# Série 3 :
# Programmation *C* - Tableau et `struct`

## Buts

Le but de cette série d'exercices est de pratiquer les bases des tableaux et structures (de données). en C.

## Rappel

Avez-vous pris connaissance des **conseils relatifs à ces séries d'exercices** ?

---

## Exercice 1 : Produit scalaire (tableaux de taille fixe, niveau 1)

Écrivez un programme `scalaire.c` qui calcule le produit scalaire de deux vecteurs (implémentés au moyen de tableaux de taille fixe).
Votre programme devra utiliser (entre autres) les éléments suivants :

- **Prototypes :** `double scalaire(const double u[], const double v[], size_t taille);` la fonction qui calcule le produit scalaire.
- Déclarations locales au `main()` :
  - une *constante* `N_MAX` représentant la taille maximale des vecteurs (inutile de lui donner une valeur trop élevée... 10 suffit amplement ici) ;
  - deux variables `v1` et `v2`, de type «tableau de réels» de taille `N_MAX`.
- *Méthode* :
  - demander à l'utilisateur d'entrer `n`, la taille effective des vecteurs.

    Rappel : pour lire un entier, utiliser `"%d"` (et pour lire un `size_t`, `"%zu"`).

  - vérifier que `n` est compris entre 1 et `N_MAX` (et demander à l'utilisateur d'entrer à nouveau une valeur tant que ce n'est pas le cas).
    On pourra pour cela utiliser le résultat de l'exercice 1.
  - demander à l'utilisateur les composantes ($v1_0$, ..., $v1_{n-1}$) et ($v2_0$, ..., $v2_{n-1}$) des vecteurs `v1` et `v2`.

    Rappel : pour lire un `double`, utiliser `"%lf"`, `"%zu"`).

  - appeler la fonction `scalaire(...)` pour calculer le produit scalaire de `v1` et `v2`.
  - afficher le résultat.

**Rappel :**
Le produit scalaire de *a* par *b* est: $a.b = a_1*b_1 + a_2*b_2 + ... + a_n*b_n$

Exemple : a = (5, 3, -1)  b = (2, 1, 2)  a.b = 11

---

## Exercice 2 : Multiplication de matrices (tableaux, typedef, niveau 2)

On cherche ici à écrire un programme `mulmat.c` qui calcule la multiplication de deux matrices (rappel de l'algorithme ci-dessous).

### Déclarations :

- définissez (`#define`) une constante `N` comme la taille ***maximale*** pour la dimension d'une matrice
- dans `main()`, déclarez deux matrices `M1` et `M2`.
- Définissez le type `Matrice` en utilisant une structure contenant un tableau et deux entiers non signés représentant le nombre de colonnes et le nombre de lignes.

### Fonctions :

- la fonction de prototype

  ```
  Matrice lire_matrice(void);
  ```

  qui lit depuis le clavier les éléments d'une matrice (après avoir demandé ses tailles de lignes et colonnes et vérifié que celles-ci sont plus petites que `N`) et retourne la matrice résultante.

- la fonction de prototype

  ```
  Matrice multiplication(const Matrice a, const Matrice b);
  ```

  qui multiplie deux matrices de tailles compatibles et affecte le résultat.

- la fonction de prototype

  ```
  void  affiche_matrice(const Matrice m);
  ```

  qui affiche le contenu d'une matrice ligne par ligne.

### Méthode :

- lire depuis le clavier les dimensions **l1** (nombre de lignes) et **c1** (nombre de colonnes) de la première matrice M1
- lire le contenu de M1.
- De même, lire les dimensions puis le contenu de la seconde matrice M2.
- Vérifier que le nombre de lignes de M2 est identique au nombre de colonnes de M1.
  Dans le cas contraire, afficher un message d'erreur "`Multiplication de matrices impossible !`"

- Effectuer la mutliplication des matrices : M = M1*M2 :
  - Les dimensions de M sont : l1 (nombre de lignes) et c2 (nombre de colonnes).
  - l'élément $M_{i,j}$ est défini par

$$M_{i,j} = \sum_{k=1}^{c1} M1_{i,k} \cdot M2_{k,j}$$

- afficher le résultat ligne par ligne.

**Exemple d'utilisation :**

```
Saisie d'une matrice :
  Nombre de lignes : 2
  Nombre de colonnes : 3
  M[1,1]=1
  M[1,2]=2
  M[1,3]=3
  M[2,1]=4
  M[2,2]=5
  M[2,3]=6
Saisie d'une matrice :
  Nombre de lignes : 3
  Nombre de colonnes : 4
  M[1,1]=1
  M[1,2]=2
  M[1,3]=3
  M[1,4]=4
  M[2,1]=5
  M[2,2]=6
  M[2,3]=7
  M[2,4]=8
  M[3,1]=9
  M[3,2]=0
  M[3,3]=1
  M[3,4]=2
Résultat :
38 14 20 26
83 38 53 68
```

---

## Exercice 3 : Placement sans recouvrement (tableaux, niveau 2)

Le but de cet exercice est de placer sans recouvrement des objets rectilignes sur une grille carrée. Cela pourrait être par exemple une partie d'un programme de bataille navale.

Dans le fichier `recouvrement.c` :

1. À l'aide d'une macro (`#define`), définissez une constante globale, nommée `DIM` et de valeur 10. Elle représentera la taille de la grille (carrée).
2. Prototypez et écrivez une fonction :

   ```
   int remplitGrille(
         Grille grille,
         size_t ligne,
        size_t colonne,
        char direction,
        size_t longueur
      );
   ```

   dont le rôle est de vérifier si le placement dans une

   `grille` (voir ci-dessous) d'un objet de dimension `longueur` est possible, en partant de la coordonnée (`ligne`,`colonne`) et dans la direction définie par `direction` (Nord, Sud, Est ou Ouest).

   Si le placement est possible, la fonction devra de plus effectuer ce placement (voir ci-dessous la description de la grille).

   La fonction devra indiquer (par la valeur de retour) si le placement a pu être réalisé ou non.

3. Dans le `main()`
   - définissez une **variable** nommée `grille`, correspondant à un tableau de `char`, de taille fixe, à **deux dimensions**, avec `DIM` comme paramètre de taille pour chaque dimension (c'est-à-dire une grille carrée).
     Le caractère `'#'` dans une case `[i][j]` de cette grille représente le fait qu'un (bout d')objet occupe la case de coordonnées (i, j) et le caractère `'.'` que la case est vide (utilisez des constantes).
   - Utilisez la fonction précédente pour remplir interactivement la grille, en demandant à l'utilisateur de spécifier la position, la taille et la direction des objets à placer.
     Indiquez à chaque fois à l'utilisateur si l'élément a pu ou non être placé dans la grille.
     Le remplissage se termine lorsque l'utilisateur entre une coordonnée strictement inférieure à 0.
   - Terminer alors en "dessinant" la grille : afficher un `'.'` si la case est vide et un `'#'` si la case est occupée.

Remarques :

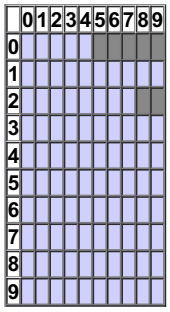
- Dans l'interface utilisateur, pour indiquer les positions, utilisez au choix soit les coordonnées du C : 0 à DIM-1 (plus facile), soit les coordonnées usuelles (1 à DIM, un peu plus difficile) , **MAIS dans tous les cas utilisez les indices de 0 à DIM-1** pour votre tableau (aspect programmeur).
- Pensez à effectuer des tests de validité sur les valeurs entrées (débordement du tableau).
- pour représenter la direction, vous pouvez soit utiliser un caractère (`'N'` pour nord. `'S'` pour sud, etc..., plus facile), soit un type enuméré (plus difficile : pensez à l'interface avec l'utilisateur).
- N'oubliez pas d'initialiser la grille en tout début de programme !
- Le prochain caractère après un `scanf("%d");` sur le clavier, est en fait le retour à la ligne que vous avez tapé ; pour le supprimer avant de lire

votre « vrai » caractère (direction), utilisez un `getchar();`.

**Exemple de fonctionnement (version facile : coordonées de 0 à 9 et lettres pour les directions) :**

```
Entrez coord. x: 2
Entrez coord. y: 8
Entrez direction (N,S,E,O): E
Entrez taille: 2
Placement en (2,8) direction E longueur 2 -> succès
Entrez coord. x: 0
Entrez coord. y: 8
Entrez direction (N,S,E,O): S
Entrez taille: 5
Placement en (0,8) direction S longueur 5 -> ECHEC
Entrez coord. x: 0
Entrez coord. y: 9
Entrez direction (N,S,E,O): O
Entrez taille: 5
Placement en (0,9) direction O longueur 5 -> succès
Entrez coord. x: -1
```

**Résultat des placements :**




---

## Exercice 4 : Nombres complexes (structures, niveau 1)

Le but de ce programme est d'effectuer les manipulations élémentaires sur les nombres complexes : addition, soustraction, multiplication et division.

Dans le fichier `complexes.c`, définissez une structure `Complexe` représentant un nombre complexe comme deux `double` (forme cartésienne).

Ensuite, prototypez puis définissez une procédure `affiche` qui prend un nombre complexe en argument et l'affiche.

Dans le `main()`, déclarez et initialisez un nombre complexe. Affichez-le. Compilez et executez votre programme pour vérifier que tout fonctionne comme prévu jusqu'ici.

Prototypez puis définissez une fonction `addition` qui prend deux nombres complexes en argument et retourne leur somme.

Ecrivez également les fonctions `soustration`, `multiplication` et `division`.

Testez toutes vos fonctions avec les calculs suivants:

```
(1,0) + (0,1) = (1,1)
(0,1) * (0,1) = (-1,0)
(1,1) * (1,1) = (0,2)
(0,2) / (0,1) = (2,0)
(2,-3) / (1,1) = (-0.5,-2.5)
```

### Rappel

la multiplication de `z=(x,y)` par `z'=(x',y')` est le nombre complexe `z*z'=(x*x'-y*y', x*y'+y*x')`.
la division de `z=(x,y)` par `z'=(x',y')` est le nombre complexe `z/z'=((x*x'+y*y')/(x'*x'+y'*y'), (y*x'-x*y')/(x'*x'+y'*y'))`.

Si vous en avez besoin, voici un **rappel** plus complet sur les nombres complexes.

---

## Exercice 5 : Nombres complexes revisités (structures, niveau 2)

On s'intéresse ici à la résolution d'équations du second degré sur le corps des complexes.

Copiez le programme `complexes.c` dans le programme `complexes2.c` et éditez ce dernier.

Écrivez la fonction `Complexe racine(Complexe);` qui calcule la racine carrée de partie réelle positive d'un nombre complexe.

On peut montrer que la racine carrée `z'=(x',y')` de partie réelle positive d'un nombre complexe `z=(x,y)` est donnée par :

$$x' = \sqrt{\frac{\sqrt{x^2+y^2}+x}{2}}$$

$$y' = \mathrm{sgn}(y) \cdot \sqrt{\frac{\sqrt{x^2+y^2}-x}{2}}$$

où $\mathrm{sgn}(y)$ représente le signe de $y$,
avec ici la convention $\mathrm{sgn}(0) = 1$.

Testez avant de continuer. Calculez par exemple la racine carrée de -1.

Déclarez le type `Solutions` comme une structure à 2 champs `Complexes` : `z1` et `z2`.

Pour finir prototypez puis définissez la fonction `resoudre_second_degre` qui prend deux arguments `Complexe b` et `c` et retourne, sous forme de `Solutions`, les solutions de l'équation:
`z*z + b*z + c= 0`

**Note :** utilisez la même formule que pour résoudre une équation du second degré avec des nombres réels (dans le cas à 2 solutions), mais exploitez vos propres opérateurs (`multiplication`, `division`, etc) sur les `Complexe` pour trouver les `Solutions` de l'équation.

### Exemples de solutions

Avec `b=0` et `c=1` on a :

```
z1=-i
z2=i
```

Avec `b=3-2i` et `c=-5+i` on a :

```
z1=-4.11442+1.76499i
z2=1.11442+0.235013i
```

---

**Dernière mise à jour le 12 mars 2016**
**Last modified: Sat Mar 12, 2016**