Déclaration :

Déclaration d'un tableau de pointeurs :

```
<Type> *<NomTableau>[<N>]
          déclare un tableau <NomTableau> de <N> pointeurs sur
des données du type <Type>.
```

Exemple :

```
double *A[10];
```

déclare un tableau de 10 pointeurs sur des rationnels du type **double** dont les adresses et les valeurs ne sont pas encore définies.

2) Initialisation :

Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.

Exemple :

```
char *JOUR[] = {"dimanche", "lundi",
                "mardi", "mercredi",
                "jeudi", "vendredi",
                "samedi"};
```

déclare un tableau **JOUR[]** de 7 pointeurs sur **char**. Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.

On peut afficher les 7 chaînes de caractères en fournissant les adresses contenues dans le tableau JOUR à **printf** (ou **puts**) :

```
int I;
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Comme JOUR[I] est un pointeur sur **char**, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur 'contenu de' :

```
int I;
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

L'expression JOUR[I]+J désigne la J-ième lettre de la I-ième chaîne. On peut afficher la troisième lettre de chaque jour de la semaine par :

```
int I;
for (I=0; i<7; I++) printf("%c\n",*(JOUR[I]+2));
```

V) Allocation dynamique de mémoire :

1) Déclaration statique de données :

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des

déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la ***déclaration statique*** des variables.

Exemples :

```
float A, B, C;          /* réservation de 12  octets */
short D[10][20];        /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10  octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40  octets */
```

a) Pointeurs :

Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation. Un pointeur est donc aussi déclaré statiquement. Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire. (En DOS : p =2 ou p = 4)

Exemples :

```
double *G;      /* réservation de p    octets */
char *H;        /* réservation de p    octets */
float *I[10];   /* réservation  de 10*p octets */
```

b) Chaînes de caractères constantes

L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement :

Exemples :

```
char *J = "Bonjour !";
           /* réservation de   p+10      octets */
 float *K[] = {"un", "deux", "trois", "quatre"};
            /* réservation de 4*p+3+5+6+7 octets */
```

2) Allocation dynamique :

Problème :

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple :

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par :

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

Allocation dynamique :

La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*. Nous parlons dans ce cas de l'***allocation dynamique*** de la mémoire.

3) La fonction `malloc` et l'opérateur `sizeof` :

La fonction **malloc** de la bibliothèque <*stdlib*> nous aide à localiser et à réserver de la mémoire au cours d'un programme.

a) La fonction malloc :

```
malloc( <N> )
fournit l'adresse d'un bloc en mémoire de <N> octets
libres ou la valeur zéro s'il n'y a pas assez de
mémoire.
```

Attention !

Sur notre système, le paramètre <N> est du type **unsigned int**. A l'aide de **malloc**, nous ne pouvons donc pas réserver plus de 65535 octets à la fois!

Exemple :

Supposons que nous ayons besoin d'un bloc en mémoire pour un texte de 4000 caractères.

Nous disposons d'un pointeur T sur **char** (**char *T**). Alors l'instruction :

```
T = malloc(4000);
```

fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T. S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur **sizeof** nous aide alors à préserver la portabilité du programme.

b) L'opérateur unaire sizeof

- `sizeof <var>`
  - `fournit la grandeur de la variable <var>`
- `sizeof <const>`
  - `fournit la grandeur de la constante <const>`
- `sizeof (<type>)`
  - `fournit la grandeur pour un objet du type <type>`

Exemple :

Après la déclaration,

```
short A[10];
char B[5][10];
```

nous obtenons les résultats suivants sur un IBM-PC (ou compatible) :

sizeof A s'évalue à 20
sizeof B s'évalue à 50
sizeof 4.25 s'évalue à 8
sizeof "Bonjour !" s'évalue à 10
sizeof(float) s'évalue à 4
sizeof(double) s'évalue à 8

Exemple :

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier :

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :"); scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

c) exit :

S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de <*stdlib*>) et de renvoyer une valeur différente de zéro comme code d'erreur du programme (voir aussi chapitre 10.4).

Exemple :

Le programme suivant lit 10 phrases au clavier, recherche des blocs de mémoire libres assez grands pour la mémorisation et passe les adresses aux composantes du tableau **TEXTE[]**. S'il n'y a pas assez de mémoire pour une chaîne, le programme affiche un message d'erreur et interrompt le programme avec le code d'erreur -1.

Nous devons utiliser une variable d'aide **INTRO** comme zone intermédiaire (non dynamique).

Pour cette raison, la longueur maximale d'une phrase est fixée à 500 caractères.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
/* Déclarations */
char INTRO[500];
char *TEXTE[10];
int I;
/* Traitement */
for (I=0; I<10; I++)
{
gets(INTRO);
/* Réservation de la mémoire */
TEXTE[I] = malloc(strlen(INTRO)+1);
/* S'il y a assez de mémoire, ... */
if (TEXTE[I])
/* copier la phrase à l'adresse */
/* fournie par malloc, ...                    */
strcpy(TEXTE[I], INTRO);
else
{
```

```
/* sinon quitter le programme */ /* après un message d'erreur.
*/ printf("ERREUR : Pas assez de mémoire \n"); exit(-1);

}
}
return (0);
}
```

4) <u>La fonction free :</u>

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque *<stdlib>*.

```
free( <Pointeur> )
libère le bloc de mémoire désigné par le <Pointeur>;
n'a pas d'effet si le pointeur a la valeur zéro.
```

<u>Attention !</u>

* La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.
* La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
* Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.