université PARIS-SACLAY

Correction TP de programmation n°6

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

Types tableau de structures et structures de tableau

Cette séance de travaux pratiques est dédiée à l'écriture et l'utilisation de programmes sur les tableaux et structures.

▸ **Exercice 1. (État civil)**

On veut coder une application qui traite des informations sur l'état civil des personnes. Une personne est décrite par : son nom avec 20 caractères maximum, son genre (masculin ou féminin), les indices (entiers) de son conjoint éventuel et de ses parents. On suppose qu'on a au maximum 50 personnes à enregistrer. Enfin on interdira toute homonymie dans le système (c'est-à-dire qu'il ne peut pas y avoir deux personnes portant le même nom).

L'information n'étant 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.

On vous donne le type `Personne` ainsi que la structure pour représenter les informations de 50 personnes :

```
const int MAXPERSONNES = 50;

enum Genre { M, F };

struct Personne {
  string nom;
  Genre genre;
  int indConjoint, indParent1, indParent2;
};

struct EtatCivil {
   Personne tableP[MAXPERSONNES];
   int nbP;
};
```

1. Coder une procédure `AjoutePersonne` qui ajoute une nouvelle personne dans l'état civil. Cette fonction prendra en paramètre le nom de cette personne, son genre et les deux indices de ses parents (les indices peuvent être -1 s'ils sont inconnus). On suppose qu'au moment de cet ajout, la personne n'a pas de conjoint. La procédure doit afficher un message d'erreur si elle ne peut pas ajouter la personne.
2. Coder une procédure `AffichePersonne` qui affiche les informations d'une personne donnée par son indice dans la base de l'état civil. Si la personne n'existe pas la procédure doit afficher un message d'erreur.
3. Coder une procédure qui affiche toutes les personnes de l'état civil.
4. Complétez le `main` pour qu'il appelle la procédure `RemplitEtatCivil` (permettant de remplir l'état civil avec 20 personnes) et utilise les fonctions et procédures précédentes afin de les tester.

5. Coder et tester une fonction qui recherche une personne donnée par son nom dans l'état civil et renvoie son indice si elle le trouve, ou -1 si la personne est non existante.
6. Coder et tester une fonction qui enregistre, dans l'état civil, le mariage de deux personnes données par leurs noms. La fonction renvoie `true` si le mariage est possible et `false` sinon. Un mariage entre deux personnes est possible si les deux personnes existent et si elles ne sont pas déjà mariées.
7. (Question difficile, vous pouvez passer à l'exercice suivant et revenir à cette question plus tard) 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.

✂ ----------

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  const int MAXPERSONNES = 50;
6
7  enum Genre { M, F };
8
9  struct Personne {
10     string nom;
11     Genre genre;
```

```
12      int indConjoint, indParent1, indParent2;
13  };
14
15  struct EtatCivil {
16      Personne tableP[MAXPERSONNES];
17      int nbP;
18  };
19
20  void AjoutPersonne(string sonNom, Genre s, int parent1, int parent2, EtatCivil &EC) {
21      int ind;
22      bool ajoutOK = true;
23      if (EC.nbP == MAXPERSONNES) {
24          ajoutOK = false;
25          cout << "La base est pleine, on ne peut pas ajouter une personne." << endl;
26      } else {
27          for (int i = 0; i < EC.nbP; i++) // interdire l'homonymie
28              if (EC.tableP[i].nom == sonNom) {
29                  ajoutOK = false;
30                  cout << "La personne " << sonNom << " existe deja dans la base." << endl;
31                  break;
32              }
33          if (ajoutOK == true) {
34              ind = EC.nbP;
35              EC.tableP[ind].nom = sonNom;
36              EC.tableP[ind].genre = s;
37              EC.tableP[ind].indConjoint = -1; // pas de conjoint
38              EC.tableP[ind].indParent1 = parent1;
39              EC.tableP[ind].indParent2 = parent2;
40              EC.nbP++; // mise a jour du nombre de personnes
41          }
42      }
43  }
44
45  void AffichePersonne(int ind, EtatCivil EC) {
46      if (ind >= EC.nbP or ind < 0) {
47          cout << "La personne d'indice " << ind << " n'existe pas." << endl;
48      } else {
49          cout << "Individu " << ind << endl;
50          cout << "Nom : " << EC.tableP[ind].nom << endl;
51          cout << "Genre : ";
52          if (EC.tableP[ind].genre==M)
53              cout << "Masculin" << endl;
54          else
55              cout << "Feminin" << endl;
56          if (EC.tableP[ind].indConjoint == -1)
57              cout << "Célibataire" << endl;
58          else
59              cout << "Conjoint: " << EC.tableP[ind].indConjoint << endl;
60          cout << "Parents : ("
61               << EC.tableP[ind].indParent1 << ", "
62               << EC.tableP[ind].indParent2 << ")" << endl << endl;
63      }
64  }
65
66  void AfficheEtatCivil(EtatCivil EC) {
```

```
67      for(int i=0; i<EC.nbP; i++) AffichePersonne(i, EC);
68  }
69
70  void RemplitEtatCivil(EtatCivil &EC) {
71      AjoutPersonne("Noemie", F, -1, -1, EC);
72      AjoutPersonne("Georges", M, -1, -1, EC);
73      AjoutPersonne("Albert", M, -1, -1, EC);
74      AjoutPersonne("Marie", F, -1, -1, EC);
75      AjoutPersonne("Luc", M, 0, -1, EC);
76      AjoutPersonne("Valerie", F, -1, 1, EC);
77      AjoutPersonne("Stephane", M, 3, 2, EC);
78      AjoutPersonne("Helene", F, 5, 4, EC);
79      AjoutPersonne("Justine", F, 7, 6, EC);
80      AjoutPersonne("Berenice", F, 5, 10, EC);
81      AjoutPersonne("John", M, 12, 11, EC);
82      AjoutPersonne("Franco", M, -1, -1, EC);
83      AjoutPersonne("Viviane", F, -1, -1, EC);
84      AjoutPersonne("Pierre", M, 9, 14, EC);
85      AjoutPersonne("Remi", M, 16, 15, EC);
86      AjoutPersonne("Boris", M, -1, 17, EC);
87      AjoutPersonne("Sharon", F, 19, 18, EC);
88      AjoutPersonne("Alexandre", M, -1, -1, EC);
89      AjoutPersonne("Augustin", M, -1, -1, EC);
90      AjoutPersonne("Johanne", F, -1, -1, EC);
91  }
92
93  int RecherchePersonne(string nom, EtatCivil EC) {
94      for (int i = 0; i < EC.nbP; i++)
95          if (EC.tableP[i].nom == nom) return i;
96      return -1;
97  }
98
99  bool Mariage(string nom1, string nom2, EtatCivil &EC) {
100     int ind1, ind2;
101     ind1 = RecherchePersonne(nom1, EC);
102     ind2 = RecherchePersonne(nom2, EC);
103     if (ind1 == -1 || ind2 == -1 ||
104             EC.tableP[ind1].indConjoint != -1 ||
105             EC.tableP[ind2].indConjoint != -1)
106         return false;
107     EC.tableP[ind1].indConjoint = ind2;
108     EC.tableP[ind2].indConjoint = ind1;
109     return true;
110 }
111
112
113 /* On utilise une fonction auxiliaire pour indiquer la génération et pour
114  * ne pas faire plusieurs fois les tests de validite des indices
115  */
116 void AfficheArbreGenePersonneAux(int ind, EtatCivil EC, int generation) {
117     int i = 0;
118     cout << "Individu ";
119     if (ind == -1) {
120         cout << "inconnu" << endl;
121     } else {
```

```
122         cout << ind << endl;
123         for (i=0; i < generation; i++)
124             cout << "   ";
125         AfficheArbreGenePersonneAux(EC.tableP[ind].indParent1, EC, generation+1);
126         for (i=0; i < generation; i++)
127             cout << "   ";
128         AfficheArbreGenePersonneAux(EC.tableP[ind].indParent2, EC, generation+1);
129     }
130 }
131
132 void AfficheArbreGenePersonne(int ind, EtatCivil EC) {
133     int generation = 1;
134     if (ind >= EC.nbP)
135         cout << "La personne d'indice " << ind << " n'existe pas." << endl;
136     else
137         AfficheArbreGenePersonneAux(ind, EC, generation);
138 }
139
140 int main() {
141     EtatCivil EC;
142     EC.nbP = 0; // il est necessaire d'initialiser le nombre de personnes
143
144     RemplitEtatCivil(EC);
145
146     // essayer d'ajouter une personne qui existe deja
147     AjoutPersonne("Nom4", M, 0, -1, EC);
148
149     AffichePersonne(0, EC);
150     AffichePersonne(1, EC);
151     AffichePersonne(2, EC);
152     AffichePersonne(3, EC);
153     AffichePersonne(20, EC); // cet appel affiche que cette personne n'existe pas
154
155     AfficheEtatCivil(EC); // affiche tout l'etat civil
156
157     // rechercher des personnes dans l'etat civil
158     cout << "Personne Nom4: " << RecherchePersonne("Nom4", EC) << endl;
159     cout << "Personne xxxx: " << RecherchePersonne("xxxx", EC) << endl;
160
161     // tester le mariage entre deux personnes (on suppose couples homosexuels OK)
162     bool mariage = Mariage("Nom4", "xxxx", EC);
163     cout << "Mariage Nom4 + xxxx: " << (mariage ? "ok" : "impossible") << endl;
164
165     mariage = Mariage("Nom4", "Nom5", EC);
166     cout << "Mariage Nom4 + Nom5: " << (mariage ? "ok" : "impossible") << endl;
167     if (mariage) {
168         AffichePersonne(RecherchePersonne("Nom4", EC), EC);
169         AffichePersonne(RecherchePersonne("Nom5", EC), EC);
170     }
171
172     mariage = Mariage("Nom6", "Nom4", EC);
173     cout << "Mariage Nom6 + Nom4: " << (mariage ? "ok" : "impossible") << endl;
174
175     // affiche l'arbre généalogique de l'individu 13
176     AfficheArbreGenePersonne(13, EC);
```

```
177
178       return 0;
179   }
```

▸ **Exercice 2. (Jeu de cartes)**

L'objectif de cet exercice est de programmer un jeu de cartes simple : la bataille. Les cartes sont représentées par leur valeur de 1 à 10 et par leur couleur Pique, Cœur, Trèfle, Carreau. On vous donne le type `Couleur`, celui d'une carte ainsi que celui d'un paquet de cartes :

```
enum Couleur { P, Co, T, Ca };

struct Carte {
  int valeur;
  Couleur couleur;
};

struct Paquet {
  int taille;
  Carte carte[40];
};
```

1. Coder une procédure `initPaquet` qui prend en paramètre un paquet de cartes et le remplit avec les 40 cartes possibles.
2. Coder une procédure qui affiche un paquet de cartes dans l'ordre dans lequel les cartes sont dans le paquet.
3. Coder une procédure qui mélange le paquet de cartes. L'idée est de tirer au hasard deux nombres entre 0 et 39 et de permuter les cartes à ces positions. Vous répéterez cette procédure suffisamment pour mélanger le paquet. Indication : utiliser la fonction `rand()` qui ne prend pas de paramètre et qui renvoie un entier aléatoire, et utiliser les opérations sur les entiers pour ramener cet entier aléatoire dans l'intervalle voulu.
4. Quand deux joueurs s'affrontent, chacun tire une carte du dessus de son paquet et celui qui a la plus forte valeur prend les deux cartes et les met à la fin de son paquet. Quand les deux cartes sont de valeurs égales, c'est la couleur qui permet de décider le vainqueur, les couleurs étant ordonnées de la plus forte à la plus faible ainsi : Pique, Cœur, Trèfle, Carreau. Coder une procédure qui prend en paramètre deux paquets et les modifie de façon à représenter un tour de jeu.
5. Le jeu se termine quand un joueur gagne toutes les cartes. Dans le `main`, répartir en deux paquets de 20 cartes le paquet initial puis afficher quel joueur gagne la partie (et éventuellement les étapes de cette partie).

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <string>
4   using namespace std;
```

```
5
6   const int NB_CARTES = 40;
7
8   enum Couleur { P, Co, T, Ca };
9
10  struct Carte {
11      int valeur;
12      Couleur couleur;
13  };
14
15  struct Paquet {
16      int taille;
17      Carte carte[NB_CARTES];
18  };
19
20  void initPaquet (Paquet &paquet) {
21      paquet.taille = NB_CARTES;
22      for(int i = 0; i < NB_CARTES/4; i++) {
23          for (int j=0; j<4; j++) {
24              paquet.carte[4*i + j] = {i+1, Couleur(j)};
25          }
26      }
27  }
28
29  const string nomCouleur[4] = {"Pique", "Coeur", "Trefle", "Carreau"};
30
31  void afficherCarte (Carte c){
32      cout << c.valeur << " de "
33           << nomCouleur[c.couleur];
34  }
35
36  void afficherPaquetCarte (Paquet paquet){
37      int i;
38      for (i = 0; i< NB_CARTES; i++) {
39          cout << "carte " << i << ": ";
40          afficherCarte(paquet.carte[i]);
41          cout << endl;
42      }
43  }
44
45  void melange (Paquet &paquet) {
46      int i, a, b;
47      Carte temp;
48      for(i = 0; i < 2*NB_CARTES; i++) {
49          a = rand() % paquet.taille;
50          b = rand() % paquet.taille;
51          temp = paquet.carte[a];
52          paquet.carte[a] = paquet.carte[b];
53          paquet.carte[b] = temp;
54      }
55  }
56
57  bool gagne(Carte carte1, Carte carte2) {
58      return not ((carte1.valeur - carte1.couleur) > (carte2.valeur - carte2.couleur));
59  }
```

```
60
61  void deplaceCarte(Paquet &paquet1, Paquet &paquet2) {
62      int i;
63      Carte top1, top2;
64      top1 = paquet1.carte[paquet1.taille - 1];
65      top2 = paquet2.carte[paquet2.taille - 1];
66      paquet1.taille++;
67      paquet2.taille--;
68      for(i = paquet1.taille-1; i > 1; i--) {
69          paquet1.carte[i] = paquet1.carte[i-2];
70      }
71      paquet1.carte[0] = top1;
72      paquet1.carte[1] = top2;
73  }
74
75  void tour(Paquet &paquet1, Paquet &paquet2) {
76      Carte c1 = paquet1.carte[paquet1.taille-1];
77      Carte c2 = paquet2.carte[paquet2.taille-1];
78      afficherCarte(c1); cout << " contre ";
79      afficherCarte(c2); cout << " : ";
80      if (gagne(c1, c2)){
81          cout << "Joueur 2 gagne une bataille" << endl;
82          deplaceCarte(paquet2, paquet1);
83      } else {
84          cout << "Joueur 1 gagne une bataille" << endl;
85          deplaceCarte(paquet1, paquet2);
86      }
87  }
88
89  int main() {
90      int i;
91      Paquet monpaquet, paquet1, paquet2;
92
93      srand(time(NULL));
94      initPaquet(monpaquet);
95      afficherPaquetCarte(monpaquet);
96      melange(monpaquet);
97      //pour afficher le paquet mélangé:
98      //afficherPaquetCarte(monpaquet);
99
100     paquet1.taille = paquet2.taille = NB_CARTES/2;
101     for (i = 0; i < NB_CARTES/2; i++)
102         paquet1.carte[i] = monpaquet.carte[i];
103     for (i = NB_CARTES/2; i < NB_CARTES; i++)
104         paquet2.carte[i - NB_CARTES/2] = monpaquet.carte[i];
105
106     while (paquet1.taille != 0 and paquet2.taille != 0)
107         tour(paquet1, paquet2);
108     if (paquet1.taille != 0)
109         cout << "Joueur 1 gagne" << endl;
110     else
111         cout << "Joueur 2 gagne" << endl;
112
113     return 0;
114 }
```

Exercice 3. Dans une variante des règles de du jeu de Bataille souvent utilisée. En cas de bataille, c'est-à-dire en cas d'égalité des valeurs des deux cartes jouées, chaque joueur pose sur la table une carte face cachée et ensuite une nouvelle carte face visible. On décide du sort de la bataille avec ces nouvelles cartes. Il se peut qu'elles soient à leur tour de même valeur, on recommence alors avec deux cartes cachées et deux cartes visibles, jusqu'à ce que

- Soit l'un des deux joueurs a terminé son paquet. Dans ce cas, il a perdu et l'autre joueur à gagné la partie.
- Soit les deux dernières cartes faces visibles ne sont pas de même valeurs. Le joueur qui a posé la plus forte carte remporte alors toutes les cartes posées sur la table.

6. Après en avoir fait une copie de sauvegarde, modifier votre programme pour l'adapter à ces nouvelles règles.