Nom, prénom, numéro d'étudiant :

Université Paris Sud, Licence MPI, Info 111 **Partiel du 23 octobre 2017 (deux heures)**

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

Calculatrices et autres gadgets électroniques interdits.
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.
Barème indicatif : en gros un point par question.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, mettre un renvoi.

Exercice 1 (Cours)**.**
Rappeler la syntaxe et la sémantique de l'affectation en C++ :

Donnez un exemple d'affectation :

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

Illustrer votre réponse avec la construction d'un tableau de 100 entiers tous égaux à 1 :

Exercice 3 (Fonctions)**.**

(1) Écrire une fonction `difference` qui prend en paramètres deux entiers et renvoie la valeur de la différence entre le plus grand des deux et le plus petit des deux.

(2) Utilisation : écrire les lignes de code qui permettent de déclarer et initialiser deux entiers avec les valeurs de votre choix, puis de stocker la différence du plus grand moins le plus petit dans une nouvelle variable. Il est obligatoire d'utiliser un **appel** à votre fonction `difference`.

Exercice 4 (Booléens)**.**

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

```
/** Fonction unPasLautre
 *  @param a un entier
 *  @param b un entier
 *  @return true si l'un des deux entiers est supérieur ou égal à 10
 *          et l'autre strictement inférieur à 10, false sinon
 **/
```

```
    ASSERT( unPasLautre(15, 9) );
    ASSERT( unPasLautre(7, 23) );
    ASSERT( not unPasLautre(12, 16) );
    ASSERT( not unPasLautre(8, 6) );
```

(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)**.**

(1) Écrire un programme qui affiche tous les multiples de 5 compris entre 0 et 100 inclus. (Rappel : les multiples de 5 sont 0, 5, 10, 15, ...).

(2) Écrire une fonction nommée `afficheCarres` qui prend en paramètre un entier m et qui affiche les carrés de tous les entiers compris entre 0 et m inclus.

(3) On suppose que l'on a à disposition 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
 **/
int puissance(int x, int n) {
```

Écrire une fonction nommée `affichePuissances` qui prend en paramètres un entier m et un entier k, et qui affiche les k-ièmes puissances de tous les entiers compris entre 0 et m inclus. Vous devez utiliser un appel à la fonction puissance.

Exercice 6 (Tableaux)**.**

(1) Écrire une fonction nommée `tousPairs` qui prend en paramètre un tableau d'entiers `t` et renvoie `true` si tous les éléments de `t` sont pairs, et `false` sinon. La fonction devra passer les tests suivants :

```
ASSERT( tousPairs({2,8,6}) );
ASSERT( not tousPairs({2,7,6}) );
```

(2) Écrire une fonction nommée `recherche` qui prend en paramètre un tableau d'entiers `t` ainsi qu'un entier v. Cette fonction doit renvoyer l'indice d'une occurrence de v dans `t` si v est présent dans `t`, et -1 si v n'est pas présent dans `t`. La fonction devra passer les tests suivants :

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

Exercice 7 (Jeu video)**.**
On se propose d'organiser un jeu vidéo dans lequel des personnages avec différentes caractéristiques s'affrontent. Chaque personnage est représenté par un tableau d'entiers de taille 4. La première case du tableau contient son nombre de points de vie, la seconde son attaque, la troisième sa défense, la quatrième son initiative (fréquence de l'attaque).

(1) Implanter la fonction `affiche_personnage` qui prend en paramètre un personnage et affiche ses caractéristiques selon l'exemple ci-dessous.

```
affiche_personnage({100,20,7,4});
```

```
Le nombre de points de vie est de 100
L'attaque est de 20
La défense est de 7
L'initiative est de 4
```

(2) Un personnage est mort si son nombre de points de vie est négatif ou nul. On veut écrire une fonction `est_mort` qui prend en paramètre un personnage et qui renvoie `true` si le personnage est mort et `false` sinon. Écrire quelques tests pour la fonction `est_mort` puis l'implanter.

(3) Quand un personnage A attaque un personnage B, le nombre de points de vie du personnage B diminue de la différence entre l'attaque de A et la défense de B. Bien sûr lorsque B se fait attaquer, son nombre de vie n'augmente pas : si l'attaque est trop faible elle n'a simplement aucun effet. Implanter la fonction `attaque` correspondante, qui vérifie la documentation et les tests suivants :

```
/** attaque
 * @param attaquant un tableau d'entiers de taille 4, du personnage
 *        qui attaque defenseur
 * @param defenseur un tableau d'entiers de taille 4 du personnage
 *        qui subit l'attaque d'attaquant
 * @return le tableau du defenseur après avoir subit l'attaque
 **/
```

```
ASSERT( attaque({100,20,7,4}, {100,13,10,4}) == {90, 13, 10, 4} );
ASSERT( attaque({100,21,7,4}, {100,13,7,4}) == {86, 13, 7, 4} );
```

(4) Un combat simple entre deux personnages est un affrontement où chacun des deux personnages s'attaque tour à tour jusqu'à ce que l'un des deux meurt. L'autre est alors déclaré victorieux. Voici les tests et la documentation de la fonction implantant un combat simple :

```
/** combat_simple
 * @param pge1 un tableau d entiers de taille 4 du personnage
 *        qui attaque le premier
 * @param pge2 un tableau d entiers de taille 4 du personnage
 *        qui attaque le second
 * @return 1 si pge1 a gagné dans un combat simple, et 2 sinon
 **/
```

```
ASSERT( combat_simple({310,34,12,4}, {100,9,5,4}) == 1 );
```

Expliquer ce que fait la fonction mystère suivante :

```
int mystere(vector<int> blabla, vector<int> bloblo){
    int schtroumpf = 3;
    if (blabla[schtroumpf] > bloblo[schtroumpf]){
        int miaou = blabla[schtroumpf] - bloblo[schtroumpf];
        for(int hiphop = 0; hiphop < miaou; hiphop++){
            blabla = attaque(bloblo, blabla);
        }
    } else {
        int glouglou = bloblo[schtroumpf] - blabla[schtroumpf];
        for(int hophip = 0; hophip < glouglou; hophip++){
            bloblo = attaque(blabla, bloblo);
        }
    }
    return combat_simple(blabla, bloblo);
}
```

(5) ♣ Implanter la fonction `combat_simple`. On veillera à utiliser la fonction `est_mort`.

(6) ♣ Une variante plus complexe de combat prend en compte l'initiative. Imaginons par exemple que le combattant A ait 7 d'initiative, contre 10 pour le combattant B. Cela signifie que A attaque aux secondes $7, 14, 21, 28, 35, \ldots$ alors que B attaque aux secondes $10, 20, 30, 40, \ldots$. Ainsi L'ordre des attaques est le suivant : A à 7 secondes, puis B à 10 secondes, puis A (14s), B (20s), A (21s), encore A (28s), puis B (30s), etc.

En cas d'égalité entre deux moments d'attaque (ici à la seconde 70 par exemple) on pourra indifféremment faire attaquer l'un avant l'autre.

Écrire des tests pour la fonction `combat_initiative` correspondante qui prend en paramètres deux personnages et renvoie 1 si le premier a gagné, et 2 si c'est l'autre, puis implanter cette fonction .