

# Chapitre 8

**Tableaux**  
**Chaînes de caractères**  
**struct**  
**typedef**

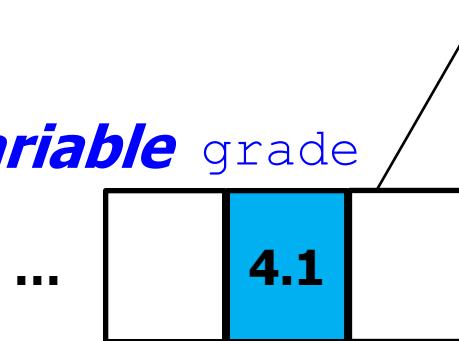
# Plan

- 1. Tableaux**
2. Chaînes de caractères
3. **struct**
4. **typedef**

## 8.1 Occupation mémoire d'un tableau

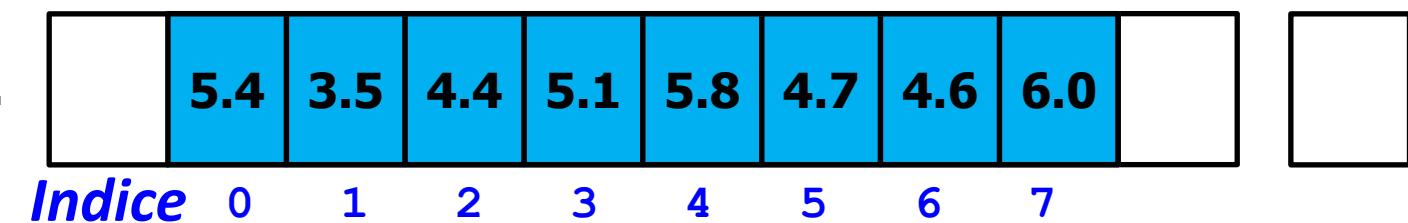
**Un** nombre réel

**nom de variable** grade



Un tableau de **huit** nombres réels

grades



Un seul **nom de variable** pour plusieurs valeurs

Chaque valeur est accessible par **un indice entier**

Ce tableau contient 8 valeurs qui sont indiquées de 0 à 7

## 8.1 Tableaux (*arrays*)

**Un tableau est un ensemble d'éléments contigus en mémoire**

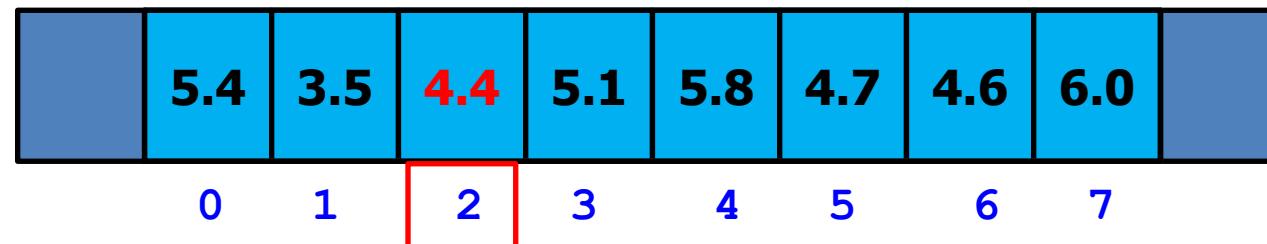
Tous les éléments d'un tableau sont **du même type**

Un tableau de taille N est indicé de **0 à N-1**

**L'accès** à un élément individuel se fait avec un **indice, un entier**

Par exemple **grades[2]** correspond à la valeur 4.4

**grades**



## 8.1 Déclaration de tableaux

```
<Type> <Nom du tableau> [<Taille>] , . . . ;
```

**On déclare une variable de type tableau en spécifiant**

Le **type** des éléments du tableau

Types primitifs, structure ou tableau...

Le **nom du tableau** de la variable représentant le tableau

La **taille** du tableau, c'est-à-dire le **nombre maximum** d'éléments que peut contenir le tableau

**Exemple**

```
double grades [8] ;
```

## 8.1 Déclaration des tableaux

### Diverses manières de définir la taille d'un tableau

```
#define NMAX 32
double price[NMAX];
```

La taille du tableau est définie  
**lors de la compilation**

```
const int NMAX=32;
double price[NMAX];

int nMax;
scanf ("%d", &nMax);
double price[nMax];
```

La taille du tableau est définie  
**à l'exécution**

**>C99**

## 8.1 Initialisation d'un tableau

### On peut

Initialiser les éléments d'un tableau directement à sa déclaration

Faire une initialisation partielle à la déclaration

### Avec les accolades, on peut aussi

Omettre la taille du tableau

Affecter une valeur à un élément particulier d'un tableau

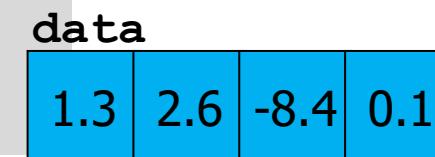
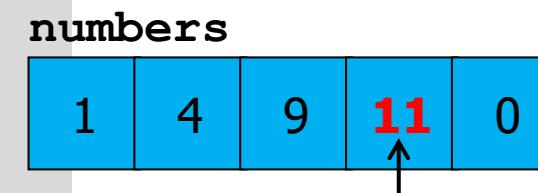
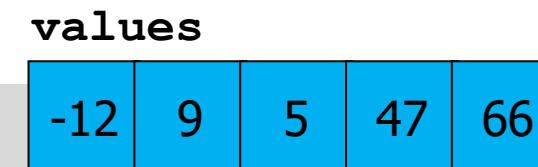
### Exemples

```
int values[5] = {-12, 9, 5, 47, 66};
```

```
int numbers[5] = {1, 4, 9};
```

```
numbers[3] = 11; ——————
```

```
double data[] = {1.3, 2.6, -8.4, 0.1};
```



## 8.1 Initialisation d'un tableau

### Pas d'initialisation

Tableau static ou global  
→ tout à 0

Sinon  
→ Valeurs indéfinies

### Initialisation partielle

Toutes les valeurs restantes  
sont initialisées à 0

```
int global_array[4];  
  
int main(void)  
{  
    int local_array[4];  
    int local_array1[4] = { 99 };  
    int local_array2[4] = { [1] = 99, [3] = 99 };  
    static int static_array[4];  
  
    return 0;  
}
```

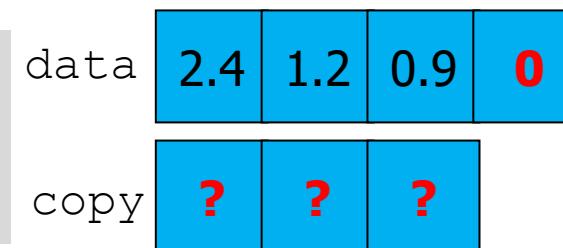
## 8.1 Utilisation d'un tableau

c>99 La taille du tableau peut être une expression quelconque. Si elle n'est pas constante, le tableau ne peut pas être initialisé à sa déclaration.

```
int size = inputSize(); // 1
double x = 1.23;
int tab[size]; ✓
double values[3] = {2.4, 1.2, x}; ✓
double val[size] = {1.1, 2.2, 3.3}; ✗
```

⚠ Il n'est **pas possible** d'affecter globalement (de recopier) tout le contenu d'un tableau à un autre avec un opérateur =

```
double data[4] ={2.4, 1.2, 0.9};
double copy[3];
copy = data; ✗
```



## 8.1 Tableaux, effets de bords, accès hors-limites

### ⚠ Attention

Le compilateur ne vérifie pas si les indices utilisés sont dans les limites autorisées !

```
int monthLength[12] = {31,28,31,30,31,30,31,31,31,30,31,30,31};
```

monthLength

?	?	31	28	31	30	31	30	31	31	30	31	30	31	?
0	1	2	3	4	5	6	7	8	9	10	11			

Accès hors tableau [0...11],  
le compilateur ne dit rien!!

```
int daysNb = monthLength[12];
```

## 8.1 Tableaux : tableau de valeurs constantes

Il est permis de déclarer des **tableaux de valeurs constantes** en qualifiant le type des éléments par **const**.

### Exemple

```
→ const double sinus[91] = { 0.000, 0.017, 0.035, 0.052, 0.070, 0.087, ... };  
  
sinus[4]; vaut 0.070 = sin(4°)  
sinus[0] = 1.0; ❌
```

interdit car les valeurs sont const

## 8.1 Tableaux : comment connaître leur taille ?

Grâce à l'opérateur **sizeof** on peut **connaître la taille** en bytes  
**d'une variable** simple ou tableau  
**d'un type**

### Exemple

```
int data[100] = {1, 2, 3, 4, 5, 6, 7};  
char vowels[] = {'a', 'e', 'i', 'o', 'u', 'y'};  
sizeof(data) // 400  
sizeof(int) // 4  
sizeof(vowels) // 6 (6 * 1 byte)
```

## 8.1 Tableaux : passage à une fonction

```
void display( int tab[] )  
{  
    int i;  
    for(i=0; i < ?? ; i++)  
    {  
        printf("%d\n",tab[i]);  
    }  
}  
  
int main(void)  
{ ...  
    int data[10]={7,2,5};  
    display(data);  
  
    return 0;  
}
```

### Problème

La fonction `display()` ne connaît pas le nombre de valeurs effectivement dans le tableau, ni sa taille.

# 8.1 Tableaux : passage à une fonction

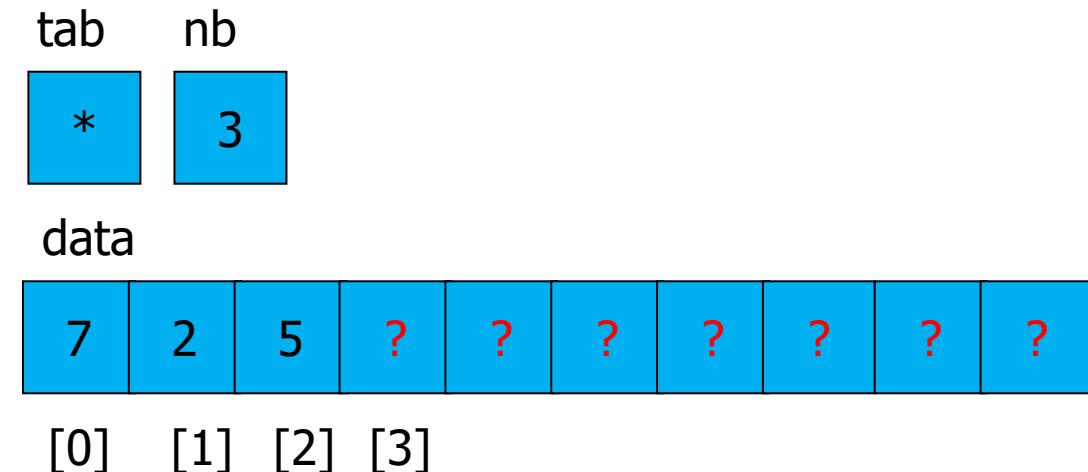
```
void display( int tab[], int nb )
{
    int i;
    for(i=0; i < nb ; i++)
    {
        printf("%d\n",tab[i]);
    }
}

int main(void)
{
    ...
    int data[10]={7,2,5};
    display(data,3);

    return 0;
}
```

## Solution

On passe en second paramètre le nombre d'éléments à traiter.



## 8.1 Tableaux multi-dimensionnels

Un tableau peut avoir plusieurs dimensions

```
<Type> <Nom de var> [N1] [N2] ... [Ni];
```

Déclaration

```
int values [20][40];  
double volume [10][10][10];
```

Initialisation

```
int matrix[2][4] = {{10,20,30,40}, {15,25,35,45}};
```

Accès aux éléments  
du tableau

```
first = values[0][0];  
volume[9][9][0] = 3.2;
```

# Plan

1. Tableaux
- 2. Chaînes de caractères**
3. `struct`
4. `typedef`

## 8.2 Chaînes de caractères (string)

### Définition

Tableau à une dimension, dont chaque élément est un caractère, **char**, et terminé par le caractère NULL, soit **0x00** ou **0** ou **'\0'**.

### Exemple

```
char greeting[6] = { 's', 'a', 'l', 'u', 't', '\0' };
```

**Remarque** Lors de sa déclaration, une chaîne de caractères peut faire l'objet d'une affectation par une chaîne de caractères constante **entre guillemets**.

```
char greeting[6] = "salut";
```

## 8.2 Chaînes de caractères

Exemple

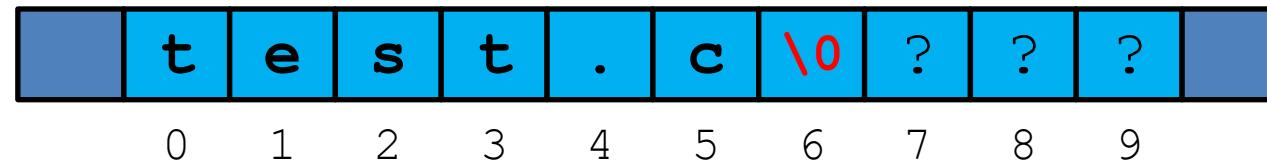
```
char filename[10] = "test.c";
```



### ⚠ Attention

Il faut toujours tenir compte du **caractère de fin** qui termine **obligatoirement** toute chaîne, **chaîne à zéro terminal**.

La taille du tableau de l'exemple doit donc être au moins de  $6 + '\backslash 0'$ , soit 7.



## 8.2 Déclaration et initialisation

### Comme les tableaux

```
char filename[] = "test.c";      ✓  
char address[] = "";            ✓  
char city[12] = "Saint-Imier";   ✓  
char name[10]; ✓
```

```
char firstname[]; ✗ // Compiler Error  
char rue[5] = "rue de la serre 17"; ✗/ Compilateur -> OK
```

## 8.2 E/S et chaînes de caractères

**printf** et **scanf**, supportent la saisie et l'affichage des chaînes de caractères avec la spécification de format **%s**.

**printf** affichera tous les caractères de la chaîne donnée en paramètre jusqu'au caractère de fin, exclu.

### Exemple

```
char fileName[16] = "test.c";
char word[] = { 'H', 'e', 'l', 'l', 'o' };

printf("%s", fileName); // test.c
printf("%s", word); // Hello....
```

## 8.2 E/S et chaînes de caractères

`scanf ("%s", ...)` récupère tous les caractères saisis jusqu'au **premier séparateur**. Ne **pas mettre** l'opérateur `&` (*adresse de*) devant le nom de la chaîne.

### Exemple

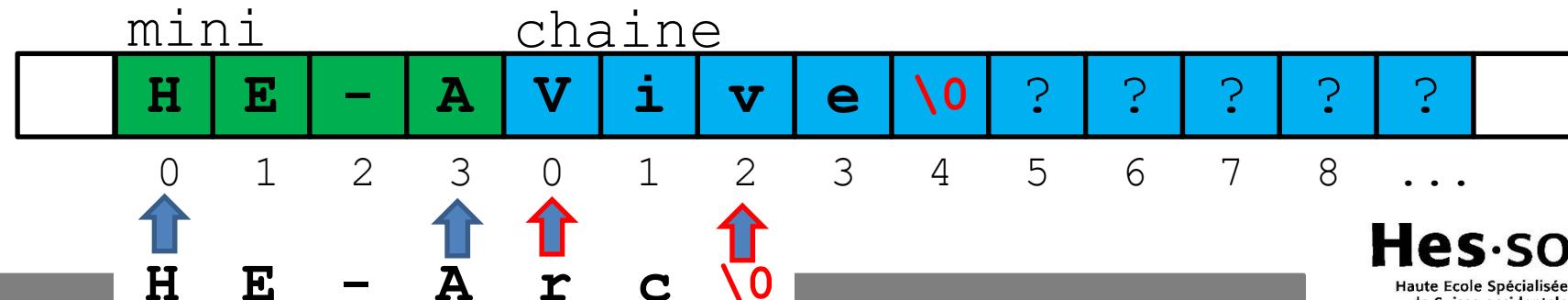
```
char chaine[256];
char mini[4];
scanf ("%s", chaine);
scanf ("%s", mini);
printf ("%s\n", chaine);
printf ("%s\n", mini);
```

Vive HE-Arc ↩  
rc  
HE-Arc

### Entrée utilisateur

### Sorties des deux printf()

⚠ gcc uniquement !



## 8.2 E/S et chaînes de caractères

### Remarques sur `scanf`

Pas de `&` devant la variable chaine lors de l'appel de `scanf`, car **un nom de tableau est déjà une adresse.**

`%s` **ajoute automatiquement un caractère nul ('\'0')** à la fin du tableau de char pour en faire une chaine de caractères valide.

Pas de contrôle du nombre de caractères récupéré → **risque de débordement de capacité**, autres variables écrasées !

## 8.2 E/S et chaînes de caractères

### Remarques sur `scanf`

L'espace est considéré comme **séparateur** donc seules les 4 premières lettres de "Vive He-Arc" sont récupérées

### Solution

Pour récupérer une ligne de texte avec des espaces, utiliser la fonction `fgets`

```
fgets(<varChaine>, <Nmax>, stdin)
```

### Exemple

```
fgets(chaine, 256, stdin)
```

# Manipulation de chaînes de caractères

Les chaînes de caractères ne peuvent pas être manipulées par les opérateurs conventionnels

- = n'est pas autorisé pour l'affectation.
- = est autorisé uniquement pour l'initialisation.
- ==, +, >, <, ...** n'ont pas le sens souhaité.

Les manipulations de chaînes de caractères se font au moyen de fonctions spéciales de la bibliothèque **string.h**

# Bibliothèque `string.h`

**Contient beaucoup de fonctions utiles aux chaînes de caractères :  
information, comparaison, copie, manipulation, recherche**

- Taille de la chaîne de caractères                    `strlen`
- Comparaison de chaînes de caractères            `strcmp`
- Copie à une chaîne de caractères                `strcpy`, `strncpy`
- Concaténation de chaînes de caractères        `strcat`
- Recherche dans une chaîne de caractères      `strstr`, `strchr`
- Extraction de sous-chaînes (*tokens*)            `strtok`

[Référence]: <http://www.cplusplus.com/reference/cstring/>

# Bibliothèque `string.h`

**int strlen(char texte[])** ← nombre de caractères de `texte`

```
int length = strlen("Hello"); // length ← 5
```

**int strcmp(char chaine1[], char chaine2[])**

// renvoie -1 si `chaine1 < chaine2` p.ex `strcmp ("ABC", "ABCD")`

// renvoie 0 si `chaine1 == chaine2` p.ex `strcmp ("A", "A")`

// renvoie +1 si `chaine1 > chaine2` p.ex `strcmp ("B", "A")`

**strcpy(char destination[], const char source[])**

// copie le contenu de `source` → `destination`

Autres fonctions utiles, celle de conversion d'une chaîne de caractère à un type numérique :  
`atof`, `atoi`, `atol`.

# Opérations sur les chaînes de caractères

```
char s1[10] = "toto";
char s2[10];
strcpy(s2, s1);
printf("%s %s", s1, s2);
```

```
char s1[10] = "toto";
char s2[10] = "toto";
strcat(s2, s1);
printf("%s \n %s", s1, s2);
```

```
const char *s1 = "toto";
char s2[] = "toto";
if (strcmp(s1,s2) == 0)
{
    printf("s1 et s2 sont identiques\n");
}
```

# Revue

Comment les deux variables suivantes sont représentées en mémoire ?

```
char letter = 'a';
char chaine[] = "a";
```

Que dit compilateur avec le code suivant ? Que va-t-il se passer ?

```
char temp [] = "";
char text [] = "Blablabla";

int i=0;
for ( ; i< strlen(text)+1 ; i++)
    temp[i] = text[i];
```

# Plan

1. Tableaux
2. Chaînes de caractères
- 3. struct**
4. **typedef**

# 8 Types

## Types de base

Entiers anonymes

Entiers nommés

Nombres flottants

## Types dérivés

### Tableaux

Fonctions

Pointeurs

### Structures

Unions

`int, short, ...`

`enum`

`float, double`

`[ ]`

`( )`

`*`

`struct`

`union`

`typedef`

## 8.3 Occupation mémoire d'une structure

Type de base

double grad = 4.1;

note



Types dérivés  
(Tableaux)

double grades[4]={5.4, 3.5, 4.4, 5.1};

grades

Même type

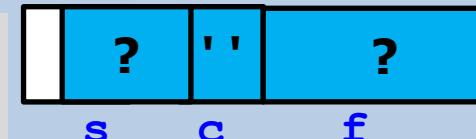


Type dérivés  
(Structures)

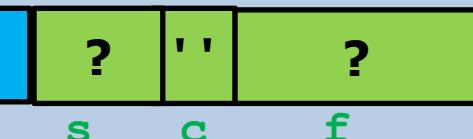
struct

```
{  
    short    s;  
    char     c;  
    float    f;  
} test, toto;
```

test



toto



Regroupe des éléments de types différents

## 8.3 Structures : déclaration de type

La **structure** permet de désigner **sous un seul nom** un **ensemble de valeurs** pouvant être de types différents\*

### Syntaxe de la déclaration d'un type structure

```
struct [identificateur]  
{  
    type1      membre1;  
    type2      membre2;  
    ...  
    typeN      membreN;  
} [var1, ...];
```

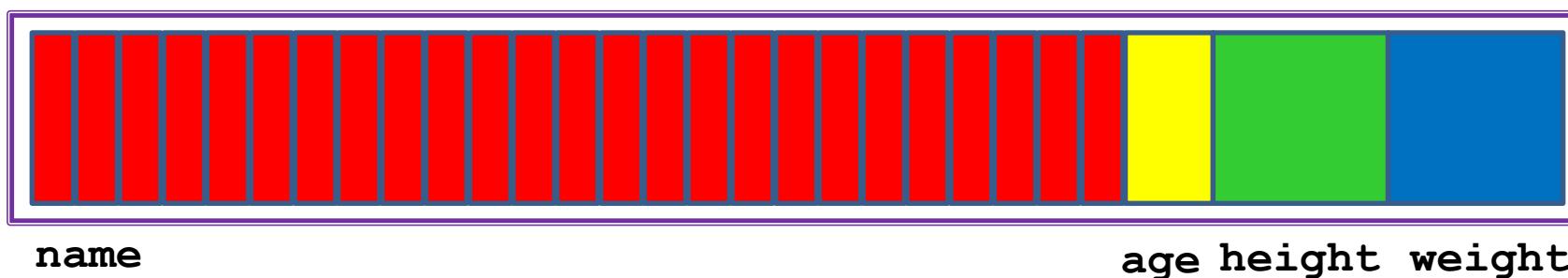
*L'un des deux est optionnel*

## 8.3 Structures : déclaration de type

Exemple

**déclaration d'un type** struct PatientFile

```
struct PatientFile // déclaration du type
{
    // déclaration des membres
    char name[32];
    short age;
    float height, weight;
};
```



## 8.3 Structures : déclaration de variables (1)

**Déclaration** d'une variable `client` de type :

`struct PatientFile`

`struct PatientFile client;`

**Déclaration et initialisation** de variables `c1`, `c2`, `c3`, de type `struct PatientFile` :

```
struct PatientFile c1 = {"Jules", 25, 176, 72},  
                      c2 = {"Eva", 18},  
                      c3 = {"Ed", .height=145, 59}; ≥C99
```



## 8.3 Structures : déclaration de variables (2)

À la déclaration d'un type structure, il est permis de **déclarer directement** une ou plusieurs variables de ce type. On peut même le faire sans nommer le type structure:

```
struct      ← type anonyme
{
    double x, y;
} point1, point2;
```

**point1** et **point2** sont des variables de type structure

## 8.3 Structures : accès aux membres (1)

**Variable de type structure, accès par l'opérateur point `.'**

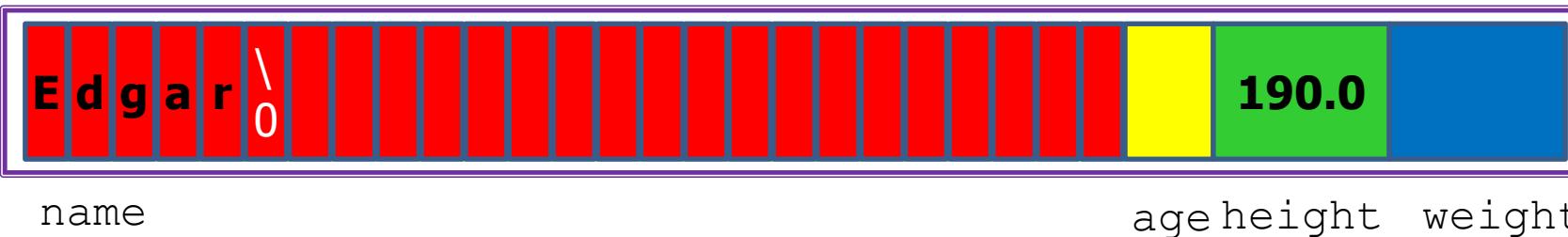
```
<variable>.<membre>
```

### Syntaxe

```
struct PatientFile myPatient;  
myPatient.height = 190.0;  
strcpy(myPatient.name, "Edgar");
```

### Exemple

myPatient



## 8.3 Structures imbriquées : déclaration

```
struct Date
{
    short day, month, year;
};
```

```
struct PatientFileB
{
    char name[32];
    struct Date birthDate;
    float height;
};
```

```
struct Date
```



```
struct FicheB
```



misterDurand

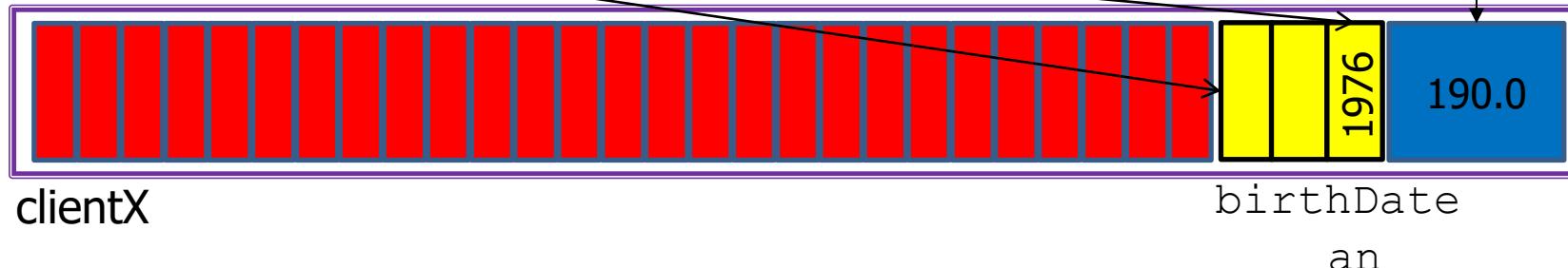
```
struct PatientFileB misterDurand = {"Durand", {13,2,1976}, 190.0};
```

## 8.3 Structures imbriquées : accès membres

```
struct FicheB
{
    char name[25];
    struct Date birthDate;
    float height;
};
```

```
struct Date
{
    short day,
          month,
          year;
};
```

```
struct FicheB clientX;
clientX.height = 190.0;
clientX.birthDate.year = 1976;
```



## 8.3 Structures : affectation

Possible si même type, même nom de type *struct* en C++

```
struct PatientFileB patientX = {"Jean", ... };  
struct PatientFileB patientY;  
  
patientY = patientX; // Tout le contenu est recopié  
patientY.height = patientX.height;
```

## 8.3 Structures et fonctions

```
struct MyStruct  
{  
};
```

Une fonction peut retourner un résultat de type structure ou un pointeur sur une structure

```
struct MyStruct getData(...);
```



Une structure peut être passée comme paramètre à une fonction

```
showData(struct MyStruct s);
```



# Plan

1. Tableaux
2. Chaînes de caractères
3. **struct**
- 4. `typedef`**

## 8.4 **typedef** (types synonymes)

Le langage C permet de renommer des types en leur donnant un synonyme

L'intérêt est de **simplifier** l'écriture et la lecture du code

La déclaration d'un nouveau type se fait avec le mot-clé **typedef**

Offre un outil **favorisant la portabilité du code**

```
typedef unsigned char Uint8;  
typedef double Sfp64;
```

## 8.4 **typedef** (types synonymes)

### Syntaxe

```
typedef <Type standard> <Nom de type>;
```

### Exemple

```
typedef int Entier;  
Entier v1; // équivalent à int v1
```

## 8.4 **typedef** (types synonymes)

### Syntaxe

```
typedef <Type std> <Nom de type>[<N>] ;
```

### Exemple

```
typedef char Chaine[80];  
Chaine v2; // équivalent à char v2[80]
```

## 8.4 **typedef** (types synonymes)

### Syntaxe

```
typedef <Déf struct> <Nom de type>;
```

### Exemple

```
typedef struct
{
    char    name [32];
    short   age;
    float   height, weight;
} PatientTy;

PatientTy v4;
```

## 8.4 Remarque

Quelle est la différence entre **struct** et **typedef struct** ?

```
struct t1 { ... };           //1.  
typedef struct { ... } T2;   //2.
```

1. **t1** est un identificateur de structure

Le type associé est **struct t1**

2. Déclare un nouveau type nommé **T2**

La seconde écriture permet **plus d'abstraction** de type

```
struct t1 v1; // struct traine  
T2 v2;        // Abstraction
```

# Exercices



Exercices du chapitre 08