Chapitre 4: LES POINTEURS

I – Définition et déclaration

Une variable est en fait une petite zone mémoire de x octets en fonction du type qui est allouée et où l'on va stocker les informations. En fait, il est possible d'accéder à une variable en utilisant :

- son nom ou identificateur
- son adresse en utilisant un pointeur

Un pointeur est une variable qui contient l'adresse d'un autre objet (variable ou fonction).

Les pointeurs sont définis par un type et se déclarent de la façon suivante :

type *nom ou **type*** nom; Exemple : int *adr;

II – Utilisation d'un pointeur

La manipulation des pointeurs nécessite l'utilisation des deux opérateurs suivants :

- L'opérateur & : cet opérateur, qui doit être obligatoirement suivi du nom d'une variable, indique la valeur de l'adresse d'une variable.

- L'opérateur * : cet opérateur désigne le contenu de la case mémoire pointée par le pointeur.

Exemple :

```
int n, *adr;  // déclaration de 2 variables n et *adr. adr est un pointeur sur un entier quelconque.
n = 98; // initialisation de la variable n mémoire de 4 octets allouée par le système.
adr = &n; // initialisation du pointeur adr : adr pointe l'entier n
```

Accès à la variable pointée

Si adr pointe sur n, alors il est possible d'accéder à la valeur de n en passant par adr.

Pour le compilateur,*adr est la variable pointée par adr, cela signifie que l'on peut, pour le moment du moins, utiliser indifféremment *adr ou n.
Ce sont deux façons de se référer à la même variable, on appelle cela de l' aliasing.

Accès à la variable pointée

```
#include<stdio.h>
 main ( )
{
     int n = 3 ;
     int * adr ;
     adr = &n ;
     printf ( "x = %d\n" , n ) ;
    *adr = 4 ;
    printf ( "x = %d\n" , n ) ;
}
```

L'affectation adr = &n fait pointer adr sur n. A partir de ce moment, *adr peut être utilisée pour désigner la variable n. De ce fait, l'affectation n = 4 peut aussi être écrite *adr = 4. Toutes les modifications opérées sur *adr seront répercutées sur la variable pointée par adr. Donc ce programme affiche

n = 3

n = 4

Exercices

Qu'affiche, a votre avis, le programme suivant ?

```
#include< stdion.h>
main ( )
{
        int x = 3 ;
        int  y = 5 ;
        int  * adr ;
        adr = &x ;
        printf("x = %d\ n" , x ) ;
        *adr = 4 ;
        printf(”x = %d\ n” , x ) ;
        adr = &y ;
        printf(” *adr = %d\ n" ,  *adr ) ;
        *adr =  adr + 1 ;
        printf( "y = %d\ n" , y ) ;
}
```

```
x = 3
x = 4
*adr = 5
y = 6
```

Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

– l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

– la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

– la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Arithmétique des pointeurs

Remarque : la somme de deux pointeurs n'est pas autorisée.

Si i est un entier et adr est un pointeur sur un objet de type type, l'expression adr + i désigne un pointeur sur un objet de type type dont la valeur est égale à la valeur de adr incrémentée de i * sizeof(type). Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation ++ et --. Par exemple, le programme

Arithmétique des pointeurs

```
main()
{
        int  i = 3;
        int *p1, *p2;
        p1 = &i;
        p2 = p1 + 1;
        printf("p1 = %ld \t p2 = %ld\n", p1,p2);
}
```

affiche p1 = 4831835984 p2 = 4831835988.

Par contre, le même programme avec des pointeurs sur des objets de type double :

Arithmétique des pointeurs

Par contre, le même programme avec des pointeurs sur des objets de type double :

```
main()
{
        double i = 3;
        double *p1, *p2;
        p1 = &i;
        p2 = p1 + 1;
        printf("p1 = %ld \t p2 = %ld\n",p1, p2);
}
```

affiche p1 = 4831835984 p2 = 4831835992.

Arithmétique des pointeurs

- Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.
- Le programme suivant imprime les éléments du tableau tab dans l'ordre croissant puis décroissant des indices.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
        int *p;
      printf("\n ordre croissant:\n");
      for (p = &tab[0]; p <= &tab[N-1]; p++)
              printf(" %d \n",*p);
   printf("\n ordre decroissant:\n");
```

Allocation dynamique

- La déclaration de variables dans le programme principal ou en global réserve de l’espace mémoire pour ces variables et pour toute la durée de vie du programme. Elle impose par ailleurs de connaître, avant le début de l’exécution, l'espace nécessaire au stockage de ces variables et en particuliers la dimension des tableaux.
- Or, dans de nombreuses applications, le nombre d'éléments d’un tableau peut varier d’une exécution du programme à l’autre. Allouer de la mémoire à l'exécution revient à réserver de manière dynamique la mémoire.

- Principal intérêt de la déclaration dynamique : allouer ou libérer la mémoire en fonction des besoins et en cours d'exécution d'un programme.

Allocation dynamique

- **allocation d'un élément de taille définie – fonction malloc**
  - void *malloc(unsigned size) : cette fonction alloue dynamiquement en mémoire, un espace de size octets et renvoie l'adresse du 1er octet de cet espace mémoire ou renvoie NULL si l'allocation n'a pu être réalisée.

- Pas d'initialisation à 0 de la variable.
- Mettre un "cast" si on veut forcer malloc à renvoyer un pointeur du type désiré.
- Utiliser la fonction sizeof(type) qui renvoie la taille en octets du type passé en paramètres.

Exemple :

```
int *p; // Déclaration du pointeur p
p = (int*) malloc (sizeof(int));
```

Allocation dynamique: exemple

- Voici un exemple illustrant l'utilisation de malloc .

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


 main ( )
{
        int *  p ;
        p = ( int * ) malloc ( 4 ) ;
        *p = 2 8 ;
        printf ( "%d\ n" , *p ) ;
}
```

Allocation dynamique:calloc

- 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 zéro. Sa syntaxe est :

  calloc(nb-objets,taille-objets)

Ainsi, si p est de type int*, l'instruction

```
p = (int*)calloc(N,sizeof(int));
```

est équivalente à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
        *(p + i) = 0;
```

L'emploi de calloc est simplement plus rapide.

Allocation dynamique: exemple

- **libération de l'espace mémoire : la fonction free**

  L'un des intérêts essentiels de la gestion dynamique est de pouvoir récupérer des emplacements dont on n'a plus besoin. La fonction free() permet de libérer la mémoire précédemment allouée dynamiquement et désignée par un pointeur adr. Sa syntaxe est :

```
free(nom-du-pointeur);
```

```
free(adr);
```

Pointeurs et tableaux

L’utilisation des pointeurs en C est, en grande partie, orientée vers la manipulation des tableaux.

Tableau unidimensionnel

Précédemment, on a utilisé les tableaux de manière intuitive par une déclaration du type (int tab[10]). L'opérateur d'indexation noté [ ] permet d'accéder à un élément du tableau via un indice. Tout tableau en C est en fait un pointeur constant. Dans la déclaration

int tab[10];

Exemple :

```
int tab[20];
int *adr;
adr = tab; est équivalente à adr = &tab[0];
```

tab est un pointeur constant (non modifiable) dont la valeur est l’adresse du premier élément du tableau. Autrement dit, tab a pour valeur &tab[0]. On peut donc utiliser un pointeur initialisé à tab pour parcourir les éléments du tableau.

Pointeurs et tableaux: Exemple

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
    {
            int i;
            int *adr;
            adr = tab;
            for (i = 0; i < N; i++)
                    {
                        printf(" %d \n",*adr);
                        adr++;
                    }
     }
```

Si adr pointe sur une composante quelconque d'un tableau, alors adr+1 pointe sur la composante suivante.

Plus généralement, adr + i pointe sur la ième composante derrière p.

Les fonctions de gestion "dynamique" de mémoire permettent de remédier à cette limitation.

Pointeurs et tableaux

Ainsi, après l'instruction, adr = tab; le pointeur adr pointe sur tab[0]
*(adr + 1) désigne le contenu de tab[1]
*(adr + 2) désigne le contenu de tab[2]
*(adr + i) désigne le contenu de tab[i].
adr[i] = *(adr + i)

Pointeurs et tableaux se manipulent donc exactement de même manière. Par exemple, le programme précèdent peut aussi s'écrire

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{
        int i;
        int *adr;
        adr = tab;
        for (i = 0; i < N; i++)
        printf(" %d \n", adr[i]);
  }
```

Pointeurs et tableaux

Toutefois, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dûs au fait qu'un tableau est un pointeur constant. Ainsi

- on ne peut pas créer de tableaux dont la taille est une variable du programme,
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'´éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à n éléments où n est une variable du programme, on écrit

```
#include <stdlib.h>
main()
{
     int n;
     int *tab;
     ...
     tab = (int*)malloc(n * sizeof(int));
```

Pointeurs et tableaux

- Si on veut en plus que tous les éléments du tableau tab soient initialisés à zéro, on remplace l'allocation dynamique avec malloc par

  tab = (int*)calloc(n, sizeof(int));

- Les éléments de tab sont manipulés avec l'opérateur d'indexation [], exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont:

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple p = &i ;

- un tableau n'est pas une Lvalue ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire tab++;).

Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :

```
int tab[M][N];
```

tab est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. tab a une valeur constante égale à l'adresse du premier élément du tableau, &tab[0][0]. De même tab[i], pour i entre 0 et M-1, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice i. tab[i] a donc une valeur constante qui est égale à &tab[i][0].

Comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

On déclare un pointeur qui pointe sur un objet de type type * (deux dimensions) de la même manière qu'un pointeur, c'est-`a-dire

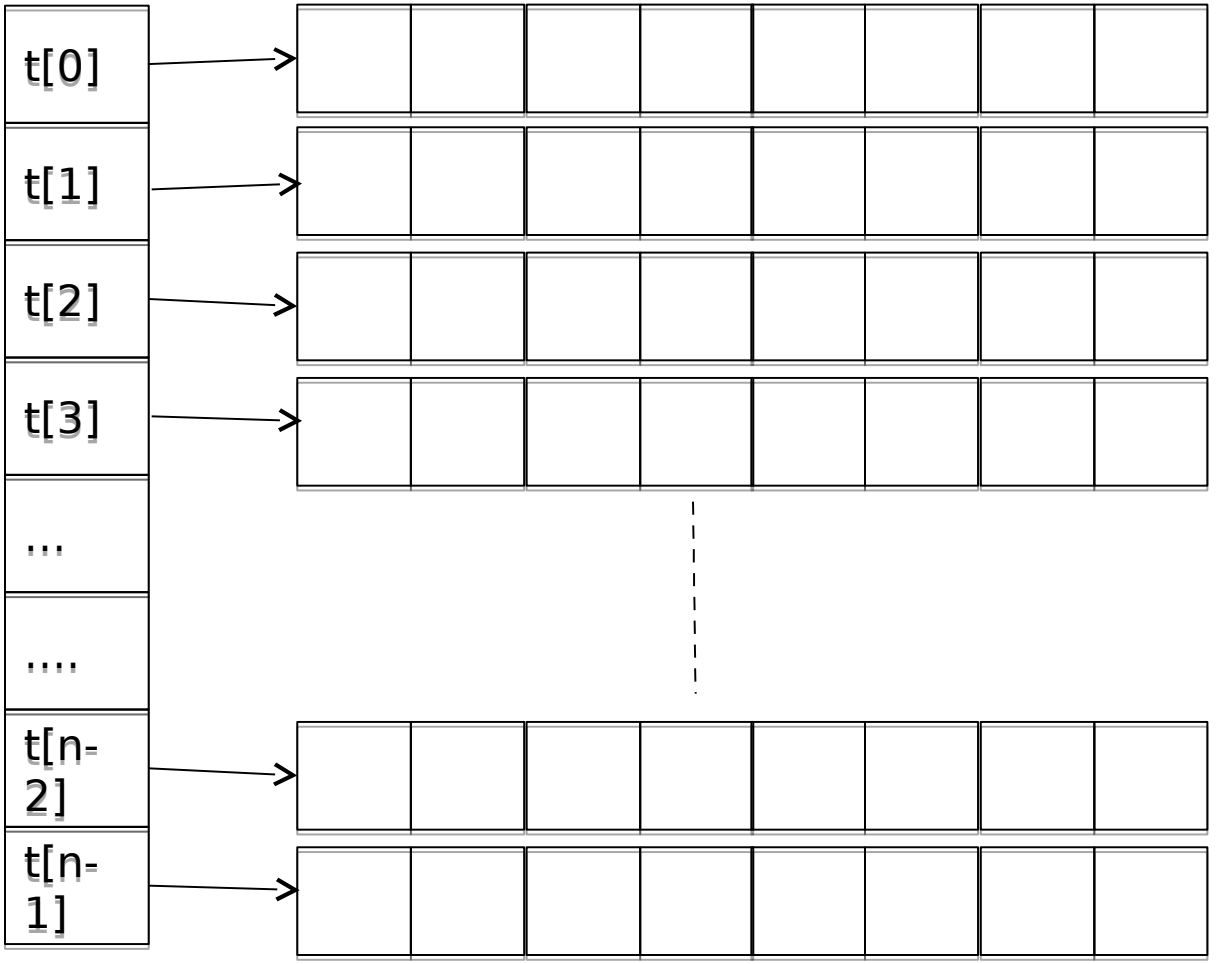
```
type **nom-du-pointeur;
```

Pointeurs et tableaux à plusieurs dimensions: Exemple

Par exemple, pour créer avec un pointeur de pointeur une matrice à k lignes et n colonnes à coefficients entiers, on écrit :

```
main()
{
        int k, n;
        int **tab;
        tab = (int**)malloc(k * sizeof(int*));
        for (i = 0; i < k; i++)
        tab[i] = (int*)malloc(n * sizeof(int));
          ....
        for (i = 0; i < k; i++)
        free(tab[i]);
        free(tab);
}
```

Tableaux à plusieurs dimensions variables



Accès aux éléménts :

t[i][j] est traduit en *(*(t+i)+j) : donc notation utilisable

Tableaux à plusieurs dimensions

Deux dimensions inconnues, la deuxième peut varier

Exemple : stocker un nombre inconnu de chaines de caractères

```
char ** dic;
int cpt = 0;
char tmp[100]; //tableau pour la lecture
int nb = 10;// nombre de chaines prévues par défaut
dic = (char **) malloc(nb * sizeof(char *));
for(i=0;i<nb;i++)
dic[i]=NULL;
printf("donner un nom"\n);
scanf("%s",tmp);
while (strlen(temp)!=0) {
      dic[cpt] = (char *) malloc(strlen(tmp)*sizeof(char));
      strcpy(dic[cpt],tmp);
      nb++;
      if (cpt> nb-1) { // on augmente la première dimension
            nb=nb+10;
            dic = (char **) realloc(dic,nb*sizeof(char*));
  for(i=nb-1;i>nb-11;i--) dic[i]=NULL;
  }
```

Exercice : trier les noms par ordre alphabétique

Pointeurs et chaînes de caractères

- On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type char, se terminant par le caractère nul '\0'. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type char. On peut faire subir à une chaîne définie par: char *chaine;

  des affectations comme

  chaine = "ceci est une chaine";

et toute opération valide sur les pointeurs, comme l'instruction chaine++;.

Pointeurs et chaînes de caractères

- Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>
main()
{
    int i;
    char *chaine;
    chaine = "chaine de caracteres";
    for (i = 0; *chaine != '\0'; i++)
    chaine++;
    printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard string.h, procède de manière identique. Il s'agit de la fonction strlen dont la syntaxe est

```
strlen(chaine);
```