Introduction au langage C

Copyleft © GUINKO Tonguim Ferdinand - IBAM - Université de Ouagadougou

13 janvier 2010

Table des matières

1 Les pointeurs 4

1.1 Définition 4
1.2 Modes de transmission des variables 5
1.2.1 Adressage direct 5
1.2.2 Adressage indirect 6
1.3 Les opérateurs de base 6
1.3.1 L'opérateur d'adresse : & 7
1.3.2 L'opérateur d'indirection : * 8
1.3.3 Déclaration d'un pointeur 9
1.4 Les opérations élémentaires sur les pointeurs 10
1.4.1 Addition et soustraction de pointeurs 10
1.4.2 Incrémentation et décrémentation de pointeurs 10
1.4.3 Comparaison de pointeurs 11
1.4.4 Affectation de pointeurs 11
1.4.5 Priorité des opérateurs * et & 12
1.4.6 Le pointeur NULL 12
1.5 Pointeurs et tableaux à une dimension 14
1.5.1 Adressage des composantes d'un tableau 14
1.6 Formalisme tableau et formalisme pointeur 16
1.7 Pointeurs et tableaux à deux dimensions 18
1.8 Tableaux de pointeurs 21
1.8.1 Déclaration 21
1.8.2 Initialisation 21
1.9 Pointeurs et chaînes de caractères 23
1.9.1 Pointeurs sur char et chaînes de caractères constantes 24
1.9.1.1 Affectation 24

1.9.1.2 Initialisation 24
1.9.1.3 Perspectives et motivation 25
1.10 Allocation dynamique de mémoire 28
1.10.1 Déclaration statique de données 28
1.10.1.1 Pointeurs 29
1.10.1.2 Chaînes de caractères constantes 29
1.10.2 Allocation dynamique 29
1.10.3 La fonction malloc() et l'opérateur sizeof 30
1.10.3.1 La fonction malloc() 30
1.10.3.2 L'opérateur unaire sizeof 30
1.10.3.3 La fonction free() 32

Chapitre 1

Les pointeurs

1.1 Définition

Toute donnée en mémoire est stockée à une adresse unique. Cette adresse est soit déterminée par le programmeur (ce cas de figure n'est pas fréquent car dangereux), soit plus généralement calculée lors de la compilation en fonction de tous les éléments qui constituent le programme (variables déclarées, fonctions, instructions, ...). Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.

Si un pointeur `P` contient l'adresse d'une variable `A`, on dit que `P pointe sur A`.

En langage C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type. Ainsi, le langage C ne gèrera pas de la même façon un pointeur vers un entier (sa valeur contient l'adresse d'un `int`) et un pointeur vers un réel (sa valeur contient l'adresse d'un `float` ou d'un `double`).

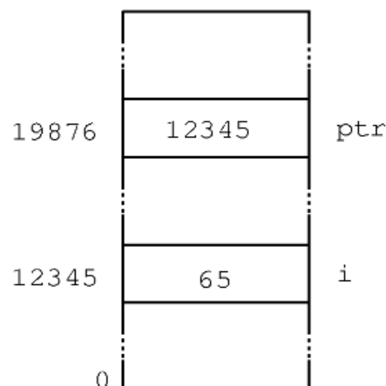

Fig. 1.1 – Représentation d'un pointeur en mémoire

1.2 Modes de transmission des variables

Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Toutefois, leur mode opératoire diffèrent ; la nuance entre les modes d'accès aux valeurs en utilisant l'un ou l'autre est mise en évidence à travers le mécanisme de transmission de paramètres aux fonction. En effet, il existe deux modes de passage des paramètres à une fonction :

– le mode de passage par valeur ou adressage direct,
– le mode de passage par adresse ou adressage indirect.

1.2.1 Adressage direct

Dans la programmation, nous utilisons des variables pour stocker des informations. La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur. Le nom de la variable nous permet alors d'accéder directement à cette valeur. L'`adressage direct`, ou le `passage de paramètre par valeur`, est donc l'accès au contenu d'une variable par le nom de la variable.

Lors d'un passage de paramètre par valeur, c'est l'ensemble des zones mémoires occupées par le paramètre qui sont dupliquées ; il en résulte que le paramètre apparaît en double dans la mémoire, pendant tout le temps d'exécution de la fonction. D'autre part, la fonction ne connaît que la version dupliquée du paramètre et ignore tout de l'original.

Les conséquences du passage de paramètre par valeur sont les suivantes :

1. le paramètre devant être dupliqué dans la mémoire, il faut tenir compte du temps effectif de duplication en cours d'exécution ; ce temps sera d'autant plus long que le paramètre occupera plus de place en mémoire ;
2. si le paramètre occupe une place trop importante en mémoire, la présence de plusieurs occurrences de ce paramètre peut provoquer un phénomène de saturation de la mémoire ;
3. la fonction ne connaît que la version dupliquée du paramètre ; si la fonction modifie le paramètre en question, les modifications n'affectent que la version dupliquée et non l'original ; il en résulte que les modifications sont effectives tant que la fonction s'exécute, mais elles sont définitivement perdues dès sortie de la fonction et retour au programme appelant.

Exemple :

```
short A;
A = 10;
```

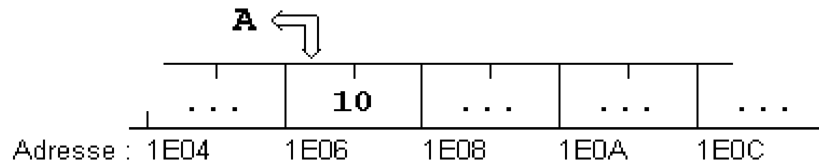

Fig. 1.2 – Exemple d'un passage de paramètre par valeur

1.2.2 Adressage indirect

Si, pour les raisons évoquées dans la section 1.2.1 nous ne voulons ou ne pouvons pas utiliser le nom d'une variable `A`, nous pouvons copier l'adresse de cette variable dans une variable spéciale pointeur `P`. Ensuite, nous pouvons retrouver l'information de la variable `A` en passant par le pointeur `P`. L'`adressage indirect`, ou le `passage de paramètre par référence`, est donc l'accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple :

Soit `A` une variable contenant la valeur `10` et `P` un pointeur qui contient l'adresse de `A`. En mémoire, `A` et `P` peuvent se présenter comme suit :

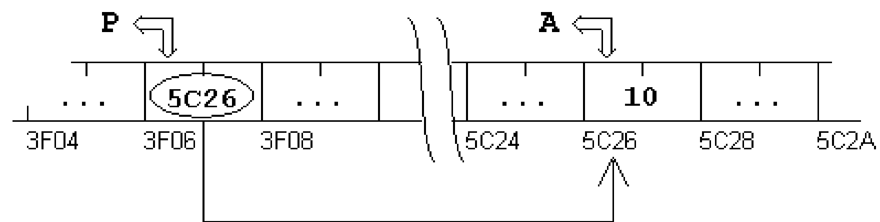

Fig. 1.3 – Exemple d'un passage de paramètre par adresse

1.3 Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin :

- d'un opérateur d'adresse : `&` pour obtenir l'adresse d'une variable ;
- d'un opérateur d'indirection : `*` pour accéder au contenu d'une adresse ;

– d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

1.3.1 L'opérateur d'adresse : &

La syntaxe suivante :

```
&<NomVariable>
```

fournit l'adresse de la variable <NomVariable> ; il permet donc d'initialiser la valeur d'un pointeur. L'opérateur & nous est déjà familier par la fonction scanf(), qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Exemple :

```
int N;
printf("Entrez un nombre entier: ");
scanf("%d", &N);
```

Attention :

L'opérateur & peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c'est à dire à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Exemple :

Soit P un pointeur non initialisé :

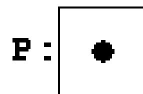

FIG. 1.4 – Représentation d'un pointeur non initialisé

et A une variable, du même type, contenant la valeur 10 :

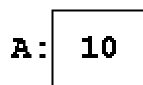

FIG. 1.5 – Représentation d'une variable simple initialisée

Alors l'instruction :

```
P = &A;
```

affecte l'adresse de la variable `A` à la variable `P`. Nous pouvons illustrer le fait que `P pointe sur A` par une flèche :

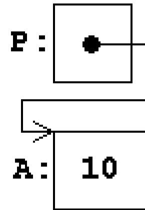

FIG. 1.6 – Représentation d'une variable initialisée par référence

1.3.2 L'opérateur d'indirection : *

L'`opérateur d'indirection` est aussi souvent appelé l'opérateur *contenu de*. La syntaxe suivante :

```
*<NomPointeur>
```

désigne le contenu de l'adresse référencée par le pointeur `<NomPointeur>`. L'opérateur * permet donc d'obtenir la valeur d'un élément dont l'adresse est contenue dans un pointeur.

Exemples :

1. Soit `A` une variable contenant la valeur `10`, `B` une variable contenant la valeur `50` et `P` un pointeur non initialisé :
   Après les instructions :

   ```
   P = &A;
   B = *P;
   *P = 20;
   ```

   - `P` pointe sur `A` ;
   - le contenu de `A` (référencé par *P) est affecté à `B`, et
   - le contenu de `A` (référencé par *P) est mis à 20.
2. ```
   int i = 65, *P;
   P = &i; // P contient l'adresse de i
   *P = *P + 1; // équivalent à: i = i + 1
   (*P)++; // équivalent à: i++
   ```
   ```

3. Il n'est pas possible d'effectuer une indirection sur un pointeur de type void.

   Exemple :

   ```
   void *P1;
   int *P2;

   *P1 = 10;          /* Incorrect */
   P2 = (int *) P1;   /* Correct */
   ```

1.3.3 Déclaration d'un pointeur

La syntaxe générale d'une définition ou déclaration d'une variable de type pointeur est de la forme :

```
<Type> *<NomPointeur>
```

Cette instruction déclare un pointeur <NomPointeur> qui peut recevoir des adresses de variables du type <Type>. Une déclaration comme :

```
int *PNUM
```

peut être interprétée comme suit :

1. *PNUM est du type int ou
2. PNUM est un pointeur sur int ou
3. PNUM peut contenir l'adresse d'une variable du type int.

Exemple :

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit :

```
|main()
|{
| /* déclarations */
| short A = 10;
| short B = 50;
| short *P;
| /* traitement */
| P = &A;
| B = *P;
| *P = 20;
| return 0;
|}
```

Ou bien

```
|main()
|{
|/* déclarations */
| short A, B, *P;
| /* traitement */
| A = 10;
| B = 50;
| P = &A;
| B = *P;
| *P = 20;
| return 0;
|}
```

Remarque :

Lors de la déclaration d'un pointeur en langage C, ce pointeur est lié explicitement à un type de données. Ainsi, la variable `PNUM` déclarée comme pointeur sur int ne peut pas recevoir l'adresse d'une variable d'un autre type que `int`.

Nous allons voir que la limitation d'un pointeur à un type de variables n'élimine pas seulement un grand nombre de sources d'erreurs très désagréables, mais permet une série d'opérations très pratiques sur les pointeurs.

1.4 Les opérations élémentaires sur les pointeurs

1.4.1 Addition et soustraction de pointeurs

La soustraction de 2 pointeurs (de même type) situés dans un même tableau, retourne le nombre d'éléments qui les séparent :

Exemple :

```
int tab[20], *P1 = &tab[2], *P2 = &tab[6];
printf("%d \n", P2 - P1); // affiche: 4
```

En règle générale, si `P` pointe sur l'élément `A[i]` d'un tableau, et `P1` et `P2` deux pointeurs qui pointent dans le même tableau :

```
P + n:  pointe sur A[i+n]
P - n: pointe sur A[i-n]
P1 - P2: fournit le nombre de composantes comprises entre P1 et P2.
```

Le résultat de la soustraction `P1-P2` est :

- négatif si `P1` précède `P2` ;
- zéro si `P1 = P2` ;
- positif si `P2` precède `P1` ;
- indéfini si `P1` et `P2` ne pointent pas dans le même tableau.

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

1.4.2 Incrémentation et décrémentation de pointeurs

L'incrémentation (ou la décrémentation) de pointeur exprime le fait que l'on veuille atteindre l'élément pointé suivant (ou précédent). Le résultat est juste seulement si l'objet pointé est situé dans un même tableau.

Ainsi avec la définition suivante `int *P ;` les instructions `P++ ;` ou `P = P + 1 ;` sont équivalentes à incrémenter la valeur du pointeur de la longueur du type de l'objet pointé : `P = P + sizeof(int)`.

Ces opérations sont impossibles avec des pointeurs sur un type `void`.

En règle générale, si `P` pointe sur l'élément `A[i]` d'un tableau, alors après l'instruction :

```
P++ ==> P pointe sur A[i+1]
P+ = n ==> P pointe sur A[i+n]
P-- ==> P pointe sur A[i-1]
P -= ==> P pointe sur A[i-n]
```

1.4.3 Comparaison de pointeurs

Un pointeur ne pourra être comparé qu'à un pointeur sur le même type (ou à la constante `NULL`). On peut comparer deux pointeurs par `<, >, <=, >=, ==, !=`. La comparaison de deux pointeurs qui pointent dans le même tableau est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

1.4.4 Affectation de pointeurs

On ne doit affecter à un pointeur qu'une valeur de pointeur sur un objet de même type.

Exemples :

1. La fonction d'allocation dynamique `malloc` retourne un pointeur sur un `void` : si l'on veut faire une allocation dynamique d'un entier, on doit écrire :

   ```
   int *t; /* t est un pointeur sur un entier */
   t = (int *) malloc(sizeof(int));
   ```

   `t` contient l'adresse de début d'une zone de mémoire pouvant contenir un entier.
2. Soit `P1` et `P2` deux pointeurs sur `int`, alors l'affectation `P1 = P2 ;` copie le contenu de `P2` vers `P1`. `P1` pointe alors sur le même objet que `P2`.

Remarque :
L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies à l'intérieur d'un tableau. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini. Toutefois, il est permis de pointer sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau). Cette règle,

introduite avec le standard ANSI C, légalise la définition de boucles qui incrémentent le pointeur avant l'évaluation de la condition d'arrêt.

1.4.5 Priorité des opérateurs * et &

En travaillant avec des pointeurs, nous devons observer les règles suivantes :

1. Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrémentation ++, la décrémentation -). Dans une même expression, les opérateurs unaires *, &, !, ++, - sont évalués de droite à gauche ;
2. Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple :

Après l'instruction P = &X ;, les expressions suivantes, sont équivalentes :

Langage algorithmique	Langage C
Y = *P+1	Y = X+1
*P = *P+10	X = X+10
*P += 2	X += 2
++*P	++X
(*P)++	X++

Dans le dernier cas, les parenthèses sont nécessaires : comme les opérateurs unaires * et ++ sont évalués de droite à gauche, sans les parenthèses le pointeur P serait incrémenté, non pas l'objet sur lequel P pointe. On peut uniquement affecter des adresses à un pointeur.

1.4.6 Le pointeur NULL

Il s'agit d'un pointeur ne pointant sur rien. Sa valeur, qui est l'entier 0 : est utilisée pour indiquer qu'un pointeur ne pointe nulle part. Elle est utilisée pour des raisons de lisibilité et de sécurité.

Exemple :

```
int *P;
       P = 0;
```

Résumons :

Après les instructions :

```
int A;
int *P;
P = &A;
```

- `A` : désigne le contenu de `A` ;
- `&A` : désigne l'adresse de `A` ;
- `P` : désigne l'adresse de `A` ;
- `*P` : désigne le contenu de `A` ;
- `&P` : désigne l'adresse du pointeur `P` ;
- `*A` : est illégal (puisque `A` n'est pas un pointeur).

Exercice :

Soit le programme suivant :

```
main()
{
    int A = 1;
    int B = 2;
    int C = 3;
    int *P1, *P2;
    P1 = &A;
    P2 = &C;
    *P1 = (*P2)++;
    P1 = P2;
    P2 = &B;
    *P1 -= *P2;
    ++ *P2;
    *P1 *= *P2;
    A =++ *P2 **P1;
    P1 = &A;
    *P2 = *P1 /= *P2;
    return 0;
}
```

Copiez le tableau suivant et complétez le pour chaque instruction du programme ci-dessus.

	A	**B**	**C**	**P1**	**P2**
Initialisation	1	2	3	/	/
P1 = &A	1	2	3	&A	/
P2 = &C					
*P1 = (*P2)++					
P1 = P2					
P2 = &B					
*P1 -= *P2					
++ *P2					
*P1 *= *P2					
A = ++ *P2 * *P1					
P1 = &A					
*P2 = *P1 /= *P					

1.5 Pointeurs et tableaux à une dimension

En langage C, il existe une relation très étroite entre tableaux et pointeurs. Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

1.5.1 Adressage des composantes d'un tableau

Le nom d'un tableau est un pointeur constant sur le premier élément du tableau. En d'autre termes : `&tableau[0]` et `tableau` sont une seule et même adresse.

Exemple :

En déclarant un tableau `A` de type `int` et un pointeur `P` sur `int` de la façon suivante :

```
int A[10];
int *P;
```

l'instruction : `P = A ;` est équivalente à `P = &A[0]`.
Si `P` pointe sur une composante quelconque d'un tableau, alors `P + 1` pointe sur la composante suivante. Plus généralement :

- `P+i` : pointe sur la `i-ième` composante derrière `P` et
- `P-i` : pointe sur la `i-ième` composante devant `P`.

Ainsi, après l'instruction : `P = A ;`, le pointeur `P` pointe sur `A[0]`, et :

- `P+1` désigne le contenu de `A[1]` ;
- `P+2` désigne le contenu de `A[2]` ;
- `...`

– P+i désigne le contenu de A[i].

Remarques :

1. Au premier coup d'oeil, il est bien surprenant que P+i n'adresse pas le i-ième octet derrière P, mais la i-ième composante derrière P ... ;
2. Un pointeur est une variable, donc des opérations comme P = A ou P++ sont permises ;
3. Le nom d'un tableau est une constante, donc des opérations comme A = P ou A++ sont impossibles.

Ceci nous permet de jeter un petit coup d'oeil derrière les rideaux :

Lors de la première phase de la compilation, toutes les expressions de la forme A[i] sont traduites en *(A+i). En multipliant l'indice i par la grandeur d'une composante, on obtient un indice en octets : <indice en octets> = <indice élément> * <grandeur élément>. Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante i du tableau. Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom adressage indexé : <adresse indexée> = <adresse> + <indice en octets>. Presque tous les processeurs disposent de plusieurs registres spéciaux (registres index) à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

Résumons :

Soit un tableau A d'un type quelconque et i un indice pour les composantes de A alors :

– A désigne l'adresse de A[0] ;
– A+i désigne l'adresse de A[i] ;
– *(A+i) désigne le contenu de A[i].

Si P = A, alors :

– P pointe sur l'élément A[0] ;
– P+i pointe sur l'élément A[i] ;
– *(P+i) désigne le contenu de A[i].

Remarques :

– En langage C, une chaîne de caractères est considérée comme un tableau de caractères, dont le dernier élément est le caractère \0 ;
– Pour travailler sur un tableau on a donc le choix entre travailler avec un tableau en utlisant un indice our travailler avec un pointeur.

1.6 Formalisme tableau et formalisme pointeur

Il nous est facile à présent de traduire un programme écrit à l'aide du formalisme tableau vu à la section 1.5 en un programme employant le formalisme pointeur.

Exemple :

Les deux programmes suivants copient les éléments positifs d'un tableau `T` dans un deuxième tableau `POS`.

Formalisme tableau

```
main()
{
   int T[10] = {-3, 4, 0, -7, 3,
8, 0, -1, 4, -9};
   int POS[10];
   int I,J;  /* indices courants
dans T et POS */
   for (J=0,I=0 ; I<10 ; I++)
      if (T[I]>0)
      {
         POS[J] = T[I];
         J++;
      }
   return 0;
}
```

Formalisme pointeur

/*Nous pouvons remplacer systématiquement la notation `tableau[I]` par `*(tableau + I)`, ce qui conduit à ce programme :*/

```
   main()
   {
      int T[10] = {-3, 4, 0, -7, 3, 8, 0,
-1, 4, -9};
      int POS[10];
      int I,J;  /* indices courants dans T
et POS */
      for (J=0,I=0 ; I<10 ; I++)
      if (*(T+I)>0)
      {
         *(POS+J) = *(T+I);
         J++;
      }
      return 0;
   }
```

Sources d'erreurs

Un bon nombre d'erreurs lors de l'utilisation du langage C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

1. Les variables et leur utilisation
   - `int A` : déclare une variable simple du type `int` ;
   - `A` désigne le contenu de `A` ;
   - `&A` désigne l'adresse de `A` ;

- `int B[]` : déclare un tableau d'éléments du type `int` ;
- `B` désigne l'adresse de la première composante de `B` (Cette adresse est toujours constante) ;
- `B[i]` désigne le contenu de la composante `i` du tableau ;
- `&B[i]` désigne l'adresse de la composante `i` du tableau.

2. en utilisant le formalisme pointeur :
   - `B+i` désigne l'adresse de la composante `i` du tableau ;
   - `*(B+i)` désigne le contenu de la composante `i` du tableau ;
   - `int *P` déclare un pointeur sur des éléments du type `int` ;
   - `P` peut pointer sur des variables simples du type `int` ou sur les composantes d'un tableau du type `int` ;
   - `P` désigne l'adresse contenue dans `P` (Cette adresse est variable) ;
   - `*P` désigne le contenu de l'adresse dans `P`.
3. Si `P` pointe dans un tableau, alors :
   - `P` : désigne l'adresse de la première composante ;
   - `P+i` : désigne l'adresse de la `i-ième` composante derrière `P` ;
   - `*(P+i)` : désigne le contenu de la `i-ième` composante derrière `P`.

Exercices :

1. Ecrire un programme qui lit deux tableaux `A` et `B` et leurs dimensions `N` et `M` au clavier et qui ajoute les éléments de `B` à la fin de `A`. Utiliser le formalisme pointeur à chaque fois que cela est possible.
2. Soit `P` un pointeur qui pointe sur un tableau `A`. On a :

   ```
   int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
   int *P;
   P = A;
   ```

   Quelles valeurs ou adresses fournissent ces expressions :

   a) *P + 2 ;
   b) *(P + 2) ;
   c) &P + 1 ;
   d) &A[4] - 3 ;
   e) A + 3 ;
   f) &A[7] - P ;
   g) P + (*P - 10) ;
   h) *(P + *(P + 8) - A[7]).
3. Ecrire un programme qui lit un entier `X` et un tableau `A` du type int au clavier et élimine toutes les occurrences de `X` dans `A` en tassant les éléments restants. Le programme utilisera les pointeurs `P1` et `P2` pour parcourir le tableau.
4. Ecrire un programme qui range les éléments d'un tableau `A` du type `int` dans l'ordre inverse. Le programme utilisera des pointeurs `P1` et `P2` et une variable numérique `AIDE` pour la permutation des éléments.

1.7 Pointeurs et tableaux à deux dimensions

L'arithmétique des pointeurs se laisse élargir avec toutes ses conséquences sur les tableaux à deux dimensions. Voyons cela sur un exemple :

Exemple :

Le tableau M à deux dimensions est défini comme suit :

```
int M[4][10] = {{0,1,2,3,4,5,6,7,8,9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe (oh, surprise...) sur le tableau M[0] qui a la valeur : 0,1,2,3,4,5,6,7,8,9.

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur : 10,11,12,13,14,15,16,17,18,19.

Explication :

Au sens strict du terme, un tableau à deux dimensions est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, le premier élément de la matrice M est le vecteur 0,1,2,3,4,5,6,7,8,9, le deuxième élément est 10,11,12,13,14,15,16,17,18,19 et ainsi de suite. L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que :

M+I désigne l'adresse du tableau M[I]

Problème :

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c'est à dire : aux éléments M[0][0], M[0][1], ... , M[3][9] ?

Discussion :

Une solution consiste à convertir la valeur de M (qui est un pointeur sur un tableau du type int) en un pointeur sur int. On pourrait se contenter de procéder ainsi :

```
int M[4][10] = {{0,1,2,3,4,5 6,7 8,9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

```
int *P;
P = (int *)M;  /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter `M` à l'aide du pointeur `P` comme un tableau unidimensionnel de dimension `4*10` :

Exemple :

Les instructions suivantes calculent la somme de tous les éléments du tableau `M` :

```
int M[4][10] = {{0,1,2 3,4,5,6,7,8,9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
     SOM += *(P+I);
```

Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel il faut calculer avec le nombre de colonnes indiqué dans la déclaration du tableau.

Exemple :

Pour la matrice `A`, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes :

```
int A[3][4];
A[0][0]=1;
A[0][1]=2;
A[1][0]=10;
A[1][1]=20;
```

Dans la mémoire, ces composantes sont stockées comme suit :

L'adresse de l'élément `A[I][J]` se calcule alors par : `A + I*4 + J`.

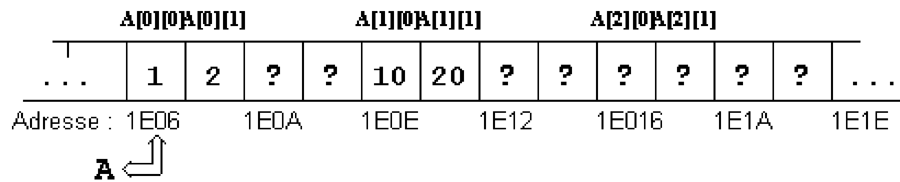

Fig. 1.7 – Pointeur et tableaux à 2 dimensions

Conclusion

Pour pouvoir travailler à l'aide de pointeurs dans un tableau à deux dimensions, nous avons besoin de quatre données :

1. l'adresse du premier élément du tableau converti dans le type simple des éléments du tableau ;
2. la longueur d'une ligne réservée en mémoire (- voir déclaration - ici : 4 colonnes) ;
3. le nombre d'éléments effectivement utilisés dans une ligne (- par exemple : lu au clavier - ici : 2 colonnes) ;
4. le nombre de lignes effectivement utilisées (- par exemple : lu au clavier - ici : 2 lignes).

Exercices :

1. Ecrire un programme qui lit une matrice `A` de dimensions `N` et `M` au clavier et affiche les données suivantes en utilisant le formalisme pointeur à chaque fois que cela est possible :
   a) la matrice `A` ;
   b) la transposée de `A` ;
   c) la matrice `A` interprétée comme tableau unidimensionnel.
2. Ecrire un programme qui lit deux matrices `A` et `B` de dimensions `N` et `M` respectivement `M` et `P` au clavier et qui effectue la multiplication des deux matrices. Le résultat de la multiplication sera affecté à la matrice `C`, qui sera ensuite affichée. Utiliser le formalisme pointeur à chaque fois que cela est possible.
3. Ecrire un programme qui lit 5 mots d'une longueur maximale de 50 caractères et les mémorise dans un tableau de chaînes de caractères `TABCH`. Inverser l'ordre des caractères à l'intérieur des 5 mots à l'aide de deux pointeurs `P1` et `P2`. Afficher les mots.

1.8 Tableaux de pointeurs

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.

1.8.1 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.

Remarque :

Le plus souvent, les tableaux de pointeurs sont utilisés pour mémoriser de façon économique des chaînes de caractères de différentes longueurs. Dans la suite, nous allons surtout considérer les tableaux de pointeurs sur des chaînes de caractères.

1.8.2 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]);
```

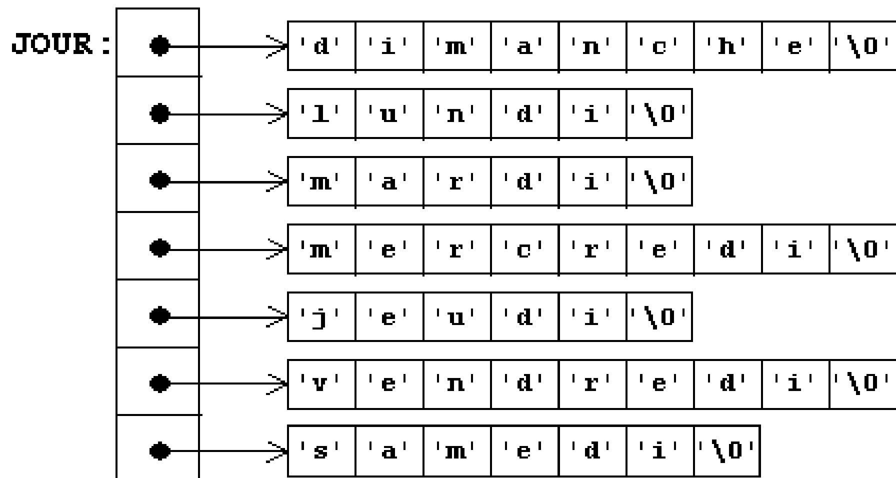

FIG. 1.8 – Représentation d'une variable initialisée par référence

Comme `jour[I]` est un pointeur sur `char`, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur d'indirection :

```
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));
```

Résumons :

le tableaux de pointeurs `int *D[] ;` déclare un tableau de pointeurs sur des éléments du type `int`

1. `D[i]` peut pointer sur des variables simples ou sur les composantes d'un tableau ;
2. `D[i]` désigne l'adresse contenue dans l'élément `i` de `D` (Les adresses dans `D[i]` sont variables) ;
3. `*D[i]` désigne le contenu de l'adresse dans `D[i]`.

Si `D[i]` pointe dans un tableau :

1. `D[i]` désigne l'adresse de la première composante ;
2. `D[i]+j` désigne l'adresse de la `j-ième` composante ;

3. `*(D[i]+j)` désigne le contenu de la `j-ième` composante.

Exercices :

1. Considérez les déclarations de `Nom1` et `Nom2` :

```
char *Nom1[] =    {"Fatou",  "Jean-Marie",
                            "Cheick-Omar", "Françoise", "Moustapha"};
char Nom2[][16] = {"Fatou",  "Jean-Marie",
                            "Cheick-Omar", "Françoise", "Moustapha"};
```

   (a) Représenter graphiquement la mémorisation des deux variables `Nom1` et `Nom2`.

   (b) Imaginez que vous devez écrire un programme pour chacun des deux tableaux qui trie les chaînes selon l'ordre lexicographique. En supposant que vous utilisez le même algorithme de tri pour les deux programmes, lequel des deux programmes sera probablement le plus rapide ?

2. Ecrire un programme qui lit le jour, le mois et l'année d'une date au clavier et qui affiche la date en français et en allemand. Utiliser deux tableaux de pointeurs, `MFRAN` et `MDEUT` que vous initialisez avec les noms des mois dans les deux langues. La première composante de chaque tableau contiendra un message d'erreur qui sera affiché lors de l'introduction d'une donnée illégale.

   Exemples :

```
Introduisez la date: 1 4 2009
Ouagadougou, le 1er avril 2009
Ouagadougou, den 1. April 2009

Introduisez la date: 2 4 2009
Ouagadougou, le 2 avril 2009
Ouagadougou, den 2. April 2009
```

1.9 Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur `int` peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur `char` peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur `char` peut en plus contenir l'adresse d'une chaîne de caractères constante et il peut même être initialisé avec une telle adresse.

1.9.1 Pointeurs sur char et chaînes de caractères constantes

1.9.1.1 Affectation

On peut attribuer l'adresse d'une chaîne de caractères constante à un pointeur sur `char` :

Exemple :

```
char *C;
C = "Ceci est une chaîne de caractères constante";
```

Nous pouvons lire cette chaîne constante (par exemple pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

1.9.1.2 Initialisation

Un pointeur sur `char` peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante :

`char *B = "Bonjour !";`

Attention : Il existe une différence importante entre les deux déclarations suivantes :

```
char A[] = "Bonjour !";   /* un tableau  */
char *B  = "Bonjour !";   /* un pointeur */
```

`A` est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison `\0`. Les caractères de la chaîne peuvent être changés, mais le nom `A` va toujours pointer sur la même adresse en mémoire. `B` est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.

Exemple :

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;            /* Impossible -> Erreur !!! */
C = "Bonjour !";  /* Impossible -> Erreur !!! */
```

Dans cet exemple, nous essayons de copier l'adresse de `B` dans `A`, respectivement l'adresse de la chaîne constante dans `C`. Ces opérations sont impossibles et illégales

parce que l'adresse représentée par le nom d'un tableau reste toujours constante. Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (par exemple dans une boucle) ou déléguer cette charge à une fonction de `<stdio>` ou `<string>`.

Conclusion :

- Utilisons des tableaux de caractères pour déclarer les chaînes de caractères que nous voulons modifier ;
- Utilisons des pointeurs sur char pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas) ;
- Utilisons de préférence des pointeurs pour effectuer les manipulations à l'intérieur des tableaux de caractères.

1.9.1.3 Perspectives et motivation

Avantages des pointeurs sur char

Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères ; nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs. Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions. En effet, pour fournir un tableau comme paramètre à une fonction, il faut passer l'adresse du tableau à la fonction. Or, les paramètres des fonctions sont des variables locales, que nous pouvons utiliser comme variables d'aide. Bref, une fonction obtenant une chaîne de caractères comme paramètre, dispose d'une copie locale de l'adresse de la chaîne. Cette copie peut remplacer les indices ou les variables d'aide du formalisme tableau.

Exemple :

La fonction `strcpy`, copie la chaîne `CH2` vers `CH1`. Les deux chaînes sont les arguments de la fonction et elles sont déclarées comme pointeurs sur `char`. La première version de `strcpy` est écrite entièrement à l'aide du formalisme tableau :

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((CH1[I]=CH2[I]) != '\0')
        I++;
}
```

Dans une première approche, nous pourrions remplacer simplement la notation `tableau[I]`

par `*(tableau + I)`, ce qui conduirait au programme :

```
void strcpy(char *CH1, char *CH2)
{
    int I;
    I=0;
    while ((*(CH1+I)=*(CH2+I)) != '\0')
        I++;
}
```

Cette transformation ne nous avance guère, nous avons tout au plus gagné quelques millièmes de secondes lors de la compilation. Un véritable avantage se laisse gagner en calculant directement avec les pointeurs `CH1` et `CH2` :

```
void strcpy(char *CH1, char *CH2)
{
    while ((*CH1=*CH2) != '\0')
    {
        CH1++;
        CH2++;
    }
}
```

Un vrai professionnel en langage C escaladerait les simplifications jusqu'à obtenir :

```
void strcpy(char *CH1, char *CH2)
{
    while (*CH1++ = *CH2++)
;
}
```

Exercices :

1. Ecrire un programme qui lit deux tableaux d'entiers `A` et `B` et leurs dimensions `N` et `M` au clavier et qui ajoute les éléments de `B` à la fin de `A`. Utiliser deux pointeurs `PA` et `PB` pour le transfer et afficher le tableau résultant `A`.
2. Ecrire de deux façons différentes, un programme qui vérifie sans utiliser une fonction de `<string>`, si une chaîne `CH` introduite au clavier est un palindrome :
   a) en utilisant uniquement le formalisme tableau
   b) en utilisant des pointeurs au lieu des indices numériques

Rappel : Un palindrome est un mot qui reste le même qu'on le lise de gauche à droite ou de droite à gauche :

Exemples :

```
Otto ==> est un palindrome
23432 ==> est un palindrome
Pierre ==> n'est pas un palindrome
```

3. Ecrire un programme qui lit une chaîne de caractères `CH` et détermine la longueur de la chaîne à l'aide d'un pointeur `P`. Le programme n'utilisera pas de variables numériques.
4. Ecrire un programme qui lit une chaîne de caractères `CH` et détermine le nombre de mots contenus dans la chaîne. Utiliser un pointeur `P`, une variable logique, la fonction isspace et une variable numérique `N` qui contiendra le nombre des mots.
5. Ecrire un programme qui lit une chaîne de caractères `CH` au clavier et qui compte les occurrences des lettres de l'alphabet en ne distinguant pas les majuscules et les minuscules. Utiliser un tableau `ABC` de dimension `26` pour mémoriser le résultat et un pointeur `PCH` pour parcourir la chaîne `CH` et un pointeur `PABC` pour parcourir `ABC`. Afficher seulement le nombre des lettres qui apparaissent au mois une fois dans le texte.

   Exemples :

   ```
   Entrez une ligne de texte (max. 100 caractères):
   Jeanne
   La chaîne "Jeanne" contient :
   1   fois la lettre 'A'
   2   fois la lettre 'E'
   1   fois la lettre 'J'
   3   fois la lettre 'N'
   ```
6. Ecrire un programme qui lit un caractère `C` et une chaîne de caractères `CH` au clavier. Ensuite toutes les occurrences de `C` dans `CH` seront éliminées. Le reste des caractères dans `CH` sera tassé à l'aide d'un pointeur et de la fonction `strcpy`.
7. Ecrire un programme qui lit deux chaînes de caractères `CH1` et `CH2` au clavier et élimine toutes les lettres de `CH1` qui apparaissent aussi dans `CH2`. Utiliser deux pointeurs `P1` et `P2`, une variable logique `TROUVE` et la fonction `strcpy`.

Exemples :

CH1	CH2	Chaîne obtenue
Bonjour	Bravo	njou
Bonjour	bravo	Bnjou
abacab	aa	bcab

8. Ecrire un programme qui lit deux chaînes de caractères `CH1` et `CH2` au clavier et supprime la première occurrence de `CH2` dans `CH1`. Utiliser uniquement des pointeurs, une variable logique `TROUVE` et la fonction `strcpy`.

   Exemples :

CH1	CH2	Chaîne obtenue
Alphonse	phon	Alse
totalement	t	otalement
abacab	aa	abacab

1.10 Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de l'allocation dynamique de la mémoire. Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

1.10.1 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.

Exemple :

```
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 */
```

1.10.1.1 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) :

Exemple :

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

1.10.1.2 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 :

Exemple :

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

1.10.2 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 ...

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.

1.10.3 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. Elle nous donne accès au `tas` (heap) ; c'est à dire à l'espace en mémoire laissé libre une fois mis en place le `DOS`, les gestionnaires, les programmes résidents, le programme lui-même et la pile.

1.10.3.1 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. 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.

1.10.3.2 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];
```

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));
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.

Exemple :

Le programme à la page suivante 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;
 }
```

1.10.3.3 La fonction free()

La fonction `free()` de la bibliothèque standard `<stdlib>` libère l'espace mémoire sur lequel pointe un pointeur `P`. Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de la fonction `malloc`, alors nous pouvons le libérer à l'aide de la fonction free(). `free( <Pointeur> )` libère le bloc de mémoire désigné par le `<Pointeur>` n'a pas d'effet si le pointeur a la valeur zéro. 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.

Attention :

La fonction `free()` peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par

Exercice :

1. Ecrire un programme qui lit 10 phrases d'une longueur maximale de 200 caractères au clavier et qui les mémorise dans un tableau de pointeurs sur `char` en réservant dynamiquement l'emplacement en mémoire pour les chaînes. Ensuite, l'ordre des phrases est inversé en modifiant les pointeurs et le tableau résultant est affiché ;
2. Ecrire un programme qui lit 10 mots au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs `MOT`. Effacer les 10 mots un à un, en suivant l'ordre lexicographique et en libérant leur espace en

mémoire. Afficher à chaque fois les mots restants en attendant la confirmation de l'utilisateur par la pression de la touche `return` du clavier ;

3. Ecrire un programme qui lit 10 mots au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs `MOT`. Copier les mots selon l'ordre lexicographique en une seule phrase dont l'adresse est affectée à un pointeur `PHRASE`. Réserver l'espace nécessaire à la `PHRASE` avant de copier les mots. Libérer la mémoire occupée par chaque mot après l'avoir copié. Utiliser les fonctions de `<string>` ;
4. Ecrire un programme qui lit 10 phrases au clavier (longueur maximale : 50 caractères) et attribue leurs adresses à un tableau de pointeurs `MOT`. Réserver dynamiquement l'emplacement en mémoire pour les mots. Trier les phrases lexicographiquement en n'échangeant que les pointeurs. Utiliser la méthode de tri par propagation (méthode de la bulle).