Plan

7. Types

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
`char`	1	-128 à 127
`short`	2	-32768 à 32767
`int`	4	-2^{31} à $2^{31}-1$
`long`	8	-2^{63} à $2^{63}-1$

Chacun de ces types peut être précédé de `unsigned` pour faire en sorte de ne représenter que des entiers positifs. On a ainsi les plages suivantes :

Nom	Plage
`unsigned char`	0 à 255
`unsigned short`	0 à 65535
`unsigned int`	0 à $2^{32}-1$
`unsigned long`	0 à $2^{64}-1$

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs **entières positives**, on utilisera les **versions non signées** des types entiers.

Quelques avantages de ce procédé :

1. possibilité de représenter des entiers plus grands ;
2. gain de lisibilité du programme.

Attention : un entier non signé **ne peut jamais devenir négatif**. Ceci peut poser problème pour les conditions de sortie au sein des boucles.

– Exemple –

```
unsigned int i;

for (i = 8 ; i >= 0 ; --i) {
    ...
}
```

Ces instructions produisent une boucle infinie. En effet, `i` étant non signé, il est toujours positif et donc la condition `i >= 0` est toujours vraie.

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- en base dix : `0`, `29`, `-322`, ...
- en octal : `01`, `0145`, `-01234567`, ...
- en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Un entier peut être représenté par un caractère car tout caractère est représenté par son code ASCII (qui est un entier compris entre `0` et `127`).

Attention : ne pas confondre les caractères chiffres avec les entiers (l'entier `'1'` a pour valeur son code ASCII qui est `49` et ne vaut donc pas `1`).

Types flottant

On se place sur une machine 64 bits.

Nom	Taille (octets)	Valeur absolue maximale
float	4	3.40282×10^{38}
double	8	1.79769×10^{308}
long double	16	1.18973×10^{4932}

Le fichier d'en-tête `float.h` contient des constantes donnant d'autres renseignements sur les types flottant.

Danger des types flottant

Les nombres flottants sont représentés de manière **approchée**.

– Exemples –

```
float x = 10000001.0;
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue, `10000001.000000`.

```
float x = 100000001.0;
printf("%f\n", x);
```

En revanche, ces instructions affichent, de manière inattendue, `100000000.000000`.

Comme ces exemples le montrent, même certains entiers, représentables de manière exacte par des types entier, ne le sont pas par des types flottant.

Un phénomène similaire apparait lors de l'addition du flottant `0.1` avec le flottant `0.2` : ce n'est pas exactement le nombre `0.3` qui est obtenu.

Danger des types flottant : une solution partielle

Les types flottants présentent divers désavantages par rapport aux types entiers :

1. **représentation non exacte** des nombres ;
2. opérations arithmétiques beaucoup **moins efficaces**.

Pour ces raisons, il est recommandé d'utiliser des types flottants que si cela est vraiment nécessaire.

Une alternative, quand cela est possible, consister à représenter par l'entier $x \times 10^k$ tout nombre x qui dispose de $k \geqslant 0$ chiffres (en base dix) après la virgule, k étant fixé.

– Exemple –

Si l'on a besoin de manipuler des nombres à $k := 2$ chiffres après la virgule, les nombres 0.15 et 331.9 sont respectivement représentés par les entiers `15` et `33190`.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un **ensemble totalement ordonné** : étant données deux valeurs, il est toujours possible de les comparer. On utilise pour cela les opérateurs relationnels

`==`, `!=`, `<=`, `>=`, `<`, `>`.

Il est possible de mélanger des comparaisons de valeurs de types entier et de types flottant. Dans ce cas, les entiers sont convertis implicitement en une valeur de type flottant avant d'effectuer la comparaison.

Sur des variables de type scalaire sont définis les opérateurs arithmétiques

`+`, `-`, `*`, `/`, `++`, `--`.

Les opérateurs `++` et `--` servent à additionner ou à retrancher 1 à la valeur des variables sur lesquels ils sont appliqués.

Plan

7. Types

Types structurés

La syntaxe

```
typedef struct ALIAS {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
} NOM;
```

permet de **déclarer un type structuré** `NOM`, constitué des champs `CHAMP_1`, `CHAMP_2`, *etc.*

L'alias `ALIAS` est facultatif.

C'est un **amalgame de données** de divers types.

– Exemple –

```
typedef struct {
    int x;
    int y;
} Couple;
```

déclare un type structuré `Couple` qui permet de représenter des couples d'entiers.

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Cette syntaxe est un raccourci pour

```
(*adr_x).ch
```

– Exemple –

Les trois suites d'instructions suivantes sont équivalentes :

```
Couple *c;
...
c->x = c->x + 1;
```

```
Couple *c;
...
(*c).x = c->x + 1;
```

```
Couple *c;
...
c->x = (*c).x + 1;
```

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

– Exemple –

```
typedef struct {
    char nom[32];
    char prenom[32];
    int age;
} Personne;
...
Personne p1, p2;
scanf(" %s", p1.nom);
scanf(" %s", p1.prenom);
p1.age = 30;
p2 = p1;
```

L'affectation en dernière ligne fait en sorte que tous les champs de `p2` contiennent les mêmes valeurs que ceux de `p1`.

Il y a recopie des tableaux statiques `p1.nom` et `p1.prenom` dans `p2.nom` et `p2.prenom`.

Ce phénomène va être étudié en détail plus loin.

Types énumérés

La syntaxe

```
typedef enum {
    ENU_1,
    ENU_2,
    ...
} NOM;
```

permet de **déclarer un type énuméré** `NOM`, constitué des énumérateurs `ENU_1`, `ENU_2`, *etc.*

Attention, on utilise des `,` et non pas des `;`.

Une variable de ce type peut prendre pour valeur exactement un des énumérateurs qui le constituent.

– Exemple –

```
typedef enum {
    FAUX,
    VRAI
} Booleen;
```

est un type qui permet de représenter des booléens.

Une valeur de type `Booleen` est soit `FAUX`, soit `VRAI`.

Types énumérés

– Exemple –

```
typedef enum {
    LUNDI,     /* = 0 */
    MARDI,     /* = 1 */
    MERCREDI,  /* = 2 */
    JEUDI,     /* = 3 */
    VENDREDI,  /* = 4 */
    SAMEDI,    /* = 5 */
    DIMANCHE   /* = 6 */
} Jour;
...
printf("%d\n", MERCREDI);
```

Les énumérateurs sont des **expressions entières**. Leur valeur est déterminée par leur ordre de déclaration dans le type.

Ces instructions affichent `2`. En effet, `LUNDI` vaut `0` car il est le 1er énumérateur déclaré et les valeurs des suivants s'incrémentent selon leur ordre de déclaration.

```
typedef enum {
    LA = 0,
    SI = 2,
    DO,        /* = 3 */
    RE = 5,
    MI = 7,
    FA = 8,
    SOL = 10
} Note;
```

Il est possible de spécifier manuellement les valeurs des énumérateurs avec la syntaxe `ENU = VAL` où `ENU` est un énumérateur et `VAL` une constante entière.

Si une valeur n'est pas spécifiée, elle est déduite de la précédente en l'incrémentant.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

– Exemple –

```
Note note;

scanf(" %d", &note);

switch (note) {
    case LA : printf("A"); break;
    case SI : printf("B"); break;
    case DO : printf("C"); break;
    case RE : printf("D"); break;
    case MI : printf("E"); break;
    case FA : printf("F"); break;
    case SOL : printf("G"); break;
    default : printf("%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Remarque : une variable d'un type énuméré peut prendre comme valeur n'importe quel entier. Ceci explique la présence de la clause `default`.

L'intérêt de l'utilisation des types énumérés est principalement **sémantique** : un programme qui les utilise est plus facile à lire et à maintenir.

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que **les énumérateurs sont des entiers**.

– Exemples –

- `printf("%d\n", SOL == SOL);` affiche `1`
- `printf("%d\n", SOL == FA);` affiche `0`
- `printf("%d\n", SI <= RE);` affiche `1`

De même, l'**opérateur d'affectation** `=` est compatible avec les types énumérés.

L'**opérateur de taille** `sizeof` renvoie `4` sur les valeurs d'un type énuméré. C'est la taille occupée par le type `int`.

8. Types structurés

Plan

8. Types structurés

Déclaration de types structurés récursifs

Il est possible de **déclarer des types structurés récursifs** en faisant usage de l'alias et du mot clé `struct` :

```
typedef struct _Liste {
    int e;
    struct _Liste *s;
} Liste;
```

Ceci fonctionne car la taille d'un pointeur vers une valeur de type `T` est connue et indépendante de la nature de `T`.

```
typedef struct _Liste {
    int e;
    struct _Liste s;
} Liste;
```

Attention, cette déclaration n'est pas valide car le **champ récursif** n'est pas un **pointeur**.

Le système **ne peut pas connaître** la taille de ce champ.

Déclaration de types structurés mutuellement récursifs

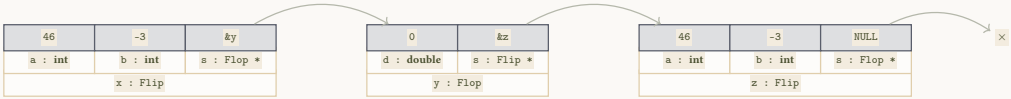
Il est possible de **déclarer des types structurés mutuellement récursifs** :

```
typedef struct _Flip {
    int a;
    int b;
    struct _Flop *s;
} Flip;

typedef struct _Flop {
    double d;
    struct _Flip *s;
} Flop;
```

– Exemple –

Une variable typique `x` de type `Flip` est de la forme



Initialisation d'une variable d'un type structuré

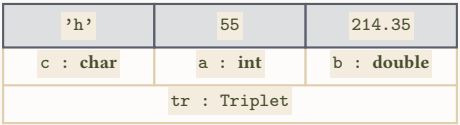
Il est possible d'**initialiser les champs** d'une variable d'un type structuré au moment de sa **déclaration**.

On utilise pour cela l'opérateur d'affectation `=` avec comme valeur droite les valeurs des champs à affecter dans des accolades et séparées par des virgules.

– Exemple –

```
typedef struct {
    char c;
    int a;
    double b;
} Triplet;
...
Triplet tr = {'h', 55, 214.35};
```

Déclare, en l'initialisant, la variable `tr`.



Plan

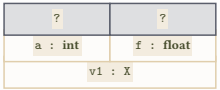
8. Types structurés

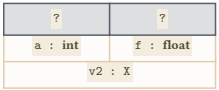
Affectation de variables d'un type structuré

Considérons le code

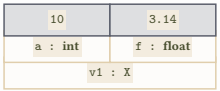
```
typedef struct {
    int a;
    float f;
} X;
...
X v1, v2;
v1.a = 10;
v1.f = 3.14;
v2 = v1;
v2.a = 20;
```

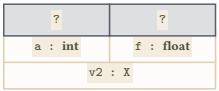
l. 6 :



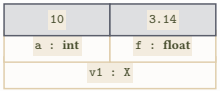


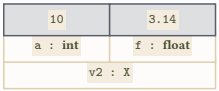
l. 8 :



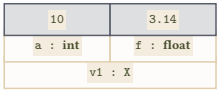


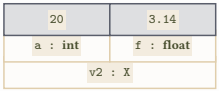
l. 9 :





l. 10 :





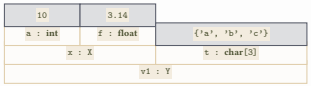
Observation : l'affectation **recopie** les champs d'une variable d'un **type scalaire**.

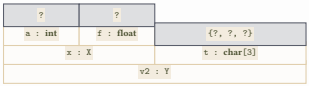
Affectation de variables d'un type structuré

Considérons le code

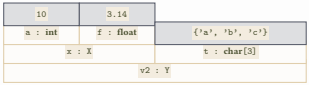
```
typedef struct {
    X x;
    char t[3];
} Y;
...
Y v1, v2;
v1.x.a = 10;
v1.x.f = 3.14;
v1.t = {'a', 'b', 'c'};
v2 = v1;
v2.x.f = 1.8;
v2.t[0] = 'g';
```

l. 9 :

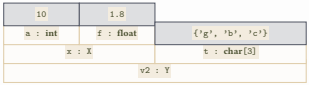




l. 10 :



l. 12 :

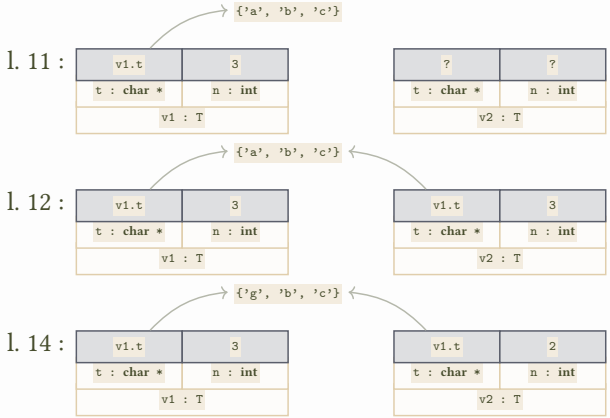


Observation : l'affectation **recopie** les champs d'une variable d'un type structuré de manière **récursive** et les **tableaux statiques**.

Affectation de variables d'un type structuré

Considérons le code

```
typedef struct {
    char *t;
    int n;
} T;
...
T v1, v2;
v1.t = malloc(3);
v1.n = 3;
v1.t[0] = 'a';
v1.t[1] = 'b';
v1.t[2] = 'c';
v2 = v1;
v2.n = 2;
v2.t[0] = 'g';
```



Observation : l'affectation **ne recopie pas** les **tableaux dynamiques**. Seule l'**adresse** d'un tableau dynamique est **recopiée**. C'est une copie de surface.

Affectation de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit (dans le même module) une fonction de prototype

```
int copier_X(const X *v1, X *v2);
```

qui **copie en profondeur** les champs de `v1` dans les champs de `v2`.

– Exemple –

La définition du type `T` précédent s'accompagne de la définition de la fonction

Cette fonction est munie du mécanisme habituel de gestion d'erreurs.

```
int copier_T(const T *v1, T *v2) {
    int i;
    assert(v1 != NULL);
    assert(v2 != NULL);
    v2->n = v1->n;
    v2->t = (char *) malloc(sizeof(char) * v1->n);
    if (v2->t == NULL) return 0;
    for (i = 0 ; i < v1->n ; ++i)
        v2->t[i] = v1->t[i];
    return 1;
}
```

Comparaison de variables d'un type structuré

Considérons le code

```
typedef struct {
    int a;
    int b;
} A;
...
A v1, v2;
...
if (v1 == v2) {...}
...
if (v1 != v2) {...}
```

Ce code est incorrect (il ne compile pas).

Le compilateur **n'accepte pas** la **comparaison de variables d'un type structuré**.

```
invalid operands to binary == (have 'A' and 'A')
```

```
invalid operands to binary != (have 'A' and 'A')
```

Comparaison de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit (dans le même module) deux fonctions de prototypes

```
int sont_ega_X(const X *v1, const X *v2);
```

```
int sont_dif_X(const X *v1, const X *v2);
```

qui **testent l'égalité** et **l'inégalité** entre `v1` et `v2`.

– Exemples –

La définition du type `A` précédent s'accompagne de la définition des fonctions

```
int sont_ega_A(A *v1, A *v2) {
    assert(v1 != NULL);
    assert(v2 != NULL);
    return (v1->a == v2->a) && (v1->b == v2->b);
}
```

```
int sont_dif_A(A *v1, A *v2) {
    assert(v1 != NULL);
    assert(v2 != NULL);
    return !sont_ega_A(v1, v2);
}
```

Attention : si `X` est composé d'un champ qui est un type structuré `Y`, il faut appeler dans `sont_ega_X` la fonction de comparaison `sont_ega_Y`.

Destruction de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré `X`, on définit (dans le même module) une fonction de prototype

```
void detruire_X(X *v);
```

qui **libère l'espace mémoire** adressé par `v`.

– Exemple –

La déclaration du type `B` suivant s'accompagne de la définition de la fonction

```
typedef struct {
    int *tab;
    int n;
} B;
```

```
void detruire_B(B *v) {
    assert(v != NULL);
    free(v->tab);
}
```

Attention : si `X` est composé d'un champ qui est un type structuré `Y`, il faut appeler dans `detruire_X` la fonction de destruction `detruire_Y`.

Plan

8. Types structurés

Renvoi d'une variable d'un type structuré

```
typedef struct {
    int x;
    int y;
} Couple;

Couple twist(Couple c) {
    Couple res;
    res.x = c.y;
    res.y = c.x;
    return res;
}
```

Ce code est correct (`twist` renvoie le couple obtenu par échange des coordonnées de celui passé en argument).

`twist` **renvoie** une variable d'un **type structuré**.

Cependant, ce code n'est pas efficace car, à chaque appel de fonction

```
d = twist(c);
```

la variable `res`, qui est située dans la pile, doit être **recopiée**.

Paramètre variable d'un type structuré

```
typedef struct {
    int tab1[2048];
    int tab2[2048];
} DeuxTab;

int prem_egaux(DeuxTab x) {
    return x.tab1[0] == x.tab2[0];
}
```

Le code est correct (`prem_egaux` teste si les premières cases des tableaux sont égales).

`prem_egaux` est **paramétrée** par une variable d'un **type structuré**.

Cependant, ce code n'est pas efficace car à chaque appel de fonction

```
prem_egaux(y);
```

les champs de l'**argument** `y` sont recopiés dans le **paramètre** `x`.

Passage par adresse *vs* passage par valeur

Soit une fonction `fct` paramétrée par une variable `x` d'un type structuré `T`.

Il est d'usage courant d'adopter la convention suivante :

- si les champs de `x` doivent être modifiés par la fonction, alors on recourt à un **passage par adresse**

```
... fct(T *x, ...) { ... }
```

- si les champs de `x` ne doivent pas être modifiés par la fonction, alors on recourt à un **passage par valeur**

```
... fct(T x, ...) { ... }
```

Cette conception est erronée car il est possible de « modifier » une variable d'un type structuré passée par valeur à une fonction.

Passage par adresse *vs* passage par valeur

Considérons en effet le code suivant :

```
typedef struct {
    int *tab;
    int n;
} Tab;
```

```
void init(Tab t, int k) {
    int i;
    for (i = 0 ; i < t.n ; ++i)
        t.tab[i] = k;
}
```

Chaque appel de fonction

```
init(s, r);
```

provoque la recopie de trois valeurs (ce qui est encore acceptable) mais « modifie » les valeurs pointées par le champ `tab` de `s`, malgré le passage par valeur.

Conclusion : écrire des fonctions avec **passage par valeur** des paramètres d'un **type structuré** ne présente que des **désavantages**.

Variables d'un type structuré dans les fonctions

En résumé, on adopte les deux règles suivantes :

1. une fonction ne renvoie jamais de valeur d'un type structuré ;
2. tous les paramètres d'un type structuré sont passés par adresse dans une fonction.

Ainsi, un prototype de fonction habituel est

```
int fct(T *x, E1 e1, ..., EN en, S1 *s1, ..., SM *sm);
```

où

- le type de retour est `int` (renvoi d'un code d'erreur) ;
- `x` est l'adresse d'une variable d'un type structuré `T` ;
- `e1`, ..., `en` sont les entrées de la fonction (adresses ou non) ;
- `s1`, ..., `sm` sont les sorties de la fonction (qui sont des adresses).

Variables d'un type structuré dans les fonctions

– Exemple –

Voici le nécessaire pour calculer la somme pondérée de deux points selon les conventions établies :

```
typedef struct {
    float x;
    float y;
} Point;

void somme_points(const Point *p1, const Point *p2, float coeff1, float coeff2, Point *res) {

    assert(p1 != NULL);
    assert(p2 != NULL);
    assert(res != NULL);

    res->x = coeff1 * p1->x + coeff2 * p2->x;
    res->y = coeff1 * p1->y + coeff2 * p2->y;
}
```

Résumé

Voici en résumé la bonne marche à suivre lors de la manipulation de types structurés :

1. on utilise l'**alias** lors de la déclaration de types structurés **récursifs** et/ou **mutuellement récursifs** ;
2. toute **déclaration d'un type structuré** s'accompagne de la définition des quatre fonctions suivantes :
   - une fonction de **copie** ;
   - une fonction de **test d'égalité** ;
   - une fonction de **test d'inégalité** ;
   - une fonction de **destruction** ;
3. on ne **renvoie jamais** de valeur d'un type structuré ;
4. on passe les **paramètres** d'un type structuré **par adresse** (ne pas oublier d'ajouter les qualificateurs `const` nécessaires).

8. Types structurés

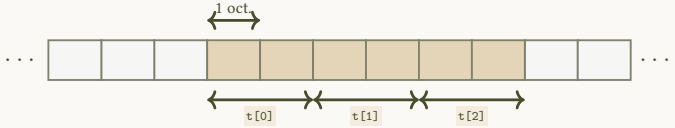
Alignement en mémoire

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

– Exemple –

Nous avons déjà vu que 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

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

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

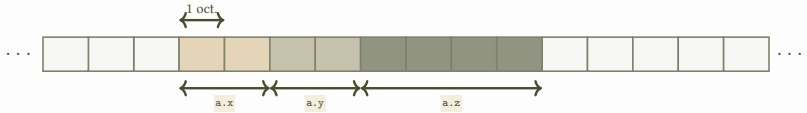
`A` et `B` sont des types structurés composés des mêmes champs. Il n'y a que l'ordre de leur déclaration qui diffère.

Cependant, l'instruction `printf("%lu %lu\n", sizeof(A), sizeof(B));` affiche `8 12`.

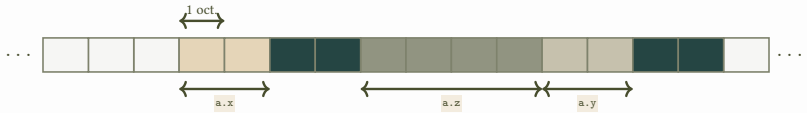
Le fait que les tailles des variables de type `A` et `B` diffèrent est dû à leur **alignements en mémoire** respectifs qui ne sont pas les mêmes.

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

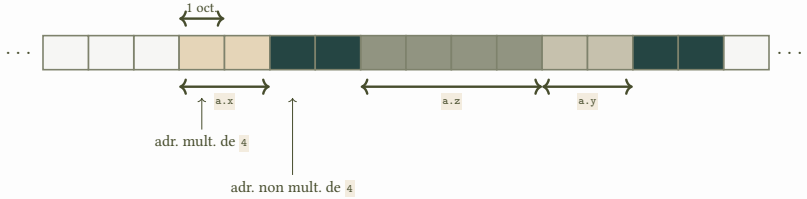


Les octets foncés intervenant dans l'alignement mémoire de `b` sont des octets de complétion.

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.