TD 6 : Fonctions, tableaux, tableaux à deux dimensions

Exercice 1 (Échauffement)**.**
Écrire des fragments de programme d'une ou deux lignes permettant d'afficher

(1) le troisième élément d'un tableau `t`.

(2) le troisième élément de la deuxième ligne d'un tableau `t` à deux dimensions.

(3) les élements de la troisième ligne d'un tableau `t` à deux dimensions.

(4) les éléments d'un tableau dans l'ordre inverse des indices. Par exemple, pour le tableau `t = {14, 32, 29, 41}`, l'affichage sera `41 29 32 14`.

(5) la table de multiplication, sous la forme :

1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 ···
···

Exercice 2.

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

```
/** Recherche d'un élément dans un tableau
 * @param t un tableau d'entiers
 * @param v un entier
 * @return l'index de la première occurence de v dans t,
 *         ou -1 si v n'est pas dans t
 **/
int recherche(vector<int> t, int v) {
```

```
    vector<int> t = { 1, 3, 2, 4, 3, -1, 2 };
    ASSERT( recherche(t, 3) == 1 );
    ASSERT( recherche(t, 4) == 3 );
    ASSERT( recherche(t, 5) == -1 );
```

(2) Adapter cette fonction et ses tests pour obtenir l'indice de la *dernière* occurrence.

(3) Écrire une fonction qui renvoie le nombre d'occurences d'un entier dans le tableau `t`.

(4) ♣ Est-il possible d'utiliser une boucle `for each` ?

Exercice 3 (Tableaux à deux dimensions)**.**
Pour chaque item suivant, spécifier et implanter une fonction qui

(1) renvoie le nombre de lignes d'un tableau à deux dimensions.

(2) renvoie le nombre de colonnes d'un tableau à deux dimensions .

(3) affiche les éléments d'un tableau à deux dimensions .

(4) teste si un tableau à deux dimensions contient un élément `x`.

(5) compte le nombre d'éléments supérieurs ou égaux à 10 dans un tableau à deux dimensions.

(6) teste si un tableau $n \times n$ est symétrique, *i.e.* si $T_{i,j} = T_{j,i}$ pour tous i et j.

(7) ♣ calcule la somme de deux matrices.

♣ Utiliser la boucle `for each` chaque fois que c'est possible dans les fonctions ci-dessus.

Exercice ♣ 4 (Fonction mystère).

```
vector<int> mystere(vector<int> foo, vector<int> bar) {
    vector<int> hop = vector<int>(foo.size() + bar.size());
    int j = 0;
    for ( int i = 0; i < foo.size(); i++ ) {
        while ( j < bar.size() and bar[j] <= foo[i] ) {
            hop[i+j] = bar[j];
            j++;
        }
        hop[i+j] = foo[i];
    }
    return hop;
}
```

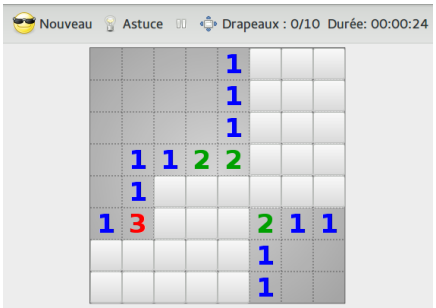
```
    ASSERT(mystere({1,3,5,9,12}, {4,5,8}) == vector<int>({1,3,4,5,5,8,9,12}));
```

(1) Exécuter pas à pas la fonction `mystere` pour les valeurs des paramètres donnés dans le test. Le test passe-t'il ?

(2) Deviner ce que cette fonction est censée faire. Donner des noms informatifs à la fonction et ses variables. Écrire la documentation en précisant bien les prérequis.

(3) Proposer quelques tests supplémentaires pour vérifier tous les cas particuliers.

L'objectif de l'exercice suivant est de réaliser une version simple du jeu du « démineur ». Il s'agit d'un jeu de réflexion dont le but est de localiser des mines cachées dans un champ virtuel avec pour seule indication le nombre de mines dans les zones adjacentes.



Plus précisément, le champ consiste en une grille rectangulaire dont chaque case contient ou non une mine. Au départ, le contenu de chaque case est masqué. À chaque étape, l'utilisateur peut :

- Démasquer le contenu d'une case ; s'il y a une mine, "BOUM !", il a perdu. Sinon, le nombre de cases adjacentes (y compris en diagonale) contenant une mine est affiché.
- Marquer une case, s'il pense qu'elle contient une mine.

L'utilisateur a gagné lorsqu'il a démasqué toutes les cases ne contenant pas de mine.

Exercice ♣ 5 (Le jeu du démineur).

Pour représenter en mémoire l'état interne de la grille, on utilisera un tableau à deux dimensions de caractères (type `vector<vector<char>>`). On utilisera les conventions suivantes pour représenter l'état d'une case :

- 'm' : présence d'une mine, 'M' : présence d'une mine, case marquée
- 'o' : absence de mine
- 'O' : absence de mine, case marquée, ' ' : absence de mine, case démasquée

(1) Implanter une fonction permettant de compter le nombre total de mines (marquées ou pas) dans une grille.

(2) Implanter une fonction permettant de tirer au hasard une grille initiale. On supposera fournie une fonction `bool boolAleatoire()` renvoyant un booléen tiré au hasard.

(3) Implanter une fonction permettant de tester si une grille est gagnante.

(4) Implanter une fonction permettant de compter le nombre de mines dans les cases adjacentes à une case donnée d'une grille.

(5) Implanter une fonction permettant de renvoyer une chaîne de caractères représentant la grille telle que doit la voir le joueur.