les tableaux

- Syntaxe :
  - <type> <nom>[<nb elem>][<nb elem>]*
  - Pour un tableau de taille *N*, l'indice du premier élément est 0 et celui du dernier est (*N* -1).
  - On peut utiliser des tableaux de dimension 2 ou plus.

Exemple : deux tableaux de 100 éléments, l'un contenant des float, l'autre des int. Le dernier tableau définit une matrice 3x3 de double.

```
float VecteurA[100];
int VecteurB[100];
double MatriceTroisTrois[3][3];
```


initialisation d'un tableau

Initialisation à la déclaration : affecter une liste de valeurs séparées par des virgules et entourée par des accolades.

exemples

```
int tab[5] = {4, 6, 8, 12, 20};
double Matrice[3][3] = {{ 1, 0, 0 },
                        { 0, 1, 0 },
                        { 0, 0, 1 }};
```

Cas particulier : initialisation d'un tableau de caractères pour laquelle on peut utiliser une chaîne de caractères. Les deux lignes suivantes sont équivalentes :

```
char chn[20] = {'B', 'o', 'n', 'j', 'o', 'u', 'r'};
char chn[20] = "Bonjour";
```


accès aux valeurs d'un tableau

Pour accéder à un élément d'un tableau, on utilise l'opérateur [] . La valeur mise entre crochets peut être un calcul.

Exemple : on stocke dans le troisième élément de Tab la valeur du $i^{ème}$ élément (on commence à 0 !!):

```
Tab[2] = Tab[i - 1];
```


adresses et tableaux

On peut récupérer l'adresse de n'importe quel objet. Par exemple, il est possible d'obtenir l'adresse d'un élément d'un tableau (dans cet exemple, le onzième élément) :

```
double a[20];
double * p;
p = & (a[10]);
```

Par convention, le nom d'un tableau est une constante égale à l'adresse du premier élément du tableau.

Les deux lignes suivantes sont équivalentes :

```
p = & a[0];
p = a;
```


Calculs sur les pointeurs

Rappel : Il est possible de faire des calculs sur les pointeurs. On peut ajouter ou soustraire une valeur entière à un pointeur.

Exemple : p pointe à la fin sur le troisième élément du tableau a (donc sur a[2]) :

```
double a[20];
double * p;
p = & (a[10]); //ci-dessous, on n'enlève pas 8
 p = p - 8; // mais 8 fois la taille d'un double
```

Pour effectuer ce calcul tous les opérateurs classiques d'addition et de soustraction sont utilisables en particulier les opérateurs d'incrémentation.

Une chaîne de caractères se termine toujours par le caractère de code ASCII 0 (\0). L'exemple suivant permet de compter le nombre de caractères stockés dans le tableau de caractères chn (le caractère nul ne fait pas partie du compte) :

```
char * p = chn;
int NbCar = 0;
while ( *p != '\0') {
    p++;
    NbCar++;
}// il y a une fonction de la Bib. <string.h> qui fait ça
```

Autre Exemple:

```
int main(){
    char *ch="bonjour", ch2[5]="salut",ch3[6]="salut",*p,
     ch4[]="au revoir";// ch2 est un tableau ch3 une chaîne
    printf("%s,%d\n",ch,strlen(ch)); //bonjour,7
//    ch[3]=0; // erreur à l'exécution
//    *(ch+3)='z'; // erreur à l'exécution
    printf("%s,%d\n",ch2,strlen(ch2)); //salut?@,7 : peut être ≠
    (mémoire)
    ch2[7]='x';  // à la position 7, il y a un zéro. Il sera remplacé par 'x'
    printf("%s,%d\n",ch2,strlen(ch2)); //salut?@xp?@,11 : peut être ≠
    printf("%s,%d\n",ch3,strlen(ch3)); //salut,5
    ch3[3]=0;
    printf("%s,%d\n",ch3,strlen(ch3)); //sal,3
    p=ch3;
    *(p+3)='0';
    printf("%s,%d\n",ch3,strlen(ch3)); //sal0t,5
    *(ch3+3)='x';
    printf("%s,%d\n",ch3,strlen(ch3)); //salxt,5
    p++;
    printf("%s,%d\n",p,strlen(p));  //al0t,4
    printf("%s,%d\n",ch4,strlen(ch4)); //au revoir,9
    return 0;
}
```



```
#include <stdio.h>
#include <string.h>
void f(char c[]){
    c[2]='T';
}
int main(){
    char ch1[6],ch2[6]="salut";
//  ch1=ch2 ; //erreur
    strcpy(ch1, ch2);
    printf("%s\n", ch1); //salut
    f(ch2);
    printf("%s,%d\n",ch2);//saTut
}
```


Conclusion :

- Les noms de tableaux sont semblable aux pointeurs
- Utilisez les [] de préférence à * pour les chaînes.
- strlen calcule, à chaque appel (recherche du \0), la longueur de la chaîne (si répétition : utiliser une variable).
- Affectation possible avec = d'une valeur constante (entre " ") sinon strcpy
- Le passage de paramètre se fait par adresse (pointeur)


```
#include <stdio.h>
#include <stdlib.h>
int main(){
unsigned char i;
    int t[] = {1, 2, 3, 4, 5} , s[10];
    int *p=(int []){1, 2, 3, 4, 5}; // cast obligatoire
    printf("%d\n",p[3]);
    printf("%d\n",*(t+1));
    printf("%d\n",p[10]);
    printf("%d\n",t[10]);
    printf("%d , %d\n",sizeof(t),sizeof(p)); // 20 , 4
    printf("%x\n",(int)p);
    printf("%x\n",(int)t);
//  *(p+1)=9; // erreur à l'exécution
//  p[2]=7; // erreur à l'exécution
    *(t+2)=6;
    t[3]=8;                                          //  *(t+i)
    for(i=0;i<sizeof(t)/sizeof(int);i++)
    printf("%d,",t[i]);//1,2,6,8,5,
    return 0;
}
```


Nombre d'éléments d'un tableau

```
#define CARD( T ) ( sizeof( T ) / sizeof( T[0] ) )
int main () {
float donnees[ ] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};
int indice = ChercheVal( donnees,CARD(donnees), 4.);
....
}
```

— CARD(donnees) vaut 10 dans main()

mais attention:

```
int ChercheVal ( float tab[ ], int taille, float val ) {  ...
}
```

— CARD(tab) != 10 dans ChercheVal() !!!

— car le paramètre **tab** n'est pas un tableau mais un **pointeur** !

— ce qui revient à :

```
int ChercheVal ( float *tab , int taille, float val ) {  ...
}
```

➔ Ajouter un paramètre : la taille pour utiliser CARD en amont


```
#include <stdio.h>
int main(){
 int n;
 printf("nb elem =");
 scanf("%d",&n);
 {int i , t[n];
   for (i=0;i<n;i++){
     printf("t[%d]=",i);
     scanf("%d",&t[i]);
   }
   for (i=0;i<n;i++){
     printf("t[%d]=%d\n",i,t[i]);
   }
 }
 return 0;
}
```

```
gcc -g -Wall -pedantic -ansi -o tb1.exe tb1.c
tb1.c:7: warning: ANSI C forbids variable-size array `t'
```

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int n, i , *t;
  printf("nb elem =");
  scanf("%d",&n);
  t=(int *) malloc(n*sizeof(int));
  for (i=0;i<n;i++){
      printf("t[%d]=",i);
      scanf("%d",&t[i]);
  }
  for (i=0;i<n;i++){
      printf("t[%d]=%d\n",i,t[i]);
  }
  free(t) ;
  return 0;
}
```



```
void *memccpy(void *, const void *, int, size_t);
void *memchr(const void *, int, size_t);
int memcmp(const void *, const void *, size_t);
void *memcpy(void *, const void *, size_t);
void *memmove(void *, const void *, size_t);
void *memset(void *, int, size_t);
char *strcat(char *, const char *);
char *strchr(const char *, int);
int strcmp(const char *, const char *);
int strcoll(const char *, const char *);
char *strcpy(char *, const char *);
size_t strcspn(const char *, const char *);
char *strdup(const char *);
char *strerror(int);
size_t strlen(const char *);
char *strncat(char *, const char *, size_t);
int strncmp(const char *, const char *, size_t);
char *strncpy(char *, const char *, size_t);
char *strpbrk(const char *, const char *);
char *strrchr(const char *, int);
size_t strspn(const char *, const char *);
char *strstr(const char *, const char *);
char *strtok(char *, const char *);
char *strtok_r(char *, const char *, char **);
size_t strxfrm(char *, const char *, size_t);
```

Bibliothèque <string.h>


```
double acos(double);
double asin(double);
double atan(double);
double atan2(double, double);
double ceil(double);
double cos(double);
double cosh(double);
double exp(double);
double fabs(double);
double floor(double);
double fmod(double, double);
double frexp(double, int *);
double ldexp(double, int);
double log(double);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sin(double);
double sinh(double);
double sqrt(double);
double tan(double);
double tanh(double);
double erf(double);
```

Bibliothèque <math.h>

```
double erfc(double);
double gamma(double);
double hypot(double, double);
double j0(double);
double j1(double);
double jn(int, double);
double lgamma(double);
double y0(double);
double y1(double);
double yn(int, double);
int    isnan(double);
double acosh(double);
double asinh(double);
double atanh(double);
double cbrt(double);
double expm1(double);
int    ilogb(double);
double log1p(double);
double logb(double);
double nextafter(double, double);
double remainder(double,
    double);
double rint(double);
double scalb(double, double);
```


déclaration d'une structure

Une structure possède un nom et est composée de plusieurs champs. Chaque champ à son propre type et son propre nom. Pour déclarer un structure on utilise le mot-clé struct :

```
struct nomStructure {
    type1 champ1;
    ...
    typeN champN;
} ;
```


exemple : nombre complexe :

```
struct complex {
      double reel; /* partie reelle */
      double imag; /* partie imaginaire */
  };
```

accès aux champs

À partir de cette déclaration, il est possible d'utiliser ce nouveau type. L'opérateur . permet d'accéder à l'un des champs d'une structure. En continuant l'exemple précédent, les lignes suivantes initialisent un complexe à la valeur (2 + 3*i*).

```
struct complex a;

  a.reel = 2;
  a.imag = 3;
```


utilisation de typedef

typedef permet d'associer un nom à un type donné. On l'utilise suivi de la déclaration d'un type (en général une structure ou une union) puis du nom qui remplacera ce type. Ceci permet, par exemple, de s'affranchir de l'emploi de struct à chaque utilisation d'un complexe. Il n'est pas alors nécessaire de donner un nom à la structure. L'exemple précédent devient :

```
typedef struct {
      double reel; /* partie reelle */
      double imag; /* partie imaginaire */
  } complexe;
  complexe a;
  a.reel = 2;
  a.imag = 3;
```

initialisation et affectation

Il est possible d'affecter une variable de type structure dans une autre variable du même type. Le contenu de chacun des champs de la première variable sera alors recopié dans le champ correspondant de la seconde variable. On peut initialiser une variable de type structure dès sa définition en lui affectant une liste de valeurs séparées par des virgules et entourées par des accolades.

```
complexe b,a = { 1, 0 }; /* le reel 1 */
b = a;
```

Il est par contre impossible de comparer ou d'effectuer des calculs entre deux structures.

les pointeurs sur structures

L'utilisation de pointeurs sur structures est très courante en C. Voici un exemple d'utilisation d'un pointeur sur un complexe :

```
complexe a = { 3.5, -5.12 };
  complexe * p = &a;
  (*p).reel = 1;
  (*p).imag = -1;
  /* a vaut (1 - i) */
```

Nous avons été obligé de mettre des parenthèses autour de *p car l'opérateur . est plus prioritaire que l'opérateur * . Cela rend difficile la lecture d'un tel programme. Heureusement, l'utilisation de pointeurs sur structures est si courante que le C définit l'opérateur -> pour accéder aux champs d'une structure via un pointeur.

Les deux expressions suivantes sont donc équivalentes :

```
(*pointeur).champ
  pointeur->champ
```

Ainsi l'exemple précédent s'écrit beaucoup plus facilement de la manière suivante :

```
complexe a = { 3.5, -5.12 };
  complexe * p = &a;
  p->reel = 1;
  p->imag = -1;
  /* a vaut (1 - i) */
```