Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2021-2022**
Examen du 13 décembre 2022 (deux heures)

Calculatrices, téléphones mobiles et tout appareil électronique non autorisé doivent être éteints et déposés avec vos affaires personnelles.
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 partie du barème sur 100. Le barème est indicatif et sujet à ajustements.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, demander un intercalaire et mettre un renvoi.
Les enseignants collecteront votre copie à votre place.

Exercice 1 (Cours : compilation séparée (10 points))**.**
On souhaite implanter une bibliothèque `bla`, utilisée par un programme `programme.cpp`.

(1) Lister ci-dessous les fichiers requis pour la bibliothèque :
- `bla.h` : fichier d'entête
- `bla.cpp` : bibliothèque
- `bla-test.cpp` : fichier de test

(2) Indiquer le rôle de chacun des extraits de code ci-dessous et le ou les fichiers le contenant :
- Rôle : documentation de la bibliothèque, fichier(s) : bla.h

```
/** Calcule ...
 **/
```

- Rôle : entêtes de la bibliothèque, fichier(s) : bla.h

```
int bla(int a, int b);
```

- Rôle : implémentation de la bibliothèque, fichier(s) : bla.cpp

```
int bla(int a, int b) {
    return a + b;
}
```

- Rôle : tests de la bibliothèque, fichier(s) : bla-test.cpp

```
    CHECK( bla(1, 2) == 3 );
```

- Rôle : inclusion d'entête, fichier(s) : bla.cpp, bla-test.cpp, programme.cpp

```
#include "bla.h"
```

- Rôle : utilisation de la bibliothèque, fichier(s) : programme.cpp

```
    int c = bla(3, 4);
```

(3) Donner les commandes pour compiler la bibliothèque `bla` et le `programme` séparément puis les lier entre eux :

```
g++ -c bla.cpp
g++ -c programme.cpp
g++ bla.o programme.o -o programme
```

Exercice 2 (Plus grand commun diviseur et l'indicatrice d'Euler : fonctions, boucles (20 points))**.** Le Plus Grand Commun Diviseur (pgcd) de deux nombres entiers x, y non nuls est définie comme le plus grand entier qui les divise simultanément. Par exemple :

$$\text{PGCD}(3,5) = 1,\ \text{PGCD}(2,6) = 2,\ \text{PGCD}(15,10) = 5.$$

Dans cet exercice, vous êtes autorisé à utiliser sans les définir les fonctions `min` et `max`, qui reçoivent deux deux entiers et renvoient leur minimum et leur maximum respectivement.

(1) Implanter en utilisant des boucles la fonction `pgcd`, dont la documentation et l'entête sont donnés ci-dessous :
Indications : On rappelle que pour tous entiers a et b, le reste de la division euclidienne de a par b est donné en C++ par `a%b`. Ce reste est nul si et seulement si a est divisible par b.

```
/** Calcul le plus grand commun diviseur de deux entiers
 * @param x: un entier
 * @param y: un entier
 * @return le pgcd de x et y
 **/
int pgcd(int x, int y) {
    int m = min(x, y);
    int diviseur;
    for (int d = 0; d <= m; d++) {
        if (x % d == 0 and y % d == 0) {
            diviseur = d;
        }
    }
    return diviseur;
}
```

(2) On peut montrer que le pgcd a une définition récursive donnée par :

$$\text{PGCD}(x,y) = \begin{cases} \text{PGCD}(x-y,y) & \text{si } x > y \\ \text{PGCD}(x,y-x) & \text{si } y > x \\ x \text{ si } & x = y \end{cases}$$

Écrire une fonction qui utilise cette récursion pour calculer le pgcd :

```
/** Calcul le plus grand commun diviseur de deux entiers de façon recursive
 * @param x: un entier
 * @param y: un entier
 * @return le pgcd de x et y
 **/
int pgcdRecursif(int x, int y) {
    if (x == y) {
        return x;
    } else if (x > y) {
        return pgcdRecursif(x - y, y);
    } else {
        return pgcdRecursif(x, y - x);
    }}
```

(3) Il existe une autre définition récursive donnée par :

$$\text{PGCD}(x,y) = \begin{cases} \max(x,y) \text{ si } \min(x,y) = 0 \\ \text{PGCD}(\min(x,y), \max(x,y)\% \min(x,y)) \text{ sinon} \end{cases}$$

Écrire une fonction qui utilise cette définition récursive pour calculer le pgcd.

```c
/** Calcul le plus grand commun diviseur de deux entiers de façon recursive
 * @param x: un entier
 * @param y: un entier
 * @return le pgcd de x et y
 **/
int pgcdRecursif2(int x, int y) {
    int minimum = min(x, y);
    int maximum = max(x, y);
    if (minimum == 0) {
        return maximum;
    } else {
        return pgcdRecursif(minimum, maximum % minimum);
    }
}
```

(4) La fonction indicatrice d'Euler φ d'un entier x, est définie comme la quantité des nombres natureles y, plus petits que x, telles que $\text{PGCD}(x,y) = 1$. En d'autres termes :

$$\varphi(x) = \#\{y < x \text{ telles que } \text{PGCD}(x,y) = 1\}.$$

Par exemple $\varphi(2) = 1,\ \varphi(9) = 6,\ \varphi(17) = 16$.

Utiliser la fonction pgcd pour écrire une fonction qui calcule l'indicatrice d'Euler d'un entier donné :

```c
/** Calcul l'indicatrice d'Euler d'un entier
 * @param x: un entier
 * @return l'indicatrice d'Euler de x
 **/
int indicatrice(int x) {
    int count = 0;
    for (int y = 1; y < x; y++) {
        if (pgcd(x, y) == 1) {
            count++;
        }
    }
    return count;
}
```

Exercice 3 (Fichiers, tableaux (30 points))**.**

Une équipe de foot veut acheter des maillots pour ses joueurs. Les tailles des maillots vont de 0 à 4. On dispose d'un fichier `maillots-joueurs.txt` dont chaque ligne contient : le nom d'un joueur (en un mot) suivi de sa taille (de 0 à 4). Voici un exemple de fichier `maillots-joueurs.txt` :

```
Mbappé 2
Pogba 4
Griezmann 1
Lloris 4
Kanté 0
Benzema 3
Pavard 0
Giroud 4
```

(1) Considérons la fonction `mystere` suivante :

```
vector<string> mystere(string abc) {
    ifstream zut;
    zut.open(abc);
    string hop;
    int plop;
    vector<string> toto;
    toto = vector<string>(0);
    while ( zut >> hop and zut >> plop ) {
        toto.push_back(hop);
    }
    zut.close();
    return toto;
}
```

Deviner ce que cette fonction est sensée faire, proposer des noms informatifs pour elle-même et pour ses variables :

— `mystere` : `nomsJoueurs`
— `abc` : `nomFichier`
— `zut` : `fichier`
— `hop` : `nom`
— `plop` : `taille`
— `toto` : `noms`

et écrire sa documentation :

```
/** Lit les noms des joueurs depuis un fichier
  * @param le nom d'un fichier qui contient une liste des joueurs
  *   et leur taille de maillots, au format <nom taille>
  * @return une liste des noms des joueurs
**/
```

(2) Écrire un test pour la fonction de la question précédente. On pourra s'inspirer des tests écrits pour les questions suivantes.

```
    CHECK( nomsJoueurs("maillots-joueurs.txt") ==
            vector<string>({"Mbappé", "Pogba", "Griezmann", "Lloris", "Kanté",
                 "Benzema", "Pavard", "Giroud",}) );
```

(3) Implanter la fonction dont le test, la documentation et l'en-tête sont donnés ci-dessous :

```
    CHECK( nombreParTaille("maillots-joueurs.txt") ==
        vector<int>({2, 1, 1, 1, 3}) );
```

```
/** Compte le nombre de maillots pour les joueurs par taille.
  *   Les tailles sont entre 1 et 5
  * @param le nom d'un fichier qui contient une liste des joueurs
  *   et leur taille de maillots, au format <nom taille>
  * @return un tableau t tel que t[i] est le nombre de maillots de
  *   taille i dans le fichier
**/
vector<int> nombreParTaille(string nomFichier){

    ifstream fichier;
    fichier.open(nomFichier);
    string nom;
    int taille;
    vector<int> liste_taille;
    vector<int> resultat (5, 0);
    liste_taille = vector<int>(0);

    while (fichier >> nom >> taille) {
        liste_taille.push_back(taille);
    }
    fichier.close();

    for (int i=0; i<liste_taille.size(); ++i) {
        resultat[liste_taille[i]] += 1;
    }
    return resultat;

}
```

(4) Implanter une fonction dont le test, la documentation et l'en-tête sont donnés ci-dessous :

```
    CHECK( nomsParTaille("maillots-joueurs.txt") ==
            vector<vector<string> >({{"Kanté", "Pavard"},
                                     {"Griezmann"},
                                     {"Mbappé"},
                                     {"Benzema"},
                                     {"Pogba", "Lloris", "Giroud"}}));
```

```
/** Donne la liste des noms de joueurs pour chaque taille de maillot
  * @param le nom d'un fichier qui contient une liste des joueurs
  *   et leur taille de maillots, au format <nom taille>
  * @return un tableau à deux dimensions t tel que t[i] contient
  *   la liste des joueurs qui portent un maillot de taille i
**/
vector<vector<string> > nomsParTaille(string nomFichier){

    ifstream fichier;
    fichier.open(nomFichier);
    string nom;
    int taille;
    vector<vector<string> > resultat;

    // allocation du tableau
    resultat = vector<vector<string> > (5);
    // note: Les sous-tableaux sont alloués de taille 0
    // initialisation
    while (fichier >> nom >> taille) {
        resultat[taille].push_back(nom);
    }
    fichier.close();
    return resultat;

}
```

Exercice 4 (Carré magique (20 points))**.**

Un *carré magique* est une grille carrée contenant des entiers telle que la somme de chaque ligne, de chaque colonne et de chaque diagonale ait la même valeur. Un carré magique de n lignes est dit *normal* s'il contient chaque entier compris entre 1 et n^2 exactement une fois. Par exemple, le tableau suivant est un carré magique normal :

6	7	2
1	5	9
8	3	4

Dans la suite de l'exercice, il s'agit d'écrire étape par étape les fonctions utiles pour tester si un tableau bidimensionnel est un carré magique normal. Vous pouvez utiliser les fonctions des étapes précédentes, même si vous ne les avez pas implantées. Pour alléger les notations, on définit un raccourci `Grille` pour les tableaux bidimensionnels d'entiers :

```
typedef vector<vector<int>>  Grille;
```

Dans les tests, on pourra supposer que l'on a à disposition la variable suivante :

```
Grille carreMagique = { { 6, 7, 2}, {1, 5, 9}, {8, 3, 4} };
```

(1) Définir deux autres variables contenant des tableaux d'entiers à deux dimensions qui ne sont pas des carrés magiques.

Correction :

```
Grille carre = { { 6, 7, 2}, {1, 6, 9}, {8, 3, 4} };
Grille rectangle = { { 6, 7}, {1, 6}, {8, 3} };
```

(2) Spécifier (par une documentation) et implanter une fonction `estCarre` qui teste si une grille donnée en paramètre est une grille carrée (c'est-à-dire un tableau dont toutes les lignes et les colonnes ont la même longueur).

Correction :

```
bool estCarre(Grille g) {
    for ( int i = 0 ; i < g.size() ; i++ ) {
        if (g[i].size()  != g.size()) {
            return false;
        }
    }
    return true;
}
```

(3) Proposer des tests pour cette fonction (en utilisant `CHECK`).

Correction :

```
void estCarreTest() {
    CHECK( estCarre(carreMagique) );
    CHECK( estCarre(carre) );
    CHECK( not estCarre(rectangle) );
}
```

(4) Implanter une fonction `sommeLigne` qui prend en paramètres un entier l et une grille et qui renvoie la somme des éléments de la l-ième ligne de la grille. Par exemple, l'appel sur l'exemple `sommeLigne(carreMagique, 1)` renvoie $1 + 5 + 9 = 15$. Le cas où l'utilisateur demanderait un l non valide ($l <= 0$ ou $l >= n$) doit être traité.

Dans la suite, on supposera que l'on dispose aussi d'une fonction `sommeColonne` analogue.

Correction :

```
int sommeLigne(Grille g, int l) {
    int somme = 0;
    for (int j = 0 ; j < g.size() ; j++)
        somme = somme + g[l][j];
    return somme;
}
```

(5) Spécifier et implanter une fonction `sommeDiagonaleMajeure` qui prend en paramètre une grille carrée et renvoie la somme des éléments de la diagonale majeure de la grille (la *diagonale majeure* est celle qui commence en haut à gauche et termine en bas à droite ; dans l'exemple ci-dessus, elle contient 6, 5 et 4).

Correction :

```
/** renvoie la somme des éléments de la diagonale majeure d'une grille
 * @param g de type Grille
 * @return la somme des éléments de la diagonale majeure de g
 **/
int sommeDiagonaleMajeure(Grille g) {
    int somme = 0;
    for (int i = 0 ; i < g.size() ; i++)
        somme = somme + g[i][i];
    return somme;
}
```

Exercice 5 (Carré magique (suite) (20 points))**.**

(1) On suppose que l'on dispose d'une fonction `sommeDiagonaleMineure` qui prend en paramètre une grille carrée et renvoie la somme des éléments de la diagonale mineure de la grille (la *diagonale mineure* est celle qui commence en bas à gauche et termine en haut à droite ; dans l'exemple ci-dessus, elle contient 8, 5 et 2). Écrire un test pour la fonction `sommeDiagonaleMineure` (ne pas écrire la fonction, seulement le test).

Correction :

```
    CHECK( sommeDiagonaleMineure(carreMagique) == 15 );
```

(2) Implanter une fonction `estCarreMagique` qui prend en paramètre une grille, et renvoie `true` s'il s'agit d'un carré magique (pas forcément normal), `false` sinon.

Correction :

```
bool estCarreMagique(Grille g) {
    int somme = sommeLigne(g, 0);
    if ( !estCarre(g))
        return false;
    for (int i = 0 ; i < g.size() ; i++)
        if (somme != sommeLigne(g, i) || somme != sommeColonne(g, i))
            return false;
    if (somme != sommeDiagonaleMajeure(g) || somme != sommeDiagonaleMineure(g))
        return false;

    return true;
}
```

(3) ♣ Implanter une fonction `histogramme` qui prend en paramètre une grille `g` et qui renvoie son histogramme, c'est-à-dire un tableau `H` de n^2 entiers (avec n le nombre de lignes de `g`) tel que pour tout `v` entre 1 et n^2, `H[v-1]` contient le nombre d'occurrences de la valeur `v` dans la grille `g`.

Correction :

```
vector <int> histogramme(Grille g) {
    int n = g.size();
    vector <int> H(n*n);
    for (int i = 0 ; i < n*n ; i++)
        H[i]=0;
    for (int i = 0 ; i < n ; i++)
        for (int j = 0 ; j < g[i].size() ; j++)
            H[g[i][j]-1] += 1;
    return H;
}
```

(4) ♣ Implanter une fonction `estCarreMagiqueNormal` qui prend en paramètre une grille, et renvoie `true` s'il s'agit d'un carré magique normal, `false` sinon.
Indication : on pourra par exemple construire l'histogramme de la grille.

Correction :

```
bool estCarreMagiqueNormal(Grille g) {
    vector <int> H;
    if ( !estCarreMagique(g))
        return false;
    H = histogramme(g);
    for (int i = 0 ; i < H.size() ; i++)
        if (H[i] !=1) return false;
    return true;
}
```