Sorbonne Université **M1 Mathématiques**
2019–2020 **MU4MA056 : Programmation en C++**

TP noté 1 : jeu de Mémory

IMPORTANT : à l'issue du TP, chaque groupe envoie par email les fichiers `memory.hpp`, `memory.cpp`, `test.cpp`, `stupid_player.cpp` et `partie.cpp`. **Il est impératif de mettre en commentaire, dans *tous* les fichiers, les nom, prénom et n° d'étudiant de chacun des membres du groupe.**

Nous vous proposons de programmer le jeu de Mémory. Ce jeu consiste en n paires de cartes identiques disposées face cachée sur une table. À chaque tour, le joueur découvre deux cartes cachées : si elles sont différentes, elles sont à nouveau cachées ; si elles sont identiques, elles restent face visible jusqu'à la fin du jeu. Le but est d'exercer sa mémoire et de découvrir toutes les cartes en un minimum de tours.

Description en C++. Nous imposons les classes suivantes à écrire dans un fichier `memory.hpp` :

```
#ifndef MEMORY_CLASS
#define MEMORY_CLASS
struct Card {
        std::string value;
        bool already_found;
};
class Memory {
        protected:
           int nb_types;
           std::vector< Card > cards; // size= 2*nb_types
           int nb_turns;
           int nb_revealed_pairs;
        public:
           void display() const;
           std::pair<std::string,std::string> one_try(int i,int j);
           bool one_player_turn(); /* asks the player two card numbers,
                          display the two values and tells if it is okay; */
};
#endif
```

La structure `Card` contient le nom de la carte (ici un nom d'animal lu dans un fichier) dans le champ `value` et un booléen qui indique si la carte est encore cachée (`false`) ou bien si la paire correspondante a été trouvée (`true`). La classe `Memory` contient les champs suivants :

- un champ `nb_types` qui contient le nombre de paires n dans le jeu,
- un champs `cards` qui contient $2n$ cases : chacune correspond à une carte du jeu,
- un champ `nb_turns` qui indique combien de tours de jeu sont déjà passés,
- un champ `nb_revealed_pairs` qui indique combien de paires ont déjà été trouvées par le joueur (il est donc toujours égal à la moitié du nombres de cartes de `card` avec un champ `already_found` égal à `true`).

Attention : **On veillera à étiqueter `const` toutes les méthodes et les arguments nécessaires.**

1. Recopier ce code dans un fichier `memory.hpp` et ajouter les `include` nécessaires. Préparer également un fichier `memory.cpp` avec les `include` nécessaires.

2. Ajouter un constructeur par défaut qui crée un jeu vide au premier tour de jeu.

3. Ajouter une méthode `add_two_cards(const std::string & S)` qui ajoute deux cartes de même valeur `S` à un jeu existant (le nombre de tours est inchangé). Lors de l'ajout ces cartes sont cachées (leur attribut `already_found` vaut `false`). *Bonus : vous pouvez remplacer* `add_two_cards` *par une surcharge de* `+=` *telle que* `M+=S` *fasse la même chose que* `M.add_two_cards(S)` *où* `M` *est de type* `Memory` *.*

4. Surcharger l'opérateur `<<` qui affiche dans un flux de sortie `std::ostream &` la configuration actuelle de jeu sous la forme :

```
0: value_of_card_0   hidden
1: value_of_card_1   revealed
...
2n-1: value_of_card_2n-1   revealed
nb_turns
nb_revealed_pairs
```

Cela vous permettra par la suite de vérifier les actions de vos différentes méthodes.

5. Écrire un accesseur `is_revealed(i)` au champ `already_found` de la carte numéro i de `cards` .

6. Écrire un accesseur `get_nb_turns()` au champ privé `nb_turns` .

7. Dans un fichier `test.cpp` , construire un jeu vide, lui ajouter 6 cartes avec les valeurs `"chien"` , `"chat"` et `"oiseau"` et l'afficher dans le terminal avec `<<` . Vérifier que vous obtenez un affichage cohérent.

Attention : **tant que vous n'obtenez pas un affichage satisfaisant, il est inutile de continuer.**

8. Ajouter un constructeur qui prend un flux de lecture `std::istream &` vers un fichier structuré de la manière suivante :

```
NB_DE_TYPES_DE_CARTES_N
VALEUR_CARTE_1
VALEUR_CARTE_2
...
VALEUR_CARTE_N
```

Vous trouverez dans `example1.dat` une source d'exemple. Nous vous rappelons que la classe `std::vector` possède une méthode `resize(n)` qui redimensionne à `n` le nombre de cases.

9. Écrire une méthode `random_initialize(std::mt19937 &)` telle que l'appel sur un jeu de memory `M` de

```
M.random_initialize(G);
```

où `G` est un générateur de nombres aléatoires de type `std::mt19937`, met les champs privés `nb_turns` et `nb_revealed_pairs` à zéro et mélange aléatoirement le vecteur `cards`.

Pour cela, nous vous indiquons que la bibliothèque `<algorithm>` contient une fonction de mélange aléatoire `std::shuffle` à 3 arguments : les deux premiers sont des itérateurs sur le début et la fin du conteneur à mélanger et la troisième est une référence sur un générateur aléatoire.

10. Écrire un programme `test.cpp` qui crée un jeu de mémory à partir du fichier `example1.dat`, l'affiche dans le terminal, l'initialise aléatoirement avec la méthode précédente et l'affiche à nouveau dans le terminal avec `<<`. Vous testerez également vos accesseurs dans ce fichier.

11. Écrire une méthode `is_game_over()` qui renvoie vrai ou faux selon que toutes les paires ont été découvertes ou non.

12. Écrire une méthode `restart_game()` qui remet le nombre de tours à 0 et remet toutes les cartes cachées. Pour cette dernière action, nous *exigeons* l'usage de `std::for_each` (qui prend comme argument deux itérateurs vers les début et fin du conteneur et applique le troisième argument (une λ-fonction avec une référence en argument) à chaque case du conteneur) ou `std::transform` (qui prend comme argument deux itérateurs de début et de fin, puis un troisième itérateur de début pour stocker les résultats, et enfin une λ-fonction qui renvoie le résultat d'une transformation souhaitée sur chaque case). Elles sont toutes les deux dans la bibliothèque `<algorithm>`.

13. Écrire le code de la méthode `display()` qui affiche une configuration de jeu sur le terminal de la manière suivante :

```
**** 4 th turn ****
Configuration:
   0: ??????
   1: ??????
   2: animal1
   3: ??????
   4: animal1
   5: ??????
```

où le premier 4 est plus généralement le numéro du tour de jeu et, pour chaque carte i, on a une ligne commençant par `i:` puis soit la valeur si la paire correspondante est déjà découverte, soit six symboles `?` si la carte est encore cachée.

Ajouter également une ligne dans `test.cpp` pour vérifier l'affichage initial du jeu précédemment chargé (on ne doit voir que des `?` et le bon nombre de lignes).

14. Écrire le code de la méthode `one_try(i,j)` qui fonctionne de la manière suivante. Les deux arguments `i` et `j` correspondent aux cartes que l'on souhaite retourner lors d'un tour de jeu. *On supposera que l'on est dans la situation idéale : on aura toujours `i` et `j` différents, que les valeurs données sont toujours entre `0` et `2n-1` et enfin qu'elles correspondent toujours à des cartes qui ne sont pas encore retournées.* Si les cartes ont des valeurs identiques, alors la paire est définitivement retournée (on mettra ainsi à jour le champ `already_revealed` et le champ `nb_revealed_pairs`),

sinon rien n'est fait. Dans tous les cas, une paire constituée des deux valeurs est renvoyé comme résultat.

On rappelle que `std::make_pair(a,b)` *renvoie une paire dont le premier élément est égal à* `a` *et le deuxième est égal à* `b` *.*

15. Écrire un programme complet `stupid_player.cpp` qui étudie, sur 1000 parties basées sur les cartes de `example2.dat` , le nombre de tours nécessaires pour gagner d'un joueur qui jouerait de la manière (stupide) suivante à chaque tour :

1. on génère deux v.a. aléatoires x et y uniformes sur $0, \dots, 2n-1$ (avec $n = 32$ ici),
2. tant que le couple (x, y) n'est pas admissible (i.e. tant que $x = y$ ou bien que x ou y correspond à une carte déjà retournée), on régénère x et y,
3. une fois qu'il est admissible, on retourne les cartes x et y avec `one_try(x,y)` ,
4. on recommence jusqu'à ce que le jeu soit terminé.

(Vous devriez trouver autour de 1000.)

16. Écrire un programme complet `partie.cpp` pour jouer avec un joueur humain. Comme précédemment, il utilisera le fichier de cartes `example2.dat` , il fera appel à `display()` pour afficher la configuration à chaque tour et à `one_try(i,j)` pour jouer les valeurs `i` et `j` demandées au joueur. Vous veillerez à couvrir toutes les situations possibles (un joueur qui entre deux fois le même numéro, des numéros de cartes qui n'existent pas, etc) et afficherez les informations nécessaires au joueur pour progresser dans la partie.

17. *Bonus : écrire librement une classe* `Player_with_small_memory` *qui permette de décrire un joueur qui se souvient exactement des* $2n$ *dernières cartes révélées et adapte sa stratégie. La tester.*