

# INTRODUCTION AU LANGAGE C

## STRUCTURES DE DONNÉES



# LIMINAIRE

Langage introduit par B. W. Kernighan et D. Ritchie au début des années 70, normalisé par l'ANSI en 1983

**Un présumé :** le programmeur « *sait ce qu'il fait* » 😊

Langage très faiblement typé : les types de données sont très restreints et proches de la représentation interne des processeurs.

# INTRODUCTION

## Les structures de données

Elles permettent de stocker et d'organiser un ensemble de données dans une « *variable commune* ». Ainsi, pour accéder à ces valeurs, il suffit de parcourir la variable de type complexe composée de « variables » de type **simple**.

Le langage C propose deux types de « *structures* »

- Les **tableaux** qui permettent de stocker plusieurs données de même type.
- Les **structures** qui peuvent contenir des données hétérogènes.

# INTRODUCTION

## Les types abstrait de données

Un type abstrait ou une structure de données abstraite est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer

On qualifie **d'abstrait** ce type de données car il correspond à un cahier des charges qu'une **structure de données** doit ensuite implémenter

**Les types abstraits sont des entités purement théoriques utilisés principalement pour simplifier la description des algorithmes**

### *Exemple*

Le type abstrait de données (TAD) pile sera défini par 2 opérations :

- PUSH qui insère un élément dans la structure
- POP qui extrait un élément de la structure, retourne l'élément qui a été empilé le plus récemment et qui n'a pas encore été dépilé

# INTRODUCTION

## Structures de données et type abstrait

- Quand la nature des données n'a pas d'influence sur les opérations à effectuer, on parle alors de **type abstrait** générique et on fait la confusion avec les structures de données
- On considérera que les **structures de données** sont indépendantes du type des données et qu'elles sont définies par l'ensemble des opérations qu'elles effectuent sur ces données

## Structures de données

- Permettent de gérer et d'organiser des données
- Sont définies à partir d'un ensemble d'opérations qu'elles peuvent effectuer sur les données

# TABLEAU

On appelle tableau un ensemble d'éléments de même type désignés par un identificateur unique; chaque élément est repéré par un « indice » précisant sa position au sein de l'ensemble.

Les éléments du tableau peuvent être :

- Des données de type simple : int, char, float, long, double, ...
- Des tableaux;
- Des structures.

Exemple de représentation d'un tableau

1	2	8	3	5	7	4	6	9
---	---	---	---	---	---	---	---	---



# TABLEAU MONODIMENSIONNEL

Un tableau monodimensionnel est un tableau qui contient des éléments simples (des éléments qui ne sont pas des tableaux).

La syntaxe de la définition :

***type Nom\_du\_tableau[Nombre\_d\_elements\_du\_tableau];***

- ***type*** définit le type d'éléments que contient le tableau (ex: int)
- ***Nom\_du\_tableau*** est le nom que l'on décide de donner au tableau, le nom du tableau suit les mêmes règles qu'un nom de variables.
- ***Nombre\_d\_elements\_du\_tableau*** est un nombre entier qui détermine le nombre d'éléments que le tableau doit contenir.

*Exemple : la définition d'un tableau qui doit contenir 10 éléments de type entier est : ***int monTab[10];****

# TABLEAU MONODIMENSIONNEL

La **taille d'un tableau** est le nombre d'éléments que contient ce tableau. La taille d'un tableau ne peut être qu'une **constante** ou une **expression constante**.

➤ Exemple : **#define** N 10

*tab[N]; tab[2\*N-1];*

Un **indice** ou position permet de repérer chaque élément du tableau

➤ L'indice du premier élément du tableau est **0**

➤ Un indice est toujours positif

➤ L'indice du dernier élément du tableau est égal nombre d'éléments-1

Pour pouvoir **accéder un élément** du tableau, il suffit de donner le nom du tableau, suivi de l'indice de l'élément entre crochets :

➤ Exemple : Nom\_du\_tableau[indice]

➤ Exemple : Nom\_du\_tableau[8] permet d'accéder au 9<sup>ème</sup> élément du tableau



# TABLEAU MONODIMENSIONNEL

Un élément du tableau (repéré par le nom du tableau et son indice) peut être manipulé exactement comme une variable, on peut donc effectuer des opérations avec (ou sur) des éléments de tableau.

Exemple :

```
int fama[10]; //définit un tableau de nom fama et de taille 10
```

```
fama[5]=14; //permet d'affecter la valeur 14 au 6ème élément
```

```
fama[9] = fama[0]+fama[1] /* permet d'affecter au 10ème  
élément de fama le résultat de l'addition du 1er et 2ème de fama*/
```

# TABLEAU MONODIMENSIONNEL : INITIALISATION

Lorsque que l'on définit un tableau, les valeurs des éléments qu'il contient ne sont pas définies, il faut donc les initialiser, c'est-à-dire leur affecter une valeur.

Exemple :

```
int tab[5] = {5,8,4,10,12}; /*place les valeurs 5,8,4,10,12 dans  
chacun des cinq éléments du tableau*/
```

```
int tab[] = {5,8,4,10,12}; /*place les valeurs 5,8,4,10,12 dans  
chacun des cinq éléments du tableau; la dimension du tableau est  
déterminé par le compilateur et est égale au nombre d'éléments  
énumérés dans l'initialisation*/
```

# TABLEAU MONODIMENSIONNEL : INITIALISATION

Une autre manière d'initialiser un tableau à un indice consiste à utiliser une boucle

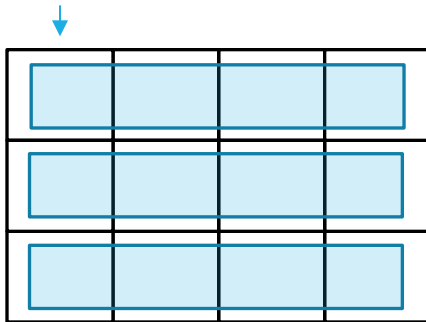
```
int tab[10];  
  
int i;  
  
for(i=0; i < 10; i++) {  
    tab[i]=0;  
}
```

Cette méthode, n'a d'intérêt que lorsque les éléments du tableau doivent être initialisés à une valeur unique ou logique (proportionnelle à l'indice par exemple)

# TABLEAUX MULTIDIMENSIONNELS

Les tableaux multidimensionnels sont des tableaux qui contiennent des tableaux.

Exemple




Un tableau bidimensionnel 3 lignes 4 colonnes.  
Ce tableau est un tableau de 3 éléments,  
chaque élément étant un tableau de quatre  
éléments.

# TABLEAUX MULTIDIMENSIONNELS

Un tableau multidimensionnel se définit de la manière suivante:

```
type Nom_du_tableau[a1][a2]...[aN]
```

Chaque élément entre crochets désigne le nombre d'éléments dans chaque dimension

Le nombre de dimensions n'est pas limité

Exemple s: un tableau d'entiers positifs à deux dimensions (3 lignes, 4 colonnes) se définira avec la syntaxe suivante :

```
int tab[3][4];
```

une représentation de ce tableau est la suivante :

<b>Tab[0][0]</b>	<b>Tab[0][1]</b>	<b>Tab[0][2]</b>	<b>Tab[0][3]</b>
<b>Tab[1][0]</b>	<b>Tab[1][1]</b>	<b>Tab[1][2]</b>	<b>Tab[1][3]</b>
<b>Tab[2][0]</b>	<b>Tab[2][1]</b>	<b>Tab[2][2]</b>	<b>Tab[2][3]</b>

# TABLEAU MULTIDIMENSIONNEL : INITIALISATION

L'initialisation d'un tableau multidimensionnel se fait à peu près de la même façon que pour les tableaux unidimensionnels. Il y a donc plusieurs façons d'initialiser un tableau multidimensionnel.

**Initialisation grâce à des boucles :**

```
int i,j;
for(i=0; i<=2; i++) {
    for(j=0; j<=3; j++) {
        tab[i][j]=0;
    }
}
```

**Initialisation à la définition :**

```
int tab[3][4] = { {1,2,3,4},
                  {5,6,7,8},
                  {9,10,11,12}}
```

# POINTEUR



Un **pointeur** est une variable contenant l'adresse d'une autre variable d'un type donné. Il permet de définir des structures dynamiques, c'est-à-dire qui évoluent au cours du temps (par opposition aux tableaux par exemple qui sont des structures de données statiques, dont la taille est figée à la définition).

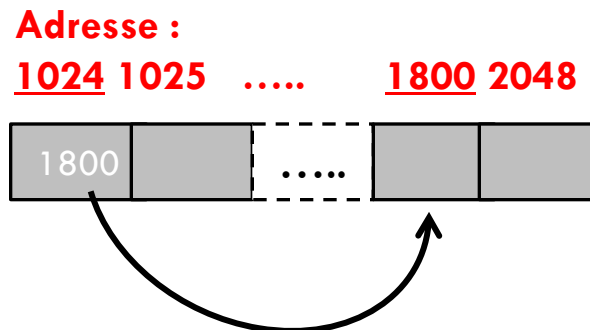
La mémoire est constituée de « cases » (octets). Une variable selon son type (donc sa taille), va occuper une ou plusieurs de ces cases appelées « blocs ».

Chacune de ces cases est identifiée par un numéro. Ce numéro s'appelle l'**adresse**.

# POINTEUR

On peut accéder à une variable de deux façons : grâce à **son nom** ou grâce à **l'adresse** du premier bloc alloué à la variable. Il suffit donc de stocker l'adresse de la variable dans un pointeur afin de pouvoir accéder à celui-ci (on dit que l'on « *pointe vers la variable* »)

Exemple :



Le pointeur stocké à l'adresse **1024**  
Pointe vers une variable stockée à  
l'adresse **1800**



# POINTEUR

L'opérateur **&** « *esperluète* » permet de désigner l'adresse d'une variable. Il suffit de faire précéder le nom de la variable par le caractère **&** pour désigner l'adresse de cette variable

Syntaxe : **&***Nom\_de\_la\_variable*

Déclaration d'un pointeur : un pointeur est une variable qui doit être définie en précisant le type de variable pointée, de la façon suivante :

*type* **\****Nom\_du\_pointeur*;

# POINTEUR : INITIALISATION

Après avoir déclaré un pointeur il faut l'**initialiser**. Cette démarche est très importante car lorsqu'on déclare un pointeur, celui-ci contient ce que la case où il est stocké contenait avant, c'est-à-dire n'importe quel nombre.

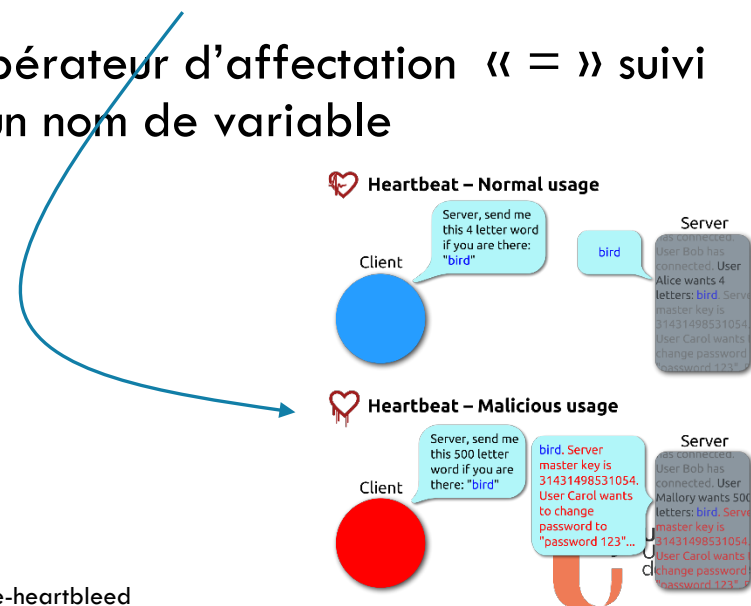
Si on initialise pas notre pointeur, celui-ci risque de pointer vers une zone hasardeuse de notre mémoire, ce qui peut être un morceau de notre programme ou ... de notre système d'exploitation (peut constituer une faille de sécurité) !

Pour initialiser un pointeur en C, il faut utiliser l'opérateur d'affectation « = » suivi de l'opérateur adresse « & » auquel est accolé un nom de variable

Exemple : `int *ptr;`

`int a = 10;`

`ptr = &a;`



# POINTEUR : ACCÉDER À UNE VARIABLE POINTÉE

L'opérateur « \* » du langage C permet d'accéder au contenu de l'adresse mémoire pointée par le pointeur. Pour pouvoir accéder au contenu le pointeur doit être déclaré et initialisé.

Exemple :

```
int *ptr1;  
int *ptr2;  
int a = 10;  
int b;  
ptr1 = &a;  
ptr2 = &b;  
*ptr1 = 15;  
*ptr2 = 20;
```

Le contenu des variables a et b sera respectivement 15 et 20

# POINTEUR : INTÉRÊT

Les pointeurs permettent de manipuler de façon simple des données pouvant être importantes. Au lieu de passer à une fonction un élément très grand (de taille), on pourra par exemple lui fournir un pointeur vers cet élément !

Les **tableaux** ne permettent de stocker **qu'un nombre fixé** d'éléments de **même type**. En stockant des pointeurs dans les cases d'un tableau, il sera possible de stocker des éléments de taille diverse, et même de rajouter des éléments au tableau en cours d'utilisation (c'est la notion de tableau dynamique qui est très étroitement liée à celle de pointeur)

Il possible de créer des structures chaînées, c'est-à-dire comportant des maillons.

# POINTEUR : LES OPÉRATIONS

## Incrémentation :

- l'opérateur ++ sur un pointeur permet de le faire passer à l'adresse suivante.
- On peut faire augmenter un pointeur d'une quantité entière quelconque.

*Exemple :*

```
int *p;
```

```
p++ ou ++p // passe à l'adresse suivante du pointeur
```

```
p = p+1 ou p +=1 // passe à l'adresse suivante du pointeur
```

```
p = p+10 ou p +=10 // passe à la deuxième adresse après celle de p
```

## Décrémentation :

- l'opérateur -- sur un pointeur permet de le faire passer à l'adresse précédente.
- On peut faire diminuer un pointeur d'une quantité entière quelconque

*Exemple :*

```
int *p;
```

```
p-- ou --p // passe à l'adresse précédente du pointeur
```

```
p = p-1 ou p -=1 // passe à l'adresse précédente du pointeur
```

```
p = p-10 ou p -=10 // passe à la deuxième adresse avant celle de p
```

# POINTEUR : LES OPÉRATIONS

## Soustraction :

- l'opérateur binaire `—` sur deux pointeurs de même type fournit le nombre d'éléments de ce type situés entre les deux adresses correspondantes aux deux pointeurs.

*Exemple :*

```
int *p;
```

```
int *q;
```

```
int tab[10];
```

```
p = tab;
```

```
q = tab;
```

```
p += 10;
```

**p-q donnera 10;**

# POINTEUR ET TABLEAU

En C, un nom de tableau est un pointeur constant. Lorsqu'un nom de tableau est employé seul, il est considéré comme un pointeur constant sur le début du tableau.

Exemple

**int** *tab*[10]; la notation *tab* est équivalente à *&tab*[0]; *tab* est considéré ici comme étant de type pointeur sur **int**, c'est-à-dire le type de *tab* est équivalent à **int\***

*tab*+1 est équivalent à *&tab*[1];

*tab*+*i* est équivalente à *&tab*[*i*];

*tab*[*i*] est équivalente à *\*(tab*+*i*);

Exemples de morceau de programme :

```
int i;
```

```
for(i=0;i<10;i++) *(tab+i)=1;
```

```
int *fin=tab+10;
```

```
int *ptr;
```

```
for(ptr=tab;ptr!=fin; ptr++) *ptr=1;
```

# CHAÎNE DE CARACTÈRES

Une **chaîne de caractère** est une suite de caractères, c'est-à-dire un ensemble de symboles faisant partie du jeu de caractères, défini par le code ASCII.

En langage C, une chaîne de caractères est un tableau, comportant plusieurs données de type **char**, dont le dernier élément est le caractère nul « **\0** », c'est-à-dire le premier caractère du code ASCII (dont la valeur est 0). Ce caractère est un **caractère de contrôle** (donc non affichable) qui permet d'indiquer une fin de chaîne de caractères. Ainsi une chaîne composée de  $n$  éléments sera en fait un tableau de  $n+1$  éléments de type **char**.

Exemple : La représentation de la chaîne « Bonjour »

B	o	n	j	o	u	r	\0
---	---	---	---	---	---	---	----



# CHAÎNE DE CARACTÈRES

Pour définir une chaîne de caractères en langage C, il suffit de définir un tableau de caractères. Le nombre maximum de caractères que comportera la chaîne sera égal au nombre d'éléments du tableau moins un (réservé au caractère de fin de chaîne).

Exemple :

```
char Nom_du_tableau[Nombre_d_elements]
```

```
char maChaine[8]; //pour initialiser la chaîne bonjour
```

```
maChaine[0]='B';  
maChaine[1]='o';  
maChaine[2]='n';  
maChaine[3]='j';  
maChaine[4]='o';  
maChaine[5]='u';  
maChaine[6]='r';  
maChaine[7]='\0';
```

# CHAINE DE CARACTÈRES : AFFICHAGE D'UNE CHAÎNE

Exemples :

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

```
void main() {
```

```
    char maChaine [7+1] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

```
    int i;
```

```
    for(i=0; i < 7; i++) {
```

```
        printf("%c", maChaine[i]);
```

```
    }
```

```
}
```

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

```
void main() {
```

```
    char* maChaine = "Bonjour";
```

```
    while(*maChaine) {
```

```
        printf("%c", *maChaine);
```

```
        maChaine++;
```

```
    }
```

```
}
```

# CHAINE DE CARACTÈRES : AFFICHAGE D'UNE CHAÎNE

*Exemples :*

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

```
int main() {
```

```
    char nom[20], prenom[20], ville[25];
```

```
    printf("quelle est votre ville : ");
```

```
    scanf("%s", ville);
```

```
    printf("quelle est votre nom et prenom : ");
```

```
    scanf("%s %s", nom, prenom);
```

```
    printf("bonjour cher %s %s qui habitez à %s" , prenom, nom, ville);
```

```
    return 0;
```

```
}
```

# STRUCTURES DE DONNÉES

Les structures de données permettent de désigner sous un seul nom un ensemble de valeurs pouvant être de types différents. L'accès à chaque élément de la structure de données (nommé champ) se fera par son nom au sein de la structure.

**Déclaration d'une structure** : lors de la déclaration de la structure, on indique les champs de la structure, c'est-à-dire le type et le nom des variables qui la composent :

*Exemple :*

```
struct Nom_Structure {  
    type_champ1 Nom_champ1;  
    type_champ2 Nom_champ2;  
    type_champ3 Nom_champ3;  
    ....  
    type_champN Nom_champN;  
};
```

# STRUCTURES DE DONNÉES

## Déclaration d'une structure :

- La dernière accolade doit être suivie d'un point-virgule !
- Le nom des champs répond aux critères des noms de variable.
- Deux champs ne peuvent avoir le même nom.
- Les données peuvent être de n'importe quel type hormis le type de la structure dans laquelle elles se trouvent.

*Exemple :*

```
struct personne {  
    char* prenom;  
    char* nom;  
    char* sexe;  
    char* tel;  
    int age;  
};
```

# STRUCTURES DE DONNÉES

**Définition d'une variable structure** : la définition d'une variable structure est une opération qui consiste à créer une variable ayant comme type celui d'une structure que l'on a précédemment déclarée, c'est-à-dire la nommer et lui réserver un emplacement en mémoire. La définition d'une variable structurée se fait comme suit :

```
struct Nom_Structure Nom_Variable_Structure;
```

Exemple avec la structure *Personne* définie précédemment :

```
struct personne p1;
```

# STRUCTURES DE DONNÉES

**Accès aux champs d'une variable structurée** : chaque variable de type structure possède des champs repérés avec des noms uniques. Toutefois le nom des champs ne suffit pas pour y accéder étant donné qu'ils n'ont de contexte qu'au sein de la variable structurée. Pour accéder aux champs d'une structure on utilise l'opérateur de champ (un simple point) placé entre le nom de la variable structurée que l'on a défini et le nom du champ:

*Nom\_Variable.Nom\_Champ;*

Exemple avec la structure **Personne** définie précédemment : pour affecter des valeurs à la variable **personne1** (de type **struct Personne** défini précédemment), on pourra écrire :

```
p1.prenom = "Moussa";  
p1.nom = "Diarra";  
p1.sexe = "masculin";  
p1.tel = "77875214";  
p1.age = 31;
```

# STRUCTURES DE DONNÉES

**Accès aux champs d'un pointeur une variable structurée** : l'opérateur `->` permet d'accéder aux différents d'une structure à partir de son adresse de début.

*Nom\_Variable->Nom\_Champ;*

Exemple avec la structure *Personne* définie précédemment :

```
struct personne * personnePtr;  
personnePtr->prenom = "Amadou";  
personnePtr->nom = "Diarra";  
personnePtr->sexe = "masculin";  
personnePtr->tel = "77875218";  
personnePtr->age = 35;
```



# STRUCTURES DE DONNÉES

Etant donné qu'une **structure** est composée d'éléments de taille fixe, il est possible de créer un **tableau** ne contenant que des éléments du type d'une structure donnée. Il suffit de créer un tableau dont le type est celui de la structure et de le repérer par un nom de variable :

```
struct Nom_Structure Nom_Tableau[Nb_Elements];
```

Exemple avec la structure *Personne* définie précédemment :

```
struct personne etudiant[368];
```

# STRUCTURES DE DONNÉES

Le mot-clé **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 de votre type

```
typedef struct {  
    char* prenom;  
    char* nom;  
    char* sexe;  
    char* tel;  
    int age;  
} personne;
```

```
personne p1;
```

# STRUCTURES DE DONNÉES

## Les unions

Les unions se déclarent de la même manière que les structures. Elles possèdent donc elles aussi des champs typés.

Mais on ne peut utiliser **qu'un seul champ** à la fois. En fait tous les champs d'une union se partagent le même espace mémoire.

Les unions sont rarement nécessaires sauf lors de la programmation système.

*Exemple :*

```
union type {  
    int entier;  
    double flottant;  
    void *pointeur;  
    char lettre;  
};
```