Haute École Léonard de Vinci
SCIENCES ET TECHNIQUES

I2181A
Langage C :
modularisation

Chap. 9
Types utilisateurs

Anthony Legrand

Les structures

Structures

- **type composé** permettant de regrouper dans une même entité des données – nommées ***champs*** – pouvant être de types différents

Définition de structure

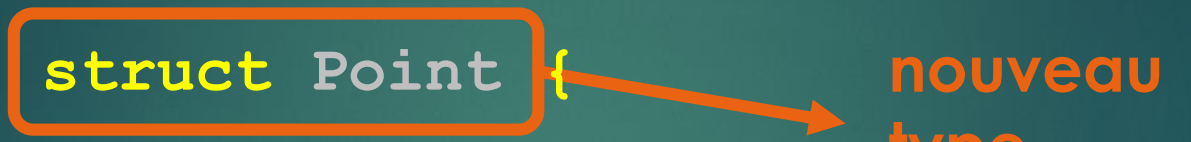
- définition de type structure:

```
struct Point {
    int abscisse;
    int ordonnee;
};
```

Définition de structure

- définition de type structure:

```
struct Point {
    int abscisse;
    int ordonnee;
};
```



Définition de structure

- définition de type structure:

```
struct Point {
    int abscisse;
    int ordonnee;
};
```

mot-clé devant accompagner l'identifiant de type

Définition de structure

- définition de type structure:

```
struct Point {
    int abscisse;
    int ordonnee;
};
```

Déclaration de variable

- avec ou sans initialisation

```
struct Point p1;

struct Point p2 = {0,0};

struct Point *ptr = &p2;
```

Accès aux champs

- nom de la variable avec nom du champs introduit par « . » ou « **->** » (cf. `man 7 operator`)

Niveau de priorité	Opérateur	Description	Associativité
15	`[ ]`	indice de tableau	de gauche à droite
	`( )`	appel de fonction	
	`.`	sélection de membre	
	`->`	sélection de membre par déréférencement	
	`++ --` (suffixe)	post incrémentation et décrémentation	
	`++ --` (préfixe)	pré incrémentation et décrémentation	
	`~`	complément à 1 (inversion des bits)	

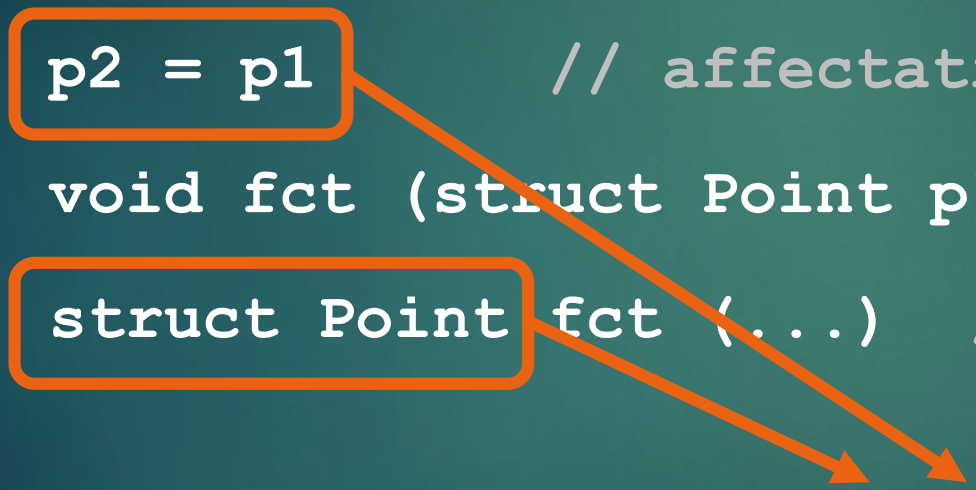
```
p1.abscisse = 5;
(*ptr).ordonnee = 8;
ptr->ordonnee = 8;
```

Opérations sur les structures


```
ptr = &p1      // adresse d'une structure
int *p = &p1.abscisse  // adresse d'un champ
p2 = p1        // affectation d'une structure
void fct (struct Point p) // paramètre
struct Point fct (...)  // valeur de retour
```

Opérations sur les structures

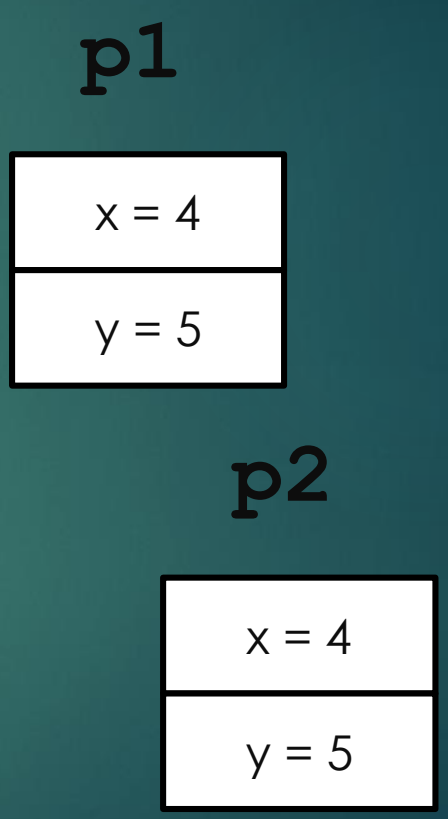

```
ptr = &p1      // adresse d'une structure
int *p = &p1.abscisse  // adresse d'un champ
p2 = p1        // affectation d'une structure
void fct (struct Point p) // paramètre
struct Point fct (...)  // valeur de retour
```



Copie bit à bit !

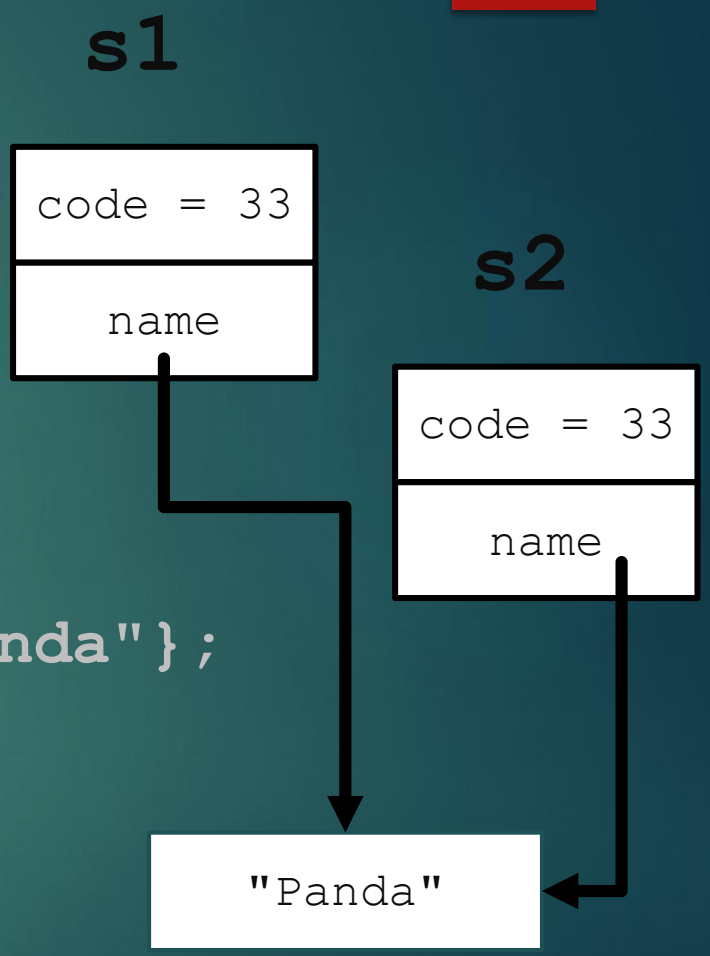
Copie bit à bit


```
struct Point {
    int x;
    int y;
};
struct Point p1 = {4, 5};
struct Point p2 = p1;
```



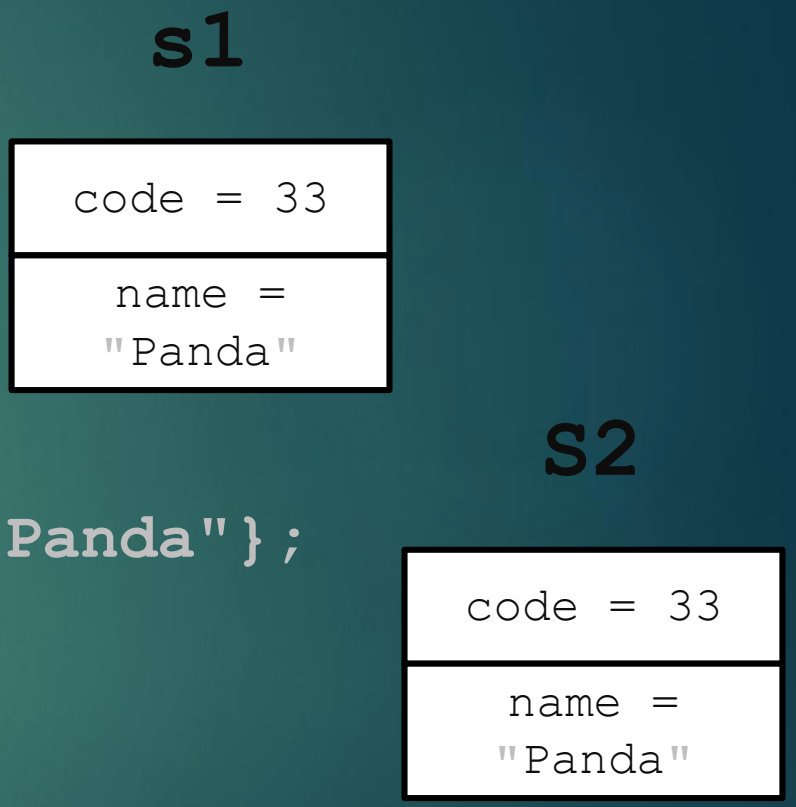
Shallow copy ou Copie superficielle


```
struct Species {
    int code;
    char *name;
};
struct Species s1 = {33, "Panda"};
struct Species s2 = s1;
```



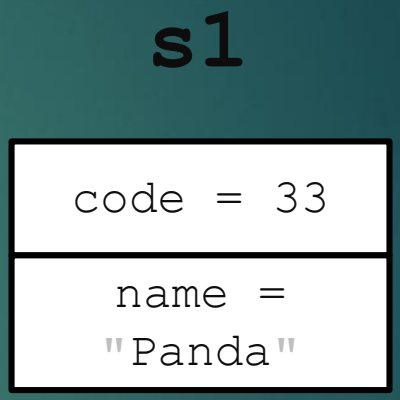
Alternative

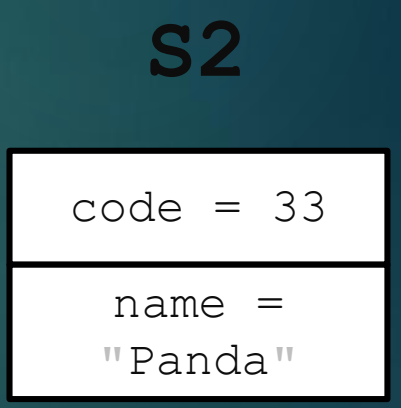

```
struct Species {
    int code;
    char name[10];
};
struct Species s1 = {33, "Panda"};
struct Species s2 = s1;
```



Alternative


```
struct Species {
    int code;
    char name[10];
};
struct Species s1 = {33, "Panda"};
struct Species s2 = s1;
```





Pas de pointeur → les données se trouvent dans la structure

Valeur de retour recopiée


```
struct Point fct (...)
```

- Copie de structure
- parfois **couteux**

Passage par valeur

- Récupération du résultat d'une fonction

```
void fct (struct Point *p)
```

  - Pour pouvoir être modifiée par une fonction, il faut fournir l'adresse de la structure en paramètre!

Passage par valeur

- Fourniture d'une donnée à une fonction

```
void fct (const struct Point *p)
```

  - On passe souvent un **pointeur** vers une structure pour éviter une copie couteuse sur la pile
  - Dans ce cas, pour éviter qu'elle soit modifiée par la fonction, la structure est déclarée **constante**

Les énumérations

Énumération

- type construit à partir d'un **ensemble de valeurs** spécifiées dans la définition du type
- permet de définir des ensembles de **constantes entières** en les regroupant par thème et en les nommant
- code plus facile à lire & plus fiable
  (ex: éviter l'utilisation de *magic numbers*,
  i.e. des constantes numériques non nommées)

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

nouveau type énuméré

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

mot-clé devant accompagner l'identifiant de type

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

ensemble des valeurs

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

- identifiant des valeurs énumérées en majuscules (= constantes)
- énumération = type entier !

```
ROUGE = 0
JAUNE = 1
VERT  = 2
...
```

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE=10, VERT, BLEU };
```

- identifiant des valeurs énumérées en majuscules (= constantes)
- énumération = type entier !

```
ROUGE = 10
JAUNE = 11
VERT  = 12
...
```

Définition d'énumération

- définition de type énuméré:

```
enum Couleur { ROUGE, VERT, BLEU };
```

- identifiant des valeurs énumérées en majuscules (= constantes)
- énumération = type entier !

⇒ usage fréquent dans des *switch* (branchement multiple en fonction d'une valeur entière)

Utilisation

- déclaration de variable

```
enum Couleur maCouleur, maFavorite;
```

- affectation

```
maCouleur = ROUGE;
```

- opérations

```
maCouleur++;  // reçoit la couleur JAUNE
```

Définition de types utilisateur : Conventions

- Un identificateur de type commence toujours par une majuscule
- Les types utilisateur sont définis après les directives du préprocesseur, généralement dans les fichiers d'entête