

## 1 Les tableaux

Un tableau est un ensemble de « cases » mémoire **consécutives**.

Toutes les « cases » ont le **même type**.

On accède **immédiatement** à 1 case particulière (« élément ») par **indexation**.

Un tableau répond au besoin de stocker plusieurs données de **même type** et d'accéder **rapidement** (tps constant) à n'importe quel élément.

Un tableau est efficace en espace s'il ne comporte pas trop de **de trous** entre les éléments effectifs (sinon gaspillage mémoire).

## 2 Tableaux statiques en C

Par **statique**, on entend « dont la **taille est connue à la compilation** ».

### 2.1 Déclaration

Comme les variables, les tableaux doivent être **déclarés**.

type-élément nom [ taille-constante ] ;

```
int main ()
{
  float tf[10] ; // Tableau de 10 flottants.
  int ti[5] ; // Tableau de 5 entiers signes.
  return (0) ;
}
```

Attention : « taille-constante » est un littéral (valeur entière comme 42, 15 ou 53) pas une variable.

```
int main ()
{
  int size = ... ;
  float tf[size] ; // NON !
  ...
}
```

### 2.2 Initialisation

Comme les variables, les tableaux doivent être **initialisés** et peuvent l'être lors de la **déclaration**.

type-élément nom [ taille ] = { val<sub>1</sub> , ... , val<sub>n</sub> } ;

```
int main ()
{
  float tf[4] = { 0.1, -1.2, 7.6 } ; // Tableau de 4 flottants.
  int ti[5] = { 1, 500, -20, 3, 0 } ; // Tableau de 5 entiers signes.
  return (0) ;
}
```

Un tableau non initialisé est comme une variable non initialisée : il y a *a priori* n'importe quoi dedans.

## 2.3 Accès à un tableau

L'élément d'indice  $i$  d'un tableau  $t$  est dénoté par  $t[i]$ .

Les indices de tableaux (« numéros de cases ») commencent à 0 ! Donc un tableau de taille  $n$  a des « numéros de cases » de 0 à  $n - 1$ .

```
#include <stdio.h>

int main ()
{
    int i ;
    int t[5] ;
    for (i = 0; i < 5; i++) { // Inferieur STRICT a 5.
        t[i] = i + 1 ; // Mettre dans la case i la valeur i + 1.
        printf ("Case %d, valeur = %d\n", i, t[i]) ;
    }
    return (0) ;
}
```

On ne peut pas affecter un tableau d'un seul coup. Donc on ne peut pas initialiser un tableau après déclaration un tableau d'un seul coup.

```
int main ()
{
    int t[5] ;
    t = { 1, 500, -20, 3, 0 } ; // Incorrect !
    return (0) ;
}
```

## 2.4 Les chaînes de caractères

Une chaîne de caractères en C est un tableau de char terminé par le caractère `'\0'`.

Fonctions standard de C (`#include <stdlib.h>`) pour transformer une chaîne de caractères en scalaire (renvoie 0 si la chaîne ne représente par l'écriture décimale d'un scalaire) :

- `atoi` : string  $\rightarrow$  int
- `atol` : string  $\rightarrow$  long int
- `atoll` : string  $\rightarrow$  long long int
- `atof` : string  $\rightarrow$  float

## 2.5 Tableaux à plusieurs dimensions

Exemple : une image de taille  $larg \times haut$  peut être représentée comme un tableau à 2 dimensions de points dont l'une est de taille  $larg$  et l'autre de taille  $haut$ .

Les tableaux à plusieurs dimensions sont déclarés et indexés par autant de dimensions.

type-élément nom [ taille-dim<sub>1</sub> ] [ taille-dim<sub>2</sub> ] ... [ taille-dim<sub>n</sub> ] ;

```
#include <stdio.h>

int main ()
{
    int t[5][3] ; // Tableau de 5 x 3 entiers.
    t[2][1] = 5 ; // Affectation.
    printf ("%d\n", t[2][1] + 4) ; // Lecture.
    return (0) ;
}
```

Le tableau `argv` du `main` est un tableau de chaînes de caractères, donc un tableau à 2 dimensions.

### 3 Structures de données élémentaires

Elles répondent au besoin de représenter autre chose que des scalaires (nombres) ou des tableaux (suites contiguës d'éléments de même type).

#### 3.1 Types énumérés

Ils répondent au besoin de représenter des valeurs **atomiques disjointes** (« *symbolique* »).

Exemple : l'ensemble des couleurs de cartes à jouer, { Coeur, Carreau, Trèfle, Pique }

Déclaration d'un **type** énuméré :

```
enum nom-type { nom-valeur1 , ... , nom-valeurn } ;
```

Déclaration d'une **variable** de type enum :

```
enum nom-type-enum nom-variable ;
```

On utilise ensuite les valeurs par leur **nom** (on peut directement initialiser une variable lors de sa déclaration).

```
enum carte_t { Coeur, Carreau, Trefle, Pique } ;

enum carte_t echange (enum carte_t c)
{
    enum carte_t res = Coeur ;
    switch (c) {
        case Coeur: res = Carreau ; break ;
        case Carreau: res = Coeur ; break ;
        case Trefle: res = Pique ; break ;
        case Pique: res = Trefle ; break ;
    }
    return (res) ;
}
```

#### 3.2 Structures

Elles répondent au besoin **d'aggréger** des données de **types différents**.

Une structure est un **groupement** de données par **champs nommés**.

Déclaration d'un **type** structure :

```
struct nom-type { nom-champ1 type-champ1 ; ... ; nom-champn type-champn } ;
```

Déclaration d'une **variable** de type struct :

```
struct nom-type nom-variable ;
```

Comme pour les tableaux, il est possible **d'initialiser directement** la valeurs des champs d'une variable de type structure lors de sa **déclaration**, mais **impossible** d'affecter d'un **seul coup** une telle variable.

L'accès à un champ de la structure se fait par notation **pointée** : expr-struct.nom-champ

```
struct circle_t {
    int x_center ;
    int y_center ;
    unsigned int radius ;
};

struct circle_t translate (struct circle_t c, int dx, int dy)
{
    struct circle_t res ;
    res.x_center = c.x_center + dx ;
    res.y_center = c.y_center + dy ;
    res.radius = c.radius ;
    return (res) ;
}
```