Chapitre 12

Les Tableaux (Listes, Matrices)

Le langage C permet de créer des tableaux, nommés parfois listes, parfois matrices. Ce chapitre va nous permettre d'apprendre à les créer, et à les utiliser, tout en nous montrant quelques exemples possibles de leur utilité.

I:\ Notions de bases au sujet des tableaux

Au cours de cette partie, nous commencerons par définir le terme de "tableaux" appliqué au langage de programmation C. Ensuite, nous verrons à quoi ils peuvent nous servir lorsque nous développons un programme.

A: Tout d'abord, quelques mots de vocabulaire

Un tableau est une variable constituée de plusieurs cases d'un type donné, chacune capable de mémoriser une information du type en question.

Un tableau à une dimension est constitué d'une seule ligne et de plusieurs colonnes (ou, selon le point de vue, de l'inverse). Un tableau à deux dimensions est constitué de plusieurs lignes et de plusieurs colonnes.

Le terme de liste est souvent utilisé pour désigner un tableau à une dimension. Le terme de matrice est parfois utilisé, comme en mathématiques, pour désigner un tableau à plus de une dimension

B: Quelques exemples classiques d'utilisation possible de tableaux

Il est assez fréquent d'avoir à utiliser de nombreuses données d'un même type... et il n'est pas vraiment pratique d'utiliser un grand nombre de variables différentes : premièrement, il faut trouver des noms pour toutes ces variables, mais, en plus, avoir des variables différentes ne permet pas de toutes les traiter dans une boucle.

Un tableau permet de remédier à ces deux problèmes : plutôt que d'utiliser de nombreuses variables nommées a1, a2, a3, ..., a150, autant utiliser une seule variable, nommée, par exemple, 'a', qui contienne les 150 données que nous aurions placé dans nos variables aX.

Une application possible des tableaux, que nous prendrons d'ailleurs comme exemple à la fin de ce chapître, est le tri de données : on place des données dans le désordre dans un tableau, et on lance un algorithme de tri sur celui-ci ; à la fin de l'exécution de l'algorithme, le tableau contiendra les données triés, que nous pourrons récupérer grâce à une boucle parcourant les cases du tableau, les unes après les autres.

II:\ Tableaux à une dimension : Listes

Nous allons commencer par étudier la création, ainsi que l'utilisation, de tableaux à une dimension, communément appelés Listes.

A: Déclaration

Pour pouvoir utiliser un tableau, il nous faut commencer par, comme pour les autres types de variables, le définir. Pour cela, on utilise la syntaxe suivante :

```
TYPE nom_variable[NOMBRE_ELEMENTS];
```

TYPE désigne le type des éléments que l'on souhaite ranger dans notre tableau.
NOMBRE_ELEMENTS correspond au nombre de cases que contiendra le tableau ; notez que ce nombre doit être fixé au moment de la compilation, du moins en standard ANSI.

Par exemple, pour déclarer une liste capable de contenir 10 entiers, nous utiliserons cette syntaxe :

```
short tab1[10];
```

Naturellement, cette syntaxe peut être réutilisée pour tous les types de variables que nous avons déjà vu, ainsi que pour ceux que nous serons amené à étudier dans le futur.

B: Initialisation

Il est possible, comme pour les autres types de variables que nous avons jusqu'à présent eu l'occasion d'étudier, d'initialiser un tableau lors de sa création, c'est-à-dire, d'affecter des valeurs à ses cases, en partant de la gauche.
Pour cela, on utilise une syntaxe de la forme suivante :

```
TYPE nom_variable[NOMBRE_ELEMENTS] = {case1, case2, case3, case4};
```

Naturellement, il faut que le nombre d'éléments compris dans la liste d'initialisation soit inférieur ou égal au nombre d'éléments que peut contenir le tableau. Dans le cas contraire, le compilateur nous renverra un warning.
Si la liste d'initialisation comporte moins d'éléments que le tableau ne peut en contenir, celui-ci sera rempli à partir du début, et les dernières cases, non initialisées, contiendront une valeur indéterminée.

Voici un exemple correspondant à l'initialisation d'un tableau d'entiers lors de sa déclaration :

```
short tab3[3] = {1, 2, 3};
```

Comme on peut selon toute logique s'y attendre, la première case du tableau tab3 prendra pour valeur 1, la seconde pour valeur 2, et la troisième et dernière aura pour valeur 3.

C: Utilisation

L'accès aux éléments d'un tableau se fait par indice : lorsque l'on le numéro de la case dans laquelle se trouve une information, le C nous permet d'accéder à celle-ci.
Notez que l'indice d'un élément dans un tableau est toujours compris entre 0 pour le premier élément du tableau, et (nombre d'éléments moins un) pour le dernier. Je me permet d'insister sur le fait que le premier élément d'un tableau est à la case d'indice 0, et non pas 1 comme dans d'autres langages, tels le TI-BASIC !
Si vous dépassez ces bornes, vous sortirez du tableau ; en lecture, vous obtiendrez des données indéterminées ; en écriture, cela peut conduire à un plantage de votre programme. Soyez donc extrêmement prudents, notamment lorsque vous utiliserez une boucle pour parcourir un tableau.

Pour accéder à une case d'un tableau, on utilise la syntaxe suivante :

```
nom_du_tableau[indice]
```

C'est l'opérateur "`[]`" (un crochet ouvrant, l'indice, un crochet fermant) qui est utilisé pour signifier l'indexation de tableau. "indice" correspond à un nombre entier, supérieur ou égal à 0, et désigne la case du tableau sur laquelle on souhaite travailler.

Voici quelques exemples d'opérations que nous pouvons effectuer avec un tableau :

```
// Création d'un tableau de 6 entiers.
// On n'initialise que les 4 premières cases du tableau.
short tab[6] = {1, 2, 3, 4};
short i = 1;

tab[4] = 15; // on met 15 dans l'avant dernière case
tab[5] = 16; // et 16 dans la dernière
                //(qui a pour indice nombre de cases moins un = 6-1 = 5)

if(tab[2] == 3)
{
    // Code à exécuter si la 3ème case du tableau vaut 3.
    // (ce qui est ici le cas)
    printf("Coucou !\n");
}

// Affichage de la valeur de la case d'indice i
// Autrement dit, ici, de la case d'indice 1
// (la seconde case du tableau)
printf("Valeur de la case d'indice %d : %d\n", i, tab[i]);
```

Comme nous pouvons le constater, on utilise toujours la même syntaxe ; la seule chose à laquelle penser est qu'il faut éviter le débordement d'indice, pour ne pas sortir du tableau, et que, en C, les indices commencent à 0 !

D: Tableaux et pointeurs

En fait, si l'on se penche d'un peu plus près sur ce qui est en mémoire, un tableau n'est rien de plus qu'une série d'emplacements mémoire consécutifs.
Donc, si on connaît l'adresse de la première case du tableau, et qu'on la stocke dans un pointeur, il est possible, en incrémentant ce pointeur comme nous l'avons vu au chapitre précédent, de le faire pointer successivement sur toutes les cases du tableau.
Pour les exemples que nous utiliserons dans cette partie, nous considérerons que nous avons déclaré le tableau tab et le pointeur p comme ceci :

```
short tab[5] = {1, 2, 3, 4, 5};
short *p;
```

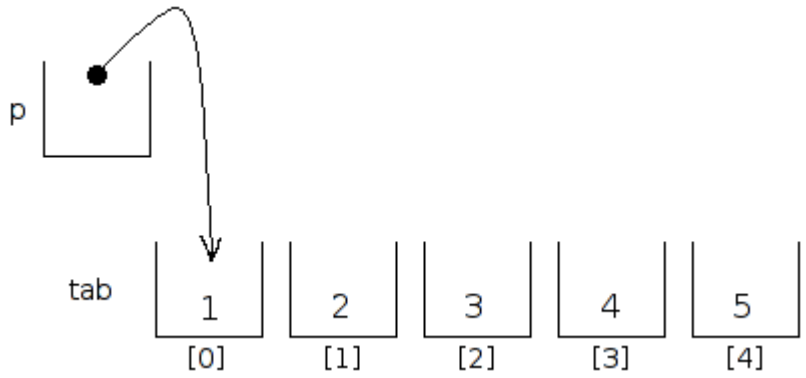
Commençons par faire pointer le pointeur p sur une des cases du tableau ; pour notre exemple, nous choisissons de le faire pointer sur la première case de celui-ci.
Pour cela, il nous suffit d'appliquer l'opérateur de référence à la première case du tableau, comme ceci :

```
p = &tab[0];    // p pointe sur la première case du tableau
```

Et si nous essayons d'afficher la donnée pointée par le pointeur, comme ceci :

```
printf("*p = %d\n", *p);
```

Nous constaterons que la valeur affichée est 1, ce qui correspond effectivement au contenu de la première case de notre tableau.
Voici un schéma représentant ce que nous venons de faire :

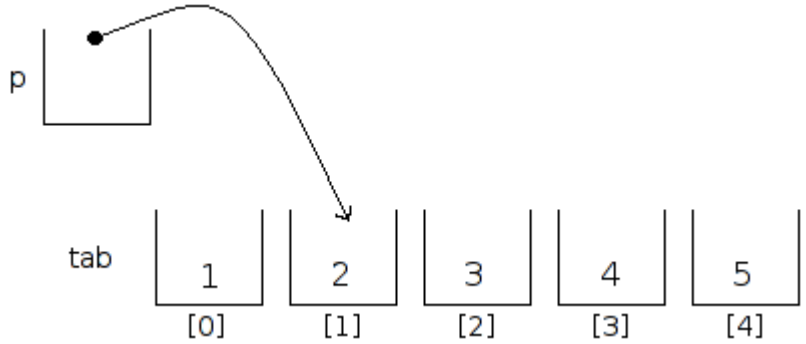


A présent, effectuons un petit peu d'arithmétique de pointeurs, et ajoutons 1 à notre pointeur, ce qui revient à le faire pointer sur l'emplacement suivant en mémoire.
Le C garantissant qu'un tableau est représenté en mémoire par une suite d'emplacements mémoires successifs, incrémenter de un notre pointeur, qui pointait sur la première case du tableau, revient à le faire pointer sur la seconde case du même tableau.
Voici la portion de code source correspondante, suivie d'un affichage, pour vérifier que *p vaut bien la valeur contenue dans la seconde case :

```
p++;
printf("*p = %d\n", *p);
```

Nous obtiendrons 2 comme valeur affichée, c'est-à-dire, la valeur de la seconde case du tableau, comme nous le souhaitions.

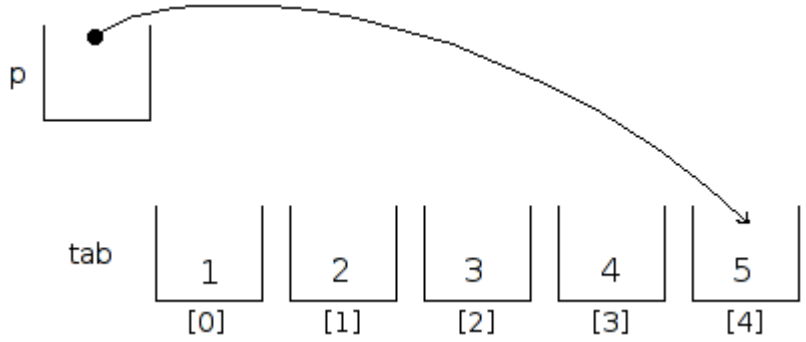
Et la représentation schématique correspondant à ce que nous venons de faire :



A présent, nous allons nous prouver que nous ne sommes pas forcé de n'incrémenter que de 1, et que nous pouvons utiliser d'autres valeurs, pour nous déplacer ailleurs dans le tableau.
Par exemple, allons voir trois cases plus loin :

```
p += 3;
printf("*p = %d\n", *p);
```

A l'affichage, nous verrons la valeur 5, qui correspond à la case d'indice 4... c'est-à-dire, à la cinquième, et dernière, case de notre tableau. Ce qui, schématiquement représenté, équivaut à ceci :



Naturellement, il est possible, au lieu d'incrémenter le pointeur pour passer à l'une des cases suivantes, de le décrémenter pour passer à une des cases précédentes.

Lorsque vous utilisez un pointeur pour accéder aux différentes cases d'un tableau, il vous faudra penser à vous assurer que vous ne sortez pas du tableau. Autrement, le comportement de votre programme sera indéfini.
(Sur nos machines, un accès en dehors d'un tableau en lecture se traduit par l'obtention de données invalides, et, en écriture, en règle général, par un plantage dans le futur.)
Cela dit, lorsque l'on travaille avec les indices des cases du tableau, il nous faut nous assurer exactement de la même chose.

Pour en finir avec l'accès aux listes par pointeurs, je tiens à préciser que la variable déclarée comme un tableau est en fait elle-même un pointeur sur la première case de celui-ci.
Dans notre exemple, donc, l'écriture

```
*tab = 10;
```

est strictement équivalente à l'écriture

```
p = &tab[0];
*p = 10;
```

Dans les deux cas, on mémorise la valeur 10 dans la première case du tableau.
Notons que la seconde écriture pourrait avoir été écrite sans utiliser de pointeur temporaire, de cette manière :

```
*(&tab[0]) = 10;
```

Mais cela n'est pas forcément très lisible, en particulier pour un débutant non encore habitué à la syntaxe des pointeurs.

Nous étudierons, au chapitre suivant, un genre particulier de tableaux, que nous aurons tendance à parcourir en utilisant un pointeur, plutôt que des indices de cases, du fait que, dans ce cas là, l'écriture que nous venons d'introduire a tendance à être plus concise que la seconde.

III:\ Deux exemples complets utilisant des Listes

Pour terminer notre étude concernant les tableaux à une dimension, nous allons étudier deux petits exemples,

A: Exemple 1

Pour notre premier exemple, nous allons remplir un tableau avec des nombres entiers générés aléatoirement, puis nous afficherons le contenu des cases de celui-ci.

La génération de nombres aléatoires se fait en utilisant la fonction dont le prototype est le suivant :

```
short random (short num);
```

Cette fonction prend en paramètre un entier, et renvoie un entier dont la valeur sera supérieure ou égale à 0, et inférieure strictement à celle passée en paramètre.
Par exemple, pour générer un nombre aléatoire compris entre 0 et 9 inclus, on utilisera ceci :

```
random(10);
```

Voici à présent le code source de notre exemple ; quelques explications suivent celui-ci, mais ne devraient pas être utiles, étant donné que nous avons déjà étudié tout ce qui est utilisé au sein de l'exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[6];
    int i;

    for(i=0 ; i<6 ; i++)
    { // Parcours de tout le tableau, case par case.
        tab[i] = random(100);  // On affecte à la case
courante un nombre aléatoirement sélectionné entre 0 et 99 compris.
    }

    clrscr();
    for(i=5 ; i>=0 ; i--)
    { // Parcours du tableau en sens inverse
        printf("case %d => %d\n", i, tab[i]);
    }

    ngetchx();
}
```

Exemple Complet

Pour résumer en quelques mots ce que fait ce programme : on commence par déclarer un tableau de six entiers. On affecte ensuite à chacune des cases de celui-ci une valeur aléatoire comprise entre 0 et 99 compris.
Puis on finit par une boucle qui parcourt le tableau en partant de la fin, en affichant pour chaque case son indice, et la valeur contenue.

B: Exemple 2

Notre second exemple va partir de l'idée illustrée au cours du précédent, à savoir, remplissage d'un tableau à l'aide de valeurs générées aléatoirement, mais ira plus loin, puisque nous trierons ces valeurs avant de les afficher.
L'algorithme de tri que nous utiliserons est un des moins efficaces qui soit, mais il présente l'avantage d'être extrêmement simple, et rapide à écrire.
Voici le code source de l'exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[10];
    short i, j;
    short temp;

    for(i=0 ; i<10 ; i++)
    { // On remplit le tableau avec des données aléatoires
        tab[i] = random(20);
    }

    clrscr();
    printf("Données non triées :\n");
    for(i=0 ; i<10 ; i++)
    { // On affiche les données (non triées) contenues dans le tableau
        printf("%d ", tab[i]);
    }
    ngetchx();


    // Tri des données par ordre croissant :
    for(i=0 ; i<10 ; i++)
    {
        for(j=i+1 ; j<10 ; j++)
        {
            if(tab[j] < tab[i])
            { // On inverse
                temp = tab[i];
                tab[i] = tab[j];
                tab[j] = temp;
            }
        }
    }

    printf("\nDonnées triées :\n");
    for(i=0 ; i<10 ; i++)
    { // On affiche les données (qui, maintenant, sont triées) contenues dans le tablea
u
        printf("%d ", tab[i]);
    }
    ngetchx();

}
```

Exemple Complet

Le fonctionnement du programme est assez basique, divisé en trois parties : tout d'abord, on crée un tableau, et le remplit de valeurs générées aléatoirement ; ensuite, on l'affiche, pour montrer que les valeurs sont bien dans le désordre ; puis, on le trie ; et finalement, on le réaffiche, pour montrer que les valeurs sont à présent dans l'ordre.
Le principe de l'algorithme de tri utilisé est, dans les grandes lignes, le suivant : on parcoure le

tableau case par case, et, à chaque itération de la boucle de parcours, on s'assure que la case suivante contienne la plus petite valeur restant en partie droite du tableau ; autrement dit, à chaque itération, les cases à gauche de la case courante sont triées, et celles à droite de la case courante restent à trier.

IV:\ Tableaux à plusieurs dimensions

Le C ne permet pas de créer que des listes : il est en effet possible de créer des tableaux à "plusieurs dimensions", qui rentrent donc véritablement dans la définition de tableaux.
Bien qu'il soit possible de créer des tableaux à autant de dimensions que l'on veut, il est rarement que l'on ait besoin de plus de deux dimensions ; nous nous consacrerons donc, au cours de cette partie, aux tableaux à deux dimensions, communément appelés matrices.

A: Création d'un tableau à plusieurs dimensions

Voici la syntaxe qu'il convient d'utiliser pour déclarer un tableau à plusieurs dimensions :

```
TYPE            nom_du_tableau[nombre_elements_dimension_1][nombre_elements_dimension_2]
[nombre_elements_dimension_3];
```

Cette syntaxe correspond au cas d'un tableau à trois dimensions, mais, naturellement, il suffit de l'adapter pour déclarer un tableau du nombre de dimensions que l'on souhaite. Par exemple, pour déclarer une matrice composée d'entiers, organisés en 10 lignes et 3 colonnes, on écrira ceci :

```
short  tab[10][3];
```

Cela dit, il nous aurait été tout à fait possible d'écrire l'inverse, soit ceci :

```
short  tab[3][10];
```

On aurait aussi obtenu un tableau de 10 lignes et 3 colonnes... organisées autrement en mémoire, mais capable de contenir la même quantité de données.
En fait, le C ne défini pas les termes de "ligne" et de "colonne" : c'est à nous, lorsque nous travaillons avec des tableaux, de nous souvenir de ce que nous considérons comme "ligne" et comme "colonne", notamment lorsque nous souhaitons y accéder. Personnellement, j'ai l'habitude de ceci :

```
TYPE nom_du_tableau[nombre_de_lignes][nombre_de_colonnes];
```

A vous de choisir la logique que vous préférez... et de vous y tenir pour ne pas vous embrouiller.

B: Utilisation d'un tableau à plusieurs dimensions

Pour utiliser un tableau à plusieurs dimensions, on utilisera le même type de syntaxe que pour les listes ; la seule différence sera que l'on utilisera non pas un indice désignant la case dans la liste, mais plusieurs ; à savoir, un par dimension.
Par exemple, pour accéder à la seconde case de la troisième ligne d'un tableau à deux dimensions (en considérant que la première dimension correspond à la ligne, et la seconde à la colonne) et y stocker la valeur 123, on utilisera cette syntaxe :

```
short tab[10][3];   // Tableau à 10 lignes et 3 colonnes
                        // (puisque je considère que la première dimension correspond
aux lignes, et la seconde aux colonnes)

tab[2][1] = 123;    // On enregistre la valeur 123
                        // à la seconde case de la 3ème ligne
```

Naturellement, tout comme pour les listes, les indices commencent à 0, et se terminent à nombre_d_éléments moins 1. Ce qui, après tout, est logique, puisqu'une matrice n'est finalement rien de plus, en mémoire, qu'une liste de listes.

C: Un petit exemple d'utilisation d'une matrice

Ci-dessous, un petit exemple qui, dans une matrice de 10 lignes et 3 colonnes, mémorise les tables de multiplication de 5, 6, et 7. Encore une fois, nous aurions pu nous dispenser d'utiliser un tableau pour un programme aussi simple, et calculer la table de multiplication lors de l'affichage (plutôt que la calculer à un moment, et l'afficher plus tard)...
Cela dit, encore une fois, un exemple bref, même s'il est simple et qu'il n'utilise pas forcément les tableaux de façon judicieuse, permet, à mon avis, de mieux présenter un sujet qu'un long exemple dans lequel on se perdrait.
Voici le code source de cet exemple :

```
#include <tigcclib.h>

void _main(void)
{
    short tab[10][3];  // Tableau à 10 lignes et 3 colonnes
                       // (puisque je considère que la première dimension correspond au
x lignes, et la seconde aux colonnes)
    short i=0,
          j=0;

    for(j=0 ; j<3 ; j++)
    {
        for(i=0 ; i<10 ; i++)
        {
            tab[i][j] = (i+1)*(j+5);   // i+1 car on veut que la table de multiplicatio
ns
                                       //aille de 1 à 10 (et non de 0 à 9)
                                       // j+5 car on veut les tables de multiplication d
e 5, 6, et 7
                                       // (j allant de 0 à 2 compris)
        }
    }

    // tab[i][1] (i variant de 0 à 9 compris) vaut
    // maintenant {6, 12, 18, 24, 30, 36, 42, 48, 54, 60)
    // Pour vérifier, affichons tab[i][1] (table de multiplication de 6) :
    clrscr();
    for(i=0 ; i<10 ; i++)
    {
        printf("6*%d = %d\n", i+1, tab[i][1]);
    }


    ngetchx();
}
```

Exemple Complet

Pour les curieux qui se demandent ce qu'aurait donné un programme n'utilisant pas de tableaux, voici une solution possible, qui affiche les tables de multiplication de 5, 6, et 7 au fur et à mesure qu'elles sont calculées :

```
#include <tigcclib.h>

void _main(void)
{
    short i=0,
          j=0;

    clrscr();
    for(i=0 ; i<10 ; i++)
    {
        printf("% 2d* ", i+1);
        for(j=0 ; j<3 ; j++)
        {
            printf("%d=% 2d ", j+5, (i+1)*(j+5));
        }
        printf("\n");
    }

    ngetchx();
}
```

Exemple Complet

Ce code source ne présentant rien de bien nouveau, et n'étant pas en rapport avec le sujet du chapître, je ne m'étendrai pas dessus. Juste à titre d'information, "% 2d" est une balise printf qui permet d'afficher un nombre entier sur au minimum deux caractères, en mettant des blancs à gauche si nécessaire.

D: Utilisation de pointeurs pour accéder aux matrices

Tout comme pour les listes, il est possible d'utiliser des pointeurs pour accéder aux différents éléments d'une matrice.
Je ne dirai que quelques mots à ce sujet, car ce n'est pas extrêmement utilisé, et il est beaucoup plus difficile de s'y retrouver que lorsqu'il s'agit de listes.
La principale différence est que vous avez à gérer plusieurs dimensions. Par exemple, pour passer à la ligne suivante d'un tableau à deux dimensions, il vous faut ajouter non pas 1, mais le nombre d'éléments que le tableau peut contenir par ligne, puisque les cases sont organisées de manière successive en mémoire.

V:\ Travail avec des tableaux

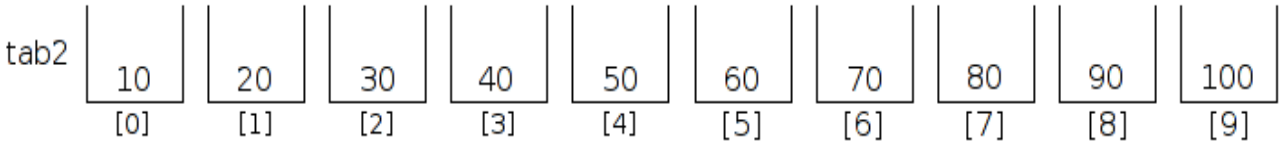
Pour finir ce chapitre, nous allons voir comment manipuler d'un seul coup plusieurs cases de tableaux ; par exemple, pour copier les données d'un tableau vers un autre tableau.

A: Copie de données d'un tableau vers un autre

Tout d'abord, pour cette sous-partie, nous allons considéré que nous avons déclaré deux tableaux d'entiers, nommés tab1 et tab2, comme ceci :

```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

A titre d'information, pour y voir plus clair, voici le schéma correspondant :



Si nous voulons copier le contenu du tableau tab1 vers le tableau tab2, nous serions tenté d'utiliser une syntaxe telle que ceci :

```
tab2 = tab1;
```

Cela dit, ceci n'est pas permis en C. En effet, l'opérateur d'affectation ne peut être appliqué à des tableaux, qui, bien que n'étant que des zones mémoires auxquelles le compilateur fait correspondre un pointeur, ne sont pas directement considéré comme des pointeurs par celui-ci.
Cette écriture est considérée une erreur au niveau du compilateur, qui refusera donc de compiler votre programme.

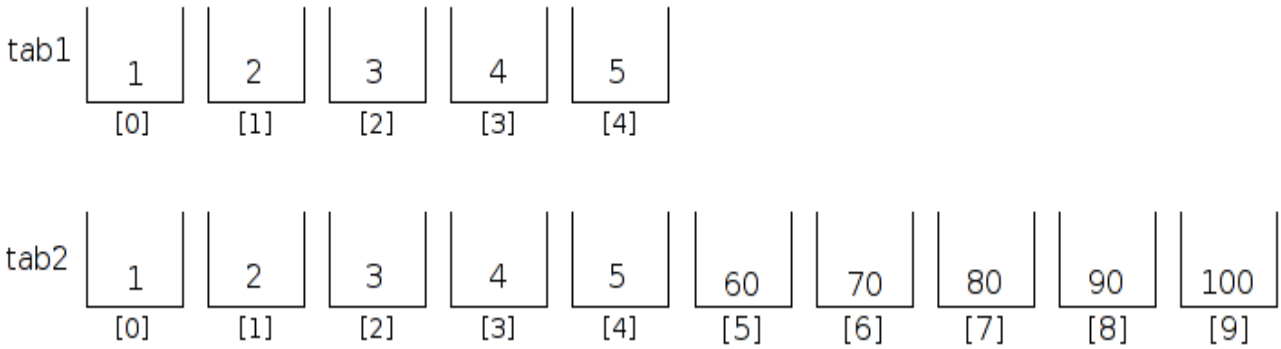
Si vous voulez copier des données d'une zone mémoire vers une autre autre mémoire, il va vous falloir employer la fonction memcpy, qui s'utilise comme ceci :

```
memcpy(destination, origine, nombre_d_octets);
```

Si vous souhaitez plusieurs cases d'un tableau vers un autre tableau, c'est ceci qu'il vous faut utiliser, puisqu'il est garanti, en C, que les cases successives d'un tableau seront organisées de manière séquentielle en mémoire. Par exemple, si nous souhaitons copier les 5 premières cases du tableau tab1 vers les 5 premières cases du tableau tab2, sachant que ces tableaux contiennent des éléments de type short, nous utiliserons ceci :

```
memcpy(tab2, tab1, 5*sizeof(short));
```

Et voici le résultat, représenté sous forme schématique :



Le dernier paramètre que l'on doit fournir à memcpy correspond à la taille en octets du bloc à copier. Il nous faut donc multiplier le nombre d'éléments que l'on souhaite copier par la taille de chacun de ces éléments, le plus propre pour obtenir la taille d'un élément étant d'utiliser sizeof.
Considérer qu'on sait qu'un short est codé sur 2 octets n'est pas une bonne idée. En effet, cela risque de poser problème si vous essayez de recompiler votre programme pour une autre plate-forme, telle un PC par exemple. Alors que sizeof(short) vaudra toujours le nombre d'octets sur lesquels un short est codé, quelque soit la plate-forme.

Naturellement, il est possible de copier une portion de tab1 vers, par exemple, le milieu de tab2.
Par exemple, pour copier vers les 6ème et 7ème cases de tab2 les 3ème et 4ème cases de tab1, nous utiliserions ceci :

```
memcpy(&tab2[5], &tab1[2], 2*sizeof(short));
```

Ou, en faisant de l'arithmétique de pointeurs, ce qui serait probablement plus naturel pour un programmeur C disposant d'un minimum d'expérience :

```
memcpy(tab2+5, tab1+2, 2*sizeof(short));
```

Naturellement, à chaque fois que nous faisons appel à la fonction memcpy, il nous faut nous prendre garde à ne pas dépasser des bornes du tableau.
En particulier, il faut nous assurer qu'on ne cherche pas à copier vers le tableau de destination plus qu'il ne peut contenir, ce qui causerait un plantage à plus ou moins long terme, et aussi, qu'on ne cherche pas à copier depuis le tableau d'origine plus de données qu'il n'en contient, ce qui reviendrait à copier des données indéterminées.
Soyez encore plus prudent si vous cherchez à copier des portions de tableaux comme nous venons juste de le faire.

Si, par curiosité, vous souhaitez afficher le contenu des deux tableaux, vous pourrez utiliser, par exemple, cette portion de code source :

```
unsigned short i;
for(i=0 ; i<5 ; i++)
     printf("tab1[%u]=%d\n", i, tab1[i]);
for(i=0 ; i<10; i++)
     printf("tab2[%u]=%d\n", i, tab2[i]);
```

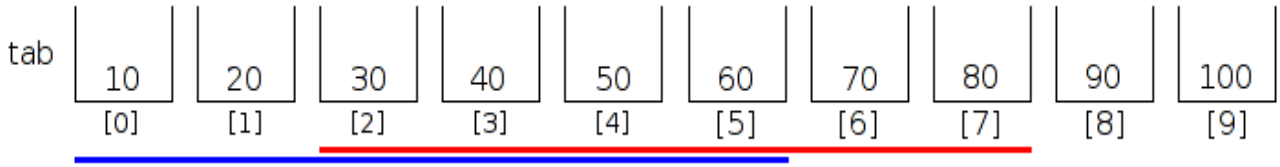
(Eventuellement, il peut être utile, selon la taille de votre écran et la taille des caractères utilisés, d'insérer un appel à ngetchx entre les deux boucles for)

B: Déplacement de données

Dans le cas où les zones mémoire d'origine et de destination sont superposées, la fonction memcpy ne doit pas être employée.
Cela se produit si, par exemple, on cherche à décaler de quelques cases les données d'un tableau.
Pour cette sous-partie, nous considérerons que nous avons déclaré le tableau tab de la façon qui suit :

```
short tab[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

Ce qui, graphiquement, correspond à ceci :



A présent, considérons le cas où nous voulons copier les données comprises dans les cases allant de la 3ème à la 8ème (zone en rouge sur le schéma) vers le début du tableau (zone en bleu sur le schéma).
De toute évidence, la région d'origine et celle de destination se recouvrent, ne serait-ce qu'en partie. Si on en croit la documentation, memcpy ne doit pas être utilisée dans une situation telle que celle-ci. Nous ferons donc appel à la fonction memmove, qui s'emploie de la même façon :

```
memmove(destination, origine, nombre_d_octets);
```

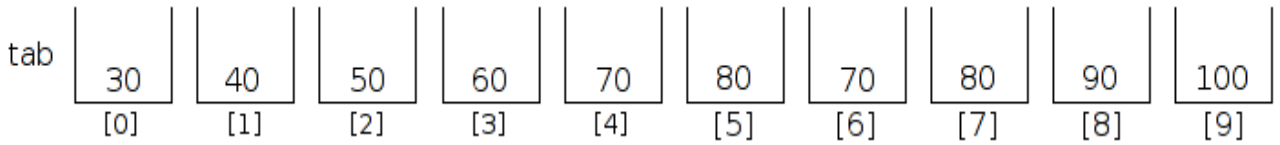
Dans le cas qui nous intéresse, il nous faudra donc écrire :

```
memove(&tab[0], &tab[2], 6*sizeof(short));
```

Ou plutôt, si vous commencez à penser comme la majorité des programmeurs C :

```
memove(tab, tab+2, 6*sizeof(short));
```

Ce qui nous permettra d'atteindre le résultat souhaité, que nous pourrions ainsi représenter, sous forme schématique :



C: Inversion de pointeurs

Pour terminer, nous allons voir une petite astuce qui peut être utile dans le cas où l'on souhaite échanger les contenus de deux tableaux.

Naturellement, on pourrait créer un tableau "temporaire", copier les données du premier tableau dedans, copier les données du second tableau dans le premier, et finir par copier les données du tableau temporaire vers le second tableau.
Un peu comme ceci :

```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[5] = {10, 20, 30, 40, 50};
short temp[10];

memcpy(temp, tab1, 5*sizeof(short));
memcpy(tab1, tab2, 5*sizeof(short));
memcpy(tab2, temp, 5*sizeof(short));
```

Cela dit, cette façon de procéder présente plusieurs inconvénients :

- Tout d'abord, il nous faut avoir un tableau temporaire suffisamment grand pour pouvoir contenir les données des deux tableaux (il faut donc que la taille du tableau temporaire soit supérieure ou égale à celle du plus grand des deux autres tableaux)
- Ensuite, si tab1 est plus petit que tab2, il ne pourra pas contenir toutes les données qui étaient dans tab2. tab2, lui, pourra contenir les données de tab1, mais il ne sera pas "rempli" : il restera, en fin de tableau, des cases inoccupées.
- Et enfin, copier des données prend du temps ; si les tableaux sont de grande taille, cela risque de ralentir notre programme.

Une autre solution est de, dès le départ, travailler avec des pointeurs.
On commence par déclarer nos deux tableaux, et, en même temps, on déclare des pointeurs sur le type des données du tableau. Comme ceci :

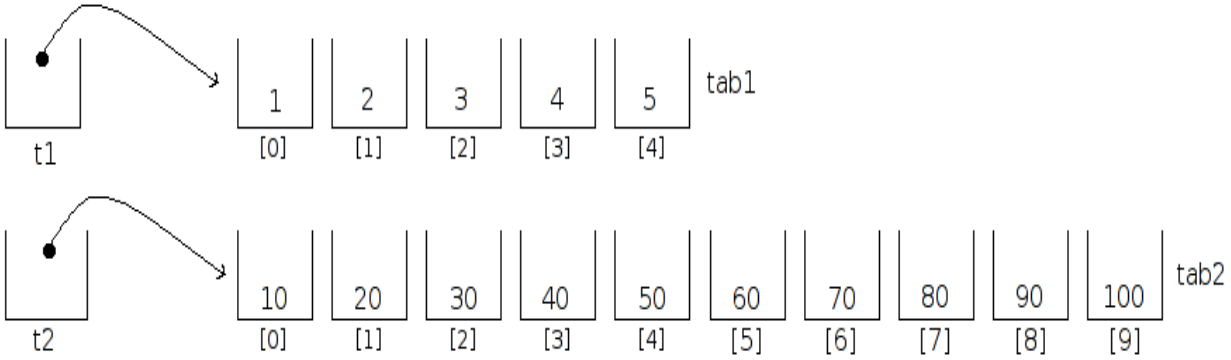
```
short tab1[5] = {1, 2, 3, 4, 5};
short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

short *t1;
short *t2;
```

Et dès le début du programme, on fait correspondre ces deux pointeurs à nos deux tableaux :

```
t1 = tab1;
t2 = tab2;
```

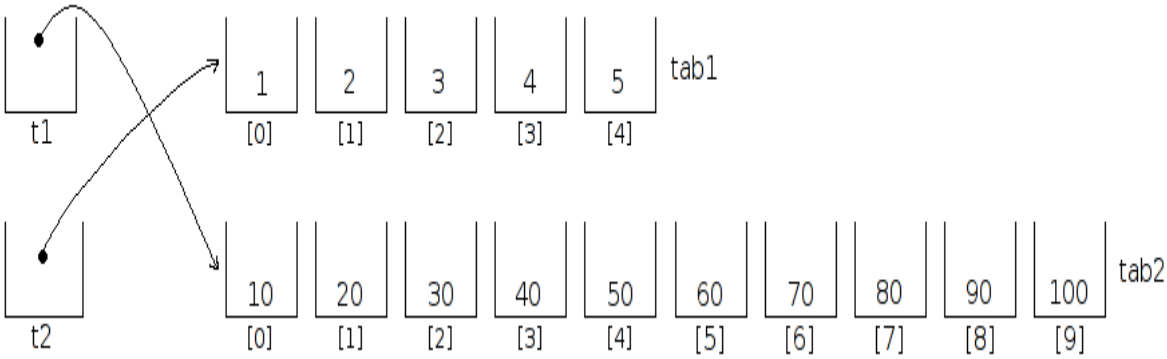
Schématiquement, on peut représenter cela ainsi :



Par la suite, au lieu de travailler avec nos tableaux, nous travaillons avec les deux pointeurs, et, au lieu d'échanger les contenu des deux tableaux, on échange les deux pointeurs :

```
short *temp;
temp = t1;
t1 = t2;
t2 = temp;
```

De la sorte, t2 pointera sur le tableau tab1, et t1 pointera sur le tableau tab2.
Si nous continuons à travailler avec nos deux pointeurs, nous aurons l'impression que les contenus des deux tableaux ont été échangés., ce qui peut se représenter ainsi :



Si on procède à un affichage des contenus pointés par les deux pointeurs, on verra que l'on peut accéder aux tableaux, exactement comme s'ils avaient été échangés.

Voici un exemple de code source complet illustrant les propos que nous venons de tenir :

```
#include <tigcclib.h>

void _main(void)
{
    unsigned short i;
    short tab1[5] = {1, 2, 3, 4, 5};
    short tab2[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    short *t1;
    short *t2;

    t1 = tab1;
    t2 = tab2;

    // Affichage des tableaux pointés
    // par les deux pointeurs, au départ :
    clrscr();
    printf("Tableau correspondant à t1:\n");
    for(i=0 ; i<5 ; i++)
        printf("t1[%u]=%d\n", i, t1[i]);
    ngetchx();
    clrscr();
    printf("Tableau correspondant à t2:\n");
    for(i=0 ; i<10; i++)
        printf("t2[%u]=%d\n", i, t2[i]);
    ngetchx();

    // Echange des deux pointeurs :
    short *temp;
    temp = t1;
    t1 = t2;
    t2 = temp;

    // Et affichage, après échange :
    clrscr();
    printf("Tableau correspondant à t1:\n");
    for(i=0 ; i<10 ; i++)
        printf("t1[%u]=%d\n", i, t1[i]);
    ngetchx();
    clrscr();
    printf("Tableau correspondant à t2:\n");
    for(i=0 ; i<5; i++)
        printf("t2[%u]=%d\n", i, t2[i]);
    ngetchx();

}
```

Exemple Complet

En premier, voici les deux premiers affichage, avant l'échange des pointeurs :

```
Tableau correspondant à t1:
t1[0]=1
t1[1]=2
t1[2]=3
t1[3]=4
t1[4]=5
```

```
Tableau correspondant à t2:
t2[0]=10
t2[1]=20
t2[2]=30
t2[3]=40
t2[4]=50
t2[5]=60
t2[6]=70
t2[7]=80
t2[8]=90
t2[9]=100
```

Et voici les deux suivants, après échange des pointeurs :

```
Tableau correspondant à t1:    Tableau correspondant à t2:
t1[0]=10                       t2[0]=1
t1[1]=20                       t2[1]=2
t1[2]=30                       t2[2]=3
t1[3]=40                       t2[3]=4
t1[4]=50                       t2[4]=5
t1[5]=60
t1[6]=70
t1[7]=80
t1[8]=90
t1[9]=100
```

Comme nous pouvons le constater, nous avons l'impression que les contenus des deux tableaux ont été échangés, alors que, finalement, nous n'avons fait qu'échanger deux pointeurs.
Nous avons gagné en rapidité (il est nettement plus rapide d'échanger deux pointeurs, ce qui revient à effectuer trois affectations, que d'échanger les contenus de deux gros tableaux !), mais aussi en espace mémoire, puisque nous n'avons plus besoin d'un (éventuellement grand) tableau temporaire.

Le chapitre suivant, traitant des chaînes de caractères, va nous permettre de travailler avec des tableaux dont le contenu est organisé de façon quelque peu particulière. Il nous permettra de mettre en application certains des points que nous venons d'étudier.