TD 7 : Tableaux à deux dimensions (tableaux 2D)

Exercice 1 (Échauffement)**.**
On considère le tableau d'entiers à deux dimensions suivant :

```
vector<vector<int>> Tab2d = { {7,-1,4,3}, {15,11,17,12}, {20,34,42,25} };
```

(1) Quelles sont les valeurs `Tab2d[0][0]`, `Tab2d[0][1]`, `Tab2d[2][3]` ?

(2) Donner le nombre de lignes et de colonnes du tableau `Tab2d`.

Correction : Les valeurs sont : 7, -1, 25. Le tableau a trois lignes et quatre colonnes.

Exercice 2 (Déclaration, allocation et initialisation de tableau 2D)**.**
Écrire un programme qui crée un tableau 2D de L lignes et C colonnes et qui l'initialise par un entier v.

Correction :

```
// Programme tableau2DInitialise
// Entrée: le nombre de lignes L, le nombre de colonnes C et l'entier v
// pour initialiser le tableau
// Sortie: le tableau 2D d'entiers

vector<vector<int>> Tab2d;
Tab2d = vector<vector<int>>(L);
for ( int i=0; i < L; i++ ) {
    Tab2d[i] = vector<int>(C);
}
for ( int i=0; i<L; i++ ) {
    for ( int j=0; j<C; j++ ) {
        Tab2d[i][j] = v;
    }
}
```

Exercice 3 (Opérations sur tableaux à deux dimensions)**.**
Dans tout cet exercice, `t` est un tableau d'entiers à deux dimensions. Pour chaque opération ci-dessous, spécifier et implanter une fonction qui prend `t` en paramètre et la réalise :

(1) renvoyer le nombre de lignes de `t`.

Correction :

```
/** renvoie le nombre de lignes d'un tableau 2D d'entiers
 * @param t le tableau 2D d'entiers
 * @return le nombre de lignes du tableau t
 **/
int nombreDeLignes(vector<vector<int>> t) { // Correction
    return t.size();
}
```

(2) renvoyer le nombre de colonnes de `t` (supposé rectangulaire).

Correction :

```
/** renvoie le nombre de colonnes d'un tableau 2D d'entiers
 * @param t le tableau 2D d'entiers
 * @return le nombre de colonnes du tableau t
 **/
int nombreDeColonnes(vector<vector<int>> t) { // Correction
    if (t.size() == 0) {
        return 0;
    } else {
        return t[0].size();
    }
}
```

(3) afficher tous les éléments de la ligne d'indice ℓ de `t`.

Correction :

```
/** affiche tous les elements de la ligne l de t.
 * @param t le tableau 2D d'entiers
 * @param l un entier
 **/
void afficheLigne(vector<vector<int>> t, int l) { // Correction
    for (int i=0; i < t[l].size(); i++) {
        cout << t[l][i] << ";" ;
    }
    cout << endl;
}
```

(4) afficher tous les éléments de la colonne d'indice c de `t` (supposé rectangulaire).

Correction :

```
/** affiche tous les elements de la colonne c de t.
 * @param t le tableau 2D d'entiers
 * @param c un entier
 **/
void afficheColonne(vector<vector<int>> t, int c) { // Correction
    for (int i = 0; i < t.size(); i++) {
        cout << t[i][c] << ";" ;
    }
    cout << endl;
}
```

(5) afficher tous les éléments de la diagonale de `t` (supposé carré).

Correction :

```
/** affiche tous les éléments de la première diagonale d'un
    tableau t supposé carré
 * @param t le tableau 2D d'entiers
 **/
void afficheDiagonale(vector<vector<int>> t) { // Correction
    for ( int i=0; i < t.size(); i++ ) {
        cout << t[i][i] << ";" ;
```

```
    }
    cout << endl;
}
```

(6) afficher tous les éléments de `t` ; pour l'exemple de l'exercice 1, l'affichage sera :

7	-1	4	3
15	11	17	12
20	34	42	25

Correction :

```
/** affiche tous les elements de la ligne l de t.
 * @param t le tableau 2D d'entiers
 * @param l un entier
 **/
void afficheLigne(vector<vector<int>> t, int l) { // Correction
    for (int i=0; i < t[l].size(); i++) {
        cout << t[l][i] << ";" ;
    }
    cout << endl;
}
```

(7) tester si un tableau à deux dimensions contient un élément `x`.

Correction :

```
/** Test d'appartenance
 * @param t un tableau d'entiers à deux dimensions
 * @param x un entier
 * @return un booleen: true si x apparaît dans t, false sinon
 **/
bool appartient(vector<vector<int>> t, int x) { // Correction
    for ( int i=0; i < t.size(); i++ ) {
        for ( int j=0; j < t[i].size(); j++ ) {
            if ( t[i][j] == x ) {
                return true;
            }
        }
    }
    return false;
}
```

Exercice 4 (Matrices)**.**
Pour être plus spécifique et éviter d'avoir à écrire `vector<vector<int>>` à tout bout de champ, on peut définir un raccourci. Dans les questions suivantes, qui traitent de matrices, on utilisera par exemple le raccourci suivant :

```
typedef vector<vector<int>> Matrice;
```

Les deux constructions suivantes sont alors totalement équivalentes :

```
vector<vector<int>> tab = { {1,2,3}, {4,5,6}, {7,8,9} };
```

```
Matrice tab = { {1,2,3}, {4,5,6}, {7,8,9} };
```

Spécifier et implanter une fonction pour chacune des questions suivantes :

(1) teste si un tableau carré est symétrique, *i.e.* si $T_{i,j} = T_{j,i}$ pour tous i et j.

Correction :

```
/** Teste si une matrice est symétrique
 * @param t une matrice carrée
 * @return true si t[i][j] == t[j][i] pour tout i,j, false sinon
 **/
bool estSymetrique(Matrice t) { // Correction
    for ( int i=0; i < t.size(); i++ ) {
        for ( int j=0; j < i; j++ ) {
            if ( t[i][j]  != t[j][i] ) {
                return false;
            }
        }
    }
    return true;
}
```

Remarque : pour la 2e boucle for, on aurait aussi pu écrire :
`for ( int j = 0; j < t.size(); j++ )` ce qui serait correct mais moins efficace. En effet on ferait deux fois chaque test. Par exemple, on testerait que `t[0][1]=t[1][0]` et que `t[1][0]=t[0][1]`.

(2) calcule la somme de deux matrices (supposées de mêmes tailles). On vous rappelle que la somme de deux matrices T et T' est une matrice C, où $C_{i,j} = T_{i,j} + T'_{i,j}$ pour tous i et j.

Correction :

```
/** somme deux matrices dont les dimensions sont identiques
 * @param t1 une matrice
 * @param t2 une matrice
 * @return la matrice t1 + t2
 **/
Matrice somme(Matrice t1, Matrice t2) { // Correction
    Matrice res;                        // Déclaration
    res = Matrice(t1.size());           // Allocation
    for ( int i=0; i<t1.size(); i++ ) { // Allocation des sous-tableaux
        res[i] = vector<int>(t1[i].size());
    }
    for ( int i=0; i < res.size(); i++ ) { // Initialisation
        for ( int j=0; j < res[i].size(); j++ ) {
            res[i][j] = t1[i][j] + t2[i][j];
        }
    }
    return res;
}
```

(3) ♣ calcule le produit de deux matrices. On vous rappelle que le produit de deux matrices T et T' est une matrice C, où $C_{i,j} = T_{i,1}T'_{1,j} + T_{i,2}T'_{2,j} + \cdots$.

Correction :

```
/** produit de deux matrices dont les dimensions sont compatibles
 * @param t1 une matrice
 * @param t2 une matrice
```

```
 * @return la matrice t1 * t2 (produit matriciel)
 **/
Matrice produit(Matrice t1, Matrice t2) { // Correction
    Matrice res;
    res = Matrice(t1.size());
    for ( int i=0; i<t1.size(); i++ ) {
        res[i] = vector<int>(t2[0].size());
    }
    for ( int i=0; i < res.size(); i++ ) {
        for ( int j=0; j < res[i].size(); j++ ) {
            res[i][j] = 0;
            for ( int k=0; k < t2.size(); k++ ) {
                res[i][j] += t1[i][k] * t2[k][j];
            }
        }
    }
    return res;
}
```

Remarque : Si `t1` est de taille $n \times p$ et `t2` de taille $p \times m$ alors `res` doit être de taille $n \times m$ *i.e.* `t1.size()` $\times$ `t2[0].size()`. De plus la somme pour calculer un coefficient de `res` porte sur p éléments, soit `t2.size()`.

Exercice 5 (Réservation de salle)**.**
Une salle de réunion peut être utilisée par différents employés d'une entreprise. La réservation se fait par plage d'une heure, de 8h00 à 19h00. Chaque plage d'une heure commence à l'heure pile (par exemple, il y a une plage 9h00-10h00 mais il n'y a pas de plage 9h15-10h15). Un tableau de booléens à deux dimensions est utilisé pour représenter si la salle est occupée (valeur `true`) ou disponible (valeur `false`) pendant une semaine. Une dimension est utilisée pour coder les jours ouvrables de 0 (lundi) à 4 (vendredi). L'autre dimension est utilisée pour les plages horaires de 0 (8h00-9h00) à 10 (18h00-19h00). Chaque case correspond à la réservation de la salle pour une plage d'un jour donné.

(1) Écrire le code pour déclarer le tableau représentant l'état d'occupation d'une salle sur une semaine.

Correction :

```
    vector<vector<bool>> occupationSalle;
```

(2) Écrire une fonction qui prend en paramètre le tableau d'état d'une salle et qui l'affiche de façon intelligible (par exemple : salle occupée le mardi de 9h00 à 10h00). Pour cela, on suppose avoir le tableau suivant :

```
vector<string> jours = {"lundi", "mardi", "mercredi", "jeudi", "vendredi"};
```

Correction :

```
void afficheOccupation(vector<vector<bool>> s) { // Correction
    for (int j = 0; j < s.size(); j++) {
        for (int h = 0; h < s[j].size(); h++) {
            if (s[j][h]) {
                cout << "salle occupee le " << jours[j]
                     << " de " << h + 8 << "h00 à " << h + 9 << "h00" << endl;
            }
        }
    }
}
```

(3) Écrire une fonction qui calcule le taux d'occupation d'une salle, c'est à dire le nombre de plages réservées divisé par le nombre total de plages.

Correction :

```
double tauxOccupation(vector<vector<bool>> s) { // Correction
    double nboccupees = 0;
    for (int j = 0; j < s.size(); j++) {
        for (int h = 0; h < s[j].size(); h++) {
            if (s[j][h]) {
                nboccupees++;
            }
        }
    }
    return nboccupees / (s.size() * s[0].size());
}
```

Exercice ♣ 6 (Le jeu du démineur)**.**
L'objectif de cet exercice est de réaliser une version simple du jeu du « démineur ». 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.

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.

Afin d'éviter d'avoir à écrire `vector<vector<char>>` à tout bout de champ on utilise un raccourci.

```
typedef vector<vector<char>> GrilleDemineur ; // tableau 2D de caractères
```

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

Correction : 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 nombreMines(GrilleDemineur grille) { // Correction
    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) Implanter une fonction permettant de tirer au hasard une grille initiale. On supposera fournie une fonction `bool boolAleatoire()` renvoyant un booléen tiré au hasard.

Correction : 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'
 **/
GrilleDemineur grilleInitiale(int n, int m) { // Correction
    // Déclaration
    GrilleDemineur result;
    // Allocation
    result = GrilleDemineur(n);
    // Allocation des sous-tableaux
    for (int i = 0; i < n; i++)
```

```
        result[i] = vector<char>(m);
    // Initialisation
    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) Implanter une fonction permettant de tester si une grille est gagnante.

Correction : 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(GrilleDemineur grille) { // Correction
    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) Implanter une fonction permettant de compter le nombre de mines dans les cases adjacentes à une case donnée d'une grille.

Correction : 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(GrilleDemineur grille, int i, int j) { // Correction
    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;
}
```

Remarque : pour simplifier la boucle, au lieu de compter uniquement dans les 8 cases adjacentes, on compte dans 9 cases (en comptant la case elle-même), puis si besoin on enlève ce qui a été compté en trop.

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

Correction : Afficher le tableau pour le joueur :

```
/** Dessine 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
 *
 * Indications:
 * - si s et t sont des chaînes de caractères, s+t est leur concaténation
 * - "\n" représente un saut de ligne
 **/
string dessinGrille(GrilleDemineur grille) { // Correction
    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 == '0')
                resultat = resultat + "M";
            else if (c == ' ')
                resultat += minesVoisines(grille, i, j) + '0';
            else
                resultat = resultat + " ";
        }
        resultat = resultat + "\n";
    }
    return resultat;
}
```