Chapitre un
LES TABLEAUX

Lorsque les données sont nombreuses et de même nature, au lieu de manipuler un grand nombre de variables, il est plus pratique de ranger ces variables dans un **tableau**. Dans la plupart des langages de programmation existe la notion de tableau indexé par des entiers ; des langages comme le PASCAL permettent de plus l'indexation par des éléments d'autres types, pas tout à fait n'importe lesquels mais les types dits *types ordinaux*. Nous allons nous intéresser dans ce chapitre aux tableaux indexés par des entiers.

La notion de **tableau** est une notion courante très utilisée, pas seulement en informatique. Il s'agit la plupart du temps de *tableaux à deux dimensions* avec des *lignes* et des *colonnes*. Mais on peut aussi considérer des tableaux à une seule dimension (ce sont plutôt des *listes* dans le langage courant) ou à plusieurs dimensions (plus difficilement représentables de façon graphique à partir de la quatrième dimension).

Voyons, d'une part, comment mettre en œuvre les tableaux en programmation et, d'autre part, leur intérêt pour les problèmes de programmation. Nous allons étudier les tableaux à une dimension, les tableaux à deux dimensions puis les tableaux à plusieurs dimensions.

1 Tableaux à une dimension

1.1 Notion

Notion intuitive.- Un **tableau à une dimension** est formé d'**éléments** tous de même nature et repérés par un **index**.

Représentation graphique.- Un tableau à une dimension est souvent représenté comme une suite de cases avec un index pointant sur une case.

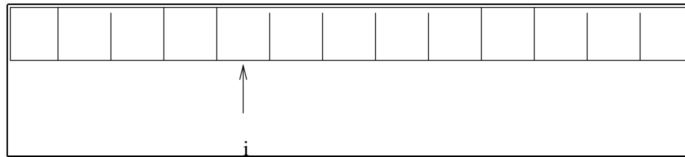

Figure 1:

Formalisation mathématique.- Nous avons dit que tous les éléments d'un tableau sont de même nature, ce qui veut dire qu'ils appartiennent à un même ensemble A. Un tableau à une dimension est un n-uplet sur l'ensemble A, donc de la forme $(x_1, x_2, ..., x_n)$. L'entier naturel n est appelé la **dimension** du tableau.

L'ensemble des tableaux (à une seule dimension) de dimension n à éléments dans l'ensemble A est donc la puissance cartésienne $A^n = A \times A \times ... \times A$.

Si $x = (x_1, x_2, ..., x_n)$ est un tel tableau, l'élément d'**index** i, avec i entier naturel compris entre 1 et n, où n est la dimension du tableau, est la i-ième projection de x, à savoir $x_i = pr_i(x)$.

1.2 Mise en place des tableaux à une dimension en langage C

1.2.1 Nom d'un tableau

Un tableau (à une dimension) porte un nom qui est, comme d'habitude, un identificateur non déjà utilisé pour autre chose, par exemple `TAB`. Il s'agit plus exactement d'une variable de tableaux.

1.2.2 Déclaration d'un tableau

Syntaxe.- La déclaration d'un tableau suit la syntaxe suivante :

```
type Nom [Entier];
```

où `Type` est un type, `Nom` est un identificateur et `Entier` une expression constante entière positive.

Sémantique.- `Type` est le type des éléments du tableau, `Nom` est le nom du tableau et `Entier` est le nombre (maximum) d'éléments du tableau.

Exemple.- On déclare un tableau TAB de cinq réels de la façon suivante :

```
float TAB [5];
```

1.2.3 Accès à un élément d'un tableau

Syntaxe.- L'accès à un élément du tableau de nom `Nom` se fait en désignant cet élément de la façon suivante :

```
Nom[index]
```

où `index` est une expression entière positive.

Sémantique.- Ceci permet de désigner l'élément du tableau dont l'index est celui désigné. En langage C, comme en général en informatique et contrairement à ce qui se passe en mathématiques, l'indexation commence à 0 (et non à 1). La valeur de l'index doit donc être comprise entre 0 et $Entier - 1$.

Exemple.- Si on veut accèder au troisième élément du tableau déclaré ci-dessus, donc celui d'index 2, il suffit d'écrire `TAB[2]`. Ceci conduit à des instructions du genre :

```
TAB[3] = 4.5;
X = 3*TAB[I] + 2*TAB[I+5];
```

à condition, bien sûr, que la variable `X` soit déclarée d'un type convenable et que `I` et `I+5` prennent leurs valeurs parmi les index possibles.

Remarque.- En langage C, comme dans la plupart des langages informatiques, il n'y a pas de vérification pour savoir si la valeur de l'index est bien comprise entre 0 et $Entier - 1$. Si on sort de l'intervalle d'indexation, on risque de récupérer des valeurs sans signification (en cas de lecture) ou, pire, de détruire une partie du contenu de la mémoire vive (en cas d'écriture). Il faut donc être extrêmement vigilant à l'égard de ce problème.

1.3 Un exemple

Introduction.- Nous allons traiter un premier exemple montrant l'intérêt de la notion de tableau pour des problèmes de programmation. Le problème suivant est suffisamment courant et exemplaire pour qu'on s'y intéresse.

Problème.- Les élèves d'une classe ont obtenu chacun une note à un certain examen. Nous voulons déterminer combien d'entre eux ont une note supérieure à la moyenne de la classe.

1.3.1 Algorithme sans tableau

Un premier algorithme.- Le premier algorithme auquel on peut penser est le suivant :

- demander le nombre `N` d'élèves ;
- saisir les `N` notes tout en calculant la somme de ces notes ;
- diviser la somme par `N` pour obtenir la moyenne de la classe ;
- mettre à zéro le compteur des notes supérieures à la moyenne de la classe ;
- redemander les `N` notes et comparer chacune d'elles à la moyenne de la classe afin d'incrémenter le compteur des notes supérieures à la moyenne de la classe ;
- afficher le résultat.

Exercice 1.- Implémenter cet algorithme en langage C.

Inconvénient de cet algorithme.- Cet algorithme possède un inconvénient majeur pour l'utilisateur : il faut saisir l'ensemble des notes deux fois. On comprend que l'ordinateur ne peut pas deviner les notes et qu'il faut donc les saisir une fois, mais l'ordinateur ne pourrait-il pas retenir les notes ?

1.3.2 Algorithme avec tableau

Un meilleur algorithme.- Soit `N` le nombre d'élèves. On peut :
- saisir les `N` notes et les conserver ;
- calculer la moyenne de ces `N` notes ;
- déterminer combien, parmi ces `N` notes, sont supérieures à la moyenne ainsi obtenue.

La notion de tableau est bien adaptée pour conserver les notes.

Programme.- Ceci nous conduit au programme suivant en supposant qu'il y ait 59 élèves :

```
/* Programme NOTE_1.c */

#include <stdio.h>

void main(void)
     {
     int i, s;
     float m;
     float note[59];

/* Saisie des notes */
     for (i=0 ; i < 59 ; i++)
         {
         printf("\nQuelle est la note numero %d : ",i+1);
         scanf("%f", &note[i]);
         }

/* Calcul de la moyenne */
     m  = 0;
     for (i = 0 ; i < 59 ; i++) m = m + note[i];
     m = m/59;
     printf("\nLa moyenne est de : %4.2f", m);

/* Calcul du nombre de notes superieures a la
 * moyenne */
     s = 0;
     for (i = 0 ; i < 59 ; i++)
     if (note[i] >= m) s = s+1;
     printf("\nLe nombre de notes superieures a la");
     printf("\nmoyenne de la classe est de : %d", s);
     }
```

Avantage et inconvénient de cet algorithme.- On voit tout de suite l'avantage de cet algorithme par rapport au précédent pour l'utilisateur. Cependant il présente également un inconvénient (on ne peut pas gagner sur tous les tableaux !, et ce n'est pas que pour faire un jeu de mots) : le nombre d'élèves doit être connu au moment de la programmation alors, que pour le premier algorithme, il était demandé à l'utilisateur.

1.4 Dimension maximum et dimension effective

Introduction.- On peut contourner en général l'inconvénient que nous venons de constater pour l'utilisation des tableaux. On déclare une **dimension maximum** du tableau puis on demande une **dimension effective** à l'utilisateur, ce qui détermine la partie du tableau réellement utilisée.

Exemple.- Ceci nous conduit, par exemple, au programme suivant, en supposant qu'il y a au plus 150 élèves par classe :

```
/* Programme NOTE_2.c */

#include <stdio.h>

void main(void)
     {
     const int max = 150;
     int n, i, s;
     float m;
     float note [max];

/* Determination du nombre effectif d'eleves */
     printf("Quel est le nombre d\'eleves ? : ");
     scanf("%d", &n);

/* Saisie des notes */
     for (i=0 ; i < n ; i++)
         {
         printf("\nQuelle est la note numero %d : ",i+1);
         scanf("%f", &note[i]);
         }

/* Calcul de la moyenne */
     m = 0;
     for (i = 0 ; i < n ; i++) m = m + note[i];
     m = m/n;
     printf("\nLa moyenne est de : %4.2f", m);

/* Calcul du nombre de notes superieures a la
 * moyenne */
     s = 0;
     for (i = 0 ; i < n ; i++)
     if (note[i] >= m) s = s+1;
     printf("\nLe nombre de notes superieures a la");
     printf("\nmoyenne de la classe est de : %d\n", s);
     }
```

Remarque.- On notera la déclaration de la constante `max` pour la dimension maximum du tableau. Si celle-ci se révèle un jour insuffisante il suffit de changer cette ligne.

Tout inconvénient a-t-il complètement disparu ?.- Bien sûr que non puisque, comme nous venons de le faire remarquer, la dimension maximum peut se révéler insuffisante. L'expérience montre cependant que pour certains problèmes de programmation, tel que celui que nous venons de considérer, il existe des dimensions maximum naturelles.

Remarquons aussi que nous avons remplacé un inconvénient par un autre. En effet le tableau avec sa dimension maximum occupe plus de place que nécessaire en mémoire centrale.

On dit que la structure de données tableau est une **structure de données statique**, par opposition aux **structures de données dynamiques** pour lesquelles l'utilisateur peut saisir la taille au moment de l'exécution (et non avant la compilation).

1.5 Émulation des tableaux

Introduction.- Nous avons déjà dit que nous pouvions programmer tout ce qui est programmable avec les primitives de programmation. Les tableaux ont l'air d'être intéressant pour le programmeur. Peut-on s'en passer ? En théorie oui.

Un exemple.- Considérons le programme `NOTE_1.c`. Écrivons un programme de comportement absolument identique pour l'utilisateur mais n'utilisant pas la notion (informatique) de tableau.

Nous allons juste considérer qu'il y a 5 élèves au lieu de 59 pour des raisons que nous allons comprendre tout de suite. Ceci nous conduit au programme suivant :

```
/* Programme NOTE_3.c */
#include <stdio.h>
void main(void)
     {
     int s;
     float m;
     float NOTE_1, NOTE_2, NOTE_3, NOTE_4, NOTE_5;

/* Saisie des notes */
     printf("Quelle est la note numero 1 : ");
     scanf("%f", &NOTE_1);
     printf("Quelle est la note numero 2 : ");
     scanf("%f", &NOTE_2);
     printf("Quelle est la note numero 3 : ");
     scanf("%f", &NOTE_3);
     printf("Quelle est la note numero 4 : ");
     scanf("%f", &NOTE_4);
     printf("Quelle est la note numero 5 : ");
     scanf("%f", &NOTE_5);

/* Calcul de la moyenne */
     m = NOTE_1 + NOTE_2 + NOTE_3;
     m = m + NOTE_4 + NOTE_5;
     m = m/5;
     printf("\nLa moyenne est de : %4.2f",m);

/* Calcul du nombre de notes superieures a la
 * moyenne */
     s = 0;
     if (NOTE_1 >= m) s = s+1;
     if (NOTE_2 >= m) s = s+1;
     if (NOTE_3 >= m) s = s+1;
     if (NOTE_4 >= m) s = s+1;
     if (NOTE_5 >= m) s = s+1;
     printf("\nLe nombre de notes superieures a la");
     printf("\nmoyenne de la classe est de : %d\n",s);
     }
```

Exercice.- Réécrire de même le programme `NOTE_2.c` sans utiliser la notion de tableau.

[Ceci sera à peu près analogue sauf qu'on fera précéder, par exemple, les instructions :

```
printf("Quelle est la note numero 1 :  ");
scanf("%f",&NOTE_1);
```

par `if (1 <= n)`.]

1.6 Exercices

Introduction.- Faites autant d'exercices que nécessaire de la liste suivante afin de vous familiariser avec la notion de tableau.

Exercice 1.- (**Saisie d'un tableau**)
Écrire un programme C qui permet de saisir les éléments d'un tableau.

[Le tableau aura moins de 100 éléments de type réel. On commencera par demander le nombre d'éléments effectif puis les valeurs de chacun des éléments, entrées au clavier.]

Exercice 2.- (**Somme des éléments d'un tableau**)
Écrire un programme C qui permet de calculer la somme des éléments d'un tableau (de moins de 100 éléments d'un type réel).

Exercice 3.- (**Nombre d'éléments nuls d'un tableau**)
Écrire un programme C qui compte le nombre d'éléments nuls d'un tableau d'entiers (de moins de 100 éléments).

Exercice 4.- (**Plus grand élément d'un tableau**)
Écrire un programme C qui donne le plus grand élément d'un tableau d'entiers (de moins de 100 éléments).

Exercice 5.- (**Produit scalaire de deux vecteurs**)
Rappelons qu'étant donnés deux éléments $u = (u_1, ..., u_n)$ et $v = (v_1, ..., v_n)$ de R^n, leur produit scalaire est : $u.v = u_1.v_1 + ... + u_n.v_n$.

Écrire un programme C qui donne le produit scalaire de deux tableaux réels (de moins de 100 éléments) de même dimension.

Exercice 6.- (**Inversion des éléments d'un tableau**)
Écrire un programme C qui, étant donné un tableau, range dans le même tableau les éléments dans l'ordre inverse.

[Par exemple $(3, 6, 7, 2, -1, 0, 4)$ devra donner $(4, 0, -1, 2, 7, 6, 3)$.]

Exercice 7.- (**Calcul des premiers termes d'une suite**)
Écrire un programme C qui demande les valeurs de u_1, de u_2 et de n puis calcule les n premiers termes de la suite $(u_n)_{1\leq n}$ définie par ces valeurs et la relation de récurrence :

$$u_{n+2} = u_{n+1} + u_n,$$

en les rangeant au fur et à mesure dans un tableau.

1.7 Initialisation d'un tableau lors de la déclaration

Introduction.- On peut initialiser un tableau de façon individuelle, élément par élément, par exemple grâce à une boucle. Mais, en langage C, on peut aussi initialiser le tableau lors de sa déclaration.

Syntaxe.- On fait suivre la déclaration du tableau du signe d'égalité puis des valeurs (qui sont des expressions constantes), séparées par des virgules et placées entre accolades :

`type Nom [Entier] =` $\{k_1, k_2, ..., k_{Entier-1}\}$ `;`

Sémantique.- Si le nombre de valeurs est inférieur au nombre d'éléments du tableau, les éléments en trop sont remplis avec des 0 (tout au moins dans le cas de tableaux de nombres).

Exemple.- On peut écrire :

```
int TAB [5] = {1, 34, 10, 24, 6};
```

et on a alors, par exemple, `TAB[2] = 10`, ce que l'on peut vérifier grâce au petit programme suivant :

```
/* init.c */

#include <stdio.h>

void main(void)
     {
     int tab[5] = {1, 34, 10, 24};
     printf("\nTAB[2] = %d", tab[2]);
     printf("\nTAB[3] = %d.\n", tab[3]);
     }
```

2 Les chaînes de caractères comme tableaux

Le type *chaîne de caractères* (*string* en anglais) n'existe pas comme type simple en langage C. Il est émulé par des tableaux de caractères.

2.1 Déclaration des chaînes de caractères

Déclaration.- Une variable chaîne de caractères se déclare comme tableau de caractères. On a, par exemple :

```
char NOM [10];
```

pour une chaîne d'au plus neuf caractères.

Nous allons voir ci-dessous pourquoi on dispose d'au plus neuf caractères et non dix comme on pourrait le penser *a priori* à juste titre.

Le problème de la longueur effective.- Comme pour tout tableau, qui est un type de données statique, on doit, d'une part, déclarer une dimension maximum et, d'autre part, indiquer la longueur effective pour connaître la fin de la chaîne de caractères.

Par exemple après saisie de "Paul" on peut se retrouver avec :

P	a	u	l	f	4	"	&	z	d

Lorsqu'on voudra afficher, par l'instruction (en fait la fonction) `printf`, il faudra bien savoir jusqu'à qu'elle lettre afficher, d'où l'importance de connaître la dimension effective du tableau.

Solution en C.- En langage C on termine la chaîne par le **caractère nul** (de numéro 0 dans le code ASCII, donc le caractère '\0'), qu'il ne faut pas confondre avec le caractère blanc. "Paul" sera donc représenté par :

P	a	u	l	\0	4	"	&	z	d

Conséquence.- Une chaîne de caractères de longueur n aura donc besoin d'un tableau de $n+1$ caractères au minimum. Ceci explique la déclaration ci-dessus.

2.2 Copie de chaînes de caractères

Affectation.- Comme c'est le cas général pour les tableaux, il n'existe pas d'affectation globale pour les chaînes de caractères. On doit donc l'émuler grâce à un programme en copiant caractère par caractère.

Les exercices suivants montrent quelques cas d'une telle émulation.

Exercice 1.- (**Lecture et affectation d'une chaîne de caractères**)

Écrire un programme C `lecture.c` pour une plate-forme MS-DOS qui lit une chaîne de caractères au clavier (sans utiliser la fonction `scanf`) et l'affecte dans une variable NOM, dans une première étape, puis l'affiche à l'écran, dans une seconde étape.

[Saisir caractère par caractère en utilisant la fonction `getche` déclarée dans `conio.h`. Remplacer le retour chariot '\n' (saisi comme '\r sur MS-DOS, ne l'oubliez pas) par '\0'.]

Exercice 2.- (**Longueur d'une chaîne de caractères**)

Écrire un programme C `longueur.c` qui lit une chaîne de caractères (dans une première étape) puis calcule (et affiche) sa longueur, dans une seconde étape.

Exercice 3.- (**Affichage d'une chaîne de caractères**)

Écrire un programme C `affiche.c` qui lit une chaîne de caractères, dans une première étape, puis l'affiche (sans utiliser la fonction `printf`), dans une seconde étape.

Exercice 4.- (**Inversion d'une chaîne de caractères**)

Écrire un programme C `inverse.c` qui lit une chaîne de caractères, la transforme en la chaîne inverse puis affiche cette chaîne inverse. Par exemple '*essor*' donnera '*rosse*'.

Exercice 5.- (**Détermination des palindromes**)

Écrire un programme C `palin.c` qui lit une chaîne de caractères et qui détermine s'il s'agit d'un palindrome.

[Un **palindrome** est une chaîne de caractères qui se lit de la même façon à l'endroit qu'à l'envers, tel que 'eve'.]

Exercice 6.- Écrire un programme C `capitale.c` qui lit une chaîne de caractères, convertit toutes les lettres minuscules de cette chaîne en majuscules et affiche le résultat.

[On simplifiera en ne changeant ni les signes spéciaux (évidemment) ni les lettres avec des signes diacritiques.]

Exercice 7.- Écrire un programme C `index.c` qui affiche la liste des mots d'un texte entré comme une chaîne de caractères.

[On supposera que les seuls caractères qui peuvent séparer deux mots sont l'espace, la virgule, le point-virgule et le point.]

2.3 Initialisation globale des chaînes de caractères

Première méthode.- Nous pouvons initialiser la chaîne de caractères `NOM` de l'une des façons habituelles pour les tableaux, à commencer par l'initialisation lors de la déclaration :

```
char NOM [10] = {'P','a','u','l','\0'};
```

Deuxième méthode.- On peut auusi faire une affectation caractère par caractère :

```
char NOM [10];
NOM[0] = 'P';
NOM[1] = 'a';
NOM[2] = 'u';
NOM[3] = 'l';
NOM[4] = '\0';
```

Troisième méthode.- Nous avons déjà vu aussi une façon plus globale :

```
char NOM [10] = "Paul";
```

qui possède l'avantage supplémentaire de ne pas nous faire gérer la fin de la chaîne.

2.4 Entrées/sorties globales

Introduction.- Il existe heureusement des fonctions standard pour saisir et afficher globalement les chaînes de caractères.

Printf et scanf.- Nous avons déjà rencontré les fonctions `printf` et `scanf` avec le format `%s`. Le nom d'une variable chaîne de caractères ne doit pas être précédé du symbole d'adressage dans le cas de la lecture (car, comme nous le verrons plus loin, NOM correspond en fait à &NOM[0]). Il est évidemment inutile de gérer le caractère nul. On a, par exemple :

```
char NOM [10];
printf("Quel est votre nom ?  ");
scanf("%s",NOM);
printf("Bonjour %s", NOM);
```

Formatage.- Une constante entière placée entre '%' et 's' permet d'indiquer le nombre maximum de caractères à saisir ou à afficher. Les autres caractères éventuels sont perdus. On a, par exemple :

```
scanf("%9s",NOM);
```

Un inconvénient.- La fonction `scanf` lit une chaîne de caractères jusqu'au premier blanc rencontré, autrement dit elle lit un mot. On ne peut donc pas entrer "Paul DUBOIS" par exemple.

On peut en fait paramétrer la fonction `scanf`. Nous renvoyons à la documentation pour cela.

Les fonctions gets et puts.- Pour lire et afficher sur l'entrée et la sortie standard (le clavier et l'écran) une chaîne de caractères quelconques on peut utiliser les fonctions `gets` et `puts`. La saisie se termine par le caractère '\n'. De même l'affichage se termine par un retour à la ligne. On a par exemple :

```
gets(PRENOM_ET_NOM);
puts("Bonjour");
puts(PRENOM_ET_NOM);
```

2.5 Fonctions concernant les chaînes de caractères

Introduction.- Il existe des fonctions concernant les chaînes de caractères dont les déclarations se trouvent dans le fichier en-tête `string.h`, implémentant certaines fonctions que nous avons déjà vues.

Liste.- Les principales fonctions sont :

- 1°) l'*affectation globale* (ou *copie*) dont la syntaxe est :

```
strcpy(nom1,nom2);
```

La sémantique est celle qui est attendue : la valeur de l'expression `nom2` est copiée dans la variable `nom1`.

Aucune vérification n'est faite quant à la longueur maximum déclarée de `nom1`. C'est au programmeur de s'assurer que `nom2` a une longueur inférieure à la longueur maximum de `nom1`.

- 2°) la *concaténation* (partielle) dont la syntaxe est :

```
strcat(nom1,nom2);
```

Il s'agit d'une concaténation restreinte (en fait d'un *ajout*) qui copie dans la variable `nom1` la valeur de la variable `nom1` suivie de la valeur de l'expression `nom2`, en enlevant bien entendu le '\0' de la valeur de `nom1`.

- 3°) la longueur, de syntaxe :

```
strlen(nom)
```

qui retourne un entier, à savoir la longueur effective de `nom`.

- 4°) la *comparaison*, dont la syntaxe est :

```
strcmp(nom1,nom2);
```

qui donne la valeur entière strictement négative, nulle ou strictement positive suivant que `nom1` précède `nom2` dans l'ordre lexicographique, que `nom1` est égal à `nom2` ou que `nom1` suit `nom2` dans l'ordre lexicographique.

Exercice 1.- Tester ces fonctions en les insérant dans des programmes complets.

Exercice 2.- Écrire une fonction de concaténation complète :

```
void strconcat(nom1,nom2,nom3);
```

qui place dans la variable `nom1` la valeur de `nom2` suivie de la valeur de `nom3`.

Exercice 3.- Écrire le corps des fonctions ci-dessus sans utiliser le fichier en-tête `string.h`.

3 Tableaux à plusieurs dimensions

3.1 Notion

Notion intuitive.- Un **tableau à deux dimensions** est formé d'éléments tous de même nature et repérés par deux **index** (et non plus un seul).

Représentation graphique.- Un tableau à deux dimensions est souvent représenté par un certain nombre de **lignes** et de **colonnes**. Un élément est repéré par son numéro de ligne et son numéro de colonne.

Formalisation mathématique.- Un tableau à deux dimensions est ce qu'on appelle une **matrice** en Mathématiques : un élément d'un tel tableau est repéré par deux indices et se note souvent $a_{i,j}$, les indices étant i et j.

Tableaux à plusieurs dimensions.- On a de même des tableaux à trois, quatre... dimensions.

3.2 Mise en place en langage C

Introduction.- La notion de tableau à plusieurs dimensions n'existe pas en langage C. On l'émule en utilisant des tableaux de tableaux.

Déclaration.- On déclare plusieurs dimensions au lieu d'une seule :

```
type Nom [dimension1] [dimension2] ... [dimensionn];
```

où `dimension1`, `dimension2`, ... , `dimensionn` sont des expressions constantes entières.

Accès aux éléments.- Les index sont également placés entre crochets :

```
Nom[index1][index2] ... [indexn]
```

Initialisation lors de la déclaration.- On peut initialiser un tableau à plusieurs dimensions lors de sa déclaration. En se rappelant que c'est un tableau de tableaux on commence par donner les éléments avec 0 pour premier index, puis ceux avec 1 pour premier index...
Par exemple avec la déclaration suivante :

```
int TAB [2][3] = {1,2,3,4,5,6};
```

on a :

```
TAB[0][0] = 1, TAB[0][1] = 2, TAB[1][2] = 6.
```

Pour des raisons de lisibilité on peut aussi introduire des paires d'accolades supplémentaires :

```
int TAB [2][3] = { {1, 2, 3 }, { 4, 5, 6} };
```

ce qui permet de bien distinguer les différentes colonnes mais n'a pas d'influence sur le compilateur.

L'utilisation de ce type d'initialisation est cependant à proscrire.

3.3 Un exemple : triangle de Pascal

Rappels.- Les *coefficients binomiaux* sont définis pour des entiers naturels n et p par :

$$C_n^p = \frac{n!}{p!.(n-p)!} \; si \; p \leq n, 0 \; sinon.$$

Le *triangle de Pascal* correspondant à un entier naturel N est le tableau formé des coefficients binomiaux C_n^p pour p et n inférieurs à N. On n'écrit pas les coefficients nuls d'où l'aspect d'un triangle.
Ce triangle se construit rapidement en remarquant que $C_0^p = 0$ pour $p \neq 0$, que $C_n^0 = 1$ et en utilisant la *formule de Pascal* :

$$C_{n+1}^{p+1} = C_n^{p+1} + C_n^p.$$

Premier programme.- Écrivons un programme qui demande un entier naturel plus petit que 100, construit le triangle de Pascal correspondant à cet entier puis permet l'interrogation de plusieurs valeurs de ce tableau en terminant par la valeur sentinelle $n = p = 0$.

```
/* pascal.c */
#include <stdio.h>
void main(void)
     {
     const dim = 101;
     int m, n, p;
     int CNP [dim] [dim];

/* Construction du tableau */
     printf("Dimension du tableau M = ");
     scanf("%d",&m);
     for (n = 0; n <= m; n++)
         {
         CNP[n][0] = 1;
         CNP[n][n] = 1;
         for (p = n+1; p <= m; p++) CNP[n][p] = 0;
         }
     for (n = 2; n <= m; n++)
     for (p = 1; p < n; p++)
     CNP[n][p] = CNP[n-1][p-1] + CNP[n-1][p];

/* Interrogation du tableau */
     do
       {
       printf("C(");
       scanf("%d", &n);
       printf(",");
       scanf("%d", &p);
       printf(") = %d \n", CNP[n][p]);
       }
```

```
     while ((n != 0) || (p != 0));
     }
```

Une séance d'utilisation ressemblera à :

```
Dimension du tableau M = 20
C[3,2] = 3
C[10,1] = 10
C[10,4] = 210
C[15,3] = 455
C[16,15] = 15504
C[0,12] = 0
C[0,0] = 1
```

Deuxième programme.- Écrivons un programme qui demande un entier naturel n et qui calcule alors $\sum_{k=0}^{n} k^2.C_n^k$.

```
/* pascal2.c */
#include <stdio.h>
void main(void)
     {
     const dim = 101;
     int m, n, p, S;
     int CNP [dim][dim];

/* Construction du tableau */
     printf("n = ");
     scanf("%d", &m);
     for (n = 0; n <= m; n++)
         {
         CNP[n][0] = 1;
         CNP[n][n] = 1;
         for (p = n+1; p <= m; p++) CNP[n][p] = 0;
         }
     for (n = 2; n <= m; n++)
     for (p = 1; p < n; p++)
         CNP[n][p] = CNP[n-1][p-1] + CNP[n-1][p];

/* Calcul de la somme des p*p*Cnp */
     S = 0;
     for (p = 1; p <= m; p++) S = S + p*p*CNP[m][p];
     printf("S = %d \n", S);
     }
```

On obtient par exemple :

```
n = 10
S = 28160.
```

3.4 Exercices

Exercice 1.- (**Vérification de l'associativité**)

Considérons une loi de composition interne définie sur un ensemble fini. Il est facile de vérifier si elle est commutative ou si elle admet un élément neutre. Il est possible, mais plus pénible, de vérifier si elle est associative. Ceci est le cas, par exemple, pour la loi suivante de la figure ci-dessous.

*	e	a	b	c
e	e	a	b	c
a	a	b	c	e
b	b	c	e	a
c	c	e	a	b

Table 1: Exemple de loi

On peut, pour cela, s'aider d'un programme, les éléments de l'ensemble étant numérotés.

Écrire un tel programme en C qui demande le nombre d'éléments puis la table de Pythagore et répond si la loi est associative ou non.

Exercice 2.- (**Opérations sur les matrices**)

Écrire un programme en C qui demande quelle opération sur les matrices à coefficients réels on veut effectuer parmi les suivantes : l'addition, la transposition, la multiplication par un scalaire et la multiplication ; il doit permettre ensuite la saisie du nombre adéquat de matrices puis afficher le résultat.

4 Pointeur et tableau

Nous n'avons pas, jusqu'ici, utilisé de tableau comme argument d'une fonction. Nous allons voir comment mettre en place ceci.

4.1 Pointeur comme index

4.1.1 Notion

Introduction.- En langage C, l'accès à un élément d'un tableau peut se faire classiquement par le nom du tableau accompagné d'un index, mais aussi par un pointeur manipulé par des opérations spécifiques, comme nous allons le voir.

Exemple.- Considérons le tableau :

```
int NOTE [30];
```

On peut accéder à la première note par un indice, par exemple :

```
NOTE[0] = 10;
```

mais on peut aussi utiliser un pointeur :

```
int *PTR;
PTR = &NOTE[0];
*PTR = 10;
```

C'est plutôt une façon compliquée de faire mais c'est possible d'après ce que nous savons de l'adressage indirect.

4.1.2 Addition des pointeurs

Introduction.- Pour accéder à l'élément d'indice 3 on peut continuer par :

```
NOTE[3] = 12;
```

mais aussi, et ceci est nouveau, par :

```
PTR = PTR + 3;
*PTR = 12;
```

On a utilisé l'**addition des pointeurs**, c'est-à-dire la possibilité d'ajouter à un pointeur une expression entière n. On augmente ainsi l'adresse, non pas de n (pour passer au n-ième octet, ou plus petit élément adressable, suivant), mais du nombre nécessaire, suivant le type pointeur, pour que l'adresse corresponde au début du n-ième élément suivant.

Syntaxe.- La syntaxe de l'addition des pointeurs est :

```
pointeur_1 = pointeur_2 + entier;
```

où `entier` désigne une expression entière.

Incrémentation.- Dans le cas d'une addition de un, on peut utiliser les abréviations usuelles du C, à savoir `PTR++` ou `++PTR`.

4.1.3 Soustraction des pointeurs

Introduction.- On peut utiliser de même la soustraction sur les pointeurs. Par exemple :

```
PTR = PTR - 2;
```

nous ramène à l'élément d'indice 1 du tableau.

Syntaxe.- La syntaxe de la soustraction des pointeurs est :

```
PTR_1 = PTR_2 - entier;
```

ou :

```
I = PTR_1 - PTR_2;
```

où `entier` est une expression entière et `I` une variable entière.

Sémantique.- Nous avons vu l'intérêt de la première forme pour revenir à des éléments antérieurs dans un tableau. La deuxième forme permet de savoir de combien d'éléments sont séparées deux adresses.

Décrémentation.- On peut utiliser la **décrémentation** `PTR--` ou `--PTR`.

4.1.4 Nom de tableau et pointeur

Introduction.- En langage C le nom d'un tableau, par exemple `NOTE`, correspond à l'adresse de son premier élément `&NOTE[0]`. Ce n'est cependant pas un pointeur, ou plus exactement une variable pointeur. Il s'agit d'un **pointeur constant** dont on ne peut pas modifier le contenu (ce n'est pas une *L-value*). On ne peut pas écrire, par exemple :

```
NOTE = NOTE + 2;
```

pour accéder à l'élément d'indice 2. Il faut, pour cela, passer par une variable pointeur auxiliaire :

```
PTR = NOTE;
PTR = PTR + 2;
```

ou même :

```
PTR = NOTE + 2;
```

Exercice.- Écrire un programme C permettant de saisir au clavier un tableau d'entiers puis de vérifier s'il s'agit d'un tableau palindrome ou non.

On utilisera des pointeurs et non pas des index.

4.2 Tableau comme paramètre de fonction

Introduction.- En langage C les tableaux sont toujours transmis par adresse à une fonction. Par exemple pour déclarer une fonction qui trie un tableau d'entiers de dimension effective n on écrira :

```
void TRI(int *TAB, int n);
```

ou mieux (mais c'est la même chose) :

```
void TRI(int TAB[], int n);
```

Commentaires.- 1°) `TAB` est un pointeur sur un entier, sur un seul entier. C'est normal puisque nous avons vu que le nom d'un tableau, par exemple `TAB`, correspond à l'adresse `&TAB[0]` de son premier élément.

2°) On passe à la fonction `TRI` l'adresse du premier élément du tableau et la dimension effective du tableau mais pas sa dimension maximum. En effet celle-ci n'est pas indispensable : le tableau que l'on passera comme argument est déclaré par ailleurs et c'est à ce moment que l'on déclare sa dimension maximum. Il faut cependant que la dimension effective soit inférieure à la dimension maximum pour ne pas avoir de problème lors de l'exécution.

Autre notation.- Au lieu de :

```
void TRI(int *TAB, int n);
```

on écrit quelquefois :

```
void TRI(int TAB[], int n);
```

qui ressemble plus à un tableau. On peut aussi écrire :

```
void TRI(int TAB[100], int n);
```

mais, comme nous l'avons déjà dit, la déclaration de la dimension maximum (ici 100) n'a absolument aucune sorte d'incidence, seule l'adresse du premier élément étant retenue.

Exercice.- 1°) Écrire le corps de cette fonction, en utilisant la méthode de tri que vous voulez.

2°) Écrire des fonctions SAISIE et AFFICHE.

3°) Écrire un programme complet.