TD 7 - fonctions, boucles et tableaux à deux dimensions

1. Échauffement

Exercice 1 (Questions de cours)**.**

(1) Quelles sont les deux zones du modèle de mémoire utilisé pour l'exécution d'un programme ? Que contiennent-elles ?

Correction L'espace mémoire d'un programme est partagé en deux zones :
- la *pile* qui contient les variable locales des fonctions,
- le *tas* qui contient le reste.

(2) Rappeler ce qu'est un tableau.

Correction Un *tableau* est une valeur *composite* formée de plusieurs valeurs du même type, auxquelles on accède par leur indice.

(3) Implanter la fonction suivante :

```
/** Teste si un tableau est de taille 9 et ne contient
    que des entiers entre 1 et 9
 * @param t un tableaux d'entiers
 * @return un booleen
 **/
bool verifie(vector<int> t) { //
```

Correction

```
bool verifie(vector<int> t) { //
    if ( t.size() != 9 ) {
        return false;
    }
    for ( int i=0; i<t.size(); i++ ) {
        if ( t[i] < 0 or t[i] > 9 ) {
            return false;
        }
    }
    return true;
}
```

(4) Implanter la fonction suivante :

```
/** Construit un tableau 2D n x n dont les valeurs sont initialisées à v
 * @param n un entier
 * @param v un entier
 * @return le tableau d'entiers
 **/
vector<vector<int>> tableau2DInitialise(int n, int v) { //
```

Correction

```
vector<vector<int>> tableau2DInitialise(int n, int v) { //
    vector<vector<int>> resultat;
    resultat = vector<vector<int>>(n);
    for ( int i=0; i<n; i++ ) {
        resultat[i] = vector<int>(n);
    }
    for ( int i=0; i<n; i++ ) {
        for ( int j=0; j<n; j++ ) {
            resultat[i][j] = v;
        }
    }
    return resultat;
}
```

2. Sudokus

L'objectif de cette séance est de concevoir un programme pour résoudre automatiquement les grilles de sudokus. Ce problème étant assez difficile en général, nous nous concentrerons sur les déductions immédiates que l'on peut effectuer sur une grille.

Commençons par introduire le problème. Une grille de sudoku M est un tableau à deux dimensions 9×9, où chaque case $M_{i,j}$ est comprise entre 0 et 9 (avec 0 dénotant une case vide). Voici une grille de sudoku incomplète et la même grille complétée :

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Exercice 2.

(1) Comment représenter une grille de sudoku en mémoire ?

(2) Spécifier et implanter une fonction `grilleVide` qui construit une grille de sudoku vide et la renvoie.

Correction (1) Une grille de sudoku, et de manière plus générale un tableaux à deux dimensions (d'entiers), peut se représenter comme un tableau de tableaux (d'entiers). En C++, on pourra définir cela par un `vector<vector<int>>`.

(2)

```
/** //
 * Créé une grille vide de sudoku
 * @return une grille vide
 **/
vector<vector<int>> grilleVide() { //
    // Déclaration
```

```
    vector<vector<int>> vide;
    // Allocation des lignes
    vide = vector<vector<int>>(9);
    // Allocation des colonnes
    for ( int i = 0; i < vide.size(); i++ )
        vide[i] = vector<int>(9);
    // Initialisation
    for ( int i = 0; i < vide.size(); i++ )
        for ( int j = 0; j < vide.size(); j++ )
            vide[i][j] = 0;
    return vide;
}
```

Pour éviter d'avoir à écrire `vector<vector<int>>` à tout bout de champ, on définit un raccourci `Grille`. En C++, cela se fait avec :

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

Il devient alors totalement équivalent d'écrire, par exemple,

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

ou

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

Exercice 3.
Spécifier et implanter une fonction `verifie` qui prend en argument un tableau d'entiers à deux dimensions et renvoie vrai si et seulement si le tableau a des tailles et des valeurs compatibles avec une grille de sudoku.

Correction

```
/** //
 * Vérifie qu'une grille est bien une grille de sudoku, c'est à dire qu'elle
 * comporte 9 lignes et 9 colonnes, et toutes les cases sont des entiers
 * compris entre 0 et 9.
 * @param grille une Grille
 * @return booléen
 **/
bool verifie(Grille grille) { //
    if (grille.size() != 9)
        return false;
    for (int ligne = 0; ligne < 9; ligne++)
        if (grille[ligne].size() !=9)
            return false;
    for (int ligne = 0; ligne < 9; ligne++)
        for (int colonne = 0; colonne < 9; colonne++)
            if (grille[ligne][colonne] < 0 || grille[ligne][colonne] > 9)
                return false;
    return true;
}
```

Exercice 4.
Afin de visualiser ce que l'on manipule, implanter (et spécifier) une fonction `affiche` qui affiche à l'écran une grille de sudoku.

♣ : rajouter des espacements pour séparer les neufs carrés 3×3 lors de l'affichage.

Correction

```
/** //
 * Affiche une grille de sudoku dans la console
 * @param grille une Grille de sudoku
 **/
void affiche(Grille grille) { //
    for (int ligne = 0; ligne < 9; ligne++) {
        for (int colonne = 0; colonne < 9; colonne++)
            cout << grille[ligne][colonne] << ' ' ;
        cout << endl;
    }
    cout << endl;
}
```

L'objectif du jeu de sudoku est de compléter la grille de telle sorte que chaque ligne, chaque colonne et chacun des neufs sous-carrés 3×3 disjoints contienne chaque chiffre de 1 à 9 exactement une fois.

Exercice 5.
Écrire une fonction `estPresentDansColonne` qui prend en argument une grille, un indice de colonne ainsi qu'une valeur et renvoie `true` si cette colonne contient la valeur, et `false` sinon. Sur le même modèle, implanter et spécifier des fonctions `estPresentDansLigne` et `estPresentDansCarre`. Cette dernière fonction prend en argument les coordonnées de la *case en haut à gauche* du carré, c'est à dire que le carré correspondra à $\{\text{ligne}, ..., \text{ligne} + 2\} \times \{\text{colonne}, ..., \text{colonne} + 2\}$.

Correction

```
/** Teste si un entier est dans une colonne
 * @param grille une Grille de sudoku
 * @param colonne un entier compris entre 0 et 9
 * @param v un entier
 * @return vrai si `v` est dans la colonne numéro `colonne` de la grille du sudoku `grille`.
 **/
bool estPresentDansColonne(Grille grille, int colonne, int v) { //
    for (int ligne = 0; ligne < 9; ligne++)
        if (grille[ligne][colonne] == v)
            return true;
    return false;
}

/** //
 * Teste si un entier est dans une ligne
 * @param grille une Grille de sudoku
 * @param ligne un entier compris entre 0 et 8
```

```
 * @param v un entier
 * @return vrai si ‘v‘ est dans la ligne numéro ‘ligne‘ de la grille du sudoku ‘grille‘.
 **/
bool estPresentDansLigne(Grille grille, int ligne, int v) { //
    for (int colonne = 0; colonne < 9; colonne++)
        if (grille[ligne][colonne] == v)
            return true;
    return false;
}

/** //
 * Teste si un entier se trouve dans un sous-carré de taille 3x3
 * @param grille une Grille de sudoku
 * @param ligne un entier compris entre 0 et 8
 * @param colonne un entier compris entre 0 et 8
 * @param v un entier
 * @return vrai si ‘v‘ est dans le sous carré 3x3 dont le bord haut-gauche se
 * trouve en position (ligne, colonne).
 **/
bool estPresentDansCarre(Grille grille, int ligne, int colonne, int v) { //
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (grille[ligne+i][colonne+j] == v)
                return true;
    return false;
}
```

Exercice 6.

(1) Spécifier et implanter une fonction `estRemplie` qui renvoie vrai si la grille est remplie, c'est-à-dire ne contient pas de 0.

(2) Spécifier et implanter une fonction `estSolution` qui renvoie `true` si la grille est une solution, c'est-à-dire si chaque ligne, chaque colonne et chacun des neufs sous-carrés contienne chaque chiffre 1 à 9 exactement une fois.

Correction

```
/** //
 * Teste si une grille est remplie, c'est à dire si elle ne contient aucun 0
 * @param grille une grille de sudoku
 * @return un booléen
 **/
bool estRemplie(Grille grille) { //
    for (int ligne = 0; ligne<9; ligne++)
        for (int colonne = 0; colonne<9; colonne++)
            if (grille[ligne][colonne] == 0)
                return false;
    return true;
}

/** //
```

```
 * Teste si une grille complète vérifie les règles du jeu
 * @param grille une grille de sudoku
 * @return un booléen
 **/
bool estSolution(Grille grille) { //
    if ( !estRemplie(grille))
        return false;

    for (int v = 1; v <= 9; v++)
        for (int ligne = 0; ligne < 9; ligne++)
            if ( !estPresentDansLigne(grille, ligne, v))
                return false;

    for (int v = 1; v <= 9; v++)
        for (int colonne = 0; colonne<9; colonne++)
            if ( !estPresentDansColonne(grille, colonne, v))
                return false;

    for (int v=1; v<=9; v++)
        for (int ligne = 0; ligne < 9; ligne += 3)
            for (int colonne = 0; colonne<9; colonne += 3)
                if ( !estPresentDansCarre(grille, ligne, colonne, v))
                    return false;

    return true;
}
```

Quelqu'un propose l'idée suivante : *Étant donnée une grille partielle, regardons s'il existe une case vide n'ayant qu'une seule valeur directement possible, et dans ce cas, remplissons la avec la valeur correspondante.*

Exercice 7.
Appliquer à la main cette idée sur les deux grilles suivantes :

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

		1				8		
	7		3	1			9	
3				4	5			7
	9		7			5		
	4	2		5		1	3	
		3			9		4	
2			5	7				4
	3			9	1		6	
		4				3		

Que concluez vous ?

Exercice 8.
Écrire une fonction `valeursPossibles` qui prend en argument une grille et une case et qui renvoie les valeurs directement possibles stockées dans un tableau. Rappel : en `C++` : si `t` est un `vector<int>` et `u` un `int`, alors `t.push_back(u)` ajoute `u` à la fin du tableau `t`.

Correction

```
/** Renvoie les valeurs possibles pour une case de sudoku selon les règles du
 * jeu
 * @param grille une grille de sudoku
 * @param ligne un entier compris entre 0 et 8
 * @param colonne un entier compris entre 0 et 8
 * @return un vecteur d'entiers contenant toutes les différentes valeurs
 * acceptées pour la case aux coordonnées (ligne, colonne).
 **/
vector<int> valeursPossibles(Grille grille, int ligne, int colonne) { //
    vector<int> possibles;
    for (int v = 1; v <= 9; v++)
        if ( !(estPresentDansLigne(grille, ligne, v) ||
                    estPresentDansColonne(grille, colonne, v) ||
                    estPresentDansCarre(grille, 3*(ligne/3), 3*(colonne/3), v)))
            possibles.push_back(v);
    return possibles;
}
```

Exercice 9.
Écrire une fonction `complete` qui prend en entrée une grille et qui retourne en sortie une grille ne possédant plus aucune case vide n'ayant qu'une seule valeur directement possible.

Correction

```
/** Complète une grille naivement
 *
 * Algorithme:
 * 1. Parcourt la grille en recherchant une case égale à 0
 *  - Calcule les valeurs possibles à insérer dans cette case
 *  - Si une seule valeur est acceptable, alors la grille est mise à jour
 *  - Si aucune valeur n'est acceptable, renvoyer une grille vide (pas de
 *  solution)
 *  - Passer à la case suivante
 *  .
 * 2. Si la grille n'a pas été mise à jour, renvoyer la grille incomplète
 * 3. Si la grille est complétée, renvoyer la grille
 * 4. Sinon continuer à 1.
 * @param grille une grille de sudoku
 * @return une grille de sudoku complétée au maximum permis par l'algorithme,
 * vide si aucune complétion possible.
 **/
Grille complete(Grille grille) { //
   bool changement = true;
   while(changement) {
       changement = false;
       for (int ligne = 0; ligne < 9; ligne++)
           for (int colonne = 0; colonne < 9; colonne++)
               if (grille[ligne][colonne] == 0) {
                   vector<int> vals = valeursPossibles(grille, ligne, colonne);
                   if (vals.size() == 0)
                       return vector<vector<int>>();
                   if (vals.size() == 1) {
                       grille[ligne][colonne] = vals[0];
                       changement = true;
                   }
               }
   }
   return grille;
}
```

Exercice ♣ 10 (Pour aller plus loin)**.**
Comme vous avez pu le constater cette fonction ne suffit pas. Pour obtenir un programme complet pour résoudre les sudokus, il faut faire des hypothèses et revenir dessus si elles ne conduisent pas à la solution (sans que l'on puisse prévoir quand). Notez que cette idée seule pourrait résoudre le problème en théorie ; cependant elle serait trop lente en pratique. C'est pourquoi les fonctions des exercices 8 et 9 sont très utiles pour accélérer l'exploration dans la plupart des cas !

Du code effectuant cette exploration sera fourni en TP, mais vous pouvez réfléchir à sa structure, notamment aux deux options possibles : avec une seule grille ou avec des copies des grilles courantes.

Exercice ♣ 11.
Écrire un programme qui prend en argument un fichier où chaque ligne est constituée de 81 entiers consécutifs

```
138570009209400176000910050003807004620394817840000 0...
```

puis résoudre le sudoku associé.

On pourra trouver de telles suite à l'adresse : http ://www.printable-sudoku-puzzles.com/wfiles/
(♣ ♣ ♣) Vous avez sans doute remarqué que certaines grilles, comme celle ci-dessous, sont "difficiles" à résoudre dans le sens où le temps de calcul est très long.

```
000000012
008030000
000000040
120500000
000004700
060000000
507000300
000620000
000100000
```

Proposer des optimisations.
(♣ ♣ ♣) Une grille peut avoir plusieurs solutions, comment les énumérer ?
(♣ ♣ ♣) Comment générer une grille de sudoku ?