



Tableaux

- Ensemble d'emplacements mémoire groupés sous un seul nom et tous du même type.
- Chaque emplacement est un élément du tableau.
- Un index permet d'accéder à un élément du tableau.



Déclaration d'un tableau

```
typeDesElements nomDuTableau [nombreElements] ;
```

typeDesElements : type de chaque élément du tableau.

Exemples :

```
float temperatures [12] ;
```

Le tableau `temperatures` contient 12 éléments de type `float`

```
int notes [100] ;
```

Le tableau `notes` contient 100 éléments de type `int`.



Utilisation d'un tableau

- **nomTableau [index]**
 - index est une valeur entière comprise entre 0 et taille du **tableau** - 1.
 - index 0 : premier élément
 - index *nb éléments - 1* : dernier élément

Chaque élément du tableau s'utilise
comme une variable du même type.



Accès à un élément

- Le 3eme élément prend la valeur 20.1 :
`temperatures[2] = 20.1 ;`
- Le 3eme élément prend la valeur du 4eme élément :
`temperatures[2] = temperatures[3] ;`
- Initialisation du 2^{ème} élément par une valeur lue au clavier :
`scanf ("%f", &temperatures[1]) ;`



Initialisation d'un tableau

- Initialisation d'un tableau par une boucle :

```
int notes [100] ;
```

```
int i;
```

```
for (i = 0 ; i < 100 ; i++) {
```

```
    notes[i] = 1 ;
```

```
}
```

Les 100 éléments du tableau sont initialisés à 1.



Initialisation avec scanf

```
int notes [100] ;  
int i;  
for (i = 0 ; i < 100 ; i++) {  
    scanf ("%d", &notes [i];  
}
```



Initialisation d'un tableau

- Initialisation statique :

```
int notes [3] = {4, 5, 6} ;
```

Le tableau est initialisé à la déclaration.

Les valeurs sont attribuées aux éléments du tableau en commençant par le premier.

Les valeurs sont entre accolades, séparées par une virgule.



Accès au tableau

- Élément par élément
 - Avec un index : `tab [i]`
- Copie de tableaux :
 - Élément par élément :

```
for (i = 0 ; i < n ; i++) {  
    tab1[i] = tab2 [i];  
}
```




Taille d'un tableau

- Nombre d'octets occupés par le tableau
- Nombre d'éléments x taille d'un élément

sizeof (NomDuTableau)

```
int note [10];
```

```
sizeof(note) => 40 octets
```



Tableaux à plusieurs dimensions

- Un tableau à n dimensions a n index
- En langage C, le nombre de dimensions n'est pas limité.
- En pratique, le programmeur est limité dans la gestion des dimensions.



Utilisation de 2 dimensions

```
float temperatures [12][31] ;
```

- Pour stocker les températures moyennes journalières pour chaque mois.

```
int notesAnnuelles [16][3];
```

- Pour stocker les 3 notes annuelles de chacun des 16 étudiants.



Utilisation de 3 dimensions

```
float notes [50][10][3] ;
```

- Pour stocker les notes des 50 étudiants
 - 10 matières différentes
 - 3 notes par matières

```
notes [no][m][i] ;
```

- Note de l'élève numéro : no dans la matière m. i est le numéro de la note parmi les 3 notes données dans la matière (0, 1, 2).



Directives define pour la taille

- Utiliser des constantes définies par `#define` pour déclarer la taille des tableaux.

```
#define NBLIGNES 10
```

```
#define NBCOL 20
```

```
int tableau [NBLIGNES] [NBCOL];
```



Initialisation dynamique

```
#define NBETUDIANTS 16
#define NBNOTES 3
int notes [NBETUDIANTS][NBNOTES] ;
for (i=0 ; i < NBETUDIANTS ; i++) {
    for (j=0 ; j < NBNOTES ; j++) {
        notes[i][j] = 1 ;
    }
}
```

- Tous les éléments sont initialisés à 1

Initialisation statique



```
int matrices [2][5] =  
    { {1, 2, 3, 4, 5},  
      {6, 7, 8, 9, 10} } ;
```

- Le programme est plus lisible si on regroupe les valeurs entre accolades. Les virgules sont obligatoires même s'il y a des accolades.



Occupation mémoire

- *type* tableau [n][m];
- Nombre d'éléments : n x m
- Taille du tableau : n x m x sizeof (*type*)
- Exemple :

int table [2][3];

Nombre d'éléments : 2 x 3 = 6 éléments

Taille mémoire : 6 x sizeof (int)



Taille maximum

- Dépend de la mémoire disponible sur votre ordinateur.
(physique et virtuelle)



Utilisation de pointeurs

- Une variable occupe un emplacement mémoire.
- Chaque emplacement mémoire a une adresse.
- Un pointeur contient l'adresse d'une variable ou d'un élément de tableau



Déclaration d'un pointeur

- Déclaration d'un pointeur :
`type * nomPointeur;`
- Type : type de la valeur stockée
- Astérisque devant le nom de la variable
`char *ptrCar ;`
`int *ptrValeur;`
`int *pointeur, valeur;`



Initialisation d'un pointeur

- L'opérateur & permet d'obtenir l'adresse d'une variable.

```
int valeur;
```

```
int *ptrValeur;
```

```
ptrValeur = & valeur;
```

- `ptrValeur` est initialisé avec l'adresse de la variable `valeur`.
- Un pointeur doit toujours être initialisé avant utilisation



Adresse d'une variable

Mémoire :

Valeur : 154

Adresse 6400

Ptrvaleur : 6350

Adresse 7200



Adresse d'un pointeur

- Un pointeur est une variable : on peut lui appliquer l'opérateur &

```
int val;
```

```
int *pointeur1;
```

```
int **pointeur2;
```

```
pointeur1 = & val;
```

```
pointeur2 = &pointeur1;
```



Indirection

- L'opérateur d'indirection `*` : contenu d'une variable à partir de son adresse

```
int valeur1 = 3;
```

```
int *ptrValeur;
```

```
ptrValeur = &valeur1;
```

```
int valeur2;
```

```
valeur2 = *ptrValeur;
```

- valeur2 prend la valeur stocké à l'adresse contenu dans ptrValeur



Indirection

Mémoire :

valeur1 : 3

ptrvaleur : 6350

Adresse 6350

valeur2 : 3



Opérateur d'indirection

- Lecture (si à droite de l'affectation)
lire la valeur stockée à l'adresse contenue dans la variable
pointeur
valeur = *pointeur;
- Ecriture (si à gauche de l'affectation)
écrire la valeur 10 à l'adresse stockée dans la variable pointeur
*pointeur = 10;
- L'indirection est valide seulement sur un
pointeurs



Pointeurs et type de variables

- Un pointeur est toujours typé.
- Le compilateur a besoin de connaître le type de valeur stockée en mémoire

```
char *ptrTexte;
```

```
int *ptrValeur;
```



Pointeurs et tableaux

- Adresse de tableau :

```
int data [10] ;
```

```
int *ptr_data ;
```

```
ptr_data = data ;
```

- Le nom d'un tableau seul : adresse du premier élément du tableau
- Cette initialisation est équivalente à :

```
ptr_data = &data[0] ;
```



Stockage d'un tableau

```
int table[3], i;  
for (i = 0 ; i < 3 ; i++) {  
    printf ("Adr table[%d] :%d\n",  
            i, &table[i]);  
}
```

- Donne le résultat suivant :

Adr table[0] : 6684140

Adr table[1] : 6684144

Adr table[2] : 6684148



Stockage d'un tableau

```
char table[3];  
int i;  
for (i = 0 ; i < 3 ; i++) {  
    printf ("Adr table[%d] %d\n" ,  
           i, &table[i]);  
}
```

- Donne le résultat suivant :

Adr table[0] : 6684148

Adr table[1] : 6684149

Adr table[2] : 6684150



Incrémentation de pointeurs

```
int table [100];
```

```
int * pointeur;
```

```
pointeur = table;
```

```
pointeur ++ ;
```

- Le pointeur est incrémenté de la taille des éléments (int donc 4 octets)



Décrémentation de pointeurs

- Le pointeur est décrémenté de la taille des éléments pointés.

pointeur -= 3 ;

- Le pointeur est décrémenté de
3 x sizeof (*éléments*)



Exemple 1 : Initialisation d'un tableau

```
int table [10];  
int *pointeur ;  
int i;  
pointeur = table;  
for (i = 0 ; i < 10 ; i++) {  
    *pointeur = 1;  
    pointeur ++;  
}
```




Exemple 2 : scanf

```
int table [10];  
int *pointeur ;  
int i;  
pointeur = table;  
for (i=0 ; i<10 ; i++) {  
    scanf (" %d ", pointeur);  
    pointeur ++;  
}
```



Autres opérations

- Différence : $\text{pointeur1} - \text{pointeur2}$
Nombre d'éléments entre les 2 éléments pointés.
- Comparaison : $\text{pointeur1} \leq \text{pointeur2}$
Est-ce que l'élément pointé par pointeur1 est avant l'élément pointé par pointeur2 ?



Pointeurs dans une fonction

```
int somme (int *table, int n) {  
    int i;  
    int *pointeur = table;  
    for (i = 0 ; i < n ; i++) {  
        somme = somme + *pointeur;  
        pointeur++;  
    }  
    return somme;  
}
```



Passage de paramètres

```
#define N 100
```

```
...
```

```
int table [N];
```

```
int total;
```

```
...
```

```
/* Appel de la fonction somme */
```

```
total = somme (table, N);
```



Pointeurs dans tableaux à plusieurs dimensions

- le nom du tableau utilisé avec n-1 index est un pointeur sur un tableau à une dimension.

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

```
int *p;
```

```
p = tab [1];
```

```
/*tab[1] pointe sur la 2ème ligne */
```

```
printf ("%d", *p);
```

Ce programme imprime la valeur 4.



Chaînes de caractères

- Tableau de caractères avec le caractère '\0' (valeur 0) dernier caractère

```
char chaine [] = "Paris" ;
```

Équivalent à

```
char chaine [] =  
    {'P', 'a', 'r', 'i', 's', '\0'} ;
```



Chaîne littérale 1

- Chaîne littérale : **"Paris"**
- 1. Réservation de tableau implicite (6 octets) (nombre de caractères + 1)
- 2. Initialisation du tableau



Chaîne littérale et tableau

- **char chaine[] = "Paris" ;**
 - 1- Réservation d'un tableau de 6 caractères
 - 2- Initialisation du tableau



Chaîne littérale et pointeur

```
char *ptr_chaine = "Salut" ;
```

- 1. Réservation d'un pointeur (ptr_chaine)
- 2. Réservation d'un tableau anonyme (6 caractères)
- 3. Initialisation du tableau anonyme
- 4. Initialisation du pointeur par l'adresse du tableau anonyme



Fonction puts

- Tableau :

```
char message[] = "Bonjour" ;  
puts (message) ;
```

- Chaîne littérale :

```
puts ("Hello") ;
```

- Pointeur :

```
char *ptr_chaine = "Salut" ;  
puts (ptr_chaine) ;
```



Fonction printf

- Tableau :

```
char message[] = "Bonjour" ;  
printf ("%s", message) ;
```

- Chaîne littérale :

```
printf ("%s", "Hello");
```

- Pointeur :

```
char *ptr_chaine = "Salut" ;  
printf ("%s", ptr_chaine);
```



Fonctions de saisie gets

- Fonction gets

```
char input[81];  
gets (input) ;
```

Lit une ligne au clavier

Taille du tableau : il doit être assez grand

Problème : On ne peut pas limiter le
nombre de caractères saisis



Fonction de saisie scanf

- **scanf ("%s", input) ;**

%s => lit tous les caractères jusqu'au prochain séparateur (espace, tabulation, Entrée)

Taille du tableau : le nombre de caractères saisis n'est pas limité



Limite de longueur en lecture

```
scanf ("%10s", input) ;
```

%s => lit tous les caractères jusqu'au prochain séparateur (espace, tabulation, Entrée) et arrêt si 10 caractères ont déjà été saisis.

Donc prévoir un tableau de 11 caractères pour le caractère '\0'



Scanf et limite

```
char codepostal [5] ;  
scanf ( " %4s ", codepostal ) ;
```

- les 4 premiers caractères non blanc seront lus et le caractère '\0' sera ajouté.
- Arrêt sur rencontre d'un séparateur
- Sur 11124, codepostal prendra la valeur 1112.
- Sur 111 24, codepostal prendra la valeur 111.
- La fonction scanf ne permet pas de lire une chaîne de caractères vide (il en faut au moins 1).

Fonction strlen



```
#include <string.h>
```

- Longueur de la chaîne de caractères (non compris le caractère '\0').

```
char chaine [] = "Paris";  
printf (" Lg chaine : %d\n",  
        strlen(chaine)) ;
```

- Donne le résultat suivant :

```
Lg chaine : 5
```




Strlen et sizeof

- Strlen donne le nombre de caractères avant le caractère `'\0'`
- Sizeof est la taille du tableau



Fonction fgets

- Idem à gets mais avec limite du nombre de caractères saisis à N-1.

`fgets (chaine, N, stdin);`

Exemple :

`char chaine [10];`

`fgets (chaine, 10, stdin);`

Lecture d'une ligne de 9 caractères max
(1 char pour '\0')



Fonction sscanf

```
char chaine [] = "123 456" ;
```

```
int a, b ;
```

```
sscanf (chaine, "%d %d", &a, &b) ;
```



fgets et sscanf

```
char ligne [100];  
int valeur;  
int n;  
/* Lire une ligne */  
fgets (ligne, 100, stdin);  
/* Initialiser une variable */  
n=sscanf (ligne,"%d",&valeur) ;
```



Concaténation de chaînes

s1, s2 : chaînes de caractères

(Tableaux de caractères avec \0)

strcat (s1, s2) : concatène s2 à la suite de la chaîne s1

strncat (s1, s2, n) : concatène au plus n caractères s2 à la suite de s1

s1 sera toujours terminée par \0

Attention à la taille du tableau s1.



Concaténation - strcat

```
char ch1[50] = "Bonjour ";  
char ch2[50] = "Monsieur";  
strcat (ch1, ch2) ;  
printf ("%s\n", ch1);
```

Résultat :

Bonjour Monsieur



Taille du tableau

- $\text{strlen}(s1) + \text{strlen}(s2) + 1$
 $\leq \text{sizeof}(s1)$



Concaténation - strncat

```
char ch1[50] = "Bonjour ";  
char ch2[50] = "Monsieur";  
strncat (ch1, ch2, 5) ;  
printf ("%s\n", ch1);
```

Résultat :

Bonjour Monsi



Copie de chaînes : strcpy

- **strcpy (destin, source)** : copie **source** (y compris '\0') dans la chaîne **destin**
- **strncpy (destin, source, n)** :
copie au plus n caractères de **source** dans **destin**.
 - Si **source** a moins de n caractères
 - Les caractères manquants sont remplacés par \0
 - Si **source** a plus que n caractères
 - Les n premiers seront pris : dans ce cas **destin** ne sera pas terminée par \0
- **Attention à la taille de destin**



Copie de chaînes - exemples

```
char ch1[] = "xxxxxxxxxxxxxxxxxxxxxxxx";  
char ch2[50];  
printf (" Donner un mot : ");  
gets (ch2) ;  
strncpy (ch1, ch2, 7) ;  
printf ("%s",ch1);
```

Donne les résultats suivants :

Donner un mot : Bon

Resultat : Bon

Et :

Donner un mot : Bonjour

Resultat : Bonjourxxxxxxxxxxxxxxxx



Comparaison de chaînes

- **strcmp (s1,s2)** : compare les 2 chaînes de caractères s1 et s2.
- **strncmp (s1,s2,n)** : compare seulement les n premiers caractères.
- Résultat :
 - >0 si s1 > s2
 - ==0 si s1 == s2
 - <0 si s1 < s2



Utilisation de strcmp

```
char chaine1 [20];  
char chaine2 [20];  
scanf ("%s %s",chaine1, chaine2);  
  
if(strcmp(chaine1,chaine2)== 0) {  
    printf (" Chaines égales ") ;  
}
```



Recherche dans une chaîne

- **strchr** (**chaine**, **car**) : recherche dans **chaine** la première position où apparaît le caractère **car**.
- **strrchr** (**chaine**, **car**) : même traitement que **strchr** mais en commençant par la fin.
- **strstr** (**chaine1**, **chaine2**) : recherche dans **chaine1** la première position où apparaît la première occurrence complète de **chaine2**.
- Résultat : **adresse** de l'information cherchée ou NULL si pas trouvée.



Exemple

```
char ligne [100];  
char *pointeur;  
int nb = 0;  
fgets (ligne, 100, stdin);  
pointeur = strchr (ligne, 'e');  
while (pointeur != NULL) {  
    nb++;  
    pointeur = strchr (pointeur+1, 'e');  
}
```



Fonctions de librairies

- **#include** <ctype.h>
- Résultat 0 si Faux et !=0 si Vrai
- **isalpha(c)** Caractère alphabétique
- **isdigit(c)** Chiffre décimal
- **isalnum(c)** Caractère alphabétique ou chiffre décimal
- **islower(c)** Lettre minuscule
- **isupper(c)** Lettre majuscule
- **ispunct(c)** Caractère imprimable de ponctuation différent de l'espace, des lettres et des chiffres
- **isspace(c)** Espace, saut de page, fin de ligne, retour chariot, tabulation



Fonctions de librairie (suite)

toupper(c) : c en majuscule

tolower(c) : c en minuscule

#include <stdlib.h>

atoi(chaine) : ascii to integer

atof(chaine) : ascii to float



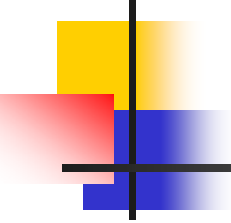
Exemple

```
void main () {  
    char chaine[100];  
    unsigned i;  
    int nbLettre, nbChiffre, nbMaj, nbMin;  
  
    nbLettre = 0; /* Nombre de lettres */  
    nbChiffre = 0; /* Nombre de chiffres */  
    nbMaj = 0; /* Nombre de majuscules */  
    nbMin = 0; /* Nombre de minuscules */  
    printf ("Entrer une ligne de texte :\n");  
    fgets (chaine, 100, stdin);
```



Exemple (suite)

```
for (i=0 ; i<strlen(chaine) ; i++) {  
    if (isalpha (chaine[i]) ){  
        nbLettre++;  
        if (islower(chaine[i])) {  
            nbMin++;  
        } else {  
            nbMaj++;  
        }  
    } else if (isdigit(chaine[i]))  
        nbChiffre++;  
}
```



```
printf ("Nombre de lettres      :  
        %d\n", nbLettre);  
printf ("Nombre de majuscules  :  
        %d\n", nbMaj);  
printf ("Nombre de minuscules  :  
        %d\n", nbMin);  
printf ("Nombre de chiffres     :  
        %d\n", nbChiffre);  
}
```



Tableau de pointeurs

- Déclaration : `type *tab [n] ;`
- `type` est le type des éléments pointés par les pointeurs, éléments du tableau
- Exemple :

```
char *jour [ ] =  
    { "lundi", "mardi", "mercredi",  
      "jeudi", "vendredi", "samedi",  
      "dimanche" } ;
```



Tableaux de pointeurs

```
char *liste_noms[20] ;
```

Tableau de 20 pointeurs sur char

```
int *tab_pointeurs[25] ;
```

Tableau de 25 pointeurs sur int



Pointeur sur pointeur

- Utilisation de 2 signes * consécutifs

```
char **ptr_tab = jour;
```

- ptr_tab est un pointeur de pointeurs, initialisé avec l'adresse du tableau de pointeurs jour.

***ptr_tab**: pointeur sur la chaîne "lundi"

****ptr_tab** : caractère 'l'



Gestion de mémoire

- Taille mémoire nécessaire inconnue à la compilation.
- Allocation d'un bloc mémoire : malloc
 - Paramètre : taille mémoire demandée
 - Résultat : adresse du bloc mémoire alloué
- Libération d'un bloc mémoire : free



Mémoire : Allocation dynamique

```
#define LGMAX 30
```

```
char *pointeur;
```

```
pointeur= (char *) malloc (LGMAX);
```

Allocation d'un bloc mémoire de 30 octets
pour un tableau de caractères (char)

```
scanf ("%29s", pointeur) ;
```

```
printf ("Chaine:%s\n ",pointeur);
```




Mémoire : allocation dynamique

```
#define NB 30
```

```
int *pointeur;
```

```
pointeur =(int *) malloc (NB*sizeof (int));
```

```
/* Nbre d'éléments * taille d'un entier */
```

```
for (i=0 ; i< NB ; i++) { pointeur[i]=33 };
```

```
free (pointeur);
```

Réservation de mémoire pour un tableau d'éléments de type int.



Les fonctions

- Une fonction :
 - bloc d'instructions, référencé par un nom
 - qui réalise une tâche
 - qui peut renvoyer une valeur au programme qui l'a appelée.
- Nom d'une fonction : identificateur
- Arguments : informations données à la fonction
- Résultat : Valeur rendue par la fonction



Prototype de la fonction

- Description de la fonction :
 - Nom de la fonction
 - Type de la valeur de retour
 - Type des arguments (et noms en option)

```
int surface_rectangle(int largeur,  
                      int longueur) ;
```



Définition de la fonction

- Entête de la fonction :
 - Nom de la fonction
 - Type de la valeur de retour
 - Type et nom des arguments
- Bloc d'instructions entre accolades.
- Instruction return
 - Sortie de la fonction
 - Transmet le résultat à la fonction appelante



Définition de la fonction

```
type_retour nom_fonction
    (type_arg1 nom-1, ..., type_argn nom-n) {
    ...
    /* Instructions */
    ...
    return valeur ; // Sortie de la fonction
}
```



Appel de la fonction

- Appel :
 - Nom de la fonction
 - Valeurs des paramètres
 - Utilisation du résultat
- Exemple
`res = surface_rectangle (3, 5);`



Exemple de définition

```
int surface_rectangle(int largeur,  
                      int longueur) {  
    int res;  
  
    res = largeur * longueur;  
    return res;  
}
```



Placement des fonctions

```
/* Debut du code */
/* prototypes des fonctions */
type1 fonct1(type_arg1 arg1, type_arg2 arg2,...);
...
void main () {
    ...
}
type1 fonct1(type_arg1 arg1, type_arg2 arg2,...) {
    ...
}
type2 fonct2() {
}
```




Exemples de fonction

```
/* Prototype ou declaration de la fonction */
int cube (int x) ;
void main () {
    int val;
    int resultat;
    printf ("Entrer une valeur : ");
    scanf ("%d", &val);
    /* Appel de la fonction cube avec un argument */
    resultat = cube (val);
    printf ("Le cube de %d est %d\n", val, resultat);
}
/* Definition de la fonction cube */
/* La fonction a un argument x et rend un résultat de type int */
int cube (int x) {
    /* Declaration d'une variable locale */
    int cube_x;
    cube_x = x * x * x;
    /* Renvoi d'une valeur */
    return (cube_x);
}
```



Données d'une fonction

- Variables locales : déclarées après l'entête de la fonction.
- Arguments : valeurs données par le programme appelant à la fonction.
Ils sont utilisés comme des variables locales dans la fonction.
- Résultat : valeur transmise au programme appelant par return.



Variables locales

```
int cube (int x) {  
    /* Declaration d'une variable locale */  
    int cube_x;  
  
    cube_x = x * x * x;  
    /* Renvoi d'une valeur */  
    return (cube_x);  
}
```

- Elle est réservée au début de l'exécution de la fonction et elle disparaît sur l'instruction return.



Résultat

- Type void si il n'y a pas de résultat :

```
void ecritBonjour () {  
    puts ("Bonjour"); /*Cette fonction n'a pas  
                        d'argument et pas de resultat*/  
}
```

- L'instruction return doit être exécutée en fin de fonction :

```
int fonct1 (int a, int b) {  
    return (a * b) ;  
    /* Le resultat est la multiplication des 2 arguments  
    */  
}
```



Passage des paramètres

- En langage C, les paramètres d'une fonction sont passés par valeur.
- Ils sont évalués et le résultat de l'évaluation est copié dans une zone mémoire réservée pour les paramètres de la fonction



Passage des paramètres

- Exemple

```
int val;
```

```
    val = 5;
```

```
fonct (val + 2);
```

La valeur 7 sera passée à la fonction



Passage de paramètres

- Les valeurs des variables passées en paramètres ne peuvent pas être modifiées par la fonction appelée.



Passage par adresse

- Lorsqu'une fonction a besoin de modifier une variable du programme appelant, on passe un pointeur sur la variable à la fonction appelée.
- Opérateur & permet d'obtenir l'adresse d'une variable
- Opérateur * permet d'obtenir la valeur d'une variable quand on a son adresse



Les pointeurs

- Le langage C permet l'utilisation d'adresses mémoire.
- Une adresse mémoire peut être stockée dans une variable de type pointeur.



Déclaration d'un pointeur (1)

- Une variable de type pointeur se déclare ainsi :

```
type *identificateur;
```

- Exemples :

```
int *ptrValeur;
```

```
char *ptrCar;
```

```
float *ptrSalaire;
```



Déclaration pointeur (2)

- Le compilateur a besoin de savoir le type de la valeur pointée parce qu'il doit savoir la longueur de la valeur pointée.
- Un pointeur contient l'adresse du premier octet de la valeur pointée.



Opérateurs sur les pointeurs

- L'opérateur **&** permet d'obtenir l'adresse d'une variable
- L'opérateur ***** permet d'obtenir la valeur stockée à l'adresse contenue dans le pointeur.



Adresses et valeurs de variable

```
/* Declaration de 2 entiers */  
int toto = 333;  
int valeur;  
/* Declaration d'un pointeur */  
int *ptrEntier ;  
  
ptrEntier = &toto;//adresse de toto  
valeur = *ptrEntier;//valeur pointée
```



Adresses et Valeurs

toto

333

(Adresse = 23450)

**ptrEntier
(&toto)**

23450

**Valeur
(*ptrEntier)**

333



Déclaration des paramètres passés par adresse

- `type-res fonction (type1 *param1, type2 *param2...);`
- param1 et param2 sont des pointeurs sur des variables
- L'opérateur * placé devant le nom du paramètre permet de déclarer un pointeur.



Exemple

```
#include <stdio.h>

/* Prototype des fonctions */
/* Les arguments sont de type int */
void doublerParValeur (int a,
                      int b, int c);

/* Les arguments sont des pointeurs */
void doublerParAdresse (int *a,
                       int *b, int *c);
```




Exemple (suite)

```
void main () {
    int i, j, k;
    i = 1;
    j = 2;
    k = 3;
    printf ("Avant : i : %d, j : %d, k : %d\n", i, j, k);
    /* On passe les valeurs des variables */
    doublerParValeur (i, j, k);
    printf ("Apres par valeur : i : %d, j : %d, k : %d\n",
            i, j, k);
    /* Dans ce cas, on passe les adresses des variables */
    doublerParAdresse (&i, &j, &k);
    printf ("Apres par adresse: i : %d, j : %d, k : %d\n",
            i, j, k);
}
```



Exemple (suite)

```
void doublerParValeur (int a, int b, int c) {  
    /* On peut modifier la valeur des arguments mais */  
    /* on ne change pas la variable du programme appelant */  
    a *= 2;  
    b *= 2;  
    c *= 2;  
}  
  
void doublerParAdresse (int *a, int *b, int *c){  
    /*      Modification      des      valeurs      en      utilisant  
    l'indirection */  
    *a *= 2;  
    *b *= 2;  
    *c *= 2;  
}
```



Résultat de l'exemple

Le résultat de ce programme est :

Avant : i : 1, j : 2, k : 3

Après par valeur : i : 1, j : 2, k : 3

Après par adresse: i : 2, j : 4, k : 6



Tableau en paramètre

- Pour passer un tableau en paramètre, on donne le nombre de dimensions :

```
int somme (int table [], int n);
```

```
int tableau [100];
```

```
int s;
```

```
...
```

```
s = somme (tableau, 100);
```



Fonction avec tableau

```
int somme (int table[], int n) {  
    int i;  
    int res = 0;  
    for (i = 0 ; i < n ; i++) {  
        res = res + table[i];  
    }  
    return res;  
}
```



Tableau en paramètre

- Le paramètre peut aussi se déclarer ainsi :

```
int somme (int *table, int n);
```

```
int tableau [100];
```

```
int s;
```

```
...
```

```
s = somme (tableau, 100);
```



Tableau à 2 dimensions en paramètre

- Déclaration du paramètre

- `int fctcalcul (int table[][nbcol],
int nblignes, int nbcolonnes);`

- Il faut préciser le nombre d'éléments dans la 2ème dimension.

- Appel de la fonction

- `fctcalcul (table, 20, 30);`

- On donne le nom du tableau