Nom, Prénom : **Numéro d'étudiant :** ␣␣␣␣␣␣␣␣␣␣␣

Université Paris Sud, Licence 1 MPI **Année Universitaire 2019-2020**

Info 111 Cours de Nicolas M. Thiéry **Partiel du lundi 21 octobre 2019 (deux heures)**

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5	Exercice 6	Exercice 7	Total
␣␣,␣␣	␣␣,␣␣	␣␣,␣␣	␣␣,␣␣	␣␣,␣␣	␣␣,␣␣	␣␣,␣␣	␣␣␣,␣

Calculatrices, téléphones mobiles et tout appareil connectable non autorisé doivent être éteints et déposés avec les affaires personnelles de l'étudiant.
Seul document autorisé : une feuille au format A4 avec, au recto, le résumé de la syntaxe C++ de la fiche de TD 2 et, au verso, des notes manuscrites. Pour les étudiants inscrits en Français Langue Étrangère, un dictionnaire est autorisé.
Les exercices sont indépendants les uns des autres ; il n'est pas nécessaire de les faire dans l'ordre ; ceux marqués d'un ♣ sont plus difficiles mais font bien partie du barème sur 20.
Le barème est indicatif.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, mettre un renvoi.

Exercice 1 (Cours (2 points))**.**
Rappeler la syntaxe (1 point) et la sémantique (1 point) de la boucle `for` en C++.

Exercice 2 (Cours (2.5 points))**.**
Quelles sont les trois étapes pour construire un tableau en C++ ?

Illustrer votre réponse avec la construction d'un tableau contenant 42 fois la chaîne de caractères "coucou". Préciser au moyen de commentaires quelles instructions correspondent à quelles étapes.

Exercice 3 (Fonctions simples (2 points))**.**

(1) Écrire une fonction `perimetreRectangle` qui prend en paramètres deux entiers représentant la longueur et la largeur d'un rectangle et renvoie son périmètre.

(2) Utilisation : écrire les lignes de code qui permettent de déclarer et initialiser deux variables avec les valeurs de votre choix pour représenter les dimensions d'un rectangle, puis de stocker son périmètre dans une nouvelle variable. Il est obligatoire d'utiliser un **appel** à votre fonction `perimetreRectangle`.

Exercice 4 (Booléens (1,5 points))**.**

(1) Écrire la fonction dont la documentation et les tests sont donnés ci-dessous :

```
/** Fonction positifs_ou_carre
 *  @param a un entier
 *  @param b un entier
 *  @return true si a et b sont tous deux positifs ou
 *        si a est le carré de b; renvoie false sinon
 **/
```

```
    ASSERT(     positifs_ou_carre(2,7) );
    ASSERT(     positifs_ou_carre(16, -4) );
    ASSERT( not positifs_ou_carre(-8, 2) );
    ASSERT( not positifs_ou_carre(-2, 4) );
```

(2) Si ce n'est pas déjà le cas, ré-écrire votre fonction pour qu'elle ne contienne pas de `if` (ni aucune autre structure de contrôle bien sûr !).

Exercice 5 (Boucles (3 points))**.**

(1) Écrire un fragment de programme d'affichant tous les nombres *non multiples* de 3 compris entre 0 et 101 inclus (c'est-à-dire 1, 2, 4, 5, 7...), en privilégiant la simplicité du code à sa performance.

(2) Écrire une fonction nommée `somme_inverses` qui prend en paramètre un entier m et qui renvoie la somme des inverses des nombres compris entre 1 et m inclus. Par exemple, le résultat pour $m = 3$ sera $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} \approx 1.833333$.

(3) Écrire une fonction nommée `affiche_puissances` qui prend en paramètres un entier m et un entier p, et qui affiche toutes les puissances de m (à partir de m^0) qui sont inférieures ou égales à p si $m \geq 2$ et rien sinon. Par exemple `affiche_puissances(2, 7)` doit afficher 1, 2 et 4, `affiche_puissances(2, 8)` doit afficher 1, 2, 4 et 8, et `affiche_puissances(1, 8)` ne doit rien afficher. Vous devez utiliser des appels à la fonction `puissance` dont la documentation est donnée ci-dessous :

```
/** La fonction puissance
 *  @param x un nombre entier
 *  @param n un nombre entier positif
 *  @return la n-ième puissance x^n de x
 **/
```

(La fonction puissance est considérée déjà définie, vous n'avez pas à écrire son code.)

Exercice 6 (Tableaux (2,5 points))**.**

(1) Écrire une fonction nommée `majore` qui prend en paramètres un tableau d'entiers `t` et un entier `m` et renvoie `true` si `m` est *strictement* plus grand que tous les éléments de `t` et `false` sinon. La fonction devra passer les tests suivants :

```
    ASSERT(     majore({2, 8, 6}, 9) );
    ASSERT( not majore({2, 8, 6}, 3) );
    ASSERT( not majore({2, 8, 6}, 8) );
```

```
bool majore(vector<int> t, int m) {




}
```

(2) ♣ Si ce n'est déjà fait, réécrivez la fonction précédente en utilisant une boucle *for each* : `for (int v:t){...}`.

```
bool majore_foreach(vector<int> t, int m) {




}
```

(3) Écrire une fonction nommée `positionPositif` qui prend en paramètre un tableau d'entiers `t`, et renvoie l'indice du premier nombre *strictement* positif dans `t`, ou -1 si `t` ne contient pas de nombre *strictement* positif. La fonction devra passer les tests suivants :

```
    ASSERT( positionPositif({ 1, 5, 3}) ==  0 );
    ASSERT( positionPositif({-2, 4, 3}) ==  1 );
    ASSERT( positionPositif({-1,-4,-3}) == -1 );
    ASSERT( positionPositif({-1, 0, 2}) ==  2 );
```

Exercice 7 (Jeu d'Éleusis (7 points))**.**

Ce problème est inspiré du jeu d'Éleusis, un jeu de carte à plusieurs. Dans ce jeu, un des joueurs est désigné "Dieu" et doit inventer une règle selon laquelle il acceptera ou refusera des cartes que lui proposent les autres joueurs. Cette règle doit être basée uniquement sur la carte elle-même et l'historique des cartes ayant déjà été acceptées. Les joueurs doivent alors, à force de proposer des cartes et de se les voir accepter ou refuser par "Dieu", trouver la règle inventée par ce dernier.

Dans la version simplifiée de ce problème, nous aurons un jeu de 40 carte sans têtes (*uniquement les cartes de 1 à 10 de chaque couleur*). Nous désignerons dans la suite de ce problème chaque carte par un *identifiant* entier (`int`) unique selon la règle suivante : les carreaux, cœurs, trèfles et piques sont respectivement représentés par les intervalles $[1, 10]$, $[11, 20]$, $[21, 30]$ et $[31, 40]$.

Par exemple :

— 5 représente un 5 de carreau (valeur : 5, couleur : 1) ;
— 11 représente un as de cœur (valeur : 1, couleur : 2) ;
— 30 représente un 10 de trèfle (valeur : 10, couleur : 3).

(1) Implanter les deux fonctions suivantes :

```
/** valeur d'une carte
 *  @param identifiant: un entier entre 1 et 40
 *  @return la valeur de la carte ayant cet identifiant:
            un entier entre 1 et 10
 **/
int valeur_carte(int identifiant) {




}
```

```
/** couleur d'une carte
 *  @param identifiant: un entier entre 1 et 40
 *  @return la couleur de la carte ayant cet identifiant:
            un entier entre 1 et 4
 **/
int couleur_carte(int identifiant) {




}
```

(2) La règle de "Dieu" sera décrite par un prédicat : fonction prenant un identifiant de carte et renvoyant `true` si la carte et acceptée et `false` sinon.

Nous commençons par écrire une règle n'acceptant que les cartes noires (trèfles et piques) dans une fonction `regle_noire`. Écrire deux tests que devront satisfaire cette fonction : un avec une carte qui doit être refusée, et l'autre acceptée :

Écrire la fonction elle-même en vous aidant des fonctions définies dans la question précédente :

(3) On souhaite maintenant écrire des règles plus intéressantes en se basant sur l'*historique des cartes acceptées*. Cet historique sera représenté sous la forme d'un tableau de type `vector<int>`, où chaque cellule est un identifiant de carte ayant été acceptée. Les cellules sont dans l'ordre chronologique : la dernière cellule est la carte acceptée la plus récente.

Nos fonctions de règles prendront donc cet historique de jeu en paramètre en plus de la carte proposée par le joueur. Voici une telle règle :

```
bool regle_mystere(vector<int> historique, int carte) {
    int foo = historique.size();
    for ( int bar = 0; bar < foo ; bar++ ) {
        if ( historique[bar] == carte ) {
            return false;
        }
    }
    return true;
}
```

Écrire deux tests de la fonction : un où elle refuse et un où elle accepte.

Que fait cette règle ?

(4) ♣ Vous allez maintenant coder la boucle de jeu elle-même. Le joueur sera désigné gagnant s'il arrive à poser 20 cartes acceptées *à la suite*. Il est désigné perdant s'il n'a pas atteint cet objectif au bout de 100 tours.

Vous avez à votre disposition les fonctions documentées ci-dessous :

```
/** affiche un tableau
 *  @param t un vecteur d'entiers
 **/
void affiche_tableau(vector<int> T);
```

```
/** demande à l'utilisateur de saisir une carte
 *  @return l'identifiant de la carte choisie (entre 1 et 40).
 **/
int demande_joueur();
```

Écrire une fonction `joue`, qui demande à répétition une carte à l'utilisateur, lui disant si sa carte est acceptée par `regle_mystere` ou pas puis affichant. Si la carte est acceptée, l'historique de la partie est mis à jour. Dans tous les cas, l'historique est ensuite affiché à l'utilisateur. Si le joueur a gagné ou perdu, le programme se termine en affichant l'issue de la partie.

(5) ♣ Écrire une nouvelle règle `regle_avancee` qui impose de jouer une carte noire (trèfle, pique) si la moyenne des valeurs cartes ayant été acceptées est strictement supérieure à 5, et une carte rouge si la moyenne est inférieure ou égale à 5. Dans le cas où aucune carte n'a été jouée, toute carte est acceptée.

(6) ♣ Écrire une règle `regle_diversite` qui maximise la variété des couleurs et des valeurs. Plus précisément, une carte ne pourra être jouée que si sa valeur et sa couleur sont celles qui sont apparues le moins dans l'historique. S'il y a plusieurs possibilités (en cas d'égalité entre couleurs ou valeurs), toutes sont acceptées.

Voici quelques tests que votre fonction devra satisfaire :

```
    ASSERT( not regle_diversite({1, 12}, 22) );
    ASSERT( not regle_diversite({1, 12}, 13) );
    ASSERT( regle_diversite({1, 12}, 23) );
    ASSERT( regle_diversite({1, 23, 14, 35}, 32) );
    ASSERT( not regle_diversite({1, 23, 14, 35}, 31) );
}
```