

# **Mise à niveau en C**

Types personnalisés: structures, enums et unions

Enseignant: P. Bertin-Johannet

# Les structures

# Les structures

- Les structures permettent de créer de nouveaux types
- Le nouveau type sera une combinaison de plusieurs champs

```
struct User{  
    int age;  
    char* name;  
};  
  
int main(){  
    struct User alain = {.age = 24, .name = "alain"};  
    printf("%d\n", alain.age);  
}
```

## Pourquoi utiliser une structure

- Une structure permet par exemple de regrouper des variables

```
int main(){
    int x1, y1, z1, x2, y2, z2;
    read_point(&x1, &y1, &z1);
    read_point(&x2, &y2, &z2);
    calc_dist(x1, x2, y1,
              y2, z1, z2);
}
```

```
struct Point{
    int x;
    int y;
}
int main(){
    struct Point p1, p2;
    read_point(&p1);
    read_point(&p2);
    calc_dist(p1, p2);
}
```

## Définition d'une structure

- Pour définir une structure on utilise le mot clé `struct`
- On écrit ensuite le nom de la structure puis la déclaration des variables qu'elle contiendra.

```
struct User{  
    int age;  
    char* name;  
    float height;  
    char* password;  
};
```

## Utilisation d'une structure

- On peut définir des variables du type de la structure en écrivant : `struct NomStructure variable;`.
- Pour accéder à un des champs contenus dans la structure, on utilise un point.

```
int main(){  
    struct User alice;  
    alice.age = 25;  
    alice.password = "Bonjour";  
    printf("%d\n", alice.age);  
}
```

## Utilisation d'une structure

- Depuis C99 , on peut instancier une structure en lui donnant des valeurs ainsi :

```
variable = {.champ1 = valeur1, .champ2 = valeur1, ...};
```

```
int main(){  
    struct User alice = {.age = 25, .password = "Bonjour"};  
    printf("%d %s\n", alice.age, alice.password);  
}
```

## Allocation de mémoire d'une structure

- On peut aussi réserver de la mémoire pour une structure en utilisant malloc .

```
int main(){  
    struct User* alice = malloc(sizeof(struct User));  
    (*alice).age = 25;  
    (*alice).password = "Bonjour";  
    printf("%d %s\n", (*alice).age, (*alice).password);  
}
```

## Pointeurs vers une structure

- Afin d'éviter d'écrire `(*nom).var` à chaque utilisation d'un pointeur vers une structure on peut utiliser l'opérateur flèche.

```
int main(){
    struct User alice = malloc(sizeof(struct User));
    alice->age = 25;
    alice->password = "Bonjour";
    printf("%d %s\n", alice->age, alice->password);
}
```

# Tableau de structures

- On peut aussi créer un tableau de structures.

```
int main(){  
    struct User utilisateurs[5]; // un tableau de 5 utilisateurs dans la  
    pile  
    struct User *administrateurs = malloc(2*sizeof(struct User)); // un  
    tableau de 2 admins dans le tas  
}
```

## Alias de type

- Afin d'éviter d'écrire `struct` à chaque utilisation, on peut utiliser le mot clé `typedef` pour créer un **alias de type**.

```
// ici on permet d'écrire User au lieu de struct User
typedef struct User User;
int main(){
    User ann = {.age = 28, .name = "ann"};
}
```

## Structure et mémoire

- Contrairement aux variables dans une fonction, les champs d'une structure sont enregistrés dans la mémoire dans l'ordre où ils ont été déclarés.
- Cela peut nous obliger à réfléchir à l'ordre dans lequel déclarer les champs d'une structure.

```
struct vecteur{  
    int taille; // taille apparaîtra en premier dans la mémoire  
    int* elements; // elements en second  
}
```

# Alignement et taille d'une structure

- A chaque type est associé une valeur d'alignement.
- Pour des raisons matérielles, une variable d'un type doit être enregistrée à une adresse multiple de son alignement.
- Par exemple si l'alignement d'un `int` est quatre, son adresse mémoire devra être un multiple de quatre.
- Pour satisfaire cette condition tout en respectant l'ordre de présence des champs d'une structure, le compilateur peut ajouter des octets de compensation.

# Alignement et taille d'une structure : Exemple

- Ici on considère un alignement de 4 pour `int` et 1 pour `char`.

```
struct IntChar{  
    int a;  
    char b;  
};  
  
int main(){  
    struct IntChar a = {.a = 5, .b = 6};  
}
```

Valeurs    Addresses

5	0xff80
	0xff81
	0xff82
	0xff83
6	0xff84

# Alignement et taille d'une structure : Exemple

- Ici on considère un alignement de 4 pour `int` et 1 pour `char`

```
struct CharInt{  
    char a;  
    int b;  
}  
  
int main(){  
    struct CharInt a = {.a = 5, .b = 6};  
}
```

Valeurs    Addresses

5	0xff80
??	0xff81
	0xff82
	0xff83
6	0xff84
	0xff85
	0xff86
	0xff87

# Les unions

## Différence avec une structure

- Une union est déclarée comme une structure mais elle ne peut contenir **qu'un seul de ses attributs** en même temps.

```
struct point {  
    int x; // la structure point contiendra toujours x et y  
    int y;  
};  
union age {  
    int age; // l'union age contiendra soit un entier soit un float  
    float age; // mais jamais les deux en même temps  
};
```

# Union et mémoire

- Parce qu'une union ne contient qu'un seul de ses attributs, sa taille mémoire sera celle du plus grand attribut.

## ! Attention

Une union utilise la même mémoire pour tous ses attributs, écrire dans l'un écrasera donc tous les autres

```
union Info { // une variable de type union Info n'utilisera que 4 octets de
mémoire
    int age;
    float taille;
    char nom[4];
};
```

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
??	0xff150
	0xff151
	0xff152
	0xff153
	0xff154
	0xff155

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
97 ('a')	0xff150
98 ('b')	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
97 ('a')	0xff150
98 ('b')	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

Affiche: "info: abc"

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
15	0xff150
	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
168.2	0xff150
	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
168.2	0xff150
	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

Affiche: "age: 22849"

# Union et mémoire: exemple

Programme:

```
int main(){  
    union Info information;  
    information.nom = "abc";  
    printf("info: %s\n", information.nom);  
    information.age = 15;  
    information.taille = 168.2;  
    printf("info: %f\n", information.age);  
}
```

Mémoire:

Valeurs	Addresses
168.2	0xff150
	0xff151
99 ('c')	0xff152
0 ('0')	0xff153
	0xff154
	0xff155

# Exemples d'utilisation d'unions

```
union ResponseBdd{  
    struct User reponse[30];  
    char message_erreur[100];  
};
```

```
union Animal{  
    struct Chat chat;  
    struct Cheval cheval;  
    struct Chien chien;  
};
```

```
union Forme{  
    struct Point triangle[3];  
    struct Point rectangle[4];  
    struct Point octogone[8];  
};
```

```
union Message {  
    char user_logout[5];  
    int user_id_change[2];  
}; // etc...
```

# Les enums

## Qu'est ce qu'un enum ?

- Utiliser un enum en C nous permet de créer un nouveau type d'entier ainsi que des constantes associées.
- Pour déclarer un enum, on précise son nom puis les valeurs des différentes constantes séparées par des **virgules**.

```
enum couleur {  
    ROUGE = 0,  
    BLEU = 1,  
    VERT = 2,  
    BLANC = 3,  
};
```

## Valeur par défaut d'un enum

- Si on ne précise pas la valeur d'une constante dans un enum, elle sera automatiquement affectée.
- La première constante aura la valeur 0, la seconde 1, etc.

```
typedef enum affichage {GAUCHE, DROITE, LIBRE = 200} affichage;
```

```
int main(){  
    affichage opt = GAUCHE;  
    printf("%d\n", opt); // affiche 0  
    printf("%d\n", DROIT); // affiche 1  
    printf("%d\n", LIBRE); // affiche 200  
}
```

## Mise en pratique

```
printf(")
```

```
    _____  
    < TP 5 >
```

```
    - - - - -
```

```
      \      ^  ^  
        _____  
      \ (oo)\_____  
        (__) \                )\ /\ /\ /\  
              || - - - - - ww - ||  
              ||                ||
```

```
    ")
```