Les Pointeurs

Pointeurs et tableaux à deux dimensions

Pointeurs et tableaux à deux dimensions(1)

- L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions.
- ***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 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}.**

Pointeurs et tableaux à deux dimensions(2)

- ***Problème***

  Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d.: 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***. Par exemple:

```
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 = M; /* conversion automatique */
```

  Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0]. (Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Pointeurs et tableaux à deux dimensions(3)

- Cette solution n'est pas satisfaisante à cent pour-cent: Généralement, on gagne en lisibilité en explicitant la conversion mise en oeuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.
- ***Solution***

  Voici finalement la version que nous utiliserons:

```
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.

Pointeurs et tableaux à deux dimensions(4)

- ***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);
```

- ***Remarque:***
  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.

Les Pointeurs

Tableaux de pointeurs

Tableaux de pointeurs(1)

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

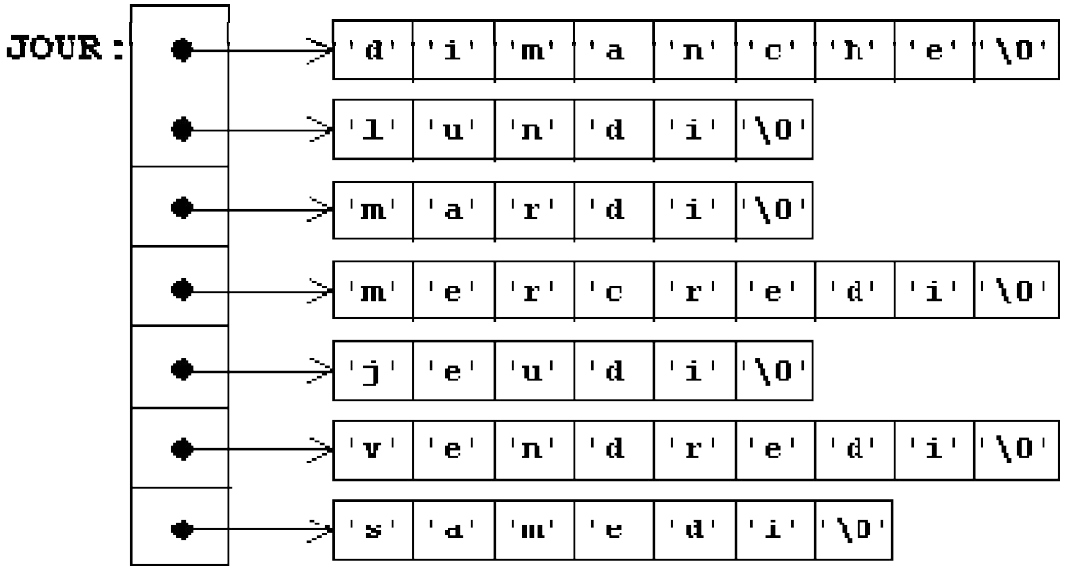
  Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.

Tableaux de pointeurs(2)

- ***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.



Tableaux de pointeurs(3)

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

```
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));
```

Tableaux de pointeurs(4)

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

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

Si D[i] pointe dans un tableau,

D[i]	désigne l'adresse de la première composante
D[i]+j	désigne l'adresse de la j-ième composante
***(D[i]+j)**	désigne le contenu de la j-ième composante

Les Pointeurs

Allocation dynamique
de mémoire

Déclaration statique de données(1)

- Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoautomatiquement ire. Jusqu'ici, la réservation de la mémoire s'est déroulée 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 */
```

Déclaration statique de données(2)

- ***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 */

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.

La fonction malloc

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

  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.
- ***Remarque:***

  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 avons 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.

L'opérateur unaire sizeof(1)

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

L'opérateur unaire sizeof(2)

- ***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(1)

- 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;
```

Exemple(2)

```
 /* 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; }
```

Remarques(1)

- Dans le programme :

```
#include <stdio.h>
main()
{
    int i ;
    int *p ;
    p=&i ;
}
```

i et ***p** sont identiques et on n'a pas besoin d'allocation dynamique puisque l'espace mémoire à l'adresse **&i** est déjà réservé pour un entier.

Remarques(2)

- La fonction **calloc** de la librairie **stdlib.h** a le même rôle que la fonction **malloc** mais elle initialise en plus l'objet pointé ***p** à **0**. Sa syntaxe est la suivante :

  calloc(nb_objets, taille_objet_octets)

- **Exemple :**

```
Int n=10;
Int *p ;
p= (int*) calloc (n, sizeof(int));    /* tableau de n entiers*/
```

- Est équivalent à:

```
Int n=10;
Int *p ;
p= (int*) malloc (n * sizeof(int));  /* tableau de n entiers*/
For (i=0; i<n; i++)
    *(p+i)=0;
```

La fonction free

- 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.
- ***Remarque:***
  - 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 de **free**, alors elle est libérée automatiquement à la fin du programme.

Les Pointeurs

Pointeurs et structures

Pointeur et structures

- **Pointeur et structures:**

  Supposons que l'on ait défini la struct personne à l'aide de la déclaration :

```
struct personne
  {
      char nom[20] ;
      unsigned int age ;
};
```

On déclare une variable de type pointeur vers une telle structure de la manière suivante :

```
struct personne  *p ;
```

On peut alors affecter à p des adresses de struct personne. Exemple :

```
struct personne pers;   /* pers est une variable de type struct personne */
struct personne *p;      /* p est un pointeur vers une struct personne */
p =  &pers;
(*p).age=20 ;                  /* parenthèses obligatoires   OU  */
p -> age=22 ;                /* -> opérateur d'accès */
```