Types composés

ICC-C Cours 8

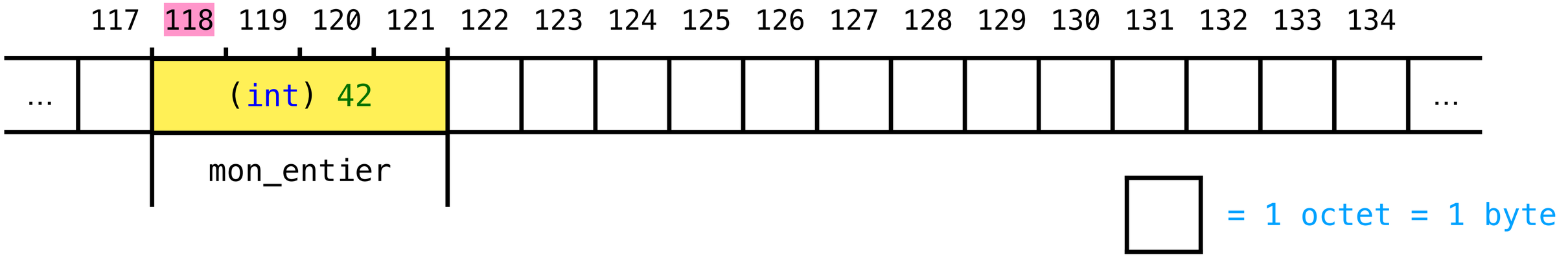
Les types de base

- “Scalaires”
- `int, short, long, char, float, double, …`
- Une seule valeur qui occupe un certain nombre d’octets
- On peut définir des constantes et des variables de ces types

Les pointeurs

```c
int mon_entier = 42;
int *ptr = &mon_entier; // une variable qui contient
                        // une adresse vers un int
```

- `&mon_entier` a un type spécial pour indiquer que c'est une adresse
- C'est un pointeur vers un `int` (= `int` *pointer*)



Les pointeurs

```
int mon_entier = 42;
int *ptr = &mon_entier; // une variable qui contient
                        // une adresse vers un int
```

- Pour affecter la valeur `100` à `mon_entier` il faut écrire:

Option 1	Option 2	Option 3	Option 4
`ptr[0] = 100`	`ptr = 100`	`&ptr = 100`	`*ptr = 100`

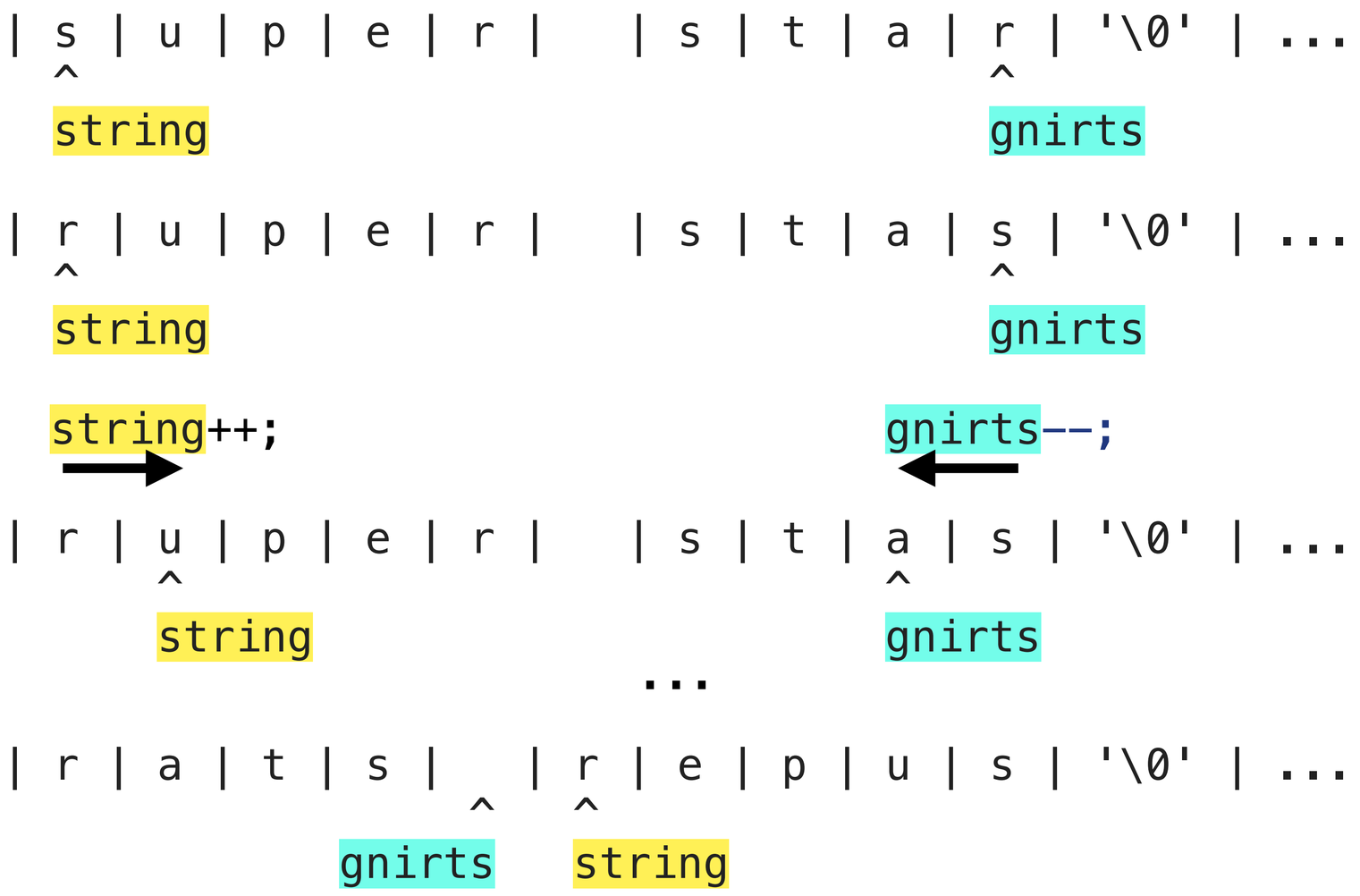
gnirts.c

Qu'affiche ce code?

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

void process(char* string);

int main()
{
    char texte[] = "super star";
    process(texte);
    printf("%s\n", texte);
}
```

```c
void process(char* string)
{
    char* gnirts =
        string + strlen(string) - 1;

    while (gnirts - string > 0)
    {
        char c = string[0];
        string[0] = gnirts[0];
        gnirts[0] = c;

        string++;
        gnirts--;
    }
}
```

gnirts.c




```
char* gnirts =
string + strlen(string) - 1;

while (gnirts - string > 0)
{
    char c = string[0];
    string[0] = gnirts[0];
    gnirts[0] = c;

    string++;
    gnirts--;
}
```

(gnirts - string) vaut -1

Tableaux

- Stocker plusieurs valeurs du même type `T`
- Taille fixe

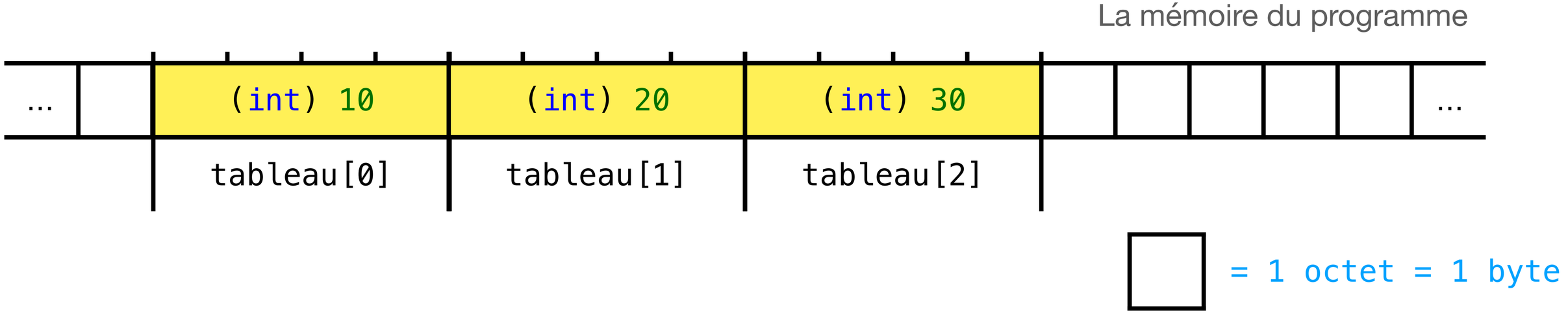
```
T nom[taille_max] (= valeur)
```

- `T` = **types de base**: `int a[3], char b[6], float c[112], ...`
- `T` = autres **tableaux**: `int d[3][5], double e[3][3][3], ...`
- `T` = **Pointeurs:** `int *f[6] // Un tableau de 6 pointeurs int`

Tableaux

```
int tableau[3] = {10, 20, 30};
```

`sizeof(tableau)` est 12



Représenter une matrice

… de taille fixe 2x4

```
short ligne1[4] = {8, 3, -1, 1},
      ligne2[4] = {5, 0, 6, 30};
```

- On peut faire mieux: tableau bi-dimensionel!
- C’est un tableau de tableaux unidimensionnels

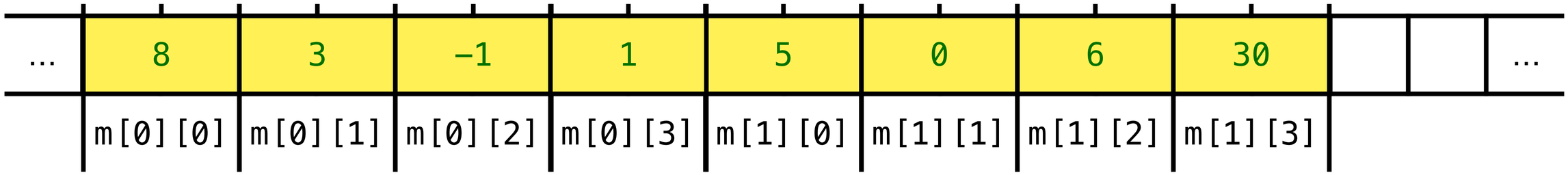
```
short m[2][4] = {
    {8, 3, -1, 1},
    {5, 0, 6, 30},
};
```

Tableaux de tableaux

- C’est aussi une zone contiguë de mémoire!

```
short m[2][4] = {
    {8, 3, -1, 1},
    {5, 0, 6, 30},
};
```

`sizeof(m)` est 16 = 2 x 4 x `sizeof(short)`



Représenter une matrice de `int`

…de taille variable

```
??? creer_matrice(int lignes, int cols);
```

- Quel est le bon type de retour?

Option 1	Option 2	Option 3	Option 4
`int[]`	`int[][]`	`int*`	`int**`

Pointeurs doubles

- Un pointeur `int*` pointe stocke l'adresse de début d'un tableau de `int`

```
int *vec = malloc(cols * sizeof(int))
```

- On l'utilise comme un tableau

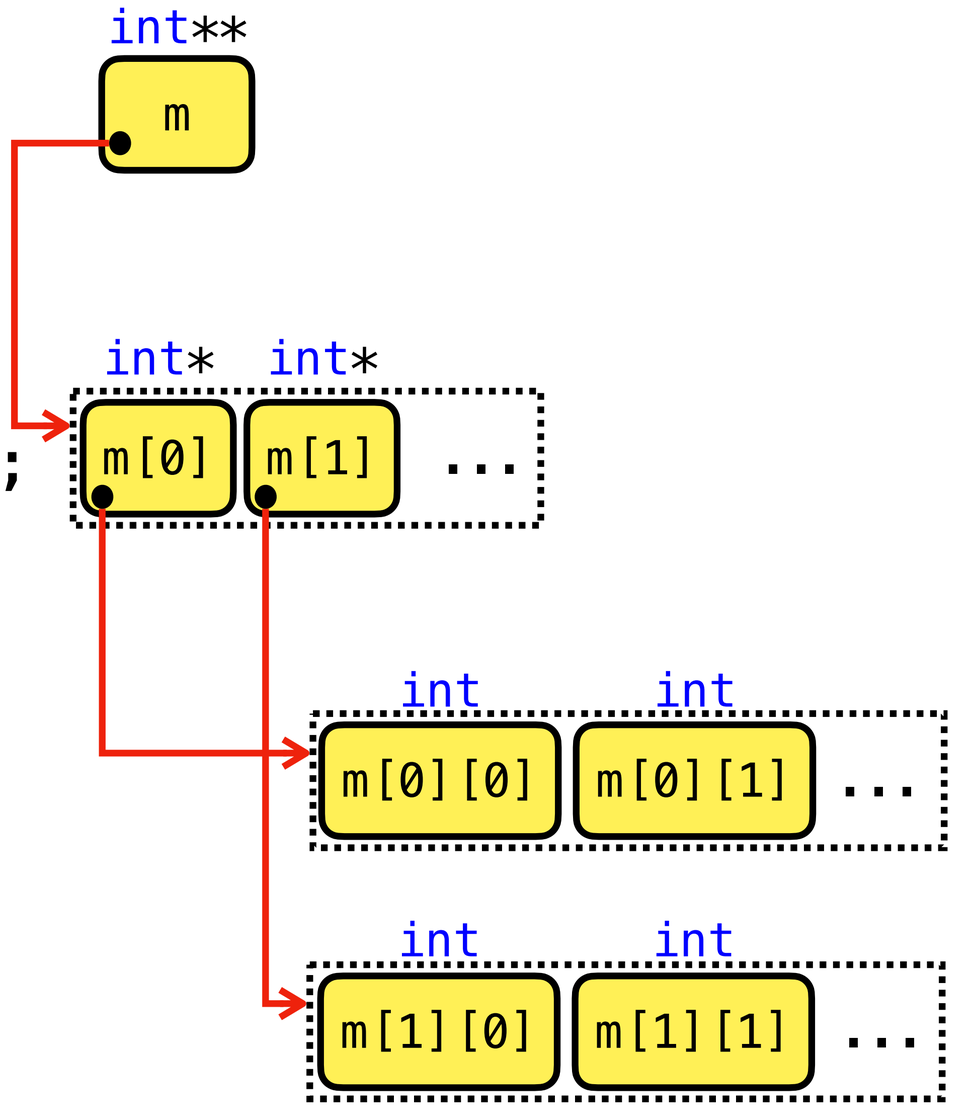
```
vec[3] = 25;  // même chose que *(vec + 3) = 25;
```

- Un pointeur `int**` pointe stocke l'adresse de début d'un tableau de `int*`

```
int **mat = malloc(lignes * sizeof(int*))
```

Matrice de taille variable

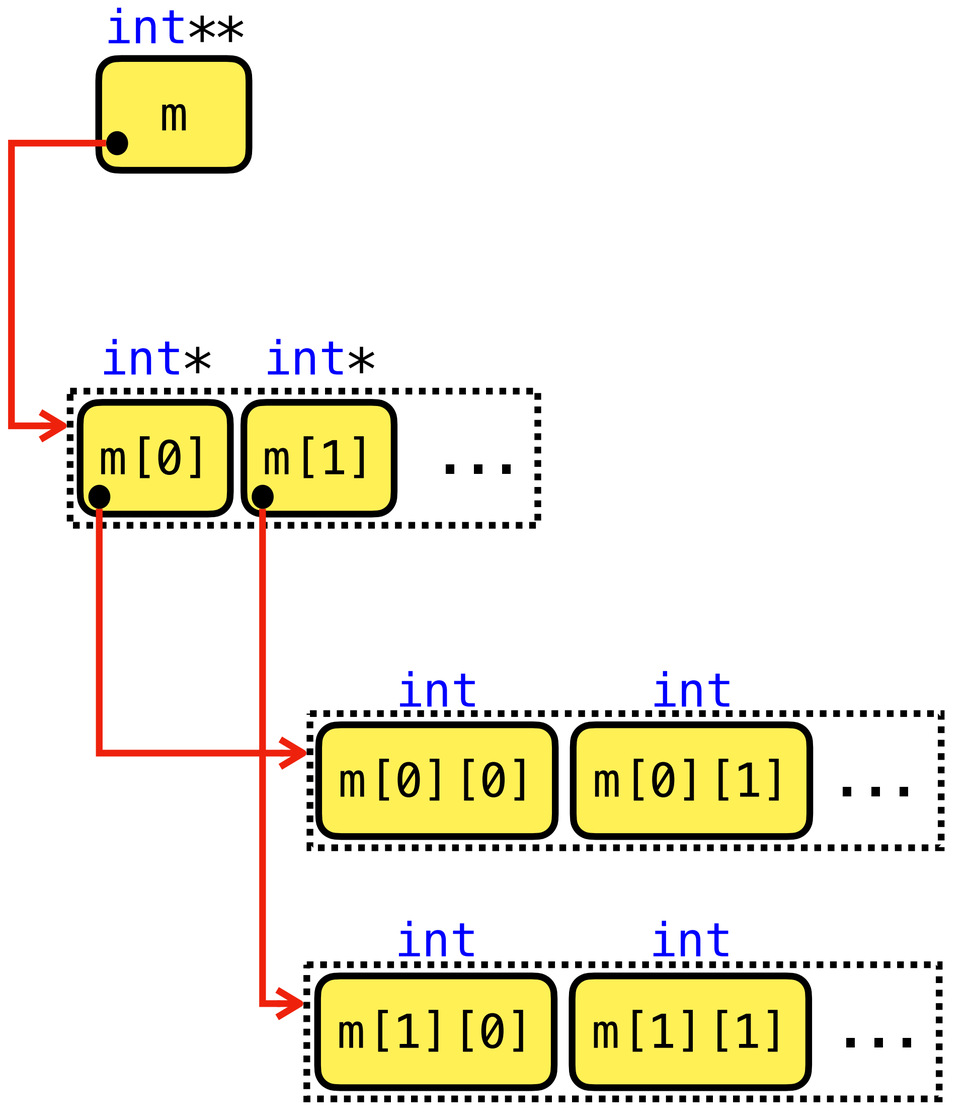
```
int **creer_matrice(int lignes, int cols)
{
    int **m = malloc(lignes * sizeof(int*));
    // m est maintenant un tableau
    // de `lignes` pointeurs vers int
    for (int i=0; i<lignes; i++)
    {
        m[i] = malloc(cols * sizeof(int));
        // m[i] pointe vers un tableau
        // de `cols` entiers
    }
    return m;
}
```



Matrice de taille variable

Free!

```
void libérer_matrice(
    int **m,
    int lignes
)
{
    for (int i=0; i<lignes; i++)
    {
        free(m[i]);
    }
    free(m);
}
```



On aimerait des types plus riches

- Un **point** en 2d a un “x” et un “y”

```
double point[2];
```

- Plusieurs (50) **points** en 2d

```
double point[50][2];
```

- Une **personne** = `nom`, `prénom`, `année` (de naissance), `taille_cm`, `poids_kg`, etc.

```
char nom[30], prénom[30];
int année;
int taille_cm;
float poids_kg;
```

- Plusieurs **personnes**

```
???
```

Données tabulaires

- Bienvenue dans Excel…
- Par exemple, des données personnelles, ou les caractéristiques d'un produit

Nom	Prénom	Année	Taille (cm)	Poids (kg)
Meunier	Alice	2002	168	55.3
Smith	Robert	1977	179	81.2

Comment pouvons-nous faire?

- Comme avant, on peut définir des tableaux par colonne

```
char nom[100][30], prénom[100][30];
int année[100];
int taille_cm[100];
float poids_kg[100];
```

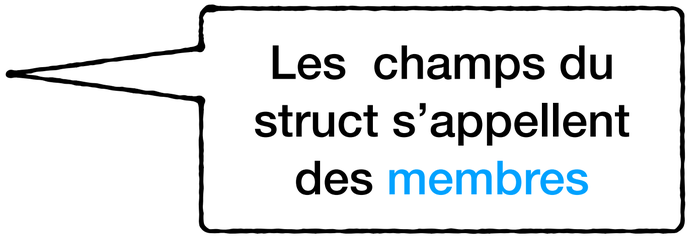
- Pour accéder aux infos d'une personne, on utilise partout le même indice
- Stockage par colonne = *Column store*
- Il y a effectivement des systèmes qui stockent les données de cette manière!

Ou alors — créer un nouveau type!

struct

- Les tableaux stockent plusieurs valeurs du même type
- Une structure stocke des valeurs de plusieurs types
- On définit le type `struct` ainsi:

```
struct nom_du_type
{
    type_1 membre_1;
    type_2 membre_2;
    ...
    type_n membre_n;
}
```



Une structure nommée

Named struct

- Une “structure” = un type composé:

```
struct personal_info
{
    char nom[30];
    char prénom[30];
    int année;
    int taille_cm;
    float poids_kg;
};
```

Ce n’est pas une variable!

Définition du type
`struct personal_info`

- La déclaration d’une variable de ce type:

```
struct personal_info bob;
```

Variable de type
`struct personal_info`

Initialisation & affectation

- Comme dans le cas des tableaux, on peut initialiser une variable `struct` lors de sa définition:

```
struct personal_info bob = {"Smith", "Robert", 1977, 179, 81.2};
```

- On peut aussi affecter le contenu d'une variable `struct` à une autre variable:

```
struct personal_info bob_copie;

bob_copie = bob;
```

- Rappel: on ne pouvait pas faire ça avec des tableaux!

Test d’égalité

- On ne peut pas directement comparer deux struct 😢
- On doit les comparer membre par membre…
- Petite limitation du C

```
struct personal_info bob_copie = bob;

if (bob_copie == bob) { ... }
```

```
error: invalid operands to binary expression ('struct personal_info' and
'struct personal_info')
   50 |     if (bob == bob_copie)
      |         ~~~ ^  ~~~~~~~~~
```

Accéder aux membres

```
struct personal_info bob = {"Smith", "Robert", 1977, 179, 81.2};
```

- Comment accède-t-on au prénom de Bob?
- On utilise l'**opérateur** d'accès aux membres "**.**"

```
bob.prénom
```

```
printf("%s %s pèse %.2f kg\n",
       bob.prénom,
       bob.nom,
       bob.poids_kg);
// Affiche: Robert Smith pèse 81.2 kg
```

Accéder aux membres

Pointeurs

- Parfois on utilise des **pointeurs** vers une variable `struct`

```
struct personal_info bob = {"Smith", "Robert", 1977, 179, 81.2};

struct personal_info *p_info = &bob;
```

- Normalement on devrait utiliser l'opérateur d'indirection *:

```
(*p_info).taille_cm;
```

- Mais on peut utiliser l'**opérateur "->"** (c'est équivalent):

```
p_info->taille_cm;
```

Comment est-ce stocké en mémoire?

- L’ordre des champs compte

bob.nom	bob.prénom	bob.année	bob.taille_cm	bob.poids_kg
“Smith”	“Robert”	1977	179	81.2
char[30]	char[30]	int	int	float

```
sizeof(struct personal_info) = 72 = 2 x 30 + 4 + 4 + 4
```

Passage par valeur

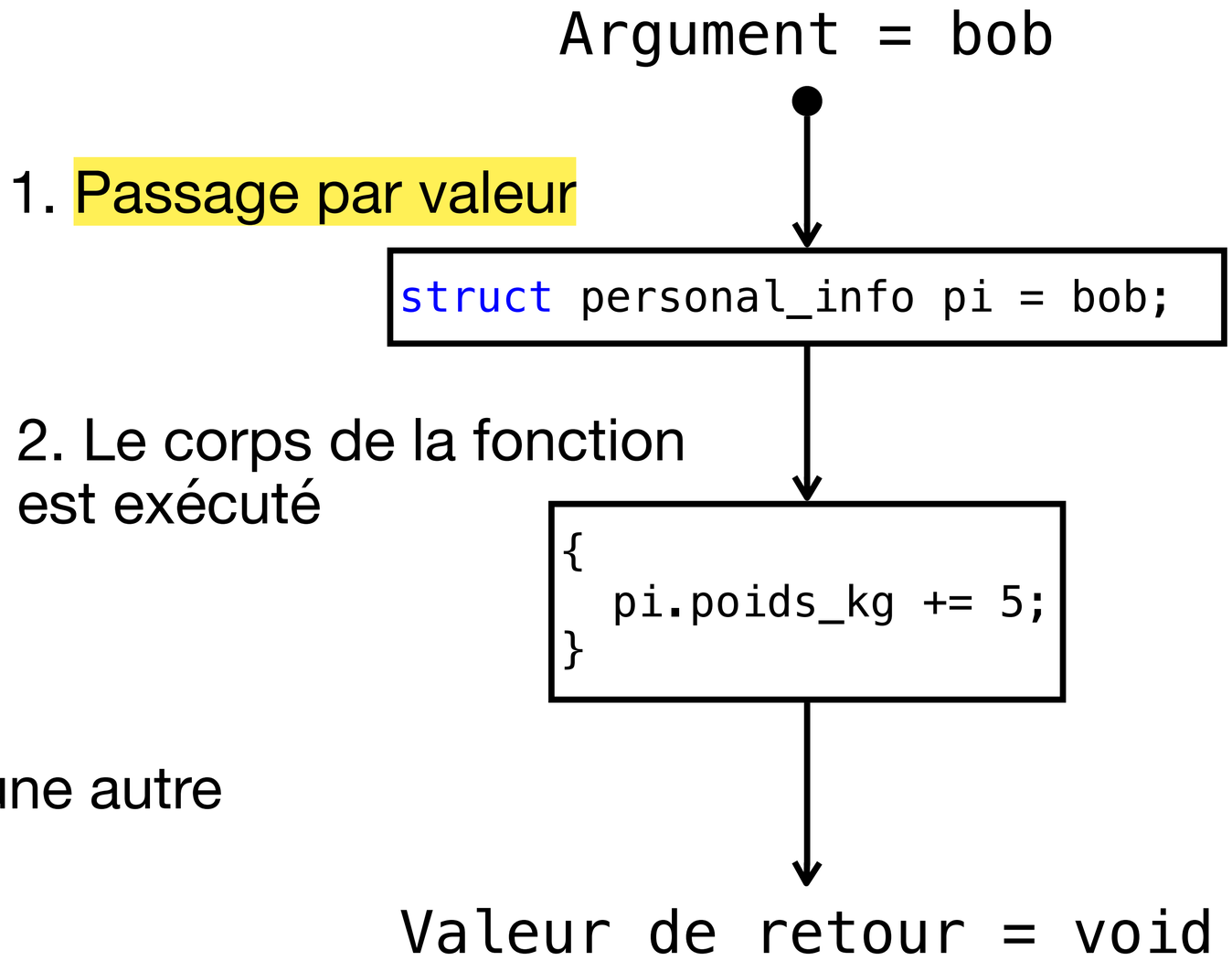
- Que se passe-t-il quand on passe une structure en argument d'une fonction?
- La fonction `sneaky` essaye de modifier le poids de Bob!
- Qu'adviendra-t-il ?

😬

```
void sneaky(struct personal_info pi)
{
    pi.poids_kg += 5;
}

...

struct personal_info bob = {
    "Smith", “Robert",
    1977, 179, 81.2};

sneaky(bob);

printf("%s %s pèse %.2f kg\n”,
       bob.prénom,
       bob.nom,
       bob.poids_kg);
```

Appel de fonction

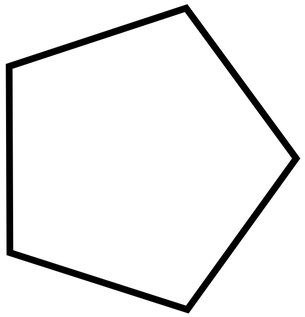
```
sneaky(bob)
```



Puisqu'on peut affecter une struct à une autre variable, `sneaky` a modifié la copie!

Tableaux de struct

```c
struct point
{
    double x, y;
};

struct point triangle[] = {
  {0, 0}, {0, 1}, {1, 0}};

struct point *pentagone = polygone(5);
```

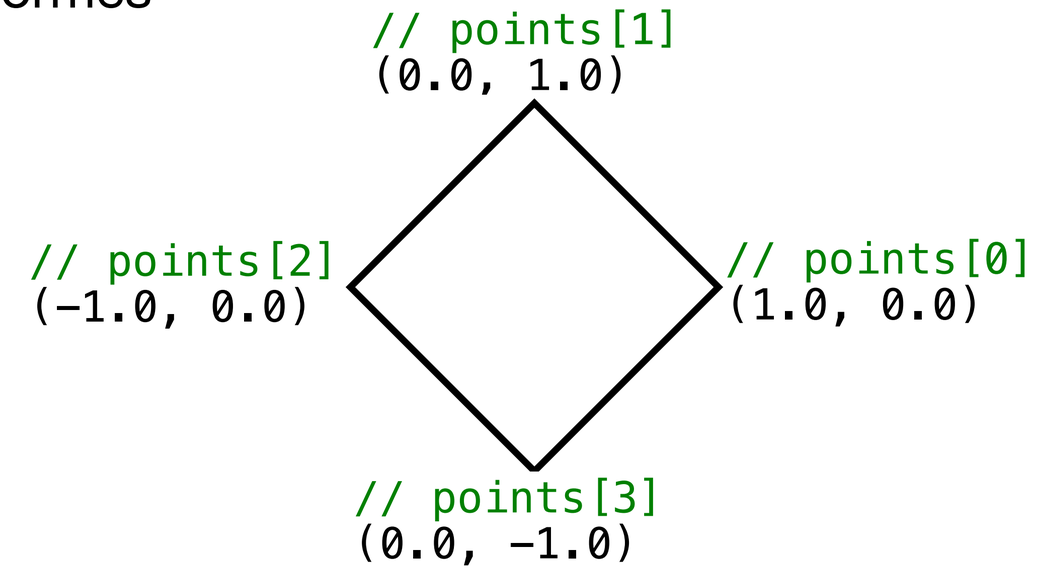


```c
struct point *polygone(int n)
{
    struct point *sommets =
        malloc(n * sizeof(struct point));

    for (int k=0; k<n; k++)
    {
        sommets[k].x = cos(2*k*M_PI/n);
        sommets[k].y = sin(2*k*M_PI/n);
    }

    return sommets;
}
```

Dessiner

- Nous avons codé un programme pour dessiner des formes
- On stocke les formes dans une struct:

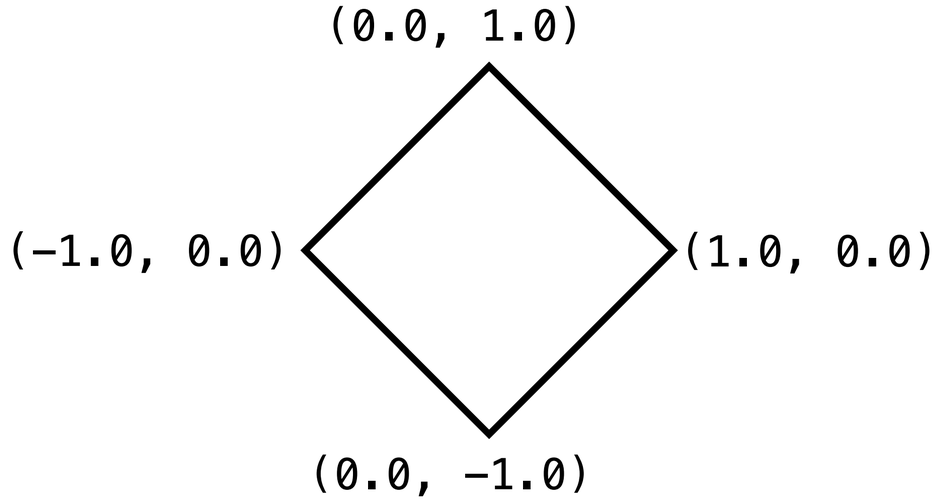
```
struct forme
{
  struct point *points;
  int n_points;
};
```



- Exemple: On appuie sur le bouton “Nouveau carré”

```
struct forme carré = {polygone(4), 4};
// carré.points est un pointeur vers le tableau de points
// carré.n_points est le nombre de points
```

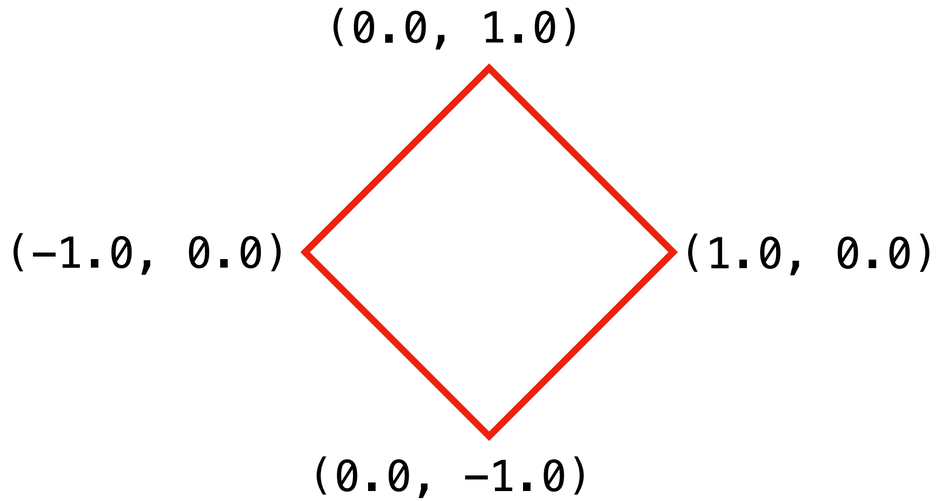
Dessiner



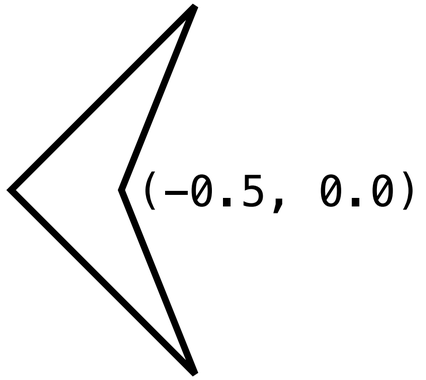

```
struct forme carré = {polygone(4), 4};
```

- Notre programme permet de copier des formes
- On copie le carré noir `Ctrl-C`
- `Ctrl-V`: on obtient le carré rouge

```
struct forme carré_rouge = carré;
```



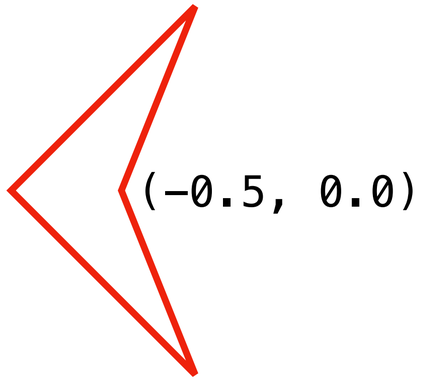
Dessiner




```
struct forme carré = {polygone(4), 4};
struct forme carré_rouge = carré;
```

- On modifie le 1er point de la copie pour en faire une flèche

```
carré_rouge.points[0].y = -0.5;
```



- Surprise, le carré noir est aussi devenu une flèche, sans qu’on y touche!
- Pourquoi?

Copie superficielle

Shallow copy

`carré`

`struct point *`	`points`	●
`int`	`n_points`	`4`

`carré_rouge`

`struct point *`	`points`	●
`int`	`n_points`	`4`

```
struct forme carré_rouge = carré;

carré_rouge.points[0].y = -0.5;
```

- Même tableau `points`,
  car le pointeur a été copié…

`points[0]`	`points[1]`	`points[2]`	`points[3]`
`(-0.5, 0.0)`	`(0.0, 1.0)`	`(-1.0, 0.0)`	`(0.0, -1.0)`

Copie profonde

Deep copy

```
struct forme carré_rouge;

carré_rouge.n_points = carré.n_points;
carré_rouge.points = malloc(carré.n_points * sizeof(struct point));

for (int i=0; i<carré.n_points; i++)
{
    carré_rouge.points[i].x = carré.points[i].x;
    carré_rouge.points[i].y = carré.points[i].y;

    // ou alors carré_rouge.points[i] = carré.points[i];
}
```

- Il faut créer un nouveau “tableau” dynamique…