

# Information, Calcul et Communication

## Composante Pratique: Programmation C++

### MOOC sem6 : typedef, structure

Comment transformer une *donnée* en une *information* ?

Assembler des données hétérogènes avec **struct**

Une structure en mémoire: alignement et padding

Accès fin à la mémoire : bits field / opérateurs bit à bit

## Comment transformer une donnée en une information ?

Avant de répondre à cette question il faut caractériser ce qu'est une donnée :

Une suite de 0 et de 1 est un **motif binaire**: 0101101100111001

On ne peut rien en faire tant qu'on ne connaît pas son **type**: `char`, `int`, `double`...

Un **motif binaire** associé à un **type de base** devient une **donnée** manipulable par le processeur avec un ensemble d'opérateurs définis pour ce type.

Ex: 11000001000100000000000000000000 associé au type `float` représente -9.0

Les types de base sont de très bas niveau ; ils sont utiles pour le compilateur pour obtenir un programme exécutable mais ils ne disent rien sur le **but** de ces données.

L'instruction `typedef` permet de transformer une **donnée** en une **information** en définissant un nouveau type synonyme du type de base:

```
typedef float Temperature;  
Temperature temp_air(-9.0);
```

## Comment transformer une donnée en une information ? (2)

Plus généralement **typedef** permet de créer un nouveau type à partir d'un type que le compilateur connaît déjà. C++11 offre une syntaxe plus intuitive avec **using**.

```
typedef type_deja_connu Nouveau_type; // par convention, un type créé
// par l'utilisateur commence
// par une Majuscule
```

*Same thing. !!!*

```
using Nouveau_type = type_deja_connu;
```

Le nom du nouveau type doit apporter des précisions sur la nature des données, leur signification, leur but etc...

```
typedef array<int,3>    RGB_color;
typedef vector<double>  Vecteur;
typedef vector<Vecteur> Matrice;
```

```
RGB_color red = {255,0,0};
Matrice    mat_nulle(3, Vecteur(3));
Vecteur    produit_vectoriel(Vecteur v1, Vecteur v2);
```

# Assembler des données hétérogènes avec struct

Avec **vector**, **array** et un **tableau-à-la-C** on peut seulement rassembler des **données de même type** sous un même identificateur.

Avec le mot clef **struct** on peut rassembler dans un nouveau type des **données de types différents**.

C'est l'outil idéal pour structurer les données d'un problème.

```
struct Nom_du_type
{
    type1  identificateur1;
    type2  identificateur2;
    ...
};
```

⇒ Bonne pratique: définir D'ABORD le type...

```
Nom_du_type x={val1,val2...};
Nom_du_type y={};
```

... AVANT de créer des variables avec ce type  
// init à 0

## Assembler des données hétérogènes avec struct (2)

```
struct Personne
{
    string  nom;
    double  taille;
    int     age;
    char    sexe;
};

Personne worker = {"Dupont", 1.65, 59, 'F'};

++(worker.age);
cout << "Bon anniversaire "
      << (worker.sexe == 'M')? "M " : "Mme "
      << worker.nom << endl;

vector<Personne> candidat(10);

...
Personne promotion(vector<Personne>& candidat);
Personne boss=promotion(candidat);

if(worker == candidat[0])
    cout << worker ;
```

// C++11: syntaxe aussi utilisable  
// pour l'affectation  
// accès à un champ avec .  
// opérateur de priorité maximum

// vector de 10 Personne

// renvoie une Personne

// l'affectation est le seul opérateur autorisé

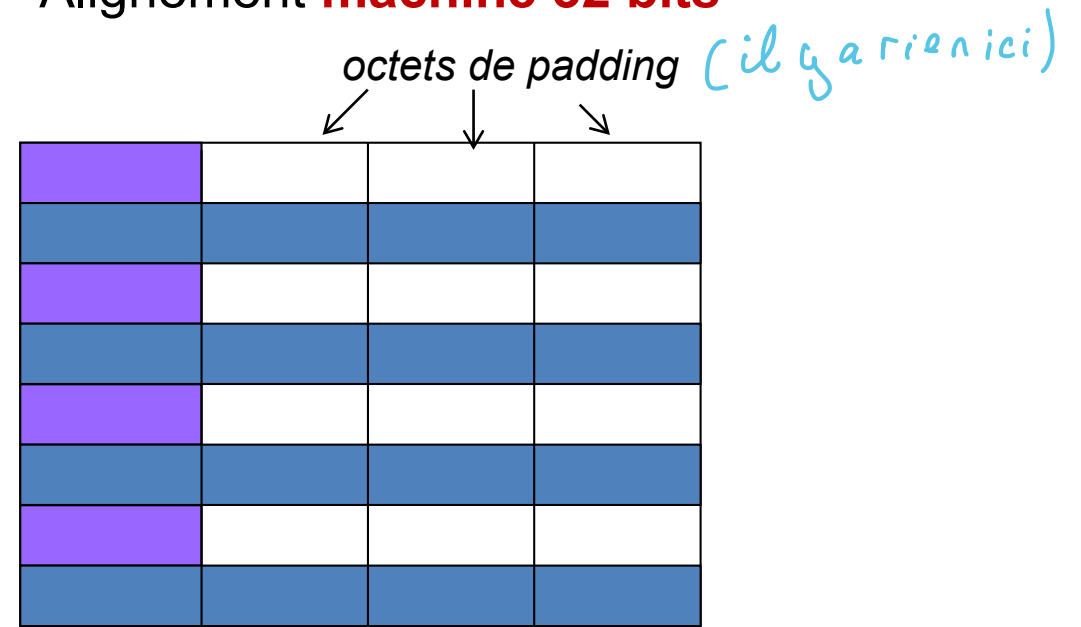
// interdit ! erreur de compilation  
// interdit ! erreur de compilation

## Une structure en mémoire: alignement et padding

l'ordre des champs peut conduire à une occupation mémoire plus importante que la somme de chacun des champs si la taille mémoire de chaque champ ne s'aligne pas sur un mot mémoire

```
struct Fiche1
{
    char    sexe;
    int     age;
    char    permis;
    int     avs;
    char    option;
    float   salaire;
    char    categorie;
    int     nb_heures;
};
```

Alignement **machine 32 bits**



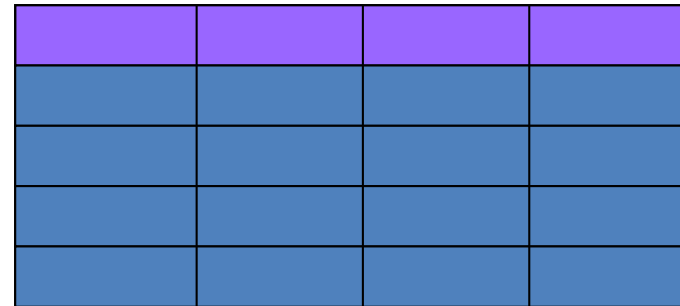
= 8x4 = 32 octets

## Une structure en mémoire: alignement et padding (2)

l'ordre des champs peut conduire à une occupation mémoire plus importante que la somme de chacun des champs si la taille mémoire de chaque champ ne s'aligne pas sur un mot mémoire

```
struct Fiche2
{
    char    sexe;
    char    permis;
    char    option;
    char    categorie;
    int     age;
    int     avs;
    float   salaire;
    int     nb_heures;
};
```

Alignement **machine 32 bits**



= 5x4 = 20 octets

## Une structure en mémoire: alignement et padding (3)

Exemple: Alignements standards sur une **machine 32 bits** : 1 mot = 4 octets

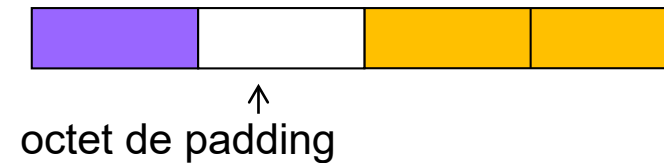
**char** peut occuper n'importe quel octet

**short** : sur les octets d'adresse paire

**int, long, float**: sur les mots

**double** occupe deux mots

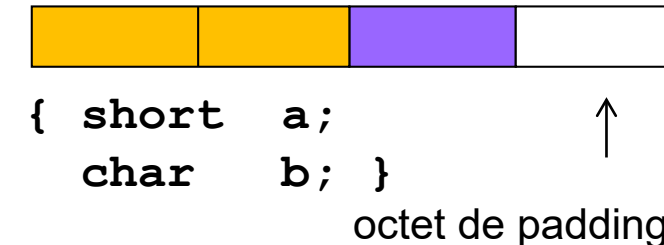
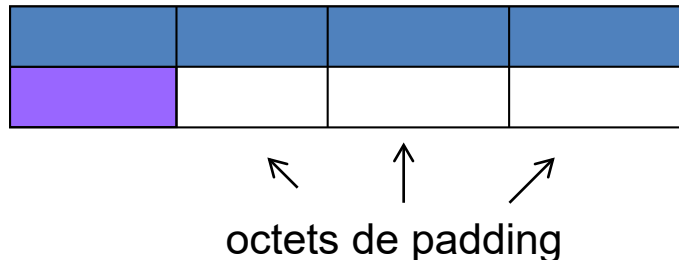
```
{ char  a;  
  short b; }
```



Les octets de paddings sont ajoutés aussi en fin de structure pour s'aligner sur le champ de plus grande taille

Pour faciliter la manipulation de tableaux, vector, array

```
{ int  n;  
  char c; }
```





## Accès fin à la mémoire : bits field

But: minimiser le temps de communication entre le processeur et les périphériques en transmettant le moins d'octets possible. *On indique le nb de bits par champ.*

```
struct Etat  
{
```

```
    unsigned int pret    : 1; // un seul bit ! C'est un véritable booléen !  
    unsigned int ok      : 1;  
    int donnee1          : 5;  
    int                : 3;  
    unsigned int ok2     : 1;  
    int donnee2          : 4;
```

```
};
```

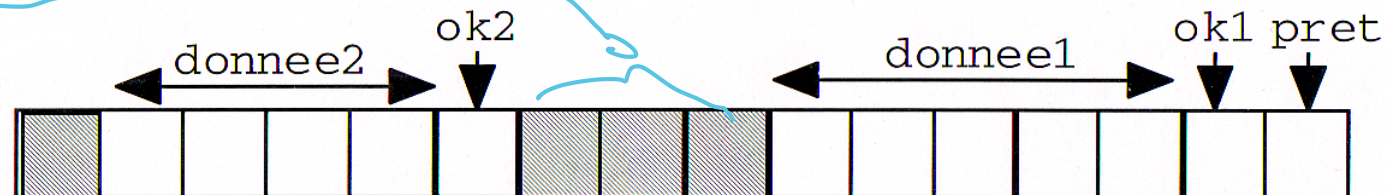
```
Etat mot;
```

```
mot.donnee1 = 13;
```

**Attention: seulement des types entiers.**

**Dépendance machine pour les entier signés**

cella  
want dire ignore



## Accès fin à la mémoire : les opérateurs bit à bit

sur 17 niveaux de priorités

Associativité: pour les  
opérateurs de même  
priorité

Gauche->Droite / Left-to-Right  
Droite->Gauche / Right-to-Left

**Opérateurs  
bit à bit**

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	Left-to-right
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of <sup>[note 1]</sup> await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
6	a+b a-b	Addition and subtraction	Left-to-right
7	<< >>	Bitwise left shift and right shift	Left-to-right
8	<==>	Three-way comparison operator (since C++20)	Left-to-right
9	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	Left-to-right
10	== !=	For relational operators = and ≠ respectively	Left-to-right
11	&	Bitwise AND	Left-to-right
12	^	Bitwise XOR (exclusive or)	Left-to-right
13		Bitwise OR (inclusive or)	Left-to-right
14	&&	Logical AND	Left-to-right
15		Logical OR	Left-to-right
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^=  =	Ternary conditional <sup>[note 2]</sup> throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR	Right-to-left
17	,	Comma	Left-to-right

# Les opérateurs logiques

`bool a, b;`

		b	
a and b		0	1
a	0	0	0
	1	0	1

$0 \text{ and } b = 0$

$1 \text{ and } b = b$

`bool a, b;`

		b	
a or b		0	1
a	0	0	1
	1	1	1

$0 \text{ or } b = b$

$1 \text{ or } b = 1$

bit à bit XOR  
EXCLUSIVE OR:  $\wedge$

		b	
a $\wedge$ b		0	1
a	0	0	1
	1	1	0

$0 \wedge b = b$

$1 \wedge b = \underline{b}$

*négation de b*

⚠ One or the other but not both!

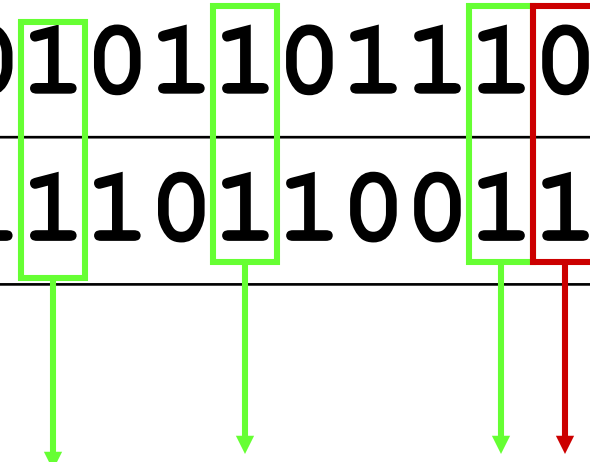
**Exemples avec les opérateurs bit à bit  
soit n et p deux variables de type entier**

<b>n</b>	<b>0000010101101110</b>
<b>p</b>	<b>0000001110110011</b>

# ET-Logique bit à bit

n & p

n	0000010101101110
p	0000001110110011
n&p	0000000100100010



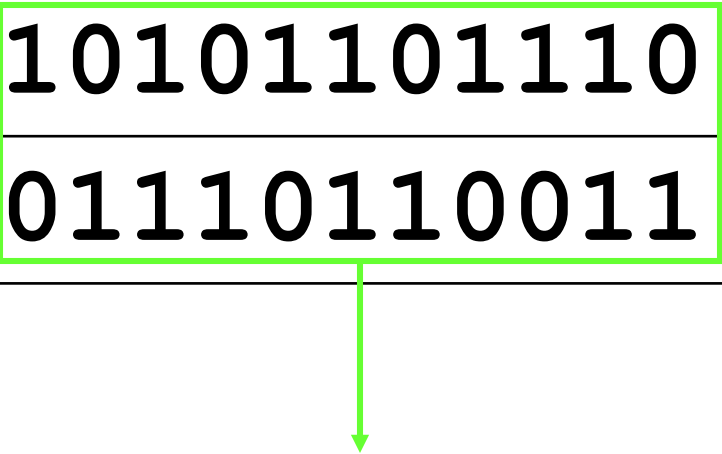
**b & 1 donne b**  
**b & 0 donne 0**

<b>b</b>	bbbbbbbbb <b>bbb</b> bbb <b>bb</b> bbb
<b>p</b>	0000000 <b>111</b> 0 <b>11</b> 00 <b>11</b>
<b>b&amp;p</b>	0000000bbb0bb00bb

# OU\_Logique bit à bit

$n \mid p$

n	0000010101101110
p	0000001110110011
$n \mid p$	0000011111111111



$b \mid 1 \text{ donne } 1$   
 $b \mid 0 \text{ donne } b$

<b>b</b>	<div> <div>bbbbbb</div> <div>bb</div> <div>b</div> <div>bb</div> <div>bb</div> <div>bb</div> </div>
<b>p</b>	<div> <div>000000</div> <div>111</div> <div>0</div> <div>11</div> <div>00</div> <div>11</div> </div>
<b>b   p</b>	<div> <div>bbbbbb</div> <div>111</div> <div>b</div> <div>11</div> <div>bb</div> <div>11</div> </div>



# Negation bit-à-bit

$\sim n$   
*↳ complément à 1*

n	0000010101101110
$\sim n$	1111101010010001

# OU\_Exclusif bit à bit

$n \wedge p$

n	00000010101101110
p	00000001110110011
$n \wedge p$	00000011011011101



$b \wedge 1$  donne  $\sim b$  (noté  $\bar{b}$ )  
 $b \wedge 0$  donne  $b$

<b>b</b>	b b b b b b b b b b b b b b b b
<b>p</b>	0 0 0 0 0 0 0 <b>1 1 1</b> 0 <b>1 1</b> 0 0 <b>1 1</b>
<b>b<sup>p</sup></b>	b b b b b b b b <u><b>b b b</b></u> b <u><b>b b</b></u> b b <u><b>b b</b></u>

# Décalage du motif binaire vers la gauche

ex:  $n \ll 3$

équivalent à une multiplication par  $2^3$

n	
$n \ll 3$	

# Décalage du motif binaire vers la droite

ex:  $n \gg 3$

équivalent à une division **entière** par  $2^3$

n	0000010101101110 →
$n \gg 3$	→ 0000000010101101

Attention: comportement dépend de la machine pour les nombres signés négatifs

Ici pas de problème car le bit de poids fort est 0, donc peu importe si le nombre est de type int ou unsigned int, dans tous les cas des 0 sont introduits

## Application système embarqué 1:

extraction d'un champ de bit à l'aide d'un masque m avec décalage d



?

extraction d'un champ de bit à l'aide d'un masque m avec décalage d

b	bbbbbbbbb <b>bbbb</b> bbbbbb
m = 0xF	0000000000000000 <b>1111</b>
b>>6	?????b <b>bbbb</b>
m & (b>>6)	0000000000000000 <b>bbbb</b>

## Application système embarqué 2:

insertion d'un champ de bit à l'aide d'un masque m avec décalage d



on veut ranger  
une nouvelle valeur ici

par exemple:  $v = 0010$



insertion d'un champ de bit à l'aide d'un masque m avec décalage d

<b>b</b>	bbbbbbbbb <b>bbbb</b> bbbbbb
<b>m = 0xF</b>	000000000000000 <b>1111</b>
<b>m&lt;&lt;6</b>	0000000 <b>1111</b> 0000000
<b>~ (m&lt;&lt;6)</b>	1111111 <b>0000</b> 1111111
<b>b &amp; ~ (m&lt;&lt;6)</b>	bbbbbbbbb <b>0000</b> bbbbbb

# insertion d'un champ de bit à l'aide d'un masque m avec décalage d (suite et fin)

$b$	bbbbbbbbb <b>bbbbbb</b> bbbbbbbbb
$b \& \sim (m \ll 6)$	bbbbbbbbb0000bbbbbbbbb
$v \ll 6$	00000000001000000000
$(b \& \sim (m \ll 6)) \mid (v \ll 6)$	bbbbbbbbb0010bbbbbbbbb