Correction TP de programmation n°5

Cours de programmation modulaire

—Licence MI/IM/MNSI - Info 3—

Révisions

Cette séance de travaux pratiques permet d'approfondir les notions de `structs`, `énumerations`, et `la surcharge d'opérateurs`.

▸ Exercice 1. (État civil)

Le but de l'application est d'enregistrer des informations d'état civil sur des personnes. Une personne est décrite par :

- son nom
- son genre (masculin ou féminin)
- son conjoint éventuel (pas forcément d'un genre différent)
- ses enfants, s'il en a
- ses parents, si on connaît cette information.

L'information n'est pas toujours complète : on peut ne pas connaître l'un ou l'autre (ou les deux) parents d'une personne. Le statut marital des personnes peut aussi évoluer dans le temps, de même que le nombre d'enfants.

On utilise pour cela les types ci-dessous : un « état civil » contient des informations sur les personnes dans un vecteur. Pour désigner une personne enregistrée dans le système, on peut utiliser l'indice de l'élément du vecteur qui contient les informations sur la personne. De plus, pour représenter une personne (ou une information) inconnue, on utilisera la valeur `-1` (qui ne peut pas être un indice). Par exemple si `Jean` est enregistré à l'indice `3` et `Marie` est enregistré à l'indice `5`, pour indiquer que `Jean` est marié à `Marie`, on mettra `5` comme valeur du champ `indConjoint` pour `Jean` et `3` comme valeur du même champ pour `Marie`. Si `Jean` n'est pas marié, son champ `indConjoint` contient `-1`.

```
12 enum class Genre { Masc, Fem };
13
14 struct Personne {
15   string nom;
16   Genre genre;
17   // indices dans le champ table de l'état civil de son conjoint et de
18   // ses parents ou -1 si l'information est inconnue
19   int indConjoint, indParent1, indParent2;
20   vector<int> enfants;  // indices des enfants (vecteur eventuellement vide)
21 };
22
23 struct EtatCivil {
24   string titre;     // Titre de l'état civil. Exemple: "Ville de Paris"
25   vector<Personne> table;  // description des personnes de cet état civil
26 };
```

Les questions sont à traiter dans le fichier `EtatCivil.cpp` fourni.

1. Surcharger l'opérateur d'affichage pour pouvoir afficher une variable de type `Genre`.

✂ ----------

```
50  // Surcharge de l'operator << pour enum Genre
51  std::ostream &operator<<(std::ostream &out, Genre g) {
52    //
53    if (g == Genre::Masc)
54      out << "Genre masculin";
55    else
56      out << "Genre feminin";
57    return out;
58    //
59  }
```

2. Dans le main, déclarez une variable de type `Genre`, donnez-lui une valeur de votre choix, et affichez-là (pour vérifier votre surcharge de l'opérateur d'affichage).

```
457   Genre g;
458   g = Genre::Masc;
459   cout << g << endl;
```

3. Écrire une procédure qui initialise un état civil : on l'intitule du titre passé en argument et on lui associe un tableau vide. Voici l'en-tête :

   ```
   void initialise(EtatCivil &a, string titre)
   ```

```
110 void initialise(EtatCivil &a, string titre) {
111   //
112   a.titre = titre;
113   a.table.clear();
114   //autre possibilité: a.table = vector<Personne>(0);
115   //
116 }
```

4. Tester la fonction d'initialisation. Pour ceci, tester que le titre a bien été enregistré et que la table des personnes est de taille `0`.

```
463   EtatCivil e;
464   initialise(e,"Annuaire");
465   cout << e.titre << endl;
466   cout << e.table.size() << endl;
```

5. Surcharger la fonction d'affichage pour une personne. On devra afficher son nom, son genre, son statut (célibataire ou marié), l'indice du conjoint, pour chaque parent on indiquera son indice s'il est enregistré ou «inconnu» sinon et le nombre des enfants.

```
63  // Surcharge de l'operator << pour struct personne
64  std::ostream &operator<<(std::ostream &out, Personne P) {
65    //
66    out << "Nom " << P.nom << endl << P.genre << endl << "Statut : ";
67    if (P.indConjoint == -1) {
68      out << "Celibat.";
69    } else {
70      out << "Mariée" << endl;
71      out << "Indice conjoint : " << P.indConjoint;
72    }
73    out << endl << "Indices parents : (";
74    if (P.indParent1 != -1) {
75      out << P.indParent1 << ", ";
76    } else {
77      out << "inconnu, ";
78    }
79    if (P.indParent2 != -1) {
80      out << P.indParent2;
81    } else {
82      out << "inconnu";
83    }
84    out << ")\n";
85    if (P.enfants.size() > 0) {
86      out << "Nb d'enfants : " << P.enfants.size() << endl;
87    }
88    return out;
89    //
90  }
```

6. Écrire une fonction

   ```
   int cherche(const EtatCivil &a, string nom);
   ```

   qui recherche le nom d'une personne dans le système et renvoie son indice si elle le trouve, ou -1 si la personne est inconnue.

```
138 int cherche(const EtatCivil &a, string nom) {
139   //
140   for (size_t i = 0; i < a.table.size(); i++) {
141     if (nom == a.table[i].nom)
142       return i;
143   }
144   return -1;
145   //
146 }
```

7. Écrire des tests pour cette fonction cherche. Pour ceci on pourra utiliser la fonction `creeEtatCivildeTest` fournie qui initialise un état civil de tests. Ne pas oublier de faire des tests positifs où l'on trouve bien la personne, mais aussi des tests négatifs où la personne n'existe pas.

```
150 TEST_CASE("cherche une personne") {
151   EtatCivil a;
152   a = creeEtatCivildeTest();
153   CHECK(cherche(a, "Noemie") == 2);
154   // Ajouter d'autre exemples ici
155   //
156   CHECK(cherche(a, "Armand") == 6);
157   CHECK(cherche(a, "Antoine") == -1);
158   //
159 }
```

8. Écrire une fonction qui affiche les informations pour une personne dont on fournit le nom.

   ```
   void imprimePersonne(const EtatCivil &a, string nom);
   ```

```
164 void imprimePersonne(const EtatCivil &a, string nom) {
165   //
166   int ind = cherche(a, nom);
167   if (ind < 0) {
168     throw invalid_argument("Nom incorrect dans imprimePersonne: ");
169   } else {
170     cout << a.table[ind];
171   }
172   //
173 }
```

9. Surcharger l'opérateur d'affichage pour un état civil. On devra afficher son titre et les personnes présente dans la table de l'état civil.

```
94  // Surcharge de l'operateur << pour struct état civil
95  std::ostream &operator<<(std::ostream &out, EtatCivil a) {
96    //
97    if (a.table.size() == 0) {
98      out << "L'état civil est vide" << endl;
99    } else {
100     for (size_t i = 0; i < a.table.size(); i += 1) {
101       imprimePersonne(a, a.table[i].nom);
102     }
103   }
104   return out;
105   //
106 }
```

10. Écrire une fonction qui enregistre dans le système une nouvelle personne. Les parents sont inconnus, il n'a pas de conjoint et pas d'enfants.

    ```
    int personne(EtatCivil &a, string nom, Genre s);
    ```

    On interdit toute homonymie dans le système (on ne doit pas avoir deux personnes de même nom) et une personne ne doit pas avoir un nom vide. La valeur de retour est l'indice de la nouvelle personne ou une valeur négative si la personne n'a pas pu être enregistrée.

✂ ----------

```
178
179 /* enregistre une personne (si possible) et renvoie son indice dans le tableau.
180  * Renvoie un nombre negatif en cas d'erreur. Dans la suite" du code on
181  * utilise les 3 valeurs suivantes pour les erreurs
182  * -1 : pas de personne du nom indique lors d'une recherche
183  * -2 : tentative d'ajout d'un homonyme
184  * -3 : nom de personne vide
185  * Si on n'est pas interesse par le detail d'une erreur, une valeur negative
186  * renvoyee a la place d'un indice indique unn cas d'erreur.
187  */
188 int personne(EtatCivil &a, string sonNom, Genre s) {
189   //
190   if (sonNom.size() == 0) {
191     return -3;
192   }; /* pas de nom vide */
193   if (cherche(a, sonNom) != -1) {
194     return -2;
195   }
196   vector<int> enfants;  // vector vide.
197   Personne p = {sonNom, s, -1, -1, -1, enfants};
198   a.table.push_back(p);
199   return a.table.size() - 1;
200   //
201 }
```

---------- ✂

11. Tester cette fonction avec des tests positifs et négatifs.

✂ ----------

```
207 TEST_CASE("Ajout d'une personne") {
208   //
209   EtatCivil a;
210   a = creeEtatCivildeTest();
211   CHECK(personne(a, "Lamia", Genre::Fem) == 8);
212   CHECK(personne(a, "Lamia", Genre::Fem) == -2);
213   CHECK(personne(a, "Antoine", Genre::Masc) == 9);
214   CHECK(personne(a, "Antoine", Genre::Masc) == -2);
215   CHECK(personne(a, "Noemie", Genre::Masc) == -2);
216   CHECK(personne(a, "", Genre::Masc) == -3);
217   //
218 }
```

---------- ✂

12. Écrire une fonction 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 mariage(EtatCivil &a, string lun, string lautre);
```

```
223 bool mariage(EtatCivil &a, string lun, string lautre) {
224   //
225   int ilun = cherche(a, lun), ilautre = cherche(a, lautre);
226   if (ilun < 0 or ilautre < 0 or ilun == ilautre) {
227     return false;
228   }
229   if (a.table[ilun].indConjoint != -1 or a.table[ilautre].indConjoint != -1) {
230     return false;
231   }
232   a.table[ilun].indConjoint = ilautre;
233   a.table[ilautre].indConjoint = ilun;
234   return true;
235   //
236 }
```

13. Tester cette fonction avec des tests positifs et négatifs.

```
240 TEST_CASE("Mariage de deux personnes") {
241   //
242   EtatCivil a;
243   a = creeEtatCivildeTest();
244   personne(a, "Lamia", Genre::Fem);
245   CHECK(mariage(a, "Lamia", "Yuri"));
246   CHECK_FALSE(mariage(a, "Guillaume", "Yuri"));
247   //
248 }
```

14. Écrire une fonction qui enregistre la naissance d'une personne. Son en-tête est :

```
bool naissance(EtatCivil &a, 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.

```
257 bool naissance(EtatCivil &a, string qui, Genre s, string p1, string p2) {
258   //
259   int iqui, ip1, ip2;
260
261   ip1 = cherche(a, p1);
262   ip2 = cherche(a, p2);
```

```
263   // ATTENTION: ici ne pas utiliser la fonction mariage de la question
264   // precedente car celle-ci ne teste pas si deux personnes sont mariees
265   // mais elle les marient s'ils etaient celibataires ! On doit donc
266   // refaire les tests, mais ici plus simples.
267   if (ip1 == -1 or ip2 == -1 or
268       a.table[ip1].indConjoint != ip2
269       // ce dernier test est inutile si la fonction mariage est correcte:
270       // si l'un est conjoint de l'autre, la reciproque doit etre vraie aussi
271       // !
272       or a.table[ip2].indConjoint != ip1) {
273     return false;
274   }
275
276   // essayer d'ajouter le nouveau-ne
277   iqui = personne(a, qui, s);
278   if (iqui < 0) {
279     return false;
280   }
281
282   // enregistrer la filiation dans les deux sens: de l'enfant vers les
283   // parents et l'inverse.
284   a.table[iqui].indParent1 = ip1;
285   a.table[ip1].enfants.push_back(iqui);
286
287   a.table[iqui].indParent2 = ip2;
288   a.table[ip2].enfants.push_back(iqui);
289
290   return true;
291   //
292 }
```

15. Tester cette fonction.

```
296 TEST_CASE("Naissance d'un bébé") {
297   //
298   EtatCivil a;
299   a = creeEtatCivildeTest();
300   personne(a, "Lamia", Genre::Fem);
301   mariage(a, "Lamia", "Yuri");
302   CHECK(naissance(a, "Christophe", Genre::Masc, "Lamia", "Yuri"));
303   CHECK_FALSE(naissance(a, "Dina", Genre::Masc, "Noemie", "Yuri"));
304   //
305 }
```

▸ Exercice 2. (Généalogie)

Cet exercice est la suite de l'exercice précédent. On travaillera dans le même fichier.

16. Écrire une version 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 ascendant(EtatCivil &a, string qui, string ancetre);
```

Tester cette fonction.

✂ ----------

```
316 bool ascendantI(const EtatCivil &a, string qui, string ancetre) {
317   //
318   vector<int> pile;
319   int iqui = cherche(a, qui), iancetre = cherche(a, ancetre);
320   if (iqui < 0 or iancetre < 0) {
321     return false;
322   }
323   pile.push_back(iqui);  // On empile l'indice de depart
324   // pile des indices des personnes a tester: on remonte dans l'arbre
325   // genealogique.
326   while (pile.size() > 0) {
327     int courant = pile[pile.size() - 1];
328     pile.pop_back();
329     if (courant == iancetre) {
330       return true;
331     } else {
332       // ajouter ses parents, si connus, dans la pile des personnes a
333       // tester
334       int i1 = a.table[courant].indParent1;
335       int i2 = a.table[courant].indParent2;
336       if (i1 != -1) {
337         pile.push_back(i1);
338       }
339       if (i2 != -1) {
340         pile.push_back(i2);
341       }
342     }
343   }
344   /* fin de la boucle: on a visite tous les ancetres sans trouver indParent */
345   return false;
346   //
347 }
348
349 TEST_CASE("ascendent itiratif") {
350   //
351   EtatCivil a;
352   a = creeEtatCivildeTest();
353   personne(a, "Lamia", Genre::Fem);
354   mariage(a, "Lamia", "Yuri");
355   naissance(a, "Christophe", Genre::Masc, "Lamia", "Yuri");
356   CHECK(ascendantI(a, "Christophe", "Remy"));
357   CHECK_FALSE(ascendantI(a, "Christophe", "Noemie"));
358   //
359 }
```

---------- ✂

17. Écrire une fonction récursive de la fonction précédente.

```
364
365 /* version recursive de la recherche d'ascendant. On utilise une fonction
366  * auxiliaire pour ne pas faire plusieurs fois les tests de validite des indices
367  */
368 bool ascendantAux(const EtatCivil &a, int iqui, int iancetre) {
369   //
370   int i1, i2;
371   if (iqui == iancetre) {
372     return true;
373   }
374   // on remonte dans l'arbre a la recherche de l'ancetre. Plus efficace
375   // que de descendre de l'ancetre vers les enfants qui explorerait aussi
376   // les cousins (eloignes)
377   i1 = a.table[iqui].indParent1;
378   i2 = a.table[iqui].indParent2;
379   return (i1 != -1 and ascendantAux(a, i1, iancetre)) or
380          (i2 != -1 and ascendantAux(a, i2, iancetre));
381   //
382 }
383
384 bool ascendantR(const EtatCivil &a, string qui, string ancetre) {
385   //
386   int iqui = cherche(a, qui), iancetre = cherche(a, ancetre);
387
388   if (iqui < 0 or iancetre < 0) {
389     return false;
390   }
391   // ca trivial qu'on regle tout de suite.
392   if (iqui == iancetre) {
393     return true;
394   }
395   return ascendantAux(a, iqui, iancetre);
396   //
397 }
398
399 TEST_CASE("ascendent recursif") {
400   //
401   EtatCivil a;
402   a = creeEtatCivildeTest();
403   personne(a, "Lamia", Genre::Fem);
404   mariage(a, "Lamia", "Yuri");
405   naissance(a, "Christophe", Genre::Masc, "Lamia", "Yuri");
406
407   CHECK(ascendantR(a, "Christophe", "Remy"));
408   CHECK_FALSE(ascendantR(a, "Christophe", "Noemie"));
409   //
410 }
```

Tester cette fonction.

18. Coder une procédure permettant d'afficher, pour un individu donné, son arbre généalogique sous la forme :

```
Individu
    Mère
        Grand-mère maternelle
```

```
            ...
        Grand-père maternel
            ...
    Père
        Grand-mère paternelle
            ...
        Grand-père paternel
            ...
```

Voici par exemple l'affichage de l'arbre généalogique de l'individu 9 :

```
Individu 9
   Individu 5
      Individu inconnu
      Individu 1
         Individu inconnu
         Individu inconnu
   Individu 10
      Individu 12
         Individu inconnu
         Individu inconnu
      Individu 11
         Individu inconnu
         Individu inconnu
```

Indication : faire fonction récursive contenant 2 appels récursifs. Démarrer l'écriture de la fonction récursive en pensant au cas d'arrêt puis ensuite réfléchissez à l'appel récursif.

✂ ----------

```
417 //
418 /* On utilise une fonction auxiliaire pour indiquer la génération et pour
419  * ne pas faire plusieurs fois les tests de validite des indices
420  */
421 void AfficheArbreGenePersonneAux(int ind, EtatCivil EC, int generation) {
422   int i = 0;
423   cout << "Individu ";
424   if (ind == -1) {
425     cout << "inconnu" << endl;
426   } else {
427     cout << ind << endl;
428     for (i=0; i < generation; i++)
429       cout << "   ";
430     AfficheArbreGenePersonneAux(EC.table[ind].indParent1, EC, generation+1);
431     for (i=0; i < generation; i++)
432       cout << "   ";
433     AfficheArbreGenePersonneAux(EC.table[ind].indParent2, EC, generation+1);
434   }
435 }
436
437 void AfficheArbreGenePersonne(int ind, EtatCivil EC) {
438   int generation = 1;
439   if (ind >= (int)EC.table.size())
440     cout << "La personne d'indice " << ind << " n'existe pas." << endl;
441   else
442     AfficheArbreGenePersonneAux(ind, EC, generation);
```

```
443  }
444  //
     ------------------------------------------------------------------------ ✂
```