

COURS 7

TYPE - IQUE
DE C

PARTIE ①

LES STRUCTURES

MOTIVATION

Parfois on aimerait condenser en une variable plusieurs valeurs
genre

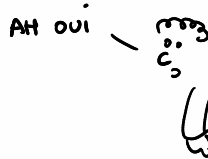
9	7.5	'a'
---	-----	-----

une seule variable

COMME UN
TABLEAU EN
FAIT?



Oui, mais un tableau peut contenir que des variables de même type



Pas de panique, c'est pour ça qu'il y a les structures.

JE PANIQUAIS PAS.



DEFINITION DE LA STRUCTURE

structure = regroupement composite de plusieurs valeurs en une seule variable
c'est comme si on définissait un nouveau type!

SYNTAXE	<pre>struct <nom structure> { type1 champ 1; type2 champ 2; : };</pre> <p>↳ ne pas oublier le point virgule, ce n'est pas une fct!</p>
---------	--

à mettre
généralement
au début du
programme, en
dehors du main.

Example

```
struct vache {
    char prenom[15];
    int nb_taches;
    float poids;
};
```

Les parties qui composent la structure (comme prénom ou nb-taches) s'appellent des membres ou des champs

Exemple de variable de type struct uache

'C'	'l'	'a'	'r'	'a'	'b'	'e'	'l'	'l'	'e'	'l'	'l'	'l'	'l'	'l'	18	208.5
															nb-taches	poïds

DÉCLARATION + INITIALISATION

On peut considérer "struct nom_struct" comme un nouveau type.
Les variables peuvent être déclarées avec :

SYNTAXE

struct nom_struct nom_var ;

EXEMPLE

Code

```
struct vache marguerite;
```

En mémoire *marguerite*

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

On peut comme les tableaux déclarer et initialiser en une seule ligne :

SYNTAXE

```
struct nom_struct nom_var = { .champ1 = val1 , .champ2 = val2.. };
```

EXEMPLE

Code

```
struct vache vache_qui_rit = { .prenom="Kiri", .nb_taches=0, .poids=0.1};
```

En mémoire *vache_qui_rit*

'K'	'i'	'r'	'i'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	0	0.1
-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	---	-----

On peut omettre certains champs. Auquel cas, ils seront initialisés à 0 (par ex, on aurait pu ici enlever ".nb_taches = 0")

LIRE ET ÉCRIRE

On peut accéder à un champ d'une structure en écrivant
`var_structure . champ`

et le modifier en écrivant :

`var_structure . champ = ...`

E
X
E
M
P
L
E

```
struct vache steak_sur_pattes = {
    .prenom = "Noiraude",
    .nb_taches = 18,
    .poids = 413
};
printf("Prénom %s, ", steak_sur_pattes.prenom);
printf("%d taches, ", steak_sur_pattes.nb_taches);
printf("%f kg.\n", steak_sur_pattes.poids);
steak_sur_pattes.nb_taches++;
steak_sur_pattes.poids = 400.5;
printf("Plutôt : %d taches, ", steak_sur_pattes.nb_taches);
printf("%f kg.\n", steak_sur_pattes.poids);
```

```
Prénom Noiraude, 18 taches, 413.000000 kg.
Plutôt : 19 taches, 400.500000 kg.
```

affiche

CONTRAIREMENT AUX TABLEAUX...

On peut affecter une variable structure par une autre variable structure
(avec le signe =)

```
struct vache v1 = { .prenom = "Meuh-meuh"};  
struct vache v2 = { .prenom = "Moo-moo"};  
v2 = v1;
```

ce qui peut paraître étrange car...



```
char prenom1[15] = "Meuh-meuh";  
char prenom2[15] = "Moo-moo";  
prenom2=prenom1;
```

ne compile pas.

On peut même les utiliser comme entrée/sortie d'une fonction :

```
struct vache deux_fois_plus_grosse(struct vache v){  
    v.poids = 2*v.poids;  
    return v;  
}
```

STRUCTURES ET FONCTIONS

CODE

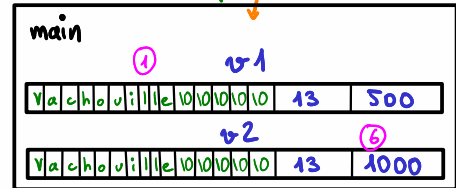
```
#include <stdio.h>
#include <stdlib.h>
struct vache {
    char prenom[15];
    int nb_taches;
    float poids;
};

struct vache deux_fois_plus_grosse(struct vache v){
    v.poids = 2*v.poids;
    return v;
}

int main() {
    struct vache v1 = {
        .prenom = "Vachouille", .nb_taches = 13, .poids = 500
    };
    struct vache v2 = deux_fois_plus_grosse(v1);
    printf("v1 pèse %fkg et v2 pèse %fkg\n", v1.poids, v2.poids);
    return EXIT_SUCCESS;
}
```

EN MÉMOIRE

deux_fois_plus_grosse



v1 pèse 500.000000kg et v2 pèse 1000.000000kg

Différentes étapes :

- ① Dans la fonction `main`, on déclare + initialise une variable `struct vache v1`, et `v2` est déclarée
- ② On appelle la fonction `deux_fois_plus_grosse`. Le paramètre `v1` est transmis dans son entièreté.
- ③ On initialise dans `deux_fois_plus_grosse` une

variable `struct vache v`.

- ④ Le champ "poids" de `v` est doublé
- ⑤ On renvoie `v` dans son entièreté
- ⑥ On recopie cette valeur dans `v2`
- ⑦ Le message est affiché

MORALITÉ Les structures sont passées par valeur et non pas par adresse

POINTEURS VERS STRUCTURE

Certaines structures occupent beaucoup de place en mémoire (avec les risques que ça engendre : baisse de performance, dépassement de pile..)

C'est pourquoi il est très fréquent de :

ne pas travailler directement
sur les structures ✗

mais d'utiliser
des pointeurs vers
ces structures ✓

L'allocation des variables se fera de manière dynamique (malloc, free)

Ex:

```
struct vache* ptr_vache = malloc(sizeof(struct vache));  
(*ptr_vache).nb_taches = 17;
```

Les fonctions ne transmettront plus les objets mais leurs adresses
Elles modifieront ainsi les structures directement:

Ex:

```
void double_poids(struct vache* ptr_vache){  
    (*ptr_vache).poids = 2 * (*ptr_vache).poids;  
}
```

OPERATEUR FLÈCHE

Quand on travaille sur des pointeurs vers des structures, on est amenés à écrire très souvent $(*ptr).champ$

(ex $(*ptr_vache).nb_taches$)

C'est lourd.

Une notation plus courte la remplace: l'opérateur flèche

SYNTAXE	$ptr \rightarrow champ$
---------	-------------------------

accès au champ de
la structure pointée par ptr

Exemple en mémoire:



Exemple en code:

```
void double_poids(struct vache* ptr_vache){  
    ptr_vache->poids = 2 * ptr_vache->poids;  
}
```

ALIAS

En parlant de lourdeur, il est pénible de devoir écrire
"struct nom-struct" voire "struct nom-struct*" à chaque fois
Heureusement, on peut donner des noms alternatifs aux types :
ce sont les alias

SYNTAXE

```
typedef <type_depart> <nouveau_nom>;
```

L'alias reste un type -
On peut toujours utiliser les anciens types.

Exemple de
code :

```
typedef struct vache vache;
typedef struct vache* pvache;

int main(){
    vache v = {.nom="Marguerite"};
    pvache pv = malloc(sizeof(vache));
    ...
}
```

il est courant de faire un
alias qui enlève le "struct"

PARTIE ~~II~~

LES ÉNUMÉRATIONS

ÇA FAIT QUOI ?

Considérons le bout de code :

```
enum debile { toto, tata, tutu };
```

Ça fait quoi ??

- Ça définit toto comme une variable constante valant 0
 - Ça définit tata comme une variable constante valant 1
 - Ça définit tutu comme une variable constante valant 2
 - Elles sont toutes les trois de type "enum debile" (mais en fait c'est pareil que int)
- Et c'est tout.

SYNTAXE

```
enum <nom enum> { <nom1>, <nom2>, ..., ... };
```

constante
qui vaut 0

constante
qui vaut 1 ...

Mais alors c'est
quoi la différence
avec

```
int toto = 0;  
int tata = 1;  
int tutu = 2;
```

?

Pratiquement aucune.
(sauf qu'on ne peut pas modifier
les variables dans une énumération)

ÇA SERT À QUOI ?

PRINCIPE DU
BON PROGRAMMEUR

En C, chaque valeur numérique
qui a du sens doit avoir un nom.

Exemple 1 : Définir des statuts

- Le lancer d'une pièce peut avoir deux statuts différents :
pile ou face

Ex

```
enum lancer { pile, face };  
enum lancer pile_ou_face() {  
    return rand()%2;  
}
```

- Dans un jeu sur une grille, les cases peuvent avoir différents statuts

Ex

```
enum statut { vide, perso, sortie, piege };
```


- Mais aussi :
- statuts d'erreur
 - directions cardinales (nord, ouest, sud, est)

ÇA SERT À QUOI ?

Exemple 2 : Indices des tableaux

Par exemple, si on veut simuler le carnet de notes d'un étudiant :

```
enum matieres { anglais, introprog, technoweb, algebre, nb_matieres };  
int notes_martin[nb_matieres];  
notes_martin[anglais] = 8;  
notes_martin[introprog] = 2;  
notes_martin[technoweb] = 19;  
notes_martin[algebre] = 8;
```



ce qui est sûrement plus explicite (mais plus long) que :

```
int notes_martin[4] = {8, 2, 19, 8};
```

ASTUCE

Mettre une variable "nombre" à la fin d'une énumération permet d'avoir automatiquement le nombre d'éléments dans l'énumération
Pratique si on veut ajouter/enlever un élément !

PARTIE ~~III~~

LES BOOLÉENS (?)

LES BOOLÉENS N'EXISTENT PAS

Booléen = vrai ou faux = valeur logique renvoyée par une condition
(ce qui va dans un `if(~)` ou un `while(~)`)

En python, il y a True et False.

En C? les booléens c'est pareil que les entiers

RÈGLE	La valeur 0 représente faux
	Tout ce qui n'est pas 0 (en particulier 1) est vrai

Ainsi

→ `if (x != 0) { ~ }` c'est pareil que `if (x) { ~ }`

→ `while (1) { }` est une boucle infinie



MAIS VU QUE "0" ET "1" ONT DU SENS,
ON DEVRAIT PAS LEUR DONNER DES NOMS?

Si, tout à fait,
élève imaginaire.

ÉMULER LES BOOLÉENS

Plusieurs solutions pour importer les booléens en C
(aucune n'est vraiment meilleure — à mon avis)
les macros constantes

MÉTHODE 1

```
#define FALSE 0  
#define TRUE 1  
typedef int boolean;
```

MÉTHODE 2

Une énumération

```
enum bool { False, True };  
typedef enum bool boolean;
```

Inclure la bibliothèque `stdbool.h`

MÉTHODE 3

```
#include <stdbool.h>  
bool vaut3(int x){  
    return x==3;  
}  
  
int main() {  
    printf("true = %d\n",true);  
    printf("false = %d",false);  
}
```

Les booléens
sont ici désignés
true et false, et
sont de type **bool**

Dans les 3 cas,
0 représente **faux**
et 1 représente **vrai**