université PARIS-SACLAY

Travaux Pratiques de programmation n°2

Cours de programmation modulaire

—Licence MI L1 S2 - Info 3—

Structures et énumérations

Dans cette séance, nous allons travailler avec les types énumérés et les structures.

Rappel : Toute séance de travail doit commencer par un chargement du TP :
— `./course.py fetch Semaine2`

La commande de compilation est la suivante

```
clang++ -std=c++11 -Wall nom_programme.cpp -o nom_programme
```

Enfin, pensez non seulement à **compiler**, mais aussi à **exécuter** et tester vos fonctions dès que possible.

On rappelle qu'un type énuméré est un type dont la liste complète des valeurs est connue. Par exemple, on peut déclarer un type `Jour` de la manière suivante :

```
enum class Jour { lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche };
```

On peut alors déclarer et initialiser des variables de type `Jour` :

```
Jour aujourdhui = Jour::lundi;
```

et tester leurs valeurs :

```
if (aujourdhui == Jour::mardi) ...
```

▸ Exercice 1.(Logique floue)

Dans cet exercice, on veut faire de la logique booléenne en présence de valeurs inconnues. On va donc avoir une logique à trois valeurs de vérité `vrai`, `faux` et `inconnu`. On appelle parfois ces valeurs de vérité des *Triléens*. Les opérateurs usuels `et`, `ou`, `non` gardent leurs sens habituels si on les applique sur `vrai` ou `faux`. La valeur `inconnu` veut dire que l'on ne sait pas si la valeur est `vrai`, ou `faux`. On va cependant essayer de prévoir le résultat du calcul au mieux avec ce que l'on sait.

Quelques exemples :
— `non(inconnu)` = `inconnu`. En effet, sachant que `non(vrai)` = `faux` et `non(faux)` = `vrai`, on ne peut pas prévoir le résultat.
— `vrai ou inconnu` = `vrai`. En effet, comme dans les deux cas `vrai ou vrai` = `vrai` et `vrai ou faux` = `vrai`, le résultat est toujours `vrai`, on est sûr que le résultat est `vrai`.
— `vrai et inconnu` = `inconnu`. En effet, comme `vrai et vrai` = `vrai` et `vrai et faux` = `faux`, on ne peut pas prévoir le résultat.

1. Dans le fichier `logic.cpp`, on a déclaré un type enuméré `Tril`, ainsi qu'une fonction de saisie. Lire les déclarations correspondantes, et vérifiez que vous les comprenez. Puis écrire le code de la fonction `affiche` pour le type `Tril`. Enfin pour vérifier le fonctionnement correct de la saisie et de l'affichage, ajoutez un appel dans le `main` (et compilez et exécutez et vérifiez ce que ça donne, à faire à chaque question même si on ne le rappelle pas !).
2. Compléter les deux fonctions `from_bool`, qui convertit un `bool` en `Tril`, et `to_bool` qui convertit un `Tril` en `bool`. Pour cette dernière fonction, si l'on essaye de convertir `inconnu`, on déclenchera une erreur (exception) avec la commande

   ```
   throw logic_error("Impossible de convertir inc en bool");
   ```

Vérifier votre fonction sur plusieurs cas, en particulier vérifier que l'erreur est bien signalée.

3. Écrire la fonction `non` et compléter la fonction de tests pour `non` ;
4. À l'aide d'une boucle pour parcourir toutes les valeurs possibles de v, ajouter un test pour vérifier l'égalité $\texttt{non}(\texttt{non}(v)) = v$ pour toutes les valeurs de vérité v ;
5. Écrire la fonction `et` ;
6. Écrire une fonction de test pour `et`. Pour tester de nombreux cas, on vérifiera aussi que pour toute valeur de vérité a et b, on a les deux identités :

$$a \texttt{ et } a = a \qquad\qquad a \texttt{ et } b = b \texttt{ et } a.$$

▸ Exercice 2.(Logique floue – avancée)

Cet exercice est pour les étudiants rapides qui ont fait l'exercice précédent en moins de une heure. Si vous n'êtes pas dans ce cas, passez directement à l'exercice suivant. Si vous êtes dans ce cas, commencez cet exercice mais n'y passez pas trop de temps, il faut garder du temps pour l'exercice suivant.

Les opérateurs `ou`, `et` et `non` de la logique booléenne, vérifient les identités suivantes :

- Idempotence :
  - $a \texttt{ ou } a = a$
  - $a \texttt{ et } a = a$
- Commutativité :
  - $a \texttt{ ou } b = b \texttt{ ou } a$
  - $a \texttt{ et } b = b \texttt{ et } a$
- Associativité
  - $a \texttt{ ou } (b \texttt{ ou } c) = (a \texttt{ ou } b) \texttt{ ou } c$
  - $a \texttt{ et } (b \texttt{ et } c) = (a \texttt{ et } b) \texttt{ et } c$
- Distributivité
  - $a \texttt{ et } (b \texttt{ ou } c) = (a \texttt{ et } b) \texttt{ ou } (a \texttt{ et } c)$.
  - $a \texttt{ ou } (b \texttt{ et } c) = (a \texttt{ ou } b) \texttt{ et } (a \texttt{ ou } c)$.
- Règles de de Morgan
  - $\texttt{non}(a \texttt{ ou } b) = (\texttt{non } a) \texttt{ et } (\texttt{non } b)$
  - $\texttt{non}(a \texttt{ et } b) = (\texttt{non } a) \texttt{ ou } (\texttt{non } b)$

Il se trouve que ces identités restent vraies dans notre logique à trois valeurs ! Elles fournissent un bon moyen de tester nos fonctions.

7. Reprendre le fichier de l'exercice précédent et ajouter une fonction `ou`.
8. Pour bien tester les fonctions `ou`, `et` et `non`, on vérifiera les identités ci-dessus pour toute valeur de vérité a et b et c.

▸ **Exercice 3.(Bridge)**

Le Bridge est un jeu de cartes qui se joue à 4 joueurs avec un jeu de 52 cartes, i.e. 13 cartes de chaque couleur dont les valeurs appartiennent au type énuméré `ValeurCarte` suivant (Remarque : les noms des valeurs d'un type énuméré doivent commencer par une lettre, d'où les noms `v2`, `v3`...).
Les valeurs seront :

```
enum class ValeurCarte { v2, v3, v4, v5, v6, v7, v8, v9, v10, Valet, Dame, Roi, As };
```

dans chacune des couleurs du type énuméré :

```
enum class CouleurCarte { pique, coeur, carreau, trefle };
```

Chacun des 4 joueurs reçoit aléatoirement une **main**, c'est-à-dire 13 cartes, dont il doit évaluer la **force**.

On choisit de se donner les représentations suivantes pour une carte, et pour une main de 13 cartes.

```
1  struct Carte {
2    ValeurCarte valeur;
3    CouleurCarte couleur;
4  };
5
6  using MainJ = array<Carte, 13>;
```

La **force** d'une main prend en compte deux aspects : les points d'Honneurs (ptH) et les points de Distribution (ptD).

Les **points d'Honneurs** d'une main s'évaluent en sommant la valeur de chacune des cartes présentes dans la main : chaque As vaut 4 points, chaque Roi, 3 points, chaque Dame 2 points et chaque Valet 1 point, les autres cartes ne valent rien. Le tableau `ptHCarte` est un tableau d'entiers (`array<int, 13>`) qui associe à chaque valeur de carte son nombre de points d'honneurs. Il est donné dans une constante globale.

`ptHCarte` :

v2	v3	v4	v5	v6	v7	v8	v9	v10	Valet	Dame	Roi	As
0	0	0	0	0	0	0	0	0	1	2	3	4

Les **points de Distribution** s'évaluent en décomptant 3 points pour une chicane (pas de carte dans une couleur), 2 points pour 1 singleton (1 seule carte dans une couleur) et 1 point pour 1 doubleton (2 cartes dans une couleur).

Exemple de main :

0	1	2	3	4	5	6	7	8	9	10	11	12
As ♠	Roi ♠	v10 ♠	v2 ♠	As ♡	Dame ♡	v10 ♡	v9 ♡	v8 ♡	v7 ♡	v4 ♡	Valet ♣	v6 ♣

La main présentée dans le tableau ci-dessus vaut donc : ptH = 14 et ptD = 4 (3 points pour la chicane à carreau + 1 point pour le doubleton à trèfle).

1. Dans le fichier `bridge.cpp`, réalisez la fonction `nbreCarteCouleur` qui prend en entrée une `MainJ` et une `CouleurCarte` et qui renvoie le nombre de cartes de la main qui ont la couleur donnée. Complétez les tests de cette fonction. Exécutez pour vérifier (à faire à chaque question, on ne le précisera plus).
2. Réalisez la fonction `evaluePtD` qui prend en entrée une `MainJ` et renvoie son nombre de points de Distribution (ptD).
3. Réalisez la fonction `evaluePtH` qui prend en entrée une `MainJ` et qui renvoie son nombre de points d'honneurs (ptH). (Pour obtenir le nombre de point d'une carte, on utilisera le tableau `ptHCarte`)

4. On souhaite supprimer le tableau `ptHCarte`. Réalisez la fonction `pointHCarte` permettant de le remplacer. Complétez la fonction de tests correspondante.

▸ Exercice 4.(Pour aller plus loin sur les cartes)

Dans cet exercice nous reprenons le fichier de l'exercice précédent. On demande de :

1. Écrire une fonction pour saisir une carte.
2. Écrire une fonction pour saisir une main.
3. Améliorer la fonction précédente en vérifiant qu'il n'y a pas deux fois la même carte dans la main.
4. Écrire une fonction pour afficher une main. On pourra faire deux affichages :
   — Un affichage texte simple (en utilisant par exemple les lettres V, D, R et A pour valet, dame, roi et as et T, K, C, P pour trefle, carreau, coeur, pique).
   — Un affichage en utilisant les caractères unicodes. Vous trouverez un exemple dans le fichier `carteUni.cpp`.
5. Écrire une fonction qui crée un jeu complet et le mélange. Indication : On stockera toutes les cartes dans un vecteur qui représentera le jeu de carte. On mélangera ensuite ce vecteur en répétant un grand nombre de fois l'échange de deux cartes tirées au hasard (voir plus loin). Remarque : ce n'est pas une bonne manière de faire (certains jeux ont plus de chance d'apparaître que d'autres), mais ça ira dans un premier temps.
6. Écrire une fonction qui, ayant mélangé le jeu, distribue les cartes à 4 joueurs.

Comment faire du hasard

Les machines ne savent pas faire de l'aléatoire. Elles ont un comportement déterministe. Du coup, on utilise du pseudo-aléatoire : On calcule une suite mathématique, qui ressemble beaucoup à de l'aléatoire mais qui n'en est pas. Un bon exemple est les décimales du nombre π. En pratique, on utilise des genres de suite récurrente (i.e. de la forme $u(n+1) = f(u(n))$), je vous ai mis un exemple donné par la norme POSIX.1-2001 à la fin du sujet. Bien évidement, si l'on démarre avec la même valeur pour $u(0)$ on obtient toujours la même suite. Cette initialisation s'appelle la «graine aléatoire». Si vous ne l'initialisez pas, le programme prendra toujours la même et donc vous aurez toujours la même suite. Si vous initialisez deux fois avec la même graine vous aurez la même suite.

La fonction `rand()` retourne un tel nombre au hasard. Si vous utilisez la fonction `rand()`, il faut d'abord faire un unique appel à

```
void srand(unsigned int seed);
```

pour initialiser la graine. Cette appel doit être **fait une seule fois, en général au début du main**. Une solution est de l'initialiser avec le nombre de secondes écoulées depuis le 1er janvier 1970 par la commande

```
srand(time(NULL));
```

Voici un exemple

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  using namespace std;
5
6  int main(){
7     srand(time(NULL));
8     cout << "Nombres au hasard : ";
9     for (int i = 0; i<5; i++)
10      cout << rand() % 100 << " ";
11    cout << endl;
12    return 0;
13 }
```

La librairie standard C++ fournit des générateurs aléatoires beaucoup plus versatiles (choix de la distribution...), mais plus complexes d'utilisation.

Enfin, aucun de ces générateurs n'est de qualité suffisante pour les applications cryptographiques : si l'on connaît quelques valeurs, on peut facilement prévoir la suite... Il faut alors passer par des mécanismes beaucoup plus sophistiqués faisant intervenir le monde physique (par exemple, la dernière décimale du temps en nanoseconde entre deux appuis de touche sur le clavier)...

```
1       static unsigned long next = 1;
2
3        /* RAND_MAX assumed to be 32767 */
4        int myrand(void) {
5            next = next * 1103515245 + 12345;
6            return((unsigned)(next/65536) % 32768);
7        }
8
9        void mysrand(unsigned int seed) {
10           next = seed;
11       }
```