Informatique II
Pointeurs et tableaux de structures

Rappels

- Les types de base (**int**, **float, char,** etc...) ne sont pas toujours suffisants pour modéliser des objets/ phénomènes de la réalité.
- La construction de ces **nouveaux types** de variables peut être faite en combinant tous les types de base et les types déjà définis. Ce sont les **structures**.

- Par exemple, on pourra avoir :
  - Un type ***Date*** qui contiendra le jour, le mois, l'année, le jour de la semaine et du mois en lettres.
  - Un type ***Personne*** qui contiendra un nom, un prénom, une adresse, une date de naissance (avec le type ***Date*** car on peut imbriquer les structures), un numéro de téléphone, etc…
  - Un type ***Classe***, qui contiendra un tableau de type ***Matière***, un tableau de type ***Personne*** qui seront les élèves, un tableau de type **N*ote***, etc…
  - Un type ***PaquetIP*** qui contiendra tous les champs d'une trame de données du protocole IP

Rappels

- Les types de bases (int, float ect.) ne sont pas toujours suffisant pour modéliser des objets/ phénomènes de la réalité.
- La construction de ces **nouveaux types** de variables peut être faite en combinant tous les types de base et les types déjà définis. Ce sont les **structures**.

Toujours une majuscule !

```
STRUCTURE Etudiant

    nom : chaîne de caractères
    prenom : chaîne de caractères
    num : entier
    groupe : entier
    note[N] : tableau de réels

FIN STRUCTURE
```

Champs (ou attributs ou membres) de la structure Etudiant

Rappels

- Une fois définie, la structure peut être utilisée comme un type : c'est un **type structuré.**

On peut aussi écrire:

```
struct _Point {
    float abscisse;
    float ordonnee;
};
…
struct _Point x;
```

```
typedef struct {
    float t abscisse;
    float t ordonnee;
} Point;

Point constructeurPoint(... ){ // fonction retournant un Point
    …
}

int main(){
    Point x; // declaration d'un Point x
    Point y; //declaration d'un Point y
    …
    return 0; ;
}
```

typedef permet de redéfinir un type : plus besoin d'écrire "struct _Point" Pour déclarer une variable

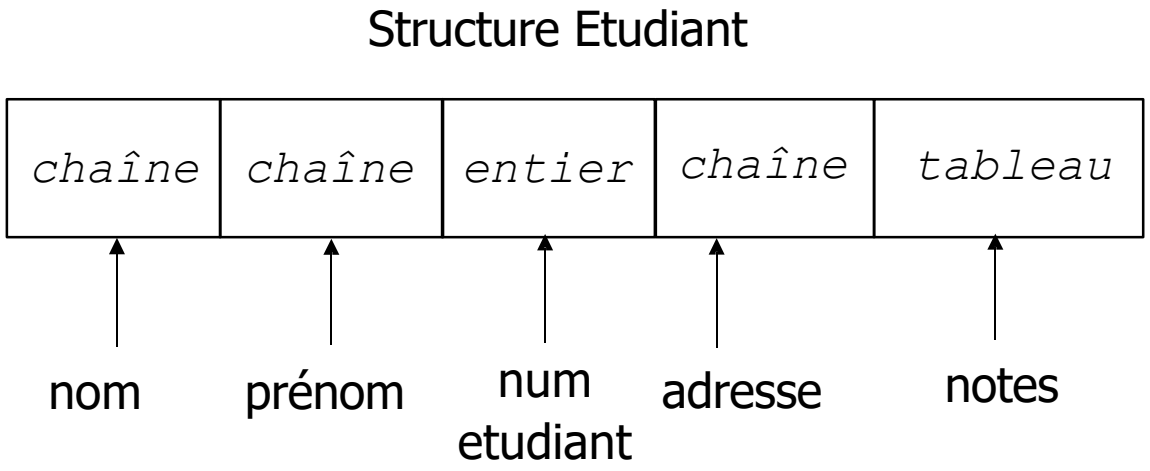
Rappels

- On peut manipuler les différents champs d'une variable de type structuré avec le caractère '.'

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

Point constructeurPoint(){ // fonction retournant un Point
    Point z;
    scanf("%f", &(z.abscisse) );
    scanf("%f" ,&(z.ordonnee) );
    return z;
}

int main(){
    Point x; // declaration d'un Point x
    Point y; //declaration d'un Point y
    x.abscisse = 10;
    x.ordonnee = 2.5;
    y = constructeurPoint();
    return 0;
}
```

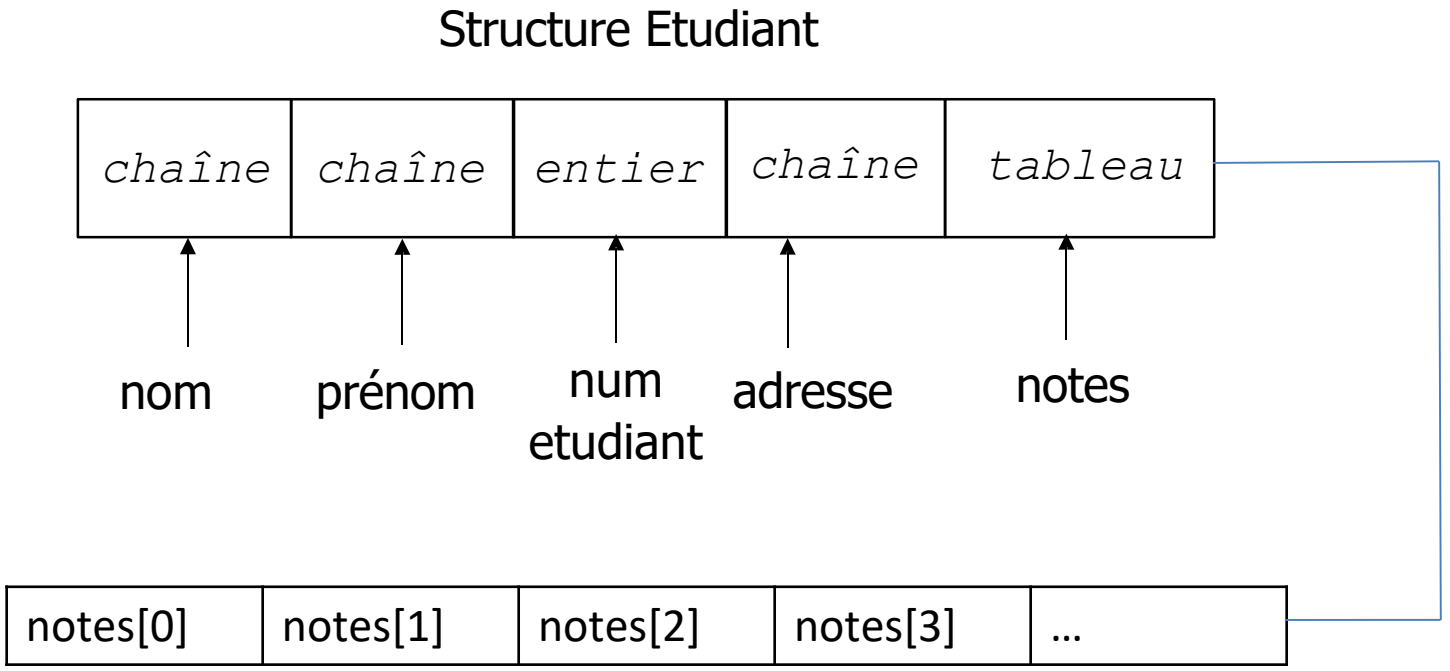
Structures et tableaux

- Les **tableaux** permettent de gérer des données de **même type** avec une seule variable.
- Les **types structurés** permettent d'avoir des **champs différentes** (différents types) dans une seule variable.



Structures et tableaux

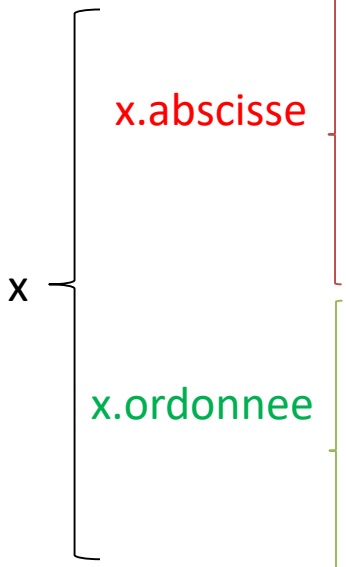
- Les **tableaux** permettent de gérer des données de **même type** avec une seule variable.
- Les **types structurés** permettent d'avoir des **champs différentes** (différents types) dans une seule variable.



➢ **Les champs d'une structure ne prennent pas tous la même place en mémoire**

Stockage d'une structure

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

int main(){
    Point x;
    x.abscisse = 1;
    x.ordonnee = 4500;
    return 0;
}
```



Adresse	Valeur
…	
1154	0000 0000
1155	0000 0000
1156	0000 0000
1157	0000 0001
1158	0000 0000
1159	0000 0000
1160	0001 0001
1161	1001 0100
…	

Stockage d'une structure

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

int main(){
    Point x;
    printf("%p\n",&x);
    printf("%p\n",&(x.abscisse));
    printf("%p\n",&(x.ordonnee));
    return 0;
}
```

```
0x7ffe3c2296f0
0x7ffe3c2296f0
0x7ffe3c2296f4
```

x.abscisse

x.ordonnee

Adresse	Valeur
…	
&x	
	???
&x+sizeof(int)	
	???
…	

Stockage d'une structure

- Les différents champs d'une structure sont à la suite dans la mémoire (comme les éléments d'un tableau). Mais contrairement au tableau, il n'y a pas de pointeur qui pointe sur la première adresse de la structure.

- L'accès aux différents champs de la structure se fait automatiquement: le compilateur connait le « saut de mémoire » à effectuer.

- Une structure étant un type, on peut connaître sa place en mémoire avec la fonction **sizeof().**

Stockage d'une structure

- Une structure étant un type, on peut connaître sa place en mémoire avec la fonction **sizeof().**

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

int main(){
    printf("La structure fait %ld octets\n",
sizeof(Point));
    return 0;
}
```

```
La structure fait 8 octets
```

Stockage d'une structure

- Une structure étant un type, on peut connaître sa place en mémoire avec la fonction **sizeof().**

```c
typedef struct{
  int a;
  int b;
  float c[3];
}Test;

int main(){
    printf("La structure fait %ld octets\n",
sizeof(Test));
    return 0;
}
```

```
La structure fait 20 octets
```

Stockage d'une structure

- Une structure étant un type, on peut connaître sa place en mémoire avec la fonction **sizeof().**

```c
typedef struct{
  int a;
  char b;
  int c;
}Test;

int main(){
    printf("La structure fait %ld octets\n",
sizeof(Test));
    return 0;
}
```

```
La structure fait 12 octets
```

Pourtant un char ne fait qu'un octet !

Stockage d'une structure

- Une structure étant un type, on peut connaître sa place en mémoire avec la fonction **sizeof().**

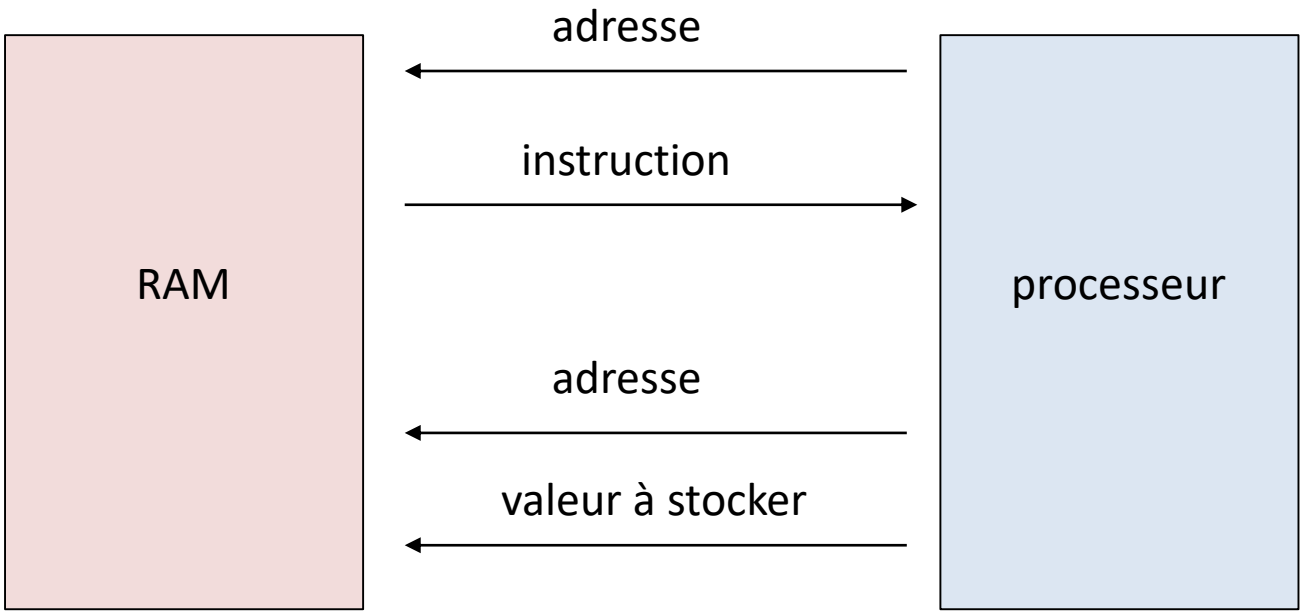
```c
typedef struct{
  int a;
  char b;
  int c;
}Test;

int main(){
    printf("La structure fait %ld octets\n",
sizeof(Test));
    Test t;
    printf("%p\n",&t.a);
    printf("%p\n",&t.b);
    printf("%p\n",&t.c);
    return 0;
}
```

```
La structure fait 12 octets
0x7ffd6ffdad30
0x7ffd6ffdad34
0x7ffd6ffdad38
```

Le champs b prends 4 octets malgré le fait qu'il soit de type **char** !

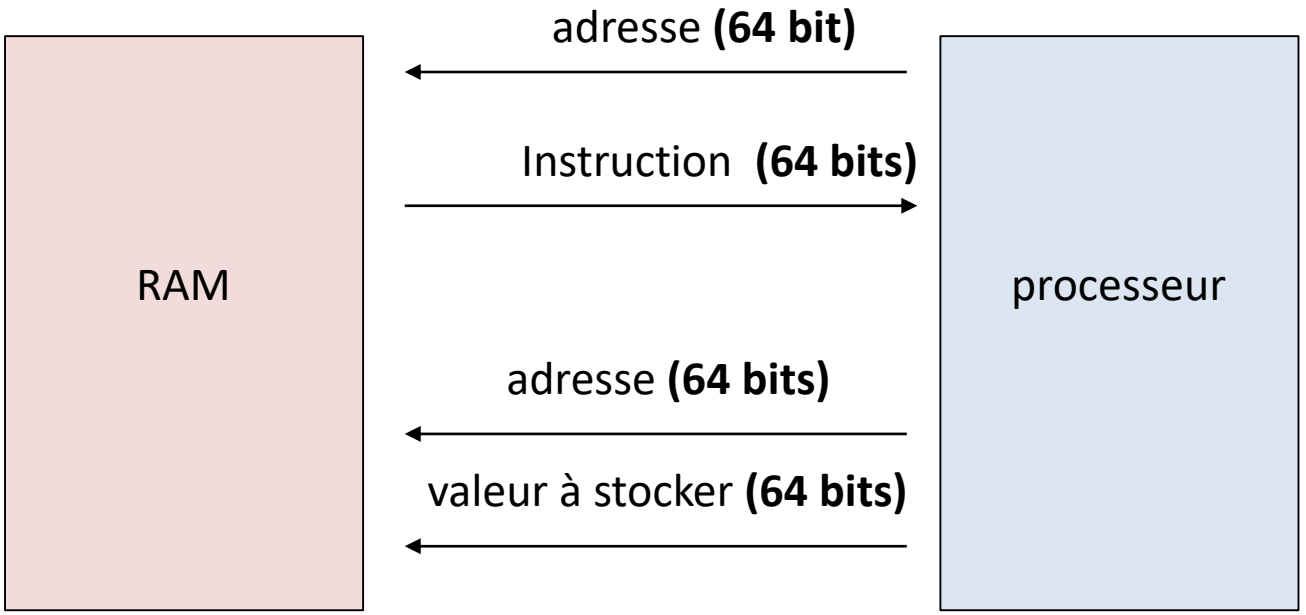
Notion de mot et alignement

- La mémoire est divisée en octets indiqués par des adresses mémoires.
- La mémoire communique avec le processeur qui effectue les instructions.



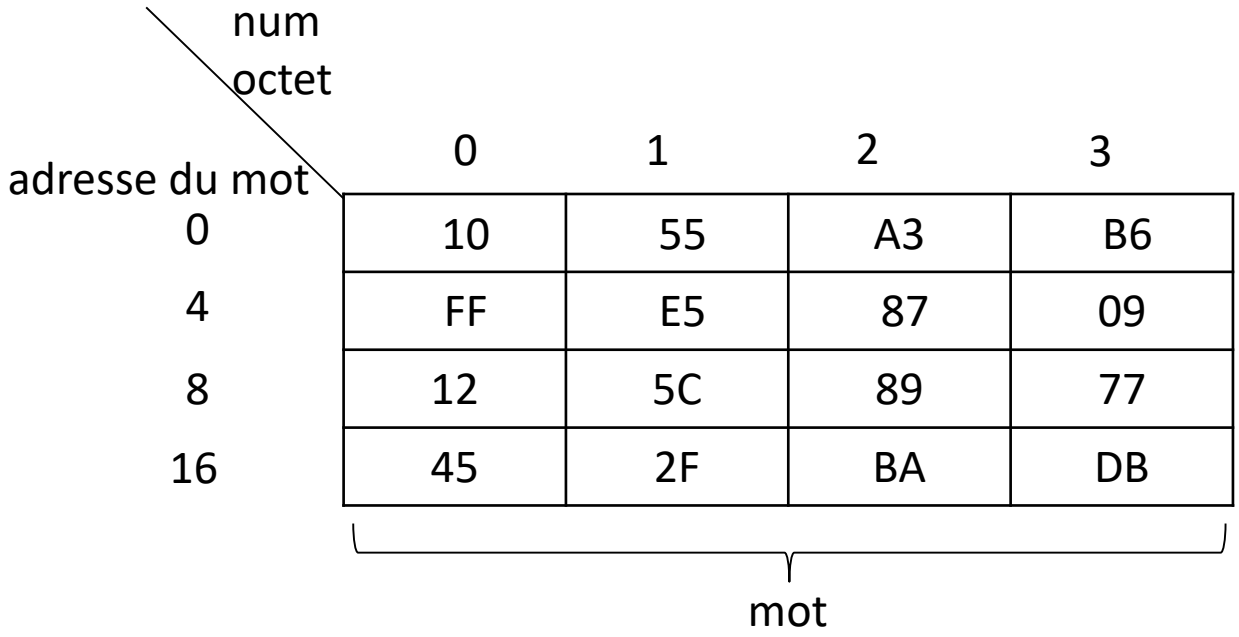
Notion de mot et alignement

- La mémoire est divisée en octets indiqués par des adresses mémoires.
- La mémoire communique avec le processeur qui effectue les instructions.
- Le processeur ne travaille pas sur des octets mais sur des **mots** dont la taille varie selon son architecture : 32 bits, **64 bits**, 128 bits



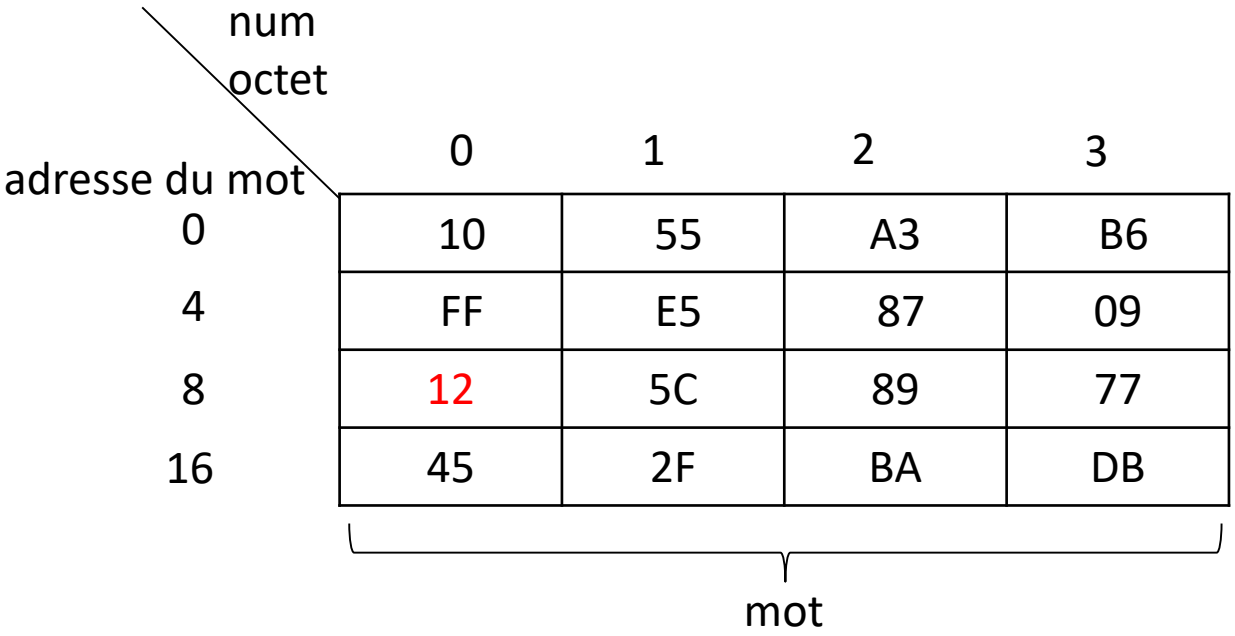
Notion de mot et alignement

- Pour assurer la rétrocompatibilité des programmes, GCC travaille avec des mots de 32 bits (4 octets).
- Illustration de la mémoire :



Notion de mot et alignement

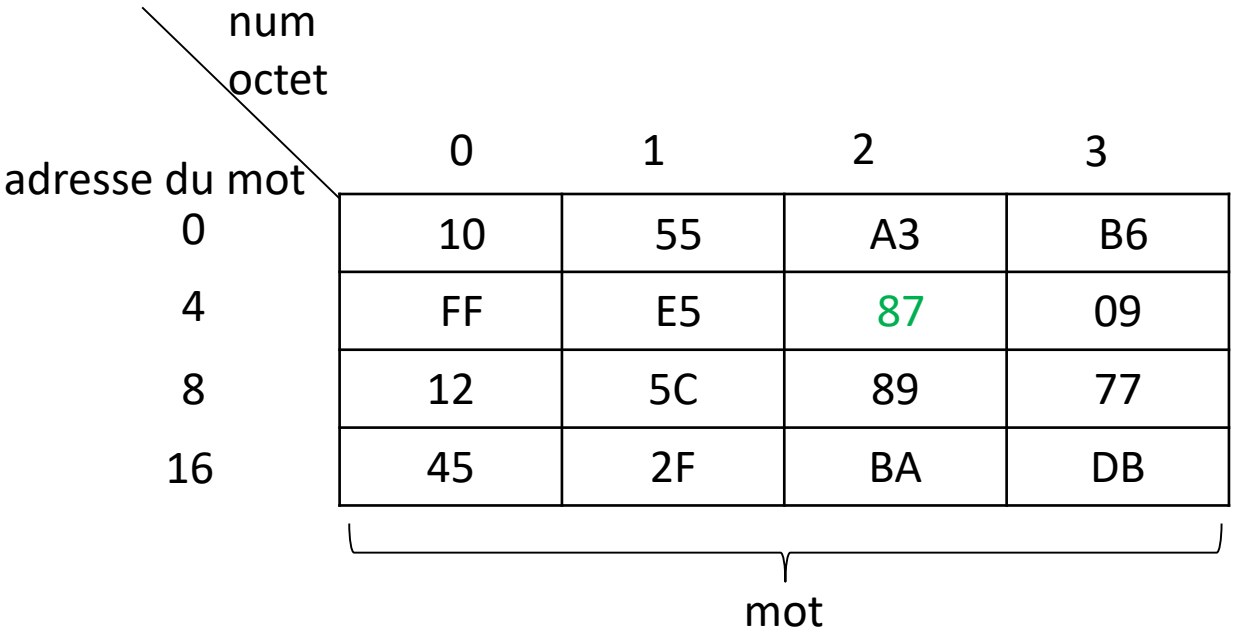
- Pour assurer la rétrocompatibilité des programmes, GCC travaille avec des mots de 32 bits (4 octets).
- Illustration de la mémoire :



adresse de 12 ? => 8

Notion de mot et alignement

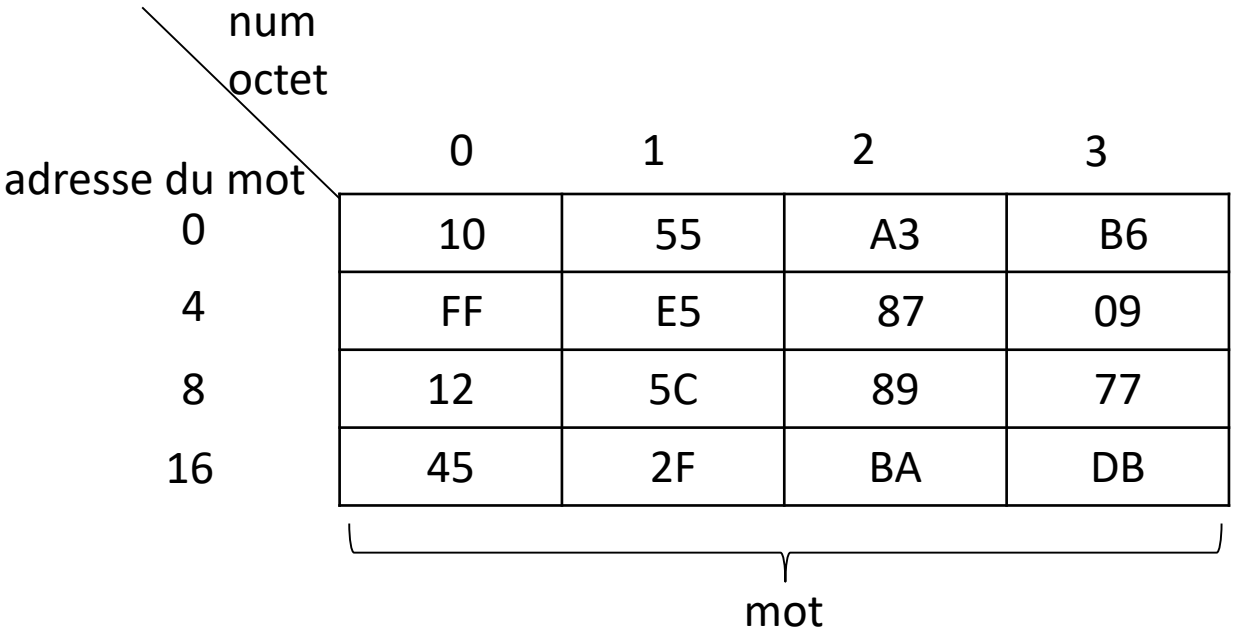
- Pour assurer la rétrocompatibilité des programmes, GCC travaille avec des mots de 32 bits (4 octets).
- Illustration de la mémoire :



adresse de 87? => 6

Notion de mot et alignement

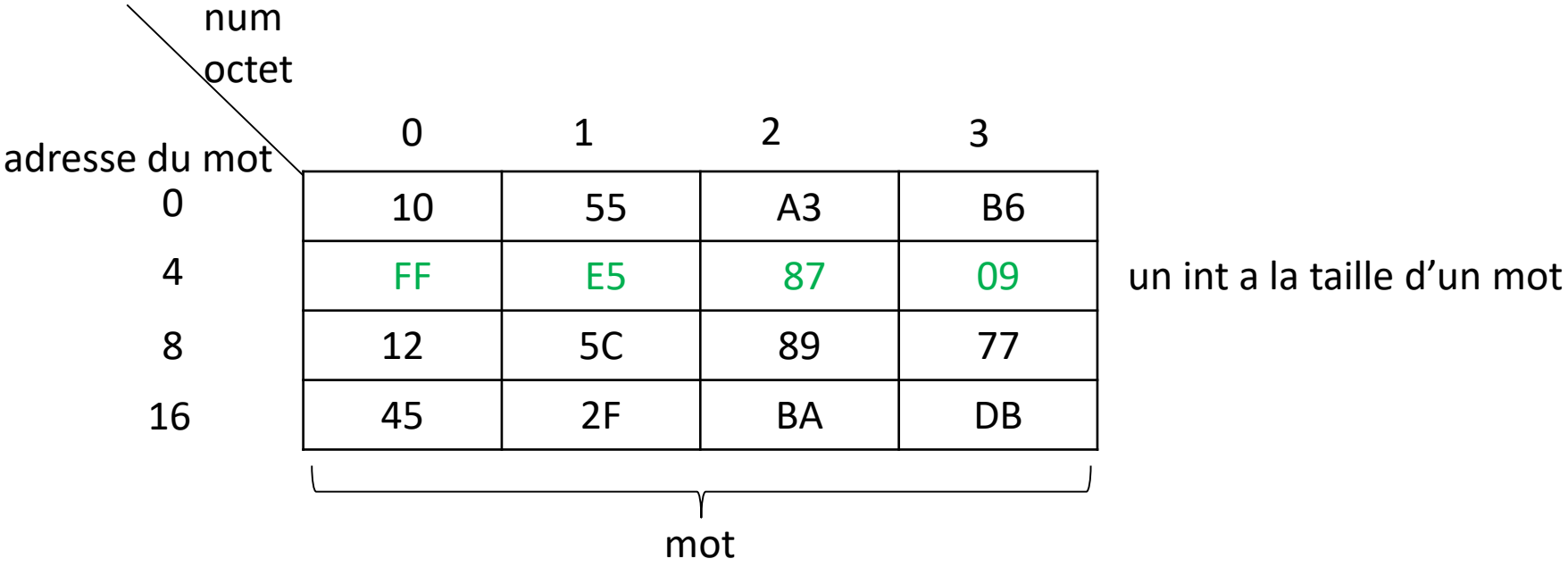
- Conséquence : **un mot est toujours à une adresse multiple de 4.**
- Une variable ne doit pas être stockée sur deux mots différents:



Notion de mot et alignement

- Conséquence : **un mot est toujours à une adresse multiple de 4.**
- Une variable ne doit pas être stockée sur deux mots différents:

- Rappel : un int est sur 4 octets:



Notion de mot et alignement

- Conséquence : **un mot est toujours à une adresse multiple de 4.**
- Une variable ne doit pas être stockée sur deux mots différents:

- Rappel : un double est sur 8 octets:

adresse du mot \ num octet	0	1	2	3
0	10	55	A3	B6
4	FF	E5	87	09
8	12	5C	89	77
16	45	2F	BA	DB

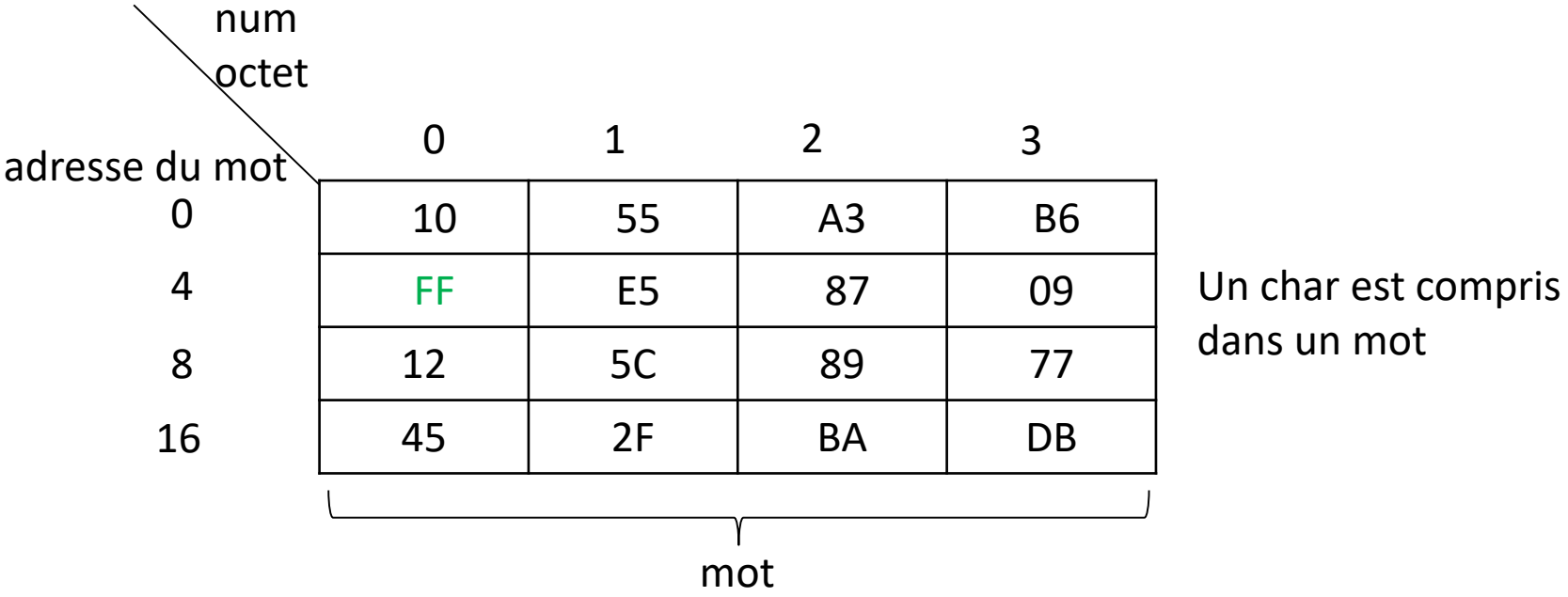
mot

Un double a la taille de deux mots

Notion de mot et alignement

- Conséquence : **un mot est toujours à une adresse multiple de 4.**
- Une variable ne doit pas être stockée sur deux mots différents:

- Rappel : un char est sur 1 octets:



Notion de mot et alignement

- Conséquence : **un mot est toujours à une adresse multiple de 4.**
- Une variable ne doit pas être stockée sur deux mots différents:

- Rappel : un int est sur 4 octets:

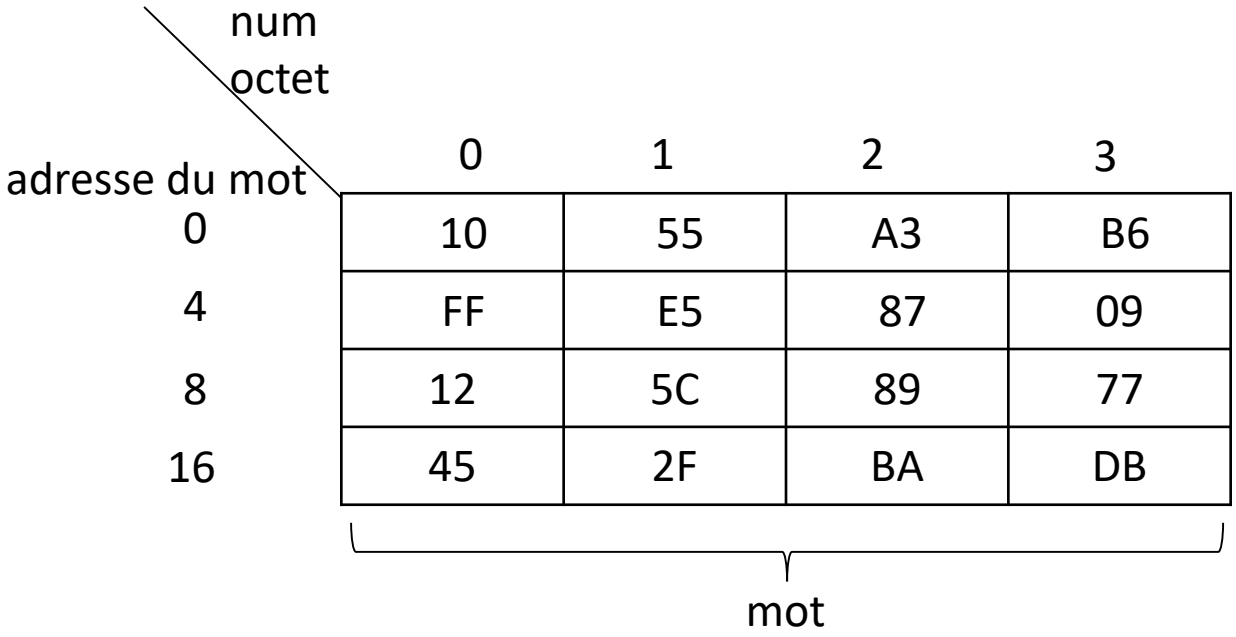
adresse du mot \ num octet	0	1	2	3
0	10	55	A3	B6
4	FF	E5	87	09
8	12	5C	89	77
16	45	2F	BA	DB

mot

Une même variable ne doit pas être à cheval sur deux mots !

Notion de mot et alignement

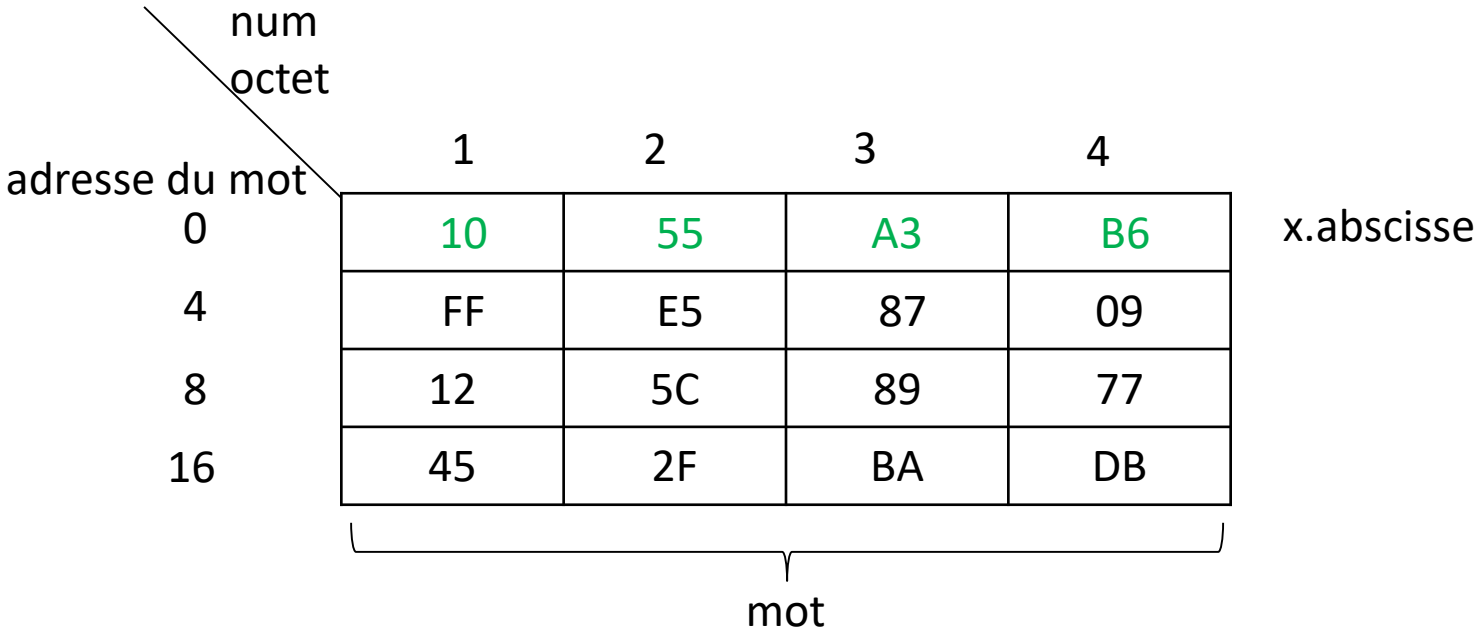
- Retour sur les structures :




```
typedef struct {
      float abscisse;
      float ordonnee;
} Point;

Point x;
```

Notion de mot et alignement

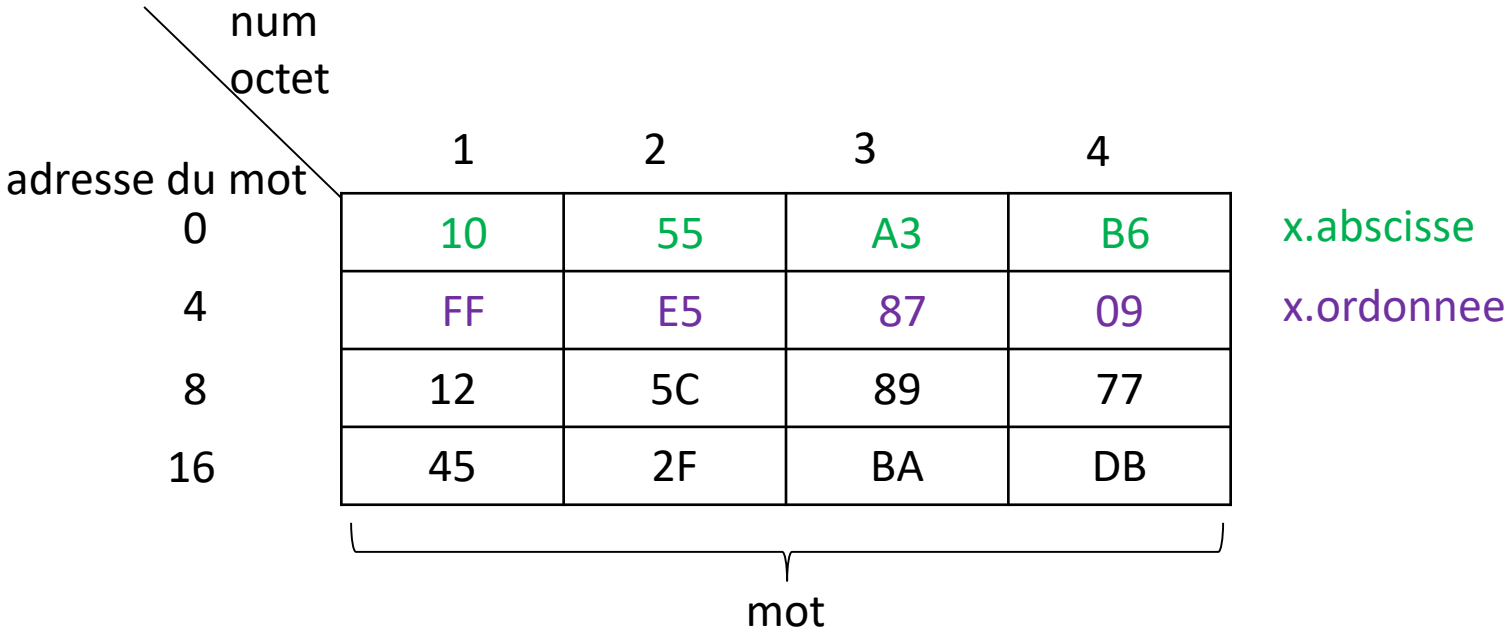
- Retour sur les structures :




```
typedef struct {
      float abscisse;
      float ordonnee;
} Point;

Point x;
```

Notion de mot et alignement

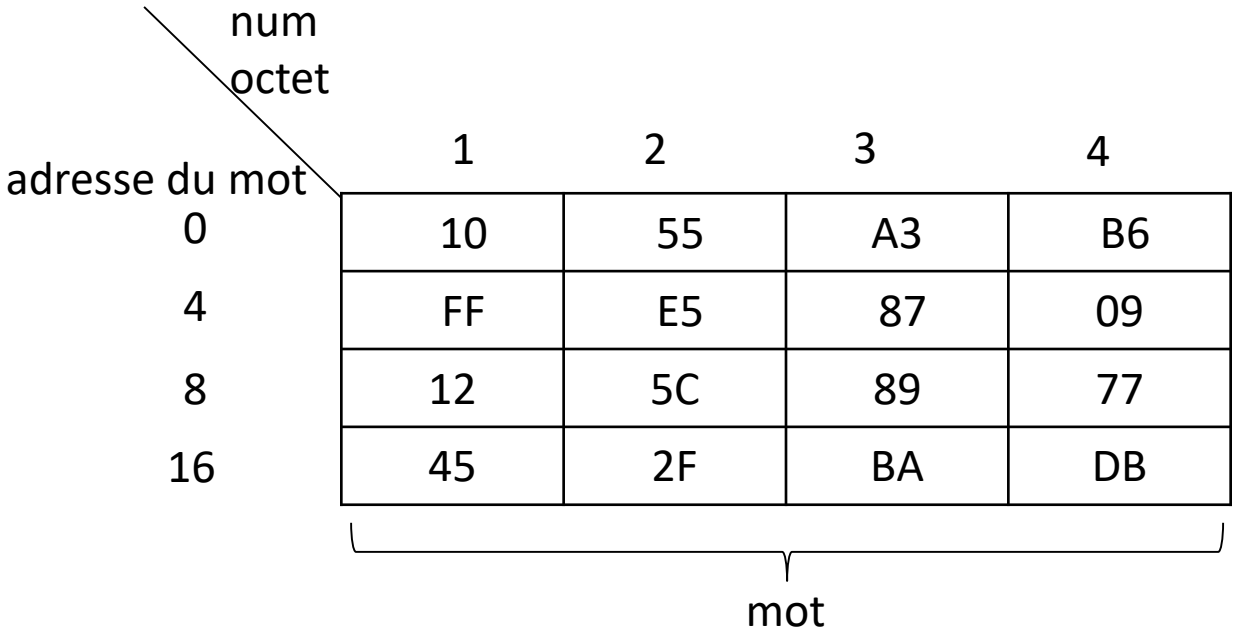
- Retour sur les structures :




```
typedef struct {
      float abscisse;
      float ordonnee;
} Point;

Point x;
```

Notion de mot et alignement

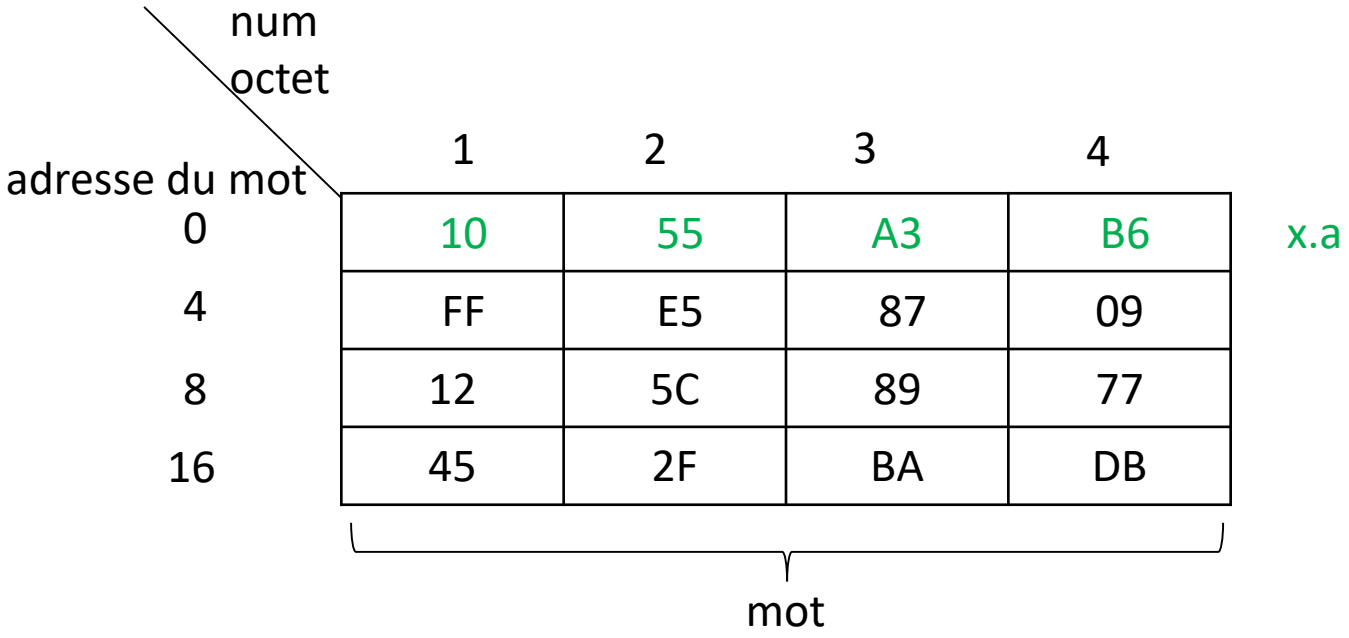
- Retour sur les structures :




```
typedef struct {
    int a;
    char b;
    int c;
} Test;

Test x;
```

Notion de mot et alignement

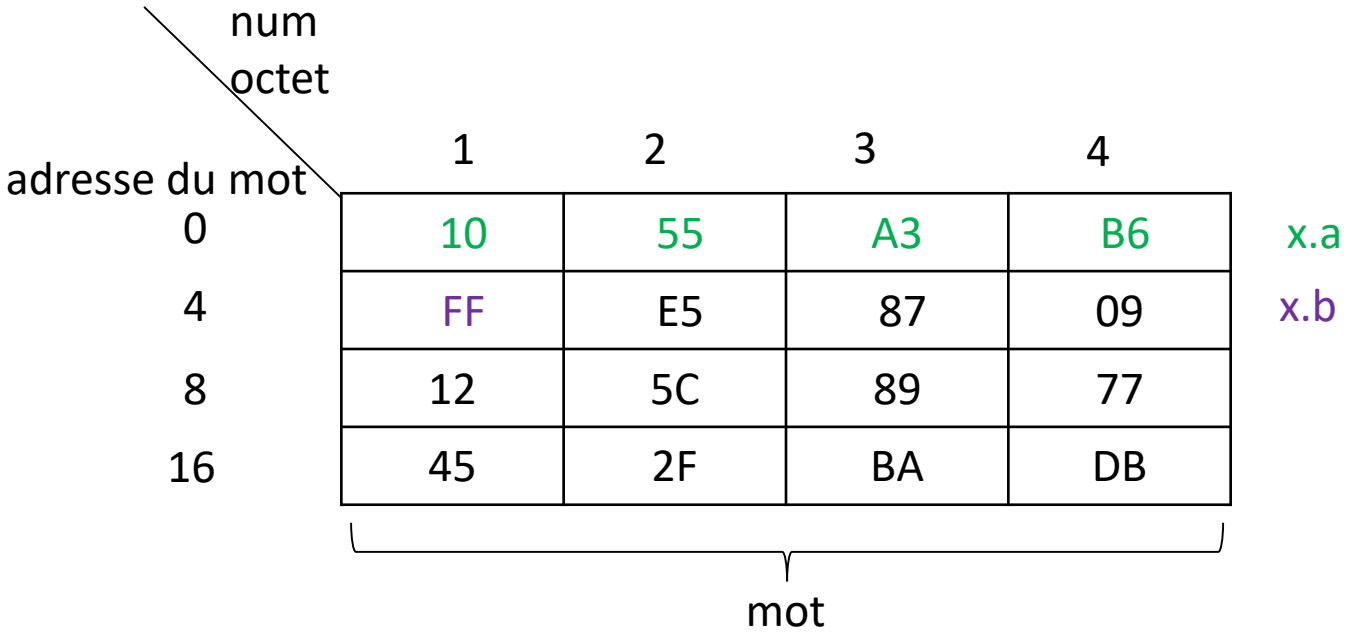
- Retour sur les structures :




```
typedef struct {
    int a;
    char b;
    int c;
} Test;

Test x;
```

Notion de mot et alignement

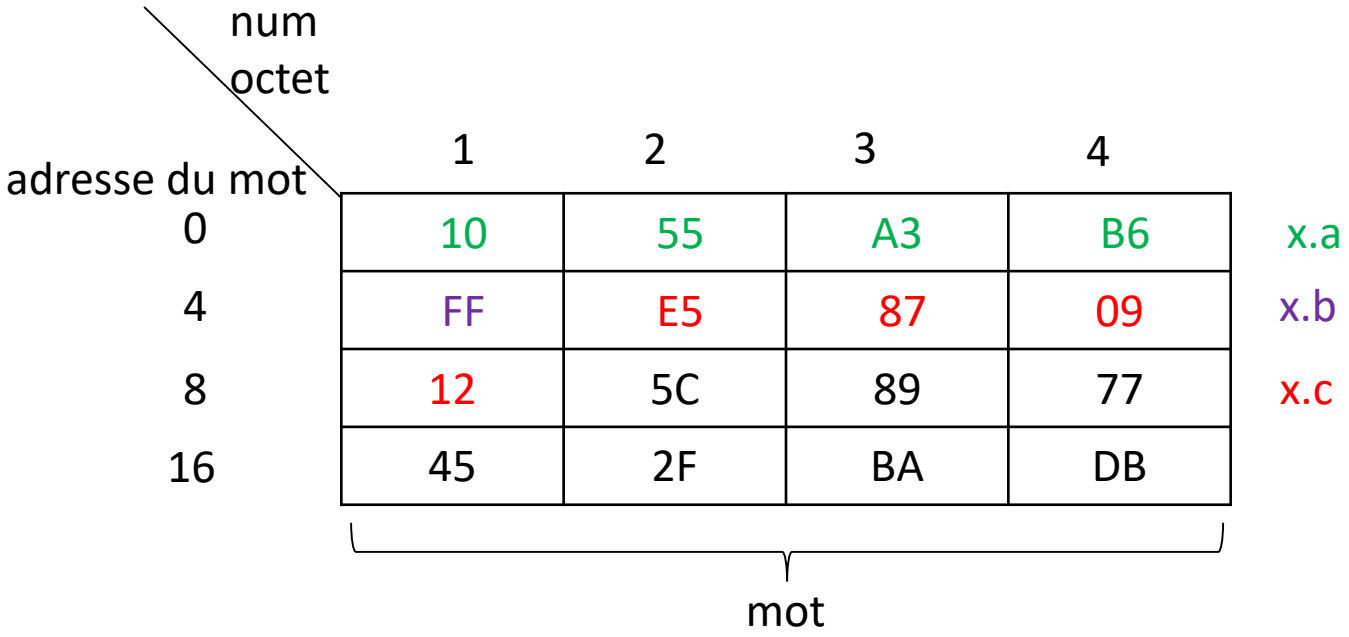
- Retour sur les structures :




```
typedef struct {
    int a;
    char b;
    int c;
} Test;

Test x;
```

Notion de mot et alignement

- Retour sur les structures :

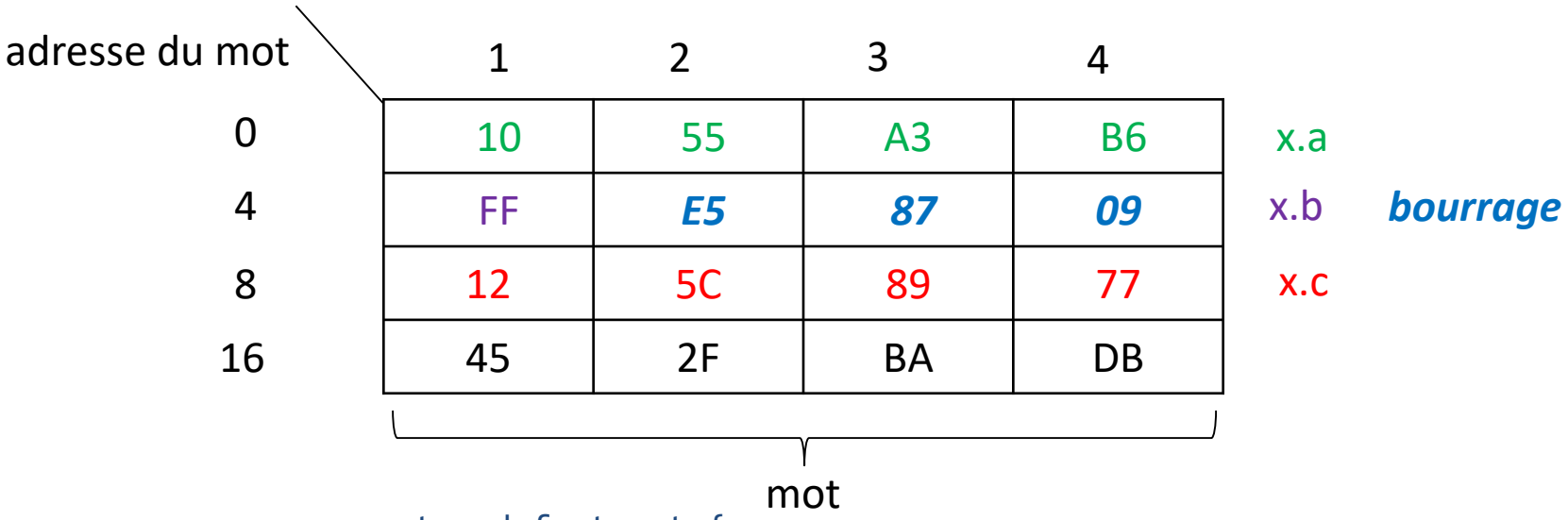



```
typedef struct {
    int a;
    char b;
    int c;
} Test;

Test x;
```

x.c est sur deux mots différents!

Notion de mot et alignement

- Pour éviter le problème de champs d'une occurrence d'une structure qui peut être sur plusieurs mots différents, les compilateurs ajoutent du **bourrage** (padding): des octets non utilisés.




```
typedef struct {
                int a;
                char b;
                int c;
                }Test;
Test x;
```

Notion de mot et alignement

- Pour éviter le problème de champs d'une occurrence d'une structure qui peut être sur plusieurs mots différents, les compilateurs ajoutent du **bourrage** (padding): des octets non utilisés.

- Le compilateur sait combien d'octets de bourrage il doit rajouter car il est nécessaire que l'adresse mémoire d'un type de variable soit multiple de sa taille. (Ex; l'adresse d'un int doit être un multiple de 4).

- C'est la **contrainte d'alignement**.

- Il est possible de connaître les contraintes d'alignement d'un type avec **_Alignof()**

- Les valeurs d'alignement dépendent de la machine.

Notion de mot et alignement

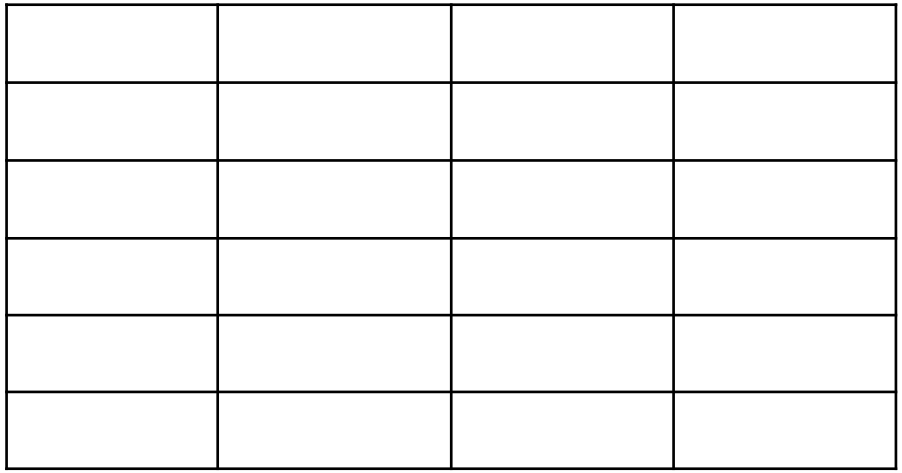
```c
typedef struct{
  int a;
  char b;
  int c;
}Test;

int main(){
    printf("alignement d'un char %ld\n",
_Alignof(char));
  printf("alignement d'un int %ld\n\n",
_Alignof(int));
  printf("Taille de Test : %ld\n",sizeof(Test));
  printf("alignement de Test %ld\n",
_Alignof(Test));
    return 0;
}
```

```
alignement d'un char 1
alignement d'un int 4

Taille de Test : 12
alignement de Test 4
```

Notion de mot et alignement

- Autre exemple : taille de la structure suivante?

```
typedef struct{
  int a;
  char b;
  float c;
  char d;
}Test2;
```



Notion de mot et alignement

- Autre exemple : taille de la structure suivante?

```
typedef struct{
  int a;
  char b;
  float c;
  char d;
}Test2;
```

a	a	a	a
b	/	/	/
c	c	c	c
d			

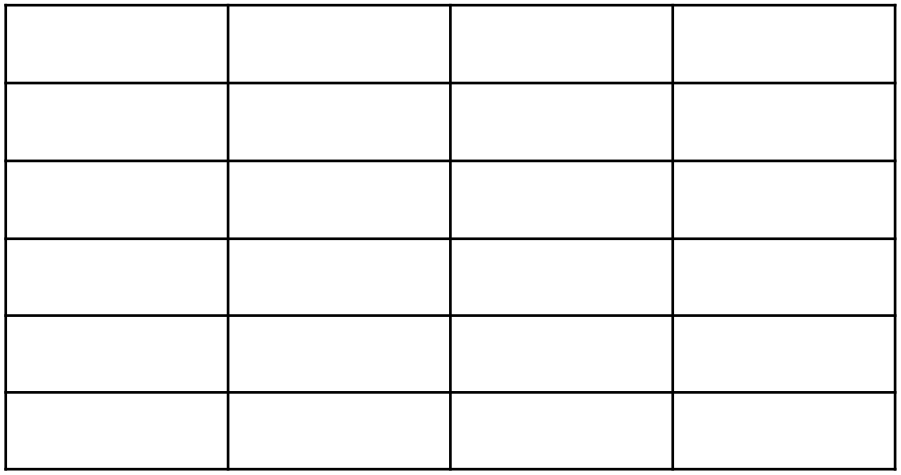
bourrage

La structure fait 13 octets

Notion de mot et alignement

- Autre exemple : taille de la structure suivante?

```
typedef struct{
  double a;
  char b[3];
  float c;
}Test3;
```



Notion de mot et alignement

- Autre exemple : taille de la structure suivante?

```
typedef struct{
  double a;
  char b[3];
  float c;
}Test3;
```

a	a	a	a
a	a	a	a
b[0]	b[1]	b[2]	/
c	c	c	c

bourrage

La structure fait 16 octets

Pointeur sur structure

- Il est possible de définir un **pointeur vers l'occurrence d'une structure**: le pointeur aura donc le type de cette structure et indiquera l'adresse de la première case de l'occurrence.

```
Type_structure* nomPointeur;
```

Pointeur sur structure

- Il est possible de définir un **pointeur vers l'occurrence d'une structure**: le pointeur aura donc le type de cette structure et indiquera l'adresse de la première case de l'occurrence.

```
Type_structure* nomPointeur;
```

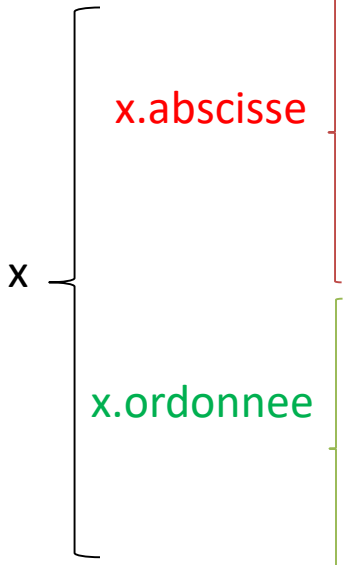
- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point  x;
    Point* px;
    px = &x;
}
```

Pointeur sur structure

- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point  x;
    Point* px;
    px = &x;
}
```



	Adresse	Valeur
	…	
	1154	0000 0000
	1155	0000 0000
	1156	0000 0000
	1157	0000 0001
	1158	0000 0000
	1159	0000 0000
	1160	0001 0001
	1161	1001 0100
	…	
px	20 000	1154

Pointeur sur structure

- Il est possible de définir un **pointeur vers l'occurrence d'une structure**: le pointeur aura donc le type de cette structure et indiquera l'adresse de la première case de l'occurrence.

```
Type_structure* nomPointeur;
```

- Pour accéder aux différents champs :

```
(*nomPointeur).nom_champ
```

- On peut également écrire :

```
nomPointeur->nom_champ
```

Pointeur sur structure

- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point x;
    Point* px;
    px = &x;
    x.abscisse = 5,5;
    px->ordonnee = 2;
    px->abscisse = px->abscisse + x.ordonnee;
    return 0;
}
```

x

x.abscisse

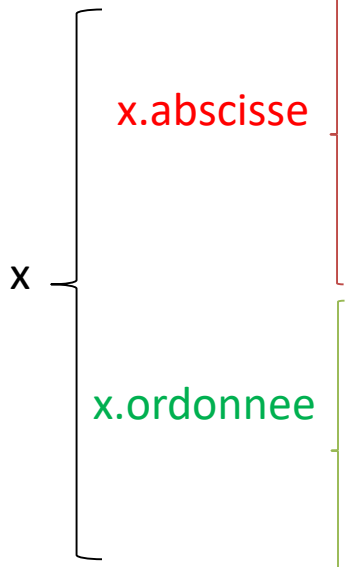
x.ordonnee

Adresse	Valeur
…	
1154	
1155	
1156	
1157	
1158	
1159	
1160	
1161	
…	
px 20 000	1154

Pointeur sur structure

- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point x;
    Point* px;
    px = &x;
    x.abscisse = 5,5;
    px->ordonnee = 2;
    px->abscisse = px->abscisse + x.ordonnee;
    return 0;
}
```

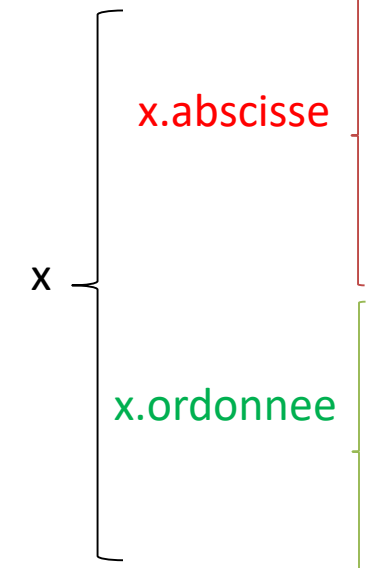


Adresse	Valeur
…	
1154	5,5
1155	
1156	
1157	
1158	
1159	
1160	
1161	
…	
px 20 000	1154

Pointeur sur structure

- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point x;
    Point* px;
    px = &x;
    x.abscisse = 5,5;
    px->ordonnee = 2;
    px->abscisse = px->abscisse + x.ordonnee;
    return 0;
}
```

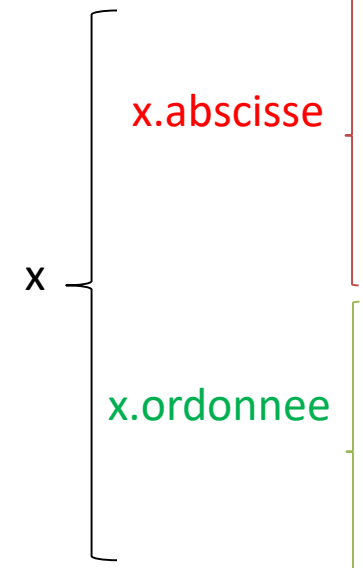


Adresse	Valeur
…	
1154	5,5
1155	
1156	
1157	
1158	2
1159	
1160	
1161	
…	
px 20 000	1154

Pointeur sur structure

- Exemple

```
typedef struct {
    float abscisse;
    float ordonnee;
} Point;

int main(){
    Point x;
    Point* px;
    px = &x;
    x.abscisse = 5,5;
    px->ordonnee = 2;
    px->abscisse = px->abscisse + x.ordonnee;
    return 0;
}
```



Adresse	Valeur
…	
1154	7,5
1155	
1156	
1157	
1158	2
1159	
1160	
1161	
…	
px 20 000	1154

Tableaux statiques de structures

- On peut définir des tableaux de structures :

```
Type_structure nom_Tableau[taille];
```

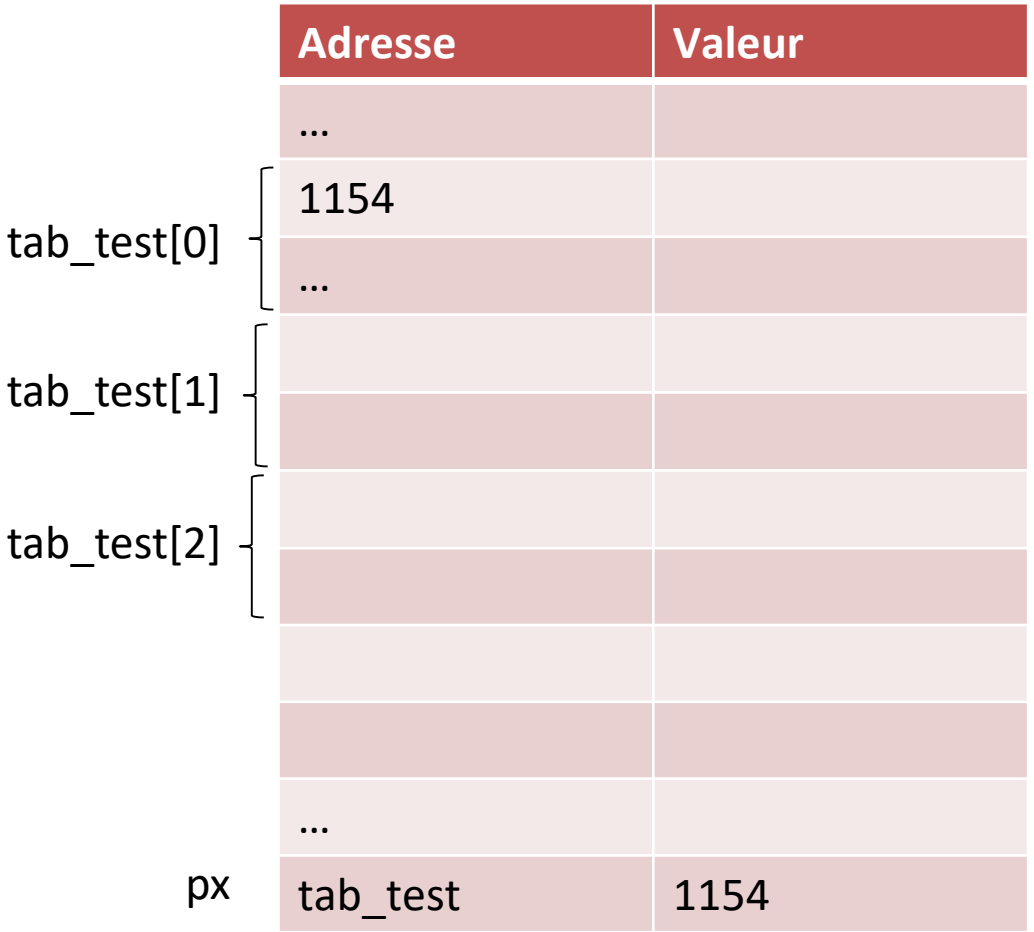
- Comme pour les tableaux classiques, les différents éléments (ici des occurrences d'une structure) seront à la suite dans la mémoire.

- Un pointeur constant est créé et initialisé avec l'adresse de la première case du tableau. Ce pointeur pointe donc sur la première occurrence du type structuré.

Pointeur sur structure

- Exemple

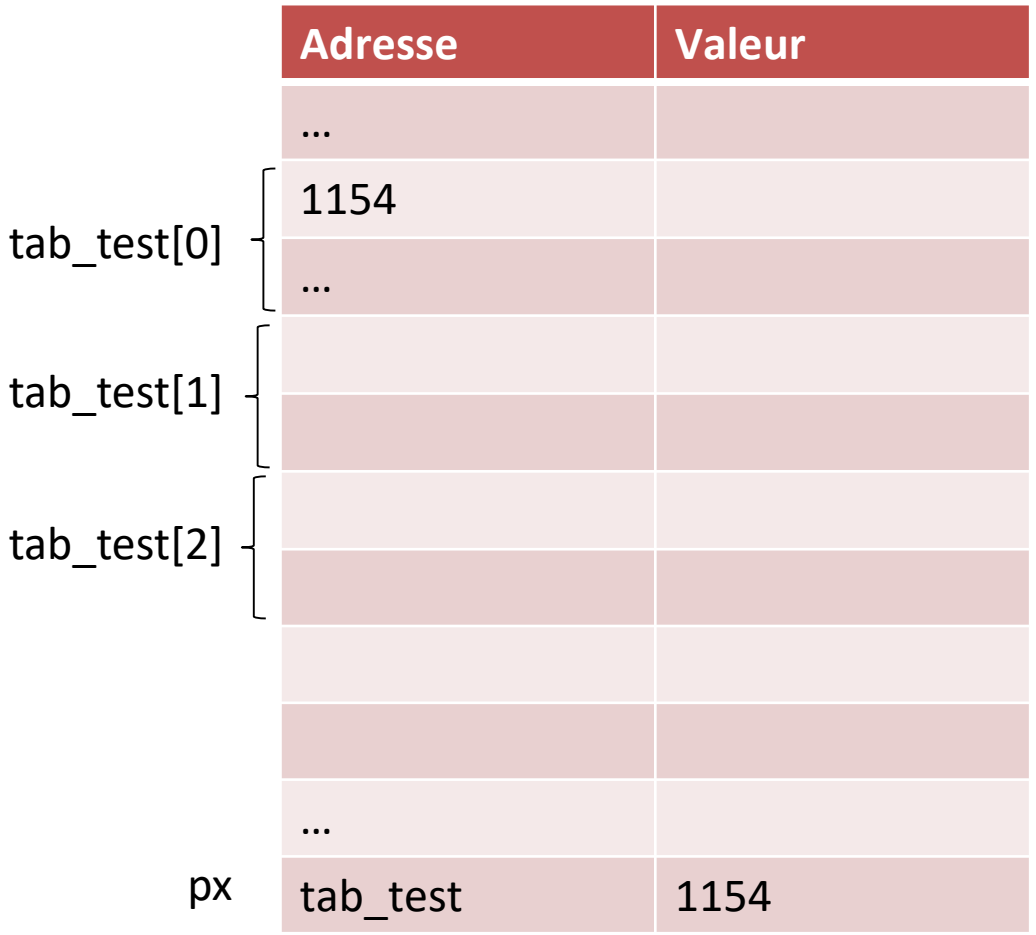
```
typedef struct {
    int a;
    char b;
    float c;
} Test;

int main(){
    // tableau de 10 structures Test
    Test tab_test[10];
    return 0;
}
```



Pointeur sur structure

- Exemple

```
typedef struct {
    int a;
    char b;
    float c;
} Test;

int main(){
    // tableau de 10 structures Test
    Test tab_test[10];
    return 0;
}
```



Question :

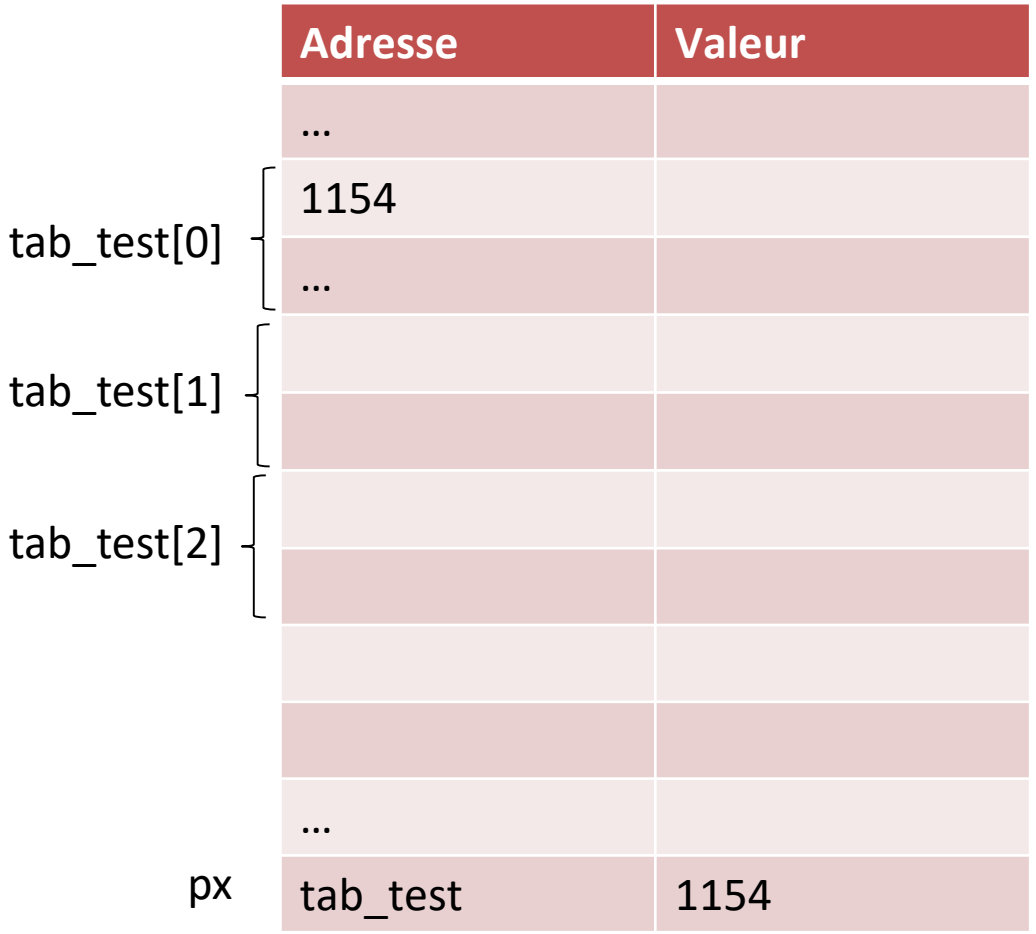
Comment accéder au champ b du 3 ème élément du tableau?

En format pointeur?

Pointeur sur structure

- Exemple

```
typedef struct {
    int a;
    char b;
    float c;
} Test;

int main(){
    // tableau de 10 structures Test
    Test tab_test[10];
    return 0;
}
```



Question :

Comment accéder au champ b du 3 ème élément du tableau?

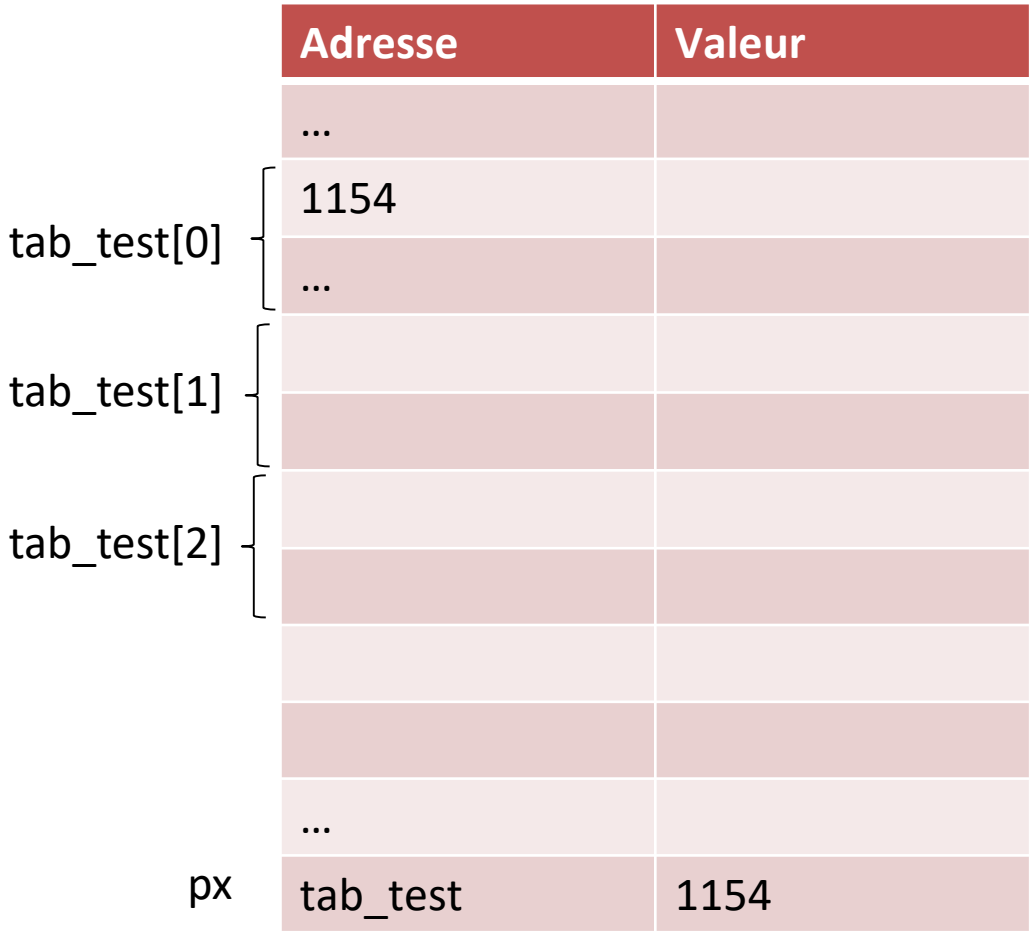
`tab_test[2].b`

En format pointeur ? `(*(tab_test+2)).b` ⇔ `(tab_test+2)->b`

Pointeur sur structure

- Exemple

```
typedef struct {
    int a;
    char b;
    float c;
} Test;

int main(){
    // tableau de 10 structures Test
    Test tab_test[10];
    return 0;
}
```



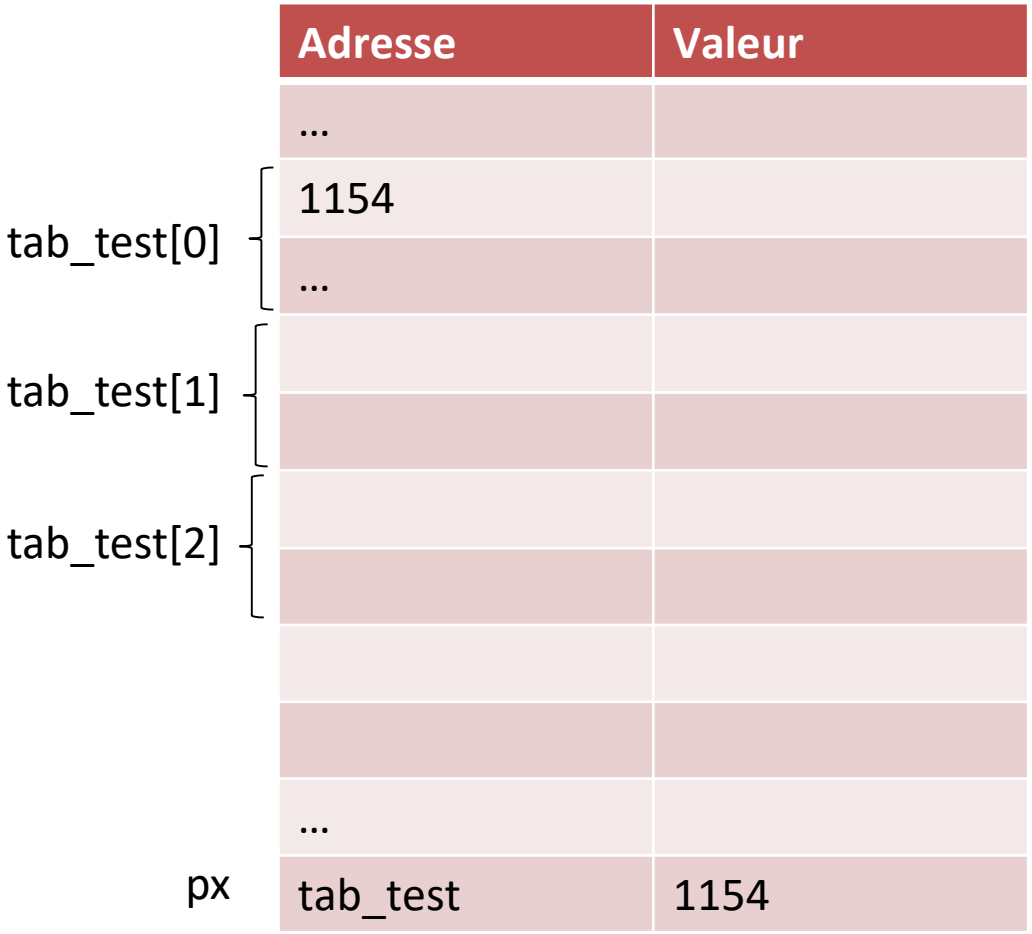
Question :

De combien d'octets se déplace-t-on en mémoire lorsque l'on fait tab_test+4 ?

Pointeur sur structure

- Exemple

```
typedef struct {
    int a;
    char b;
    float c;
} Test;

int main(){
    // tableau de 10 structures Test
    Test tab_test[10];
    return 0;
}
```



Question :

De combien d'octets se déplace-t-on en mémoire lorsque l'on fait tab_test+4 ?
On se déplace de 4*sizeof(Test), soit de 4*12=48 octets.

Tableaux statiques de structures

- On peut définir des tableaux de structures :

```
Type_structure nom_Tableau[taille];
```

- Comme pour les tableaux classiques, les différents éléments (ici des occurrences d'une structure) seront à la suite dans la mémoire.

- Un pointeur constant est créé et initialisé avec l'adresse de la première case du tableau. Ce pointeur pointe donc sur la première occurrence du type structuré.

- Pour accéder aux différents champs :

```
nom_Tableau[ind].nom_champ
⇕
*(nom_tableau+ind).nom_champ
⇕
(nom_tableau+ind)->nom_champ
```

Tableaux dynamiques de structures

- Pour allouer en mémoire un tableau de structures il faut connaître la taille totale de ce tableau. Il faut donc connaître:
  - 1. Le nombre d'elements à stocker
  - 2. La taille de la structure

- Les commandes pour allour dynamiquement un tableau de structure est donc :

```
Type_structure* tab = NULL;
int nb_elements = 7;
tab = malloc( nb_elements * sizeof(Type_structure) );
```

- Le tableau peut ensuite être utilisé comme un tableau statique

- Toujours vérifier que l'allocation mémoire a bien eu lieu.

- Ne pas oublier de libérer (free) l'espace mémoire alloué !

Tableaux dynamiques de structure

- Exemple

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

typedef struct{
    char nom[50];
    char prenom[50];
    int num;
    int groupe;
    int note[10];
} Etudiant;
```

```c
int main(){
    int nb_etu, i;
    Etudiant* tab_etu = NULL;

    printf("Nb étudiants dans la classe?\n");
    scanf("%d", &nb_etu);
    tab_etu = malloc( nb_etu*sizeof(Etudiant) );
    if(tab_etu == NULL){
        printf("Problème allocation mémoire! \n ");
        exit (1);
    }
    printf("Quel est le nom du : \n");
    for (i=0; i<nb_etu; i++){
        printf("  > %d ieme eleve?\n", i+1);
        scanf("%s", tab_etu[i].nom);
    }
    return 0;
}
```

Comment modifier la 4eme note du 5eme étudiant?

`tab_etu[4].note[3]` ⇔ `(tab_etu+4)->note[3]` ⇔ `(tab_etu+4)->*(note+3)`

Recherche et parcours d'un tableau de structures

- La recherche dans un tableau de structure se fait généralement en recherchant des valeurs de champs (ex: quels étudiants sont dans le groupe 4?).

- Si l'on souhaite faire des opérations sur les éléments recherchés, il existe plusieurs approches :

    - 1. On sauvegarde chaque occurrences recherchées puis on leur applique des opérations
    - 2. On effectue des opérations sur les occurrences recherchées à mesure que l'on parcours le tableau

Recherche et parcours d'un tableau de structures

- La recherche et le traitement de donnée se bases donc sur ces deux principe. On va choisir la méthode selon notre priorité:
  - 1.On parcours une fois le tableau et on sauvegarde les occurrences recherchée.

    -> Un seul parcours => rapidité

    -> sauvegarde de donnée => utilisation d'espace

    On peut optimiser le parcours et l'espace utilisé en travaillant sur un tableau trié, mais le tri prend beaucoup de temps !

  - 2. On parcourt une fois l'ensemble des données pour chaque information à récupérer. Celles-ci n'ont donc pas besoin d'être retenues, mais le temps d'exécution peut être beaucoup plus long.

    ->pas de sauvegarde : minimum d'espace utilisé

    -> plusieurs parcours : algorithme lent

Recherche et parcours d'un tableau de structures

- Exemple:

On stocke dans une structure les chiffres d'affaires de commerciaux d'une entreprise.

```
STRUCTURE Commercial
        nom : chaine de caractère
        region : Entier        // 0=Est ; 1=Nord ; 2=Ouest ; 3=Sud
        ca : Entier    // chiffre d'affaires
FIN STRUCTURE

   Tcom : Tableau de taille n de Commercial
```

- Objectif: afficher le chiffre d'affaire par région.

Recherche et parcours d'un tableau de structures

- Exemple:

Indice	Nom	Région	Chiffre d'Affaires
0	Néo	Nord	500
1	Yoda	Sud	2500
2	Cobb	Est	1000
3	Ripley	Ouest	4000
4	McFly	Nord	900
5	Spock	Sud	600
6	Lilou	Est	2000
7	Atréide	Ouest	1500
8	Neytiri	Nord	3000
9	Connor	Sud	800

```
CA de la région Est   : 3000
CA de la région Nord  : 4400
CA de la région Ouest : 5500
CA de la région Sud   : 3900
```

Recherche et parcours d'un tableau de structures

- Algorithme 1:
  - On trie au préalable le tableau selon le champs recherchée (ici la région)
  - On parcourt le tableau et on affiche le résultat à chaque changement de région.

```
PROCEDURE ALGO1(Tcom: tableau de taille n de Commercial)
VARIABLE
    somme, reg, i: ENTIER
DEBUT
    Tri(Tcom, region) // On trie le tableau avant par rapport à la région
    somme <- Tcom[0].ca // On initialise la somme et la région à la valeur de la 1ere case
    reg <- Tcom[0].region //1 ere region
    POUR i DE 1 A N FAIRE
        SI Tcom[i].region = reg ALORS
            somme <- somme + Tcom[i].ca
        SINON         // Changement de région on affiche la précédente
            ECRIRE ("CA de la Région" + reg + " : " +somme)
            somme <- Tcom[i].ca // On réinitialise la somme et la région
            reg <- Tcom[i].region
        FIN SI
    FIN POUR
    ECRIRE ("CA de la Région" + reg + " : " +somme)  // pour la dernière région
FIN
```

  - Avantage: 1 seul parcours, pas d'espace mémoire supplémentaire
  - Inconvénient : il faut trier le tableau
  - Complexite : O(n+n*log(n))

Recherche et parcours d'un tableau de structures

- Algorithme 2:
  - On fait un parcours du tableau par régions

```
PROCEDURE ALGO2(Tcom: tableau de taille n de Commercial)
VARIABLE
    somme, reg, i: ENTIER
DEBUT
    POUR reg DE 0 A NB_REGION FAIRE  // parcours d'une région
        somme <- 0
        POUR i DE 0 A n FAIRE // parcours du tableau entier
            SI Tcom[i].region = reg ALORS // somme si c'est la bonne région
                somme <- somme + Tcom[i].ca
            FIN SI
        FIN POUR
        ECRIRE ("CA de la Région" + reg + " : " +somme)
    FIN POUR
FIN
```

  - Avantage: pas d'espace mémoire supplémentaire
  - Inconvénient : Temps dépendant du nombre de région.
  - Complexité : O(n*NB_REGION)

Recherche et parcours d'un tableau de structures

- Algorithme 3:
  - Un compromis: on sauvegarde les différentes sommes dans un tableau -> un seul parcours.

```
PROCEDURE ALGO3(Tcom: tableau de taille n de Commercial)
VARIABLE
    reg, i: ENTIER
    somme : tableau de taille NB_REGION d'ENTIER
DEBUT
    POUR i DE 0 A NB_REGION FAIRE  // initialiser le tableau resultat
        somme[i]<-0-0
    FIN POUR
    POUR i DE 0 A n FAIRE E // parcours du tableau entier
        somme[Tcom[i].region] = somme[Tcom[i].region] + Tcom[i].ca
    FIN POUR
    POUR i DE 0 A NB_REGION FAIRE
        // Affichage des resultats
        ECRIRE ("CA de la Région" + i + " : " +somme[i])
    FIN POUR
FIN
```

  - Avantage: Un seul parcours de tableau
  - Inconvénient : 1 tableau supplémentaire
  - Complexité : O(n+2*NB_REGION)

Conclusion et résumé

- Les structures permettent de créer de nouveaux types en combinant des types existants.

- Les différents champs d'une structure sont à la suite dans la mémoire. Le compilateur peut ajouter du **bourrage** dans la mémoire afin de conserver un **alignement** correct en mémoire. Un ordre des champs défini par l'utilisateur (avec la connaissance de l'alignement mémoire) permet de gagner de l'espace mémoire.

- On peut créer des pointeurs sur structures: l'accès aux différents champs se faite avec l'opérateur **->**

- La gestion d'un tableau de structures se fait de la même façon que pour les tableaux de types classiques. L'arithmétique des pointeurs est la même.

- Les opérations de recherches et d'opérations sur ces tableaux nécessitent de trouver un compromis entre rapidité et espace mémoire selon la situation (mais ce compromis existe dans tous les aspects du développement logiciel).