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`.

Correction

```
/** Affichage du 3ème élément d'un tableau **/
vector<int> tab1= { 1,  2, 3, 5 };
cout << tab1[2] << endl;
```

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

Correction

```
/** Affichage du 3ème élément de la deuxième ligne d'un tableau à deux dimensions **/
vector<vector<int>> tab2 = { { 1,  2, 3 },
                             { 4, 11, 6 },
                             { 9, 12, 7 }};
cout << tab2[1][2] << endl;
```

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

Correction

```
/** Affichage des éléments de la troisième ligne d'un tableau à deux dimensions **/

for ( int j=0; j < tab2[2].size(); j++ ) {
            cout << tab2[2][j] << " ";
}
        cout << endl;
```

(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`.

Correction

```
/** Affichage des éléments d'un tableau dans l'ordre décroissant des indices **/

for ( int i=tab1.size()-1; i >= 0; i-- ) {
            cout << tab1[i] << " ";
}
        cout << endl;
```

(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 );
```

Correction

```
    for ( int i = 0; i < t.size(); i++ ) {
        if ( t[i] == v ) {
            return i;
        }
    }
    return -1;
}
```

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

Correction

```
/** Recherche d'un élément dans un tableau
 * @param t un tableau d'entiers
 * @param v un entier
 * @return l'index de la dernière occurence de v dans t, ou -1
 **/
int rechercheDernier(vector<int> t, int v) {
    int index = -1;
    for ( int i = 0; i < t.size(); i++ ) {
        if ( t[i] == v ) {
            index = i;
        }
    }
    return index;
}

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

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

Correction

```
int nbreOccurence(vector<int> t, int v) {
    int nbre = 0;
    for ( int i = 0; i < t.size(); i++ ) {
        if ( t[i] == v ) {
            nbre = nbre+1;
        }
    }
    return nbre;
}

int nbreOccurenceTest() {
    vector<int> t = { 1, 3, 2, 4, 3, 2, 2 };
    ASSERT( nbreOccurence(t, 3) == 2 );
    ASSERT( nbreOccurence(t, 4) == 1 );
    ASSERT( nbreOccurence(t, 2) == 3 );
}
```

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

Correction Non, car on a besoin de connaître les positions et pas seulement les valeurs.

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 .

Correction

```
/** Affiche un tableau d'entiers à deux dimensions
 * @param t un tableau d'entiers à deux dimensions
 **/
void affiche(vector<vector<int>> t) {
    for ( int i=0; i < t.size(); i++ ) {
        for ( int j=0; j < t[i].size(); j++ ) {
            cout << t[i][j] << " ";
        }
        cout << endl;
    }
}
```

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

Correction

```
vector<vector<int> > tabCarre       = { { 1,  2, 3 },
                                        { 4, 11, 6 },
                                        { 9, 12, 7 }};
vector<vector<int> > tabRectangulaire={ { 1,  2, 3 },
                                        { 9, 12, 7 }};
vector<vector<int> > tabBizarre     = { { 1,  2, 3 },
```

```
                                        { 4,  5 },
                                        { 6,  7, 8, 10 }};
vector<vector<int> > tabSymetrique = { { 1,  2, 3 },
                                        { 2, 11, 4 },
                                        { 3,  4, 0 }};

/** NombreDeLignes
 * @param t un tableau d'entiers à deux dimensions
 * @return un entier: le nombre de lignes de t
```

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

Correction

```
int nombreDeLignes(vector<vector<int>> t) { //
    return t.size();
}

void nombreDeLignesTest() {
    ASSERT( nombreDeLignes(tabVide) == 0 );
    ASSERT( nombreDeLignes(tabCarre) == 3 );
    ASSERT( nombreDeLignes(tabRectangulaire) == 2 );
    ASSERT( nombreDeLignes(tabBizarre) == 3 );
}

/** NombreDeColonnes
 * @param t un tableau d'entiers à deux dimensions
 * @return un entier: le nombre de colonnes de t
 **/
```

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

Correction

```
    if (t.size() == 0)
        return 0;
    else
        return t[0].size();
}

void nombreDeColonnesTest() { //
    ASSERT( nombreDeLignes(tabVide) == 0 );
    ASSERT( nombreDeLignes(tabCarre) == 3 );
    ASSERT( nombreDeLignes(tabRectangulaire) == 3 );
    ASSERT( nombreDeLignes(tabBizarre) == 3 );
}
```

(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.

Correction

(1) Oui, le test passe.

(2) Cette fonction fusionne deux tableaux triés dans l'ordre croissant en un seul tableau trié dans l'ordre croissant, à condition que le maximum se trouve dans le premier tableau.

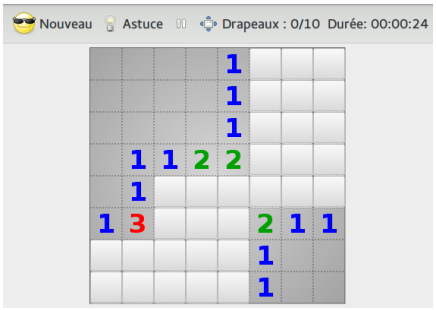
```
/** Fusionne deux tableaux triés en ordre croissant en un seul tableau
*  trié en ordre croissant
* @param tab1 le premier tableau trié
* @param tab2 le deuxième tableau trié, la valeur la plus grande
* de tab2 doit être inférieure à la valeur la plus grande de tab1
* @return le tableau trié en ordre croissant qui contient tous les
*         éléments des deux tableaux donnés en entrée
**/

vector<int> fusionCroissante(vector<int> tab1, vector<int> tab2) {
    vector<int> tabRes = vector<int>(tab1.size() + tab2.size());
    int j = 0;
    for ( int i = 0; i < tab1.size(); i++ ) {
        while ( j < tab2.size() and tab2[j] <= tab1[i] ) {
            tabRes[i+j] = tab2[j];
            j++;
        }
        tabRes[i+j] = tab1[i];
    }
    return tabRes;
}
```

(3) nombre de comparaisons est entre (n, n+m]

(4) Tests :

```
\lstinputlisting[firstline=30,lastline=37]{mystere-correction.cpp}
```



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.

Correction (1) On va parcourir la grille, représentée par un tableau 2D :

```
/** Compte le nombre de mines dans une grille
 * @param grille un tableau 2D de caractères (vector<vector<char> >)
 * @return le nombre de mines dans grille
 **/
int nombreMine(vector<vector<char> > grille) {
    int compteur = 0;
    for(int i=0; i<grille.size(); i++)
        for(int j=0; j<grille[i].size(); j++)
            if (grille[i][j] == 'm' or grille[i][j] == 'M')
                compteur++;
    return compteur;
}
```

(2) On va générer une grille avec une taille donnée :

```
/** Construit une grille initiale
 * @param n un entier positif
```

```
 * @param m un entier positif
 * @return un tableau de caractères de n lignes et m colonnes rempli
 *  aléatoirement et ne contenant que des 'm' ou 'o'
 **/
vector<vector<char> > grilleInitiale(int n, int m) {
        // Déclaration
        vector<vector<char> > result;
        // Allocation
        result = vector<vector<char> >(n);
        // Allocation des sous-tableaux
        for (int i = 0; i < n; i++)
                result[i] = vector<char>(m);
        // Remplit le tableau
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                        if (boolAleatoire())
                                result[i][j] = 'm';
                        else
                                result[i][j] = 'o';
                }
        }
        return result;
}
```

(3) On gagne la partie lorsque toutes les cases non minées sont démasquées.

```
/** Teste si une grille est gagnante
 * @param grille un tableau 2D de caractères (vector<vector<char> >)
 * @return un booléen vrai si grille est une grille gagnante, ie si
 *  toutes les cases qui ne sont pas des mines ont été démasquées
 **/
bool grilleEstGagnante(vector<vector<char> > grille) {
    for (int i=0; i < grille.size(); i++) {
        for (int j=0; j < grille[i].size(); j++) {
            if (grille[i][j]=='O' or grille[i][j]=='o') {
                return false;
            }
        }
    }
    return true;
}
```

(4) Il suffit de parcourir les 8 cases adjacentes et de compter le nombre de M et m :

```
/** Renvoie le nombre de mines voisines à ième ligne, jème colonne
 * @param grille un tableau 2D de caractères (vector<vector<char> >)
 * @param i un entier décrivant une ligne de grille
 * @param j un entier décrivant une colonne de grille
 * @return un entier entre 0 et 8 comptant le nombre de mines
 *  adjacentes à grille[i][j]
 **/
int minesVoisines(vector<vector<char> > grille, int i, int j) {
```

```
    int resultat = 0;
    int n = grille.size();
    int m = grille[0].size();
    for (int i1=max(i-1,0) ; i1 <= min(i+1,n-1); i1++ ) {
        for (int j1=max(j-1,0); j1 <= min(j+1,m-1); j1++ ) {
            if (grille[i1][j1] == 'm' or grille[i1][j1] == 'M') {
                resultat++;
            }
        }
    }
    if (grille[i][j] == 'm' or grille[i][j] == 'M')
        resultat--;
    return resultat;
}
```

(5) Afficher le tableau pour le joueur :

```
/** Affiche une grille
 * @param grille un tableau 2D de caractères (vector<vector<char> >)
 * @return une chaîne de caractères (string) décrivant la grille de
 *  gauche à droite et de bas en haut, un retour à la ligne séparant
 *  chaque ligne de grille
 **/
string dessinGrille(vector<vector<char> > grille) {
    string resultat;
    for (int i=0; i < grille.size(); i++) {
        for (int j=0; j < grille[i].size(); j++) {
            char c = grille[i][j];
            if (c == 'M' or c == 'O')
                resultat = resultat + "M";
            else if (c == ' ')
                resultat += minesVoisines(grille, i, j) + '0';
            else
                resultat = resultat + " ";
        }
        resultat = resultat + "\n";
    }
    return resultat;
}
```