# Cours Langage C/C++
# Annexe sur les structures

Thierry Vaira

BTS IRIS Avignon

tvaira@free.fr 

# Rappel : struct

- Une structure est un **objet agrégé comprenant un ou plusieurs champs (membres)** d'éventuellement différents types que l'on regroupe sous un seul nom afin d'en faciliter la manipulation et le traitement.
- Chacun des champs peut avoir n'importe quel type, y compris une structure, à l'exception de celle à laquelle il appartient.

## Déclaration d'une structure

```
[classe de memorisation] struct [etiquette]
{
   type champ_1;
   ...
   type champ_n;
} [identificateur];
```

# Alignement dans les structures

- Le mot clé `__attribute__` permet de spécifier des attributs spéciaux de `struct` (et types d'`union`) lorsque on définit ces types.
- Ce mot clé est suivie par une spécification attribut à l'intérieur des parenthèses doubles. Il vous faut consulter la documentation de `gcc` pour en savoir plus sur ces attributs :

  Lien : gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
- Cette partie est essentielle dès que vous êtes concerné par la manipulation des structures pour lire et écrire des méta-données dans des fichiers (ou pour des échanges de données structurés par le "réseau").

# Problème

- Si la machine respecte un alignement 32 bits pour les allocations mémoires des types, vous allez être surpris du comportement de votre programme pour cette structure.
- Car pour préserver l'alignement 32 bits, il y aura un ajout de 3 octets nulles après le `char`
- Soit un total de : $4 + 1 + 3$ (*padding*) $+ 4 = 12$ octets

## Exemple d'une structure :

```
struct test_t
{
  int  a; /* taille : 4 octets */
  char b; /* normalement : 1 octet mais avec un alignement 32 bits il aura une taille de 4 octets ! */
  int  c; /* taille : 4 octets */
};
```

# Solution

- L'attribut `packed` va permettre de modifier ce comportement.
- En effet, cet attribut précise que la **mémoire minimale requise** doit être utilisée pour représenter le type.
- Donc, on aura un total de : $4 + 1 + 4 = 9$ octets

## Exemple d'une structure `packed` :

```
struct test_t_a
{
  int   a; /* taille : 4 octets */
  char b; /* taille minimale pour un char : 1 octet */
  int   c; /* taille : 4 octets */
} __attribute__((__packed__));
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

## Comparaison des tailles de structures :

```
int main()
{
   printf("La structure test contient : int, char, int\n\n");
   printf("sizeof(struct test_t) = %d octets\n", sizeof(struct test_t));
   printf("sizeof(struct test_t_a) = %d octets\n", sizeof(struct test_t_a));
   printf("\n");
   return 0;
}
```

## L'exécution du programme d'essai permet de vérifier cela :

```
La structure test contient : int, char, int
sizeof(struct test_t)  = 12 octets
sizeof(struct test_t_a) = 9 octets
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

# Conclusion

- Les processeurs n'aiment pas (du tout) réaliser des échanges mémoires d'une taille différente de leur mode car cela les oblige à faire des manipulations supplémentaires. Par défaut, le compilateur active dont un alignement qui correspond à un multiple d'un octet (c'est encore souvent 4 octets ou 32 bits qui est l'alignement par défaut pour le type de base `int`).
- Quand on manipule des données structurées, il faut être certain de connaître leur organisation mémoire car sinon on interprètera "mal" ces données ! En conclusion, cet alignement par défaut de 4 octets ne nous arrangera pas pour des structures composées de type `char` ou `short int` (c'est-à-dire pour des champs sur 8 bits ou 16 bits) car le compilateur va "bourrer" pour conserver un alignement 32 bits.
- On peut aussi utiliser la directive de pré-compilation `#pragma` :

```
#pragma pack(1) // alignement 1 octet
... // ici nos structures
#pragma pack() // retour à l'alignement par défaut
```