Tableaux (1)

Agrégation d'entités d'un MEME TYPE (représentée par un bloc contigu de mémoire).

Définitions:

/* Définition de tableaux à une dimension */

int t[10];

/* Définition de tableaux à deux dimensions */

float t[5][10];

Définitions et initialisation:

/* Tableau de taille 4 déterminée par la liste d'initialisation */

int t[]={0, 1, 2, 3};

int matrice[2][3] = {{1, 2, 3},{4, 5, 6}};

Remarques: la taille est définitivement fixée à la compilation.

Tableaux (2)

/* Utilisation de l'opérateur d'indexation [...] */

int t[10];

/* t[**0**] t[1] ...t[**9**] LA FIN !!! */

float t[5][10];

/* t[0][0] t[0][1] ...t[0][9]

t[1][0] t[1][1] ...t[1][9]

...

t[4][0] ... t[4][9] LA FIN !!! */

int i, j;

```
for (i = 0;i < 5; i++) {
        for (j = 0;j < 10; j++)
                printf(``%f ``,t[i][j]);
        printf("\n");
}
```

Remarques: - les indices varient de **0** jusqu'à **taille - 1**;

- le dépassement de borne n'est pas vérifié à la compilation.

Chaînes de caractères

Pas de type spécifique - une chaîne est simplement un tableau de caractères:

```
/* chaîne de caractères de taille 5 */
    char ch[] = "toto" ;
/* tableau de caractères */
    char ch1[100];
/* chaîne de caractères de taille 5 dans un tableau de caractères de taille 10 */
    char ch2[10] = "titi" ;
/* d'autres chaînes de caractères */
    char ch3[] = "Une jolie chaîne\n" ;
    char ch4[] = "Une avec un \ ou un \\." ;
```

Convention: On marque la fin logique d'une chaîne de caractères par le caractère nul ('\0');

taille d'une chaîne = longueur + 1.

Manipulation des chaînes de caractères

Utilisation de la bibliothèque standard du C. Les déclarations des fonctions sont incluses dans <string.h>. Les fonctions suivantes sont disponibles:

Longueur d'une chaîne:

int strlen(**const char** *s)

Comparaison de chaînes:

int strcmp(**const char** *s1, **const char** *s2)

int strncmp(**const char** *s1, **const char** *s2, **int** n)

Concaténation de chaînes:

char *strcat(**char** *dest, **const char** *orig)

char *strncat(**char** *dest, **const char** *orig, **int** n)

Copie de chaînes:

char *strcpy(**char** *dest, **const char** *orig)

char *strncpy(**char** *dest, **const char** *orig, int n)

Les pointeurs

L'OPERATEUR ADRESSE &

L'opérateur adresse & retourne l'adresse d'une variable en mémoire.

Exemple:

```
int i = 8;
printf("VOICI i:  %d \n", i);
printf("VOICI SON ADRESSE EN HEXADECIMAL: %p \n", &i);
```

On remarque que le format d'une adresse est %p (hexadécimal) ou %d (décimal) dans printf.

LES POINTEURS

Définition: Un pointeur est une adresse en mémoire. On dit que le pointeur pointe sur cette adresse.

Déclaration des pointeurs

Une variable de type pointeur se déclare à l'aide de l'objet pointé précédé du symbole * (opérateur d'indirection).

Exemples:

```
char *pc;  /* pc est un pointeur pointant sur un objet de type char */
int *pi;      /* pi est un pointeur pointant sur un objet de type int */
float *pr;   /* pr est un pointeur pointant sur un objet de type float */
```

L'opérateur * désigne en fait le contenu de l'adresse.

Exemples:

```
char *pc, c ;
pc = &c ;            /* initialisation du pointeur pc */
*pc = 34;
printf("CONTENU DE LA CASE MEMOIRE: %c \n", *pc);
printf("VALEUR DE L'ADRESSE EN HEXADECIMAL: %p \n", pc);
```

Tableaux et pointeurs

Un tableau d'entités de type *Z* est un pointeur constant sur le type *Z*.

Exemple: une chaîne de caractères est un pointeur constant sur caractère.

Le nom du tableau est un pointeur implicite sur son premier élément.

```
int *pi,  t[10],  u[] = {1,2,3};
/* t et u sont deux pointeurs sur des entiers */
/* t est l'adresse de t[0] */
/* u est l'adresse de u[0] */

pi = t;  /* expression d'affectation légale */
t = pi;  /* erreur car t est un pointeur CONSTANT */
t = u;  /* erreur car t est un pointeur CONSTANT */
t[3] ≈  pi[3] ≈  (t+3) ≈  (pi+3)  /* expressions équivalentes */
```

Remarque: pour les opérations arithmétiques avec les pointeurs, l'unité est la taille de l'objet pointé.

Pointeurs et tableaux

/* Définition d'un pointeur sur caractères initialisé par une chaîne */

char *s = "bonjour";

char u[] = "salut";

s[3] ≈ *(s+3) ≈ *("bonjour"+3) ≈ "bonjour"[3] /* expressions équivalentes */

s = u; /* expression d'affectation légale */

u = s; /* erreur car u est un pointeur CONSTANT */

/* Manipulation des pointeurs sur les éléments d'un tableau */

double X, T[10], *pT1, *pT2;

pT1 = &T[4];

X = *(pT1 + 1);

pT2 = &T[7];

int i = pT2 - pT1; /* i vaut 3 ; par contre, pT1 + pT2 est interdit */

Attention: Une définition de tableau réserve implicitement la place mémoire pour TOUS les éléments du tableau, rien de tel avec les pointeurs.

Tableau en argument d'une fonction

La syntaxe pour le passage des tableaux est:

```
int sort (int base[],  int taille) ;
```

L'écriture suivante est aussi possible:

```
int sort (int *base,  int taille) ;
```

Retour d'une fonction et tableaux: une fonction NE PEUT PAS retourner un tableau.

- **Solution 1**: retourner un pointeur, MAIS attention - la valeur du pointeur doit exister après l'exécution de la fonction.
- **Solution 2**: passer le tableau résultat en argument de la fonction.

Arithmétique des pointeurs

On peut essentiellement déplacer un pointeur dans un plan mémoire à l'aide des opérateurs d'addition, de soustraction, d'incrémentation, de décrémentation.

Exemples:

```
int *pi;            /*  pi pointe sur un objet de type entier, codé sur 4 octets */
float *pr;          /*  pr pointe sur un objet de type réel, codé sur 4 octets */
char *pc;           /* pc pointe sur un objet de type caractère, codé sur 1 octet */
*pi = 421;  /*  421 est le contenu de la case mémoire pi et des 3 suivantes */
*(pi + 1) = 53;    /*  on range 53  4 cases mémoire plus loin  */
*(pi + 2) = 0xabcd;     /*  on range 0xabcd  8 cases mémoire plus loin  */
*pr = 45.7;   /*  45.7 est rangé dans la case mémoire pr et les 3 suivantes  */
pr ++;         /*  incrémente la valeur du pointeur pr (de 4 cases mémoire)  */
printf("l'adresse pr vaut: %p\n",pr); /* affichage de la valeur de l'adresse pr */
*pc = 'j';     /*  le contenu de la case mémoire pc est le code ASCII de 'j'  */
pc --;         /*  décrémente la valeur du pointeur pc (d'une case mémoire)  */
```

Allocation dynamique de la mémoire (1)

Lorsque l'on déclare une variable **char**, **int** ou **float**, un nombre de cases mémoire bien défini est réservé pour cette variable. Il n'en est pas de même avec les pointeurs.

Exemple:

```
char *c;
*c = 'a';  /* le code ASCII de a est rangé dans la case mémoire pointée par c */
*(c+1) = 'b';  /* le code ASCII de b est rangé une case mémoire plus loin */
*(c+2) = 'c';  /* le code ASCII de c est rangé une case mémoire plus loin */
*(c+3) = 'd';  /* le code ASCII de d est rangé une case mémoire plus loin */
```

Dans cet exemple, le compilateur a attribué une valeur au pointeur c, les adresses suivantes sont donc bien définies; mais le compilateur n'a pas réservé ces places: il pourra très bien les attribuer un peu plus tard à d'autres variables. Le contenu des cases mémoires c, c+1, c+2 et c+3 sera donc perdu.

Allocation dynamique de la mémoire (2)

Il existe en langage C, des fonctions permettant d'allouer de la place en mémoire à un pointeur.

Exemple (la fonction malloc):

```
char *pc;
int *pi, *pj, *pk;
float *pr;
pc = (char*)malloc(10);      /* on réserve 10 cases mémoire, soit la place pour 10 caractères */
pi = (int*)malloc(16);       /* on réserve 16 cases mémoire, soit la place pour 4 entiers */
pr = (float*)malloc(24);  /* on réserve 24 places, soit la place pour 6 réels */
pj = (int*)malloc(sizeof(int));  /* on réserve la place pour 1 entier */
pk = (int*)malloc(3*sizeof(int));  /* on réserve la place pour 3 entiers */
```

Libération de la mémoire (la fonction free):

```
free(pi);          /* on libère la place précédemment réservée pour pi */
free(pr);          /* on libère la place précédemment réservée pour pr */
```

Affectation d’une valeur à un pointeur

On ne peut pas affecter directement une valeur à un pointeur. L'écriture suivante est interdite: **char** *pc; pc = 0xfffe;

On peut cependant être amené à définir par programmation la valeur d'une adresse. On utilise pour cela l'opérateur de "cast"(jeu de deux parenthèses).

Exemple 1:

```
char *pc;
pc = (char*)0x1000;  /* pc est l'adresse 0x1000 et pointe sur un caractère */
int *pi;
pi = (int*)0xfffa;        /* pi est l'adresse 0xfffa et pointe sur un entier */
```

L'opérateur de "cast", permet d'autre part, à des pointeurs de types différents de pointer sur la même adresse.

Exemple 2:

```
char *pc;                 /* pc pointe sur un objet de type caractère */
int *pi;                  /* pi pointe sur un objet de type entier */
pi = (int*)malloc(4);   /* allocation dynamique pour pi */
pc = (char*)pi;  /* pc et pi pointent sur la même adresse */
```

```
/* Exemple d’une fonction qui échange les valeurs des paramètres */
#include <stdio.h>
void echanger (float *, float *);

void main( )
{
        float x = 1, y = 5;
        printf("Donnez deux données réelles : \n");
        scanf("%f  %f", &x,  &y);
        printf("Avant échange, dans x : %f ; dans y : %f\n", x, y);
        echanger(&x, &y);
        printf("Après échange, dans x : %f ; dans y : %f\n", x, y);
}

void echanger (float * ad_f1, float * ad_f2)
{
        float tampon;
        tampon = *ad_f1;
        *ad_f1 = *ad_f2;
        *ad_f2 = tampon;
}
```

```
/* Exemple d'une fonction qui n'échange pas les valeurs des param. */
#include <stdio.h>
void echanger (float , float );

void main( )
{
        float x = 1, y = 5;
        printf("Donnez deux données réelles : \n");
        scanf("%f  %f", &x,  &y);
        printf("Avant échange, dans x : %f ; dans y : %f\n", x, y);
        echanger(x, y);
        printf("Après échange, dans x : %f ; dans y : %f\n", x, y);
}

void echanger (float ad_f1, float ad_f2)
{
        float tampon;
        tampon = ad_f1;
        ad_f1 = ad_f2;
        ad_f2 = tampon;
}
```

```
/* Exemple d'une fonction utilisant l'allocation de la mémoire */
#include <stdio.h>
…
void RETICULATIONS (int, double **, double **, long int *, double *, int *, int **,
int, double **, double **, int);
extern void odp (double**,int*,int*,int*);

void RETICULATIONS (int n, double **DISS, double **D, long int *ARETE,
double *LONGUEUR, int *Y, int **Continuite, int OptionFunction, double
**DISTret, double **W, int Iternumber)
{ int i, j, k, p, P, *X;
  double *L, **DIST, EQminGLold;

  X = (int *)malloc((n + 1)*sizeof(int));
  L = (double *)malloc((n + 1)*sizeof(double));
  DIST = (double **)malloc((2*n-1)*sizeof(double*));
  for (i = 0;i <= 2*n-2; i++)
  {      DIST[i]=(double*)malloc((2*n-1)*sizeof(double));
         if (DIST[i]==NULL)
                exit(1);
  }
…
  free(X);
  free(L);
  for (i=0;i<=2*n-2;i++)
         free (DIST[i]);
  return 0;
}
```