_CASE("Naissance d'un bébé") {
308   //
309   EtatCivil a;
310   a = creeEtatCivildeTest();
311   a.personne("Lamia", Genre::Fem);
312   a.mariage("Lamia", "Yuri");
313   CHECK(a.naissance("Christophe", Genre::Masc, "Lamia", "Yuri"));
314   CHECK_FALSE(a.naissance("Dina", Genre::Masc, "Noemie", "Yuri"));
315   //
316 }
320 bool EtatCivil::ascendantI(string qui, string ancetre) const {
321   //
322   vector<int> pile;
323   int iqui = cherche(qui), iancetre = cherche(ancetre);
324   if (iqui < 0 or iancetre < 0) {
325     return false;
326   }
327   pile.push_back(iqui);  // On empile l'indice de départ
328   // pile des indices des personnes à tester: on remonte dans l'arbre
329   // généalogique.
330   while (pile.size() > 0) {
331     int courant = pile[pile.size() - 1];
332     pile.pop_back();
333     if (courant == iancetre) {
334       return true;
335     } else {
```

```
336       // ajouter ses parents, si connus, dans la pile des personnes a
337       // tester
338       int i1 = table[courant].indParent1;
339       int i2 = table[courant].indParent2;
340       if (i1 != -1) {
341         pile.push_back(i1);
342       }
343       if (i2 != -1) {
344         pile.push_back(i2);
345       }
346     }
347   }
348   /* fin de la boucle: on a visite tous les ancetres sans trouver indParent */
349   return false;
350   //
351 }
352
353 TEST_CASE("ascendant itératif") {
354   //
355   EtatCivil a;
356   a = creeEtatCivildeTest();
357   a.personne("Lamia", Genre::Fem);
358   a.mariage("Lamia", "Yuri");
359   a.naissance("Christophe", Genre::Masc, "Lamia", "Yuri");
360   CHECK(a.ascendantI("Christophe", "Remy"));
361   CHECK_FALSE(a.ascendantI("Christophe", "Noemie"));
362   //
363 }
367 bool EtatCivil::ascendantAux(int iqui, int iancetre) const {
368   //
369   int i1, i2;
370   if (iqui == iancetre) {
371     return true;
372   }
373   // On remonte dans l'arbre à la recherche de l'ancêtre. Plus efficace
374   // que de descendre de l'ancêtre vers les enfants qui explorerait aussi
375   // les cousins (éloignés)
376   i1 = table[iqui].indParent1;
377   i2 = table[iqui].indParent2;
378   return (i1 != -1 and ascendantAux(i1, iancetre)) or
379          (i2 != -1 and ascendantAux(i2, iancetre));
380   //
381 }
382
383 bool EtatCivil::ascendantR(string qui, string ancetre) const {
384   //
385   int iqui = cherche(qui), iancetre = cherche(ancetre);
386
387   if (iqui < 0 or iancetre < 0) {
388     return false;
389   }
390   // cas trivial qu'on regle tout de suite.
391   if (iqui == iancetre) {
392     return true;
393   }
394   return ascendantAux(iqui, iancetre);
395   //
396 }
397
398 TEST_CASE("ascendant recursif") {
```

```
399     //
400     EtatCivil a;
401     a = creeEtatCivildeTest();
402     a.personne("Lamia", Genre::Fem);
403     a.mariage("Lamia", "Yuri");
404     a.naissance("Christophe", Genre::Masc, "Lamia", "Yuri");
405
406     CHECK(a.ascendantR("Christophe", "Remy"));
407     CHECK_FALSE(a.ascendantR("Christophe", "Noemie"));
408     //
409 }
```



▸ **Exercice 3. (Gestion du stock d'une Pharmacie)**

Un pharmacien souhaite informatiser le traitement des prescriptions de ses clients. Un médicament est représenté par son nom, le nombre de comprimés par boîte, le prix de la boîte et le nombre de boîtes en stock. L'ensemble des médicaments existants est stocké dans le tableau `table` de la classe `Stock`. Chaque prescription est composée d'un nom de médicament, d'un nombre de comprimés à prendre par jour (au plus 6 comprimés par jour), pendant un certain nombre de jours (au plus 31 jours). Les structures de données choisies sont donc les suivantes :

```
struct Medicament {
  string nom;
  int nbBoites;
  int nbParBoite;
  float prixBoite;
};
```

```
struct Stock {
  vector<Medicament> table;
};
```

```
struct Prescription {
  string med;
  int nbCparJour;
  int nbJours;
};
```

On testera toutes les fonctions après les avoir écrites, et on créera un TEST CASE à chaque fois que c'est possible.

1. Dans le fichier `Pharmacie.cpp` fourni, écrire et tester la méthode `float prixComprime()` de la classe `Medicament` qui renvoie le prix d'un seul comprimé du médicament.

```
87  float Medicament::prixComprime() const {
88    //
89    return prixBoite / nbParBoite;
90    //
91  }
92
93  TEST_CASE("test prix comprime") {
94    Stock s = creeUnStockdeTest();
95    CHECK(s.table[0].prixComprime() == 0.875f);
96    CHECK(s.table[1].prixComprime() == 0.5f);
97    CHECK(s.table[2].prixComprime() == 1.25f);
98  }
```

2. Écrire la méthode `void changePrix(float nouvPrix)` de la classe `Medicament` qui prend en paramètre le nouveau prix par boîte d'un médicament et qui modifie son prix par boîte.

```
102 void Medicament::changePrix(float nouvPrix) {
103   //
104   prixBoite = nouvPrix;
105   //
106 }
```

3. Réaliser la méthode `int indiceMedicament(string nomMedicament)` de la classe `Stock` qui permet de chercher un médicament avec son nom dans la base de données du stock et renvoie son indice dans le tableau ou -1 si elle ne le trouve pas. Tester la fonction.

```
int Stock::indiceMedicament(string nomMedicament) const {
  //
  int nbMeds = table.size();
  for (int i = 0; i < nbMeds; i++) {
    if (nomMedicament == table[i].nom)
      return i;
  }
  return -1;
  //
}

TEST_CASE("trouver un medicament dans le stock par son nom") {
  Stock s = creeUnStockdeTest();
  CHECK(s.indiceMedicament("panadol") == 0);
  CHECK(s.indiceMedicament("xanax") == -1);
}
```

4. Ecrire la méthode `void ajouteMedicament()` de la classe `Stock` qui permet d'ajouter un nouveau médicament à la base de données du stock, à partir de données entrées au clavier par l'utilisateur. Vérifier que le médicament n'existe pas déja.

```
void Stock::ajouteMedicament() {
  //
  Medicament med;
  do {
    cout << "entrez le nom du medicament" << endl;
    cin >> med.nom;
  } while (indiceMedicament(med.nom) >= 0);
  med.nbBoites = lireValeurBornee("entrez nombre de boites ", 0, 100);
  med.nbParBoite = lireValeurBornee("entrez nombre de comprimés par boite ", 0, 100);
  cout << "entrez le prix " << endl;
  cin >> med.prixBoite;
  table.push_back(med);
  //
}
```

5. Ecrire la méthode `void lirePrescription(Stock s)` de la classe `Prescription` qui permet de saisir au clavier les informations relatives à une prescription (nom du médicament, nombre de comprimés par jour et durée du traitement). Sachant que le pharmacien n'accepte pas des préscriptions des médicaments qu'il n'a pas dans sa base de données, il faut donc chercher le nom du médicament dans le stock.

   Vous pouvez utiliser la fonction

   ```
   int lireValeurBornee(string texteAEcrire, int min, int max);
   ```

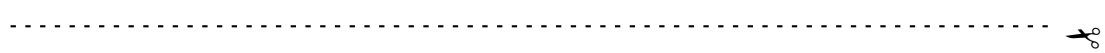
   fournie pour vous vous assurez que le nombre de comprimés à prendre par jour ne dépasse pas 6 comprimés et la durée du traitement ne dépasse pas un mois.

```
175 void Prescription::lirePrescription(const Stock &s) {
176   //
177   int indiceMed = 0;
178   do {
179     cout << "entrez le nom du medicament" << endl;
180     cin >> nomMed;
181     indiceMed = s.indiceMedicament(nomMed);
182   } while (nomMed == "" or indiceMed == -1);
183   nbCparJour = lireValeurBornee("Nbre de comprimes par jour ", 1, 6);
184   nbJours = lireValeurBornee("Duree du traitement ", 1, 31);
185   //
186 }
```

6. Écrire et tester la méthode `int nbBoites` de la classe `Prescription` qui renvoie le nombre de boîtes nécessaires pour couvrir la prescription. Par exemple, si le médicament est vendu par boîte de 20 comprimés, il ne faut qu'une boîte pour couvrir une prescription de 6 comprimés par jour pendant 3 jours, mais il faut 2 boîtes si le traitement dure 4 jours.

```
191 int Prescription::nbBoites(const Stock &s) const{
192   //
193   Medicament med = s.table[s.indiceMedicament(nomMed)];
194   int nbC = nbCparJour * nbJours;// nb de comprimes necessaires
195   if (med.nbParBoite >= nbC) return 1;
196   int res = nbC / med.nbParBoite;
197   if (nbC % med.nbParBoite != 0) res++;
198   return res;
199   //
200 }
201
202 TEST_CASE("la quantité de boites nécessaires pour un médicament") {
203   Stock s = creeUnStockdeTest();
204   Prescription p = {"panadol", 3, 7};
205   CHECK(p.nbBoites(s) == 6);
206   CHECK(Prescription {"panadol", 2, 7}.nbBoites(s) == 4);
207 }
```

7. Écrire et tester la méthode `float coutTotal` de la classe `Prescription` qui renvoie le prix total des boîtes nécessaires pour couvrir la prescription et met à jour la quantité du médicament dans le stock. Si la quantité dans le stock ne couvre pas le nombre des boîtes necessaires, on donne quand même au patient le nombre de boîtes présentes en affichant un message d'avertissement.

```
211 float Prescription::coutTotal(Stock &s){
212   //
213   int nbB = nbBoites(s);
214   int indiceMed = s.indiceMedicament(nomMed);
215   int nbStock = s.table[indiceMed].nbBoites;
216
```

```
217   if (nbStock >=  nbB) {
218     s.table[indiceMed].nbBoites = nbStock - nbB;
219     return nbB*s.table[indiceMed].prixBoite;
220   } else {
221     cout << "stock insuffisant, il manque " << nbB - nbStock << " boites";
222     s.table[indiceMed].nbBoites = 0;
223     return nbStock*s.table[indiceMed].prixBoite;
224   }
225   //
226 }
227
228 TEST_CASE("le cout total d'une prescription") {
229   Stock s = creeUnStockdeTest();
230   Prescription p = {"panadol", 2, 5};
231   CHECK(p.coutTotal(s) == 10.5f);
232   CHECK(Prescription {"panadol", 3, 20}.coutTotal(s) == 52.5f);
233 }
```