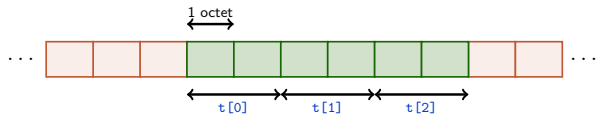
Alignement en mémoire

L'alignement en mémoire d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

Par exemple, les **tableaux** de taille `n` d'éléments d'un type `T` sont organisés en un segment contigu de `sizeof(T) * n` octets.

Ainsi, un tableau `t` de `3` éléments de type `short` est organisé en



On peut se poser de la même manière la question de l'**alignement mémoire** des variables d'un **type structuré**.

Alignement en mémoire des variables d'un type structuré

Considérons les déclarations de types

```
1  typedef struct {
2      short x;
3      short y;
4      int z;
5  } A;
```

```
6  typedef struct {
7      short x;
8      int z;
9      short y;
10 } B;
```

`A` et `B` sont des types structurés composés des mêmes champs. Il n'y a que l'ordre de leur déclaration qui diffère.

Cependant,

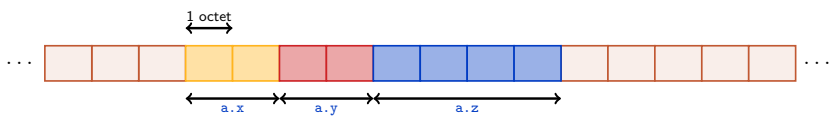
```
1      printf("%lu␣%lu\n", sizeof(A), sizeof(B));
```

affiche `8 12`.

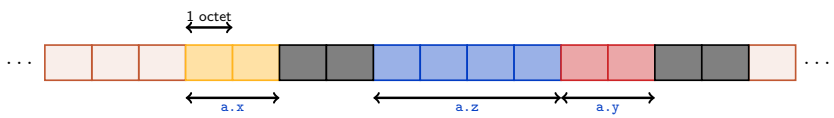
Le fait que les tailles des variables de type `A` et `B` diffèrent est dû à leur **alignement en mémoire**.

Alignement en mémoire des variables d'un type structuré

Soit `a` une variable de type `A`. Cette variable est organisée en mémoire en



Soit `b` une variable de type `B`. Cette variable est organisée en mémoire en

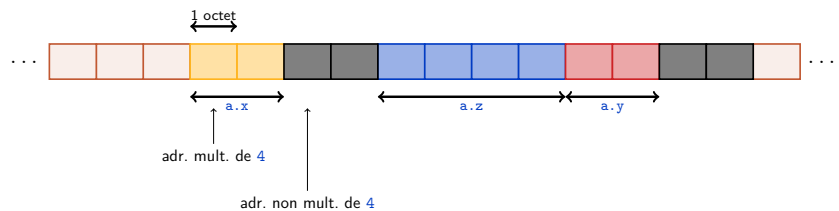


Les octets en gris intervenant dans l'alignement mémoire de `b` sont des octets de complétion.

Octets de complétion

Des octets de complétion sont introduits pour que chaque champ `c` d'une variable d'un type structuré commence à une adresse multiple d'un entier dépendant du type de `c`.

Dans notre exemple, en sachant que tout champ de type `short` (resp. `int`) doit commencer à une adresse multiple de `2` (resp. `4`), on explique l'alignement en mémoire de `b` précédent :



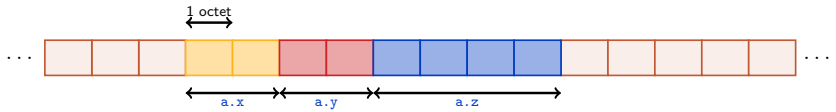
Les derniers octets de complétion sont introduits pour que les tableaux de variables de type `B` puissent être représentés en vérifiant cet alignement en mémoire.

Accès manuel aux champs

Soit `a` une variable de type `A` initialisée par

```
A a = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `a` de la manière suivante :

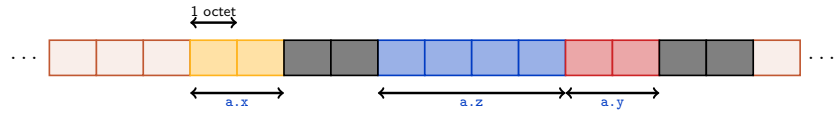
```
short x, y;
int z;
void *p;
p = &a;
x = *((short *) p); /* Equivalent a x = a.x; */
p += 2;
y = *((short *) p); /* Equivalent a y = a.y; */
p += 2;
z = *((int *) p); /* Equivalent a z = a.z; */
```


Accès manuel aux champs

Soit `b` une variable de type `B` initialisée par

```
B b = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `b` de la manière suivante :

```
short x, y;
int z;
void *p;
p = &b;
x = *((short *) p); /* Equivalent a x = a.x; */
p += 4;
z = *((int *) p); /* Equivalent a z = a.z; */
p += 4;
y = *((short *) p); /* Equivalent a y = a.y; */
```


L'option `Wpadded`

L'option du compilateur `-Wpadded` permet d'obtenir un avertissement sanctionnant la déclaration d'un type structuré nécessitant des octets de complétion.

Par exemple, avec le type structuré `B` défini par

```
typedef struct {
    short x;
    int z;
    short y;
} B;
```

on obtient l'avertissement

```
Prog.c:3:9: warning: padding struct to align ‘z’ [-Wpadded]
     int z;
         ^
Prog.c:5:1: warning: padding struct size to alignment boundary [-Wpadded]
 } B;
 ^
```


Alignement en mémoire — ce qu'il faut retenir

L'alignement mémoire d'une variable d'un type structuré dépend de beaucoup de paramètres, notamment :

- de l'architecture de la machine exécutant ou ayant compilé le programme ;
- du compilateur ayant compilé le programme.

Il est donc important de savoir que le calcul de la **taille** d'une variable d'un **type structuré** n'est pas immédiat et **dépend du contexte**.