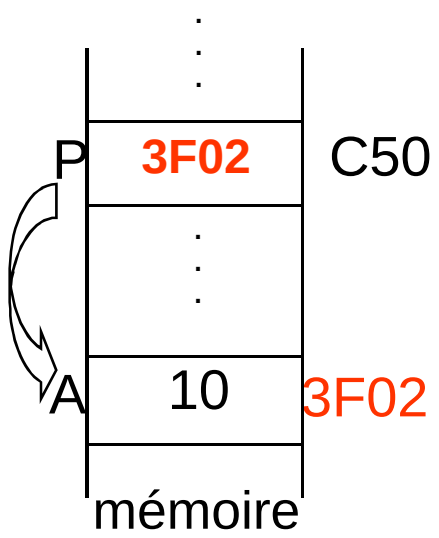
Chapitre 6

Les pointeurs

Pointeurs : définition

- Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.
- Exemple : Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A *(on dit que P pointe sur A)* .
- Remarques :
  - Le nom d'une variable permet d'accéder *directement* à sa valeur (**adressage direct**).
  - Un pointeur qui contient l'adresse de la variable, permet d'accéder *indirectement* à sa valeur (**adressage indirect**).
  - Le nom d'une variable est lié à la même adresse, alors qu'un pointeur peut pointer sur différentes adresses



Intérêts des pointeurs

- Les pointeurs présentent de nombreux avantages :
  - Ils sont indispensables pour permettre le passage par référence pour les paramètres des fonctions
  - Ils permettent de créer des structures de données (listes et arbres) dont le nombre d'éléments peut évoluer dynamiquement. Ces structures sont très utilisées en programmation.
  - Ils permettent d'écrire des programmes plus compacts et efficaces

Déclaration d'un pointeur

- En C, chaque pointeur est limité à un type de donnée (même si la valeur d'un pointeur, qui est une adresse, est toujours un entier).
- Le type d'un pointeur dépend du type de la variable pointée. Ceci est important pour connaître la taille de la valeur pointée.
- On déclare un pointeur par l'instruction : **type *nom-du-pointeur ;**
  - type est le type de la variable pointée
  - ***** est l'opérateur qui indiquera au compilateur que c'est un pointeur
  - Exemple :
    int *pi; //pi est un pointeur vers une variable de type int
    float *pf; //pf est un pointeur vers une variable de type float
- Rq: la valeur d'un pointeur donne l'adresse du premier octet parmi les n octets où la variable est stockée

Opérateurs de manipulation des pointeurs

- Lors du travail avec des pointeurs, nous utilisons :
  - un **opérateur 'adresse de': &** pour obtenir l'adresse d'une variable
  - un **opérateur 'contenu de': *** pour accéder au contenu d'une adresse

- Exemple1 :
  - **int * p;** //on déclare un pointeur vers une variable de type int
  - **int i=10, j=30;** // deux variables de type int
  - **p=&i;** // on met dans p, l'adresse de i (p pointe sur i)
  - **printf("*p = %d \n",*p);** //affiche : *p = 10
  - ***p=20;** // met la valeur 20 dans la case mémoire pointée par p (i vaut 20 après cette instruction)
  - **p=&j;** // p pointe sur j
  - **i=*p;** // on affecte le contenu de p à i (i vaut 30 après cette instruction)

Opérateurs de manipulation des pointeurs

- Exemple2 : **float a, *p;**
  p=&a;
  printf("Entrez une valeur : \n");
  scanf("%f ",p); //supposons qu'on saisit la valeur 1.5
  printf("Adresse de a= %x, contenu de a= %f\n" , p,*p);
  ***p+=0.5;**
  printf ("a= %f\n" , a); //affiche a=2.0

- **Remarque :** si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X
  - X+=2 équivaut à *P+=2
  - ++X équivaut à ++ *P
  - X++ équivaut à (*P)++ // les parenthèses ici sont obligatoires car l'associativité des opérateurs unaires * et ++ est de droite à gauche

Initialisation d'un pointeur

- A la déclaration d'un pointeur p, on ne sait pas sur quel zone mémoire il pointe. Ceci peut générer des problèmes :
  - **int *p;**
    ***p = 10;** //provoque un problème mémoire car le pointeur p n'a pas été initialisé
- **Conseil :** Toute utilisation d'un pointeur doit être précédée par une initialisation.
- On peut initialiser un pointeur en lui affectant :
  - l'adresse d'une variable (Ex: **int a, *p1; p1=&a;**)
  - un autre pointeur déjà initialisé (Ex: **int *p2; p2=p1;**)
  - la valeur 0 désignée par le symbole NULL, défini dans <stddef.h>.
    Ex: **int *p; p=0;ou p=NULL;** (on dit que p pointe 'nulle part': aucune adresse mémoire ne lui est associé)
- Rq: un pointeur peut aussi être initialisé par une allocation dynamique (voir fin du chapitre)

Pointeurs : exercice

```
main()
 {  int A = 1, B = 2, C = 3, *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;
 }
```

Donnez les valeurs de A, B,C,P1 et P2 après chaque instruction

Opérations arithmétiques avec les pointeurs

- La valeur d'un pointeur étant un entier, certaines opérations arithmétiques sont possibles : ajouter ou soustraire un entier à un pointeur ou faire la différence de deux pointeurs

- Pour un entier i et des pointeurs p, p1 et p2 sur une variable de type T
  - **p+i (resp p-i)** : désigne un pointeur sur une variable de type T. Sa valeur est égale à celle de p incrémentée (resp décrémentée) de i*sizeof(T).

  - **p1-p2 :** Le résultat est un entier dont la valeur est égale à (différence des adresses)/sizeof(T).
- Remarque:
  - on peut également utiliser les opérateurs ++ et -- avec les pointeurs
  - la somme de deux pointeurs n'est pas autorisée

Opérations arithmétiques avec les pointeurs

Exemple :

```
float *p1, *p2;
float z =1.5;
p1=&z;
printf("Adresse p1 = %x \n",p1);
p1++;
p2=p1+1;
printf("Adresse p1 = %x \t Adresse p2 = %x\n",p1,p2);
printf("p2-p1 = %d \n",p2-p1);
```

Affichage :

Adresse p1 = 22ff44

Adresse p1 = 22ff48 Adresse p2 = 22ff4c

p2-p1=1

Pointeurs et tableaux

- Comme on l'a déjà vu au chapitre 5, le nom d'un tableau T représente l'adresse de son premier élément (T=&T[0]). Avec le formalisme pointeur, on peut dire que T est un **pointeur constant** sur le premier élément du tableau.

- En déclarant un tableau T et un pointeur P du même type, l'instruction P=T fait pointer P sur le premier élément de T (P=&T[0]) et crée une liaison entre P et le tableau T.

- A partir de là, on peut manipuler le tableau T en utilisant P, en effet :
  - **P pointe sur T[0] et *P désigne T[0]**
  - **P+1 pointe sur T[1] et *(P+1) désigne T[1]**
  - **….**
  - **P+i pointe sur T[i] et *(P+i) désigne T[i]**

Pointeurs et tableaux : exemple

- Exemple: **short x, A[7]={5,0,9,2,1,3,8};**
  short *P;
  P=A;
  x=*(P+5);

- Le compilateur obtient l'adresse P+5 en ajoutant 5*sizeof(short) = 10 octets à l'adresse dans P

- D'autre part, les composantes du tableau sont stockées à des emplacements contigus et &A[5]= &A[0]+sizeof(short)*5=A+10

- Ainsi, x est égale à la valeur de A[5] (x= A[5])

Pointeurs : saisie et affichage d'un tableau

Version 1:

```
main()
 { float T[100] , *pt;
   int i,n;
   do {printf("Entrez n \n " );
        scanf(" %d" ,&n);
       }while(n<0 ||n>100);

   pt=T;
   for(i=0;i<n;i++)
       { printf ("Entrez T[%d] \n ",i );
         scanf(" %f" , pt+i);
        }

    for(i=0;i<n;i++)
         printf (" %f \t",*(pt+i));
 }
```

Version 2: sans utiliser i

```
main()
 { float T[100] , *pt;
   int n;
   do {printf("Entrez n \n " );
        scanf(" %d" ,&n);
       }while(n<0 ||n>100);


    for(pt=T;pt<T+n;pt++)
{ printf ("Entrez T[%d] \n ",pt-T );
          scanf(" %f" , pt);
        }

    for(pt=T;pt<T+n;pt++)
         printf (" %f \t",*pt);
 }
```

Pointeurs et tableaux à deux dimensions

- Le nom d'un tableau A à deux dimensions est un pointeur constant sur le premier élément du tableau càd A[0][0].

- En déclarant un tableau A[n][m] et un pointeur P du même type, on peut manipuler le tableau A en utilisant le pointeur P en faisant pointer P sur le premier élément de A (P=&A[0][0]), Ainsi :

  - **P pointe sur A[0][0] et *P désigne A[0][0]**
  - **P+1 pointe sur A[0][1] et *(P+1) désigne A[0][1]**
  - **….**
  - **P+M pointe sur A[1][0] et *(P+M) désigne A[1][0]**
  - **….**
  - **P+i*M pointe sur A[i][0] et *(P+i*M) désigne A[i][0]**
  - **….**
  - **P+i*M+j pointe sur A[i][j] et *(P+i*M+j) désigne A[i][j]**

Pointeurs : saisie et affichage d'une matrice

```
#define N 10
#define M 20
main( )
  { int i, j, A[N][M], *pt;
    pt=&A[0][0];
    for(i=0;i<N;i++)
         for(j=0;j<M;j++)
         { printf ("Entrez A[%d][%d]\n ",i,j );
           scanf(" %d" , pt+i*M+j);
          }

    for(i=0;i<N;i++)
        { for(j=0;j<M;j++)
             printf (" %d \t",*(pt+i*M+j));
          printf ("\n");
        }
  }
```

Pointeurs et tableaux : remarques

En C, on peut définir :

- **Un tableau de pointeurs :**

  Ex : **int *T[10];** //déclaration d'un tableau de 10 pointeurs d'entiers

- **Un pointeur de tableaux :**

  Ex : **int (*pt)[20];** //déclaration d'un pointeur sur des tableaux de 20 éléments

- **Un pointeur de pointeurs :**

  Ex : **int **pt;** //déclaration d'un pointeur pt qui pointe sur des pointeurs d'entiers

Allocation dynamique de mémoire

- Quand on déclare une variable dans un programme, on lui réserve implicitement un certain nombre d'octets en mémoire. Ce nombre est connu avant l'exécution du programme

- Or, il arrive souvent qu'on ne connaît pas la taille des données au moment de la programmation. On réserve alors l'espace maximal prévisible, ce qui conduit à un gaspillage de la mémoire

- Il serait souhaitable d'allouer la mémoire en fonction des données à saisir (par exemple la dimension d'un tableau)

- Il faut donc un moyen pour allouer la mémoire lors de l'exécution du programme : c'est l'allocation dynamique de mémoire

La fonction malloc

- La fonction **malloc** de la bibliothèque <*stdlib*> permet de localiser et de réserver de la mémoire, sa syntaxe est : **malloc(N)**
- Cette fonction retourne un pointeur de type char * pointant vers le premier octet d'une zone mémoire libre de N octets ou le pointeur NULL s'il n'y a pas assez de mémoire libre à allouer.
- Exemple : Si on veut réserver la mémoire pour un texte de 1000 caractères, on peut déclarer un pointeur pt sur **char** (**char *pt**).
  - L'instruction: **T = malloc(1000);** fournit l'adresse d'un bloc de 1000 octets libres et l'affecte à T. S'il n'y a pas assez de mémoire, T obtient la valeur zéro (NULL).
- Remarque : Il existe d'autres fonctions d'allocation dynamique de mémoire dans la bibliothèque <*stdlib*>

La fonction malloc et free

- Si on veut réserver de la mémoire pour des données qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.

- Exemple : on peut réserver la mémoire pour 2 variables contiguës de type int avec l'instruction : p = (int*)malloc(2 * sizeof(int)); où p est un pointeur sur **int** (**int *p**).

- Si on n'a plus besoin d'un bloc de mémoire réservé par **malloc**, alors on peut le libérer à l'aide de la fonction **free** , dont la synatxe est : **free(pointeur)**;

- Si on ne libère pas explicitement la mémoire à l'aide de **free**, alors elle est libérée automatiquement à la fin du programme.

malloc et free : exemple

Saisie et affichage d'un tableau

```
#include<stdio.h>
#include<stdlib.h>
main()
 { float *pt;
   int i,n;
   printf("Entrez la taille du tableau \n" );
   scanf(" %d" ,&n);

   pt=(float*) malloc(n*sizeof(float));
   if (pt==Null)
    {
      printf( " pas assez de mémoire \n" );
     system(" pause " );
    }
```

```
printf(" Saisie du tableau \n " );
for(i=0;i<n;i++)
  { printf ("Élément %d ? \n ",i+1);
    scanf(" %f" , pt+i);
   }

printf(" Affichage du tableau \n " );
for(i=0;i<n;i++)
     printf (" %f \t",*(pt+i));
free(pt);
}
```