Plan

Types structurés
- Déclaration et initialisation
- Affectation et comparaison
- Dans les fonctions
- Alignement en mémoire

Plan

Types structurés
- Déclaration et initialisation
- Affectation et comparaison
- Dans les fonctions
- Alignement en mémoire

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` :

```
1 typedef struct _Liste {
2     int e;
3     struct _Liste *s;
4 } Liste;
```

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

Attention, la déclaration

```
1 typedef struct _Liste {
2     int e;
3     struct _Liste s;
4 } Liste;
```

n'est pas valide car le **champ récursif** n'est pas un **pointeur**.

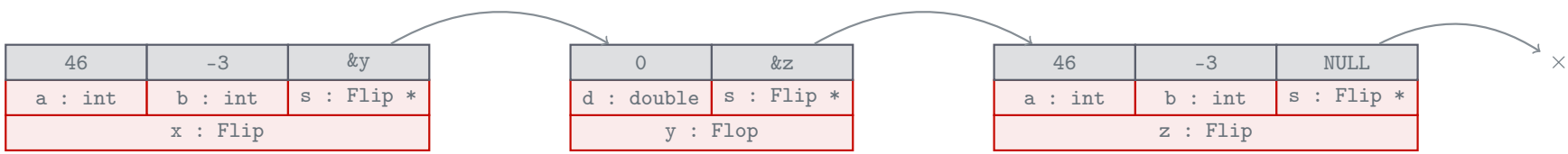
Le système ne peut pas connaître pas la taille de ce champ.

Déclaration de types structurés mutuellement récursifs

Il est possible de déclarer des types structurés mutuellement récursifs :

```
1  typedef struct _Flip {
2      int a;
3      int b;
4      struct _Flop *s;
5  } Flip;
6
7  typedef struct _Flop {
8      double d;
9      struct _Flip *s;
10 } Flop;
```

P.ex., `x` est une variable de type `Flip` représentée par



Initialisation d'une variable d'un type structuré

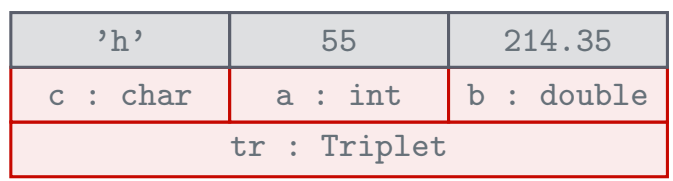
Il est possible d'initialiser les champs d'une variable d'un type structuré au moment de sa déclaration.

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.

Par exemple,

```
1  typedef struct {
2      char c;
3      int a;
4      double b;
5  } Triplet;
6  ...
7  Triplet tr = {'h', 55, 214.35};
```

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



Plan

Types structurés
- Déclaration et initialisation
- Affectation et comparaison
- Dans les fonctions
- Alignement en mémoire

Affectation de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

Affectation de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

l. 6 :

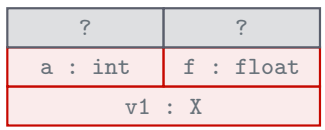
Affectation de variables d'un type structuré

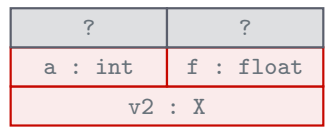
Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

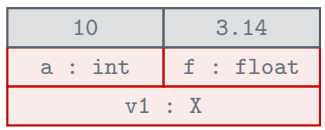
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

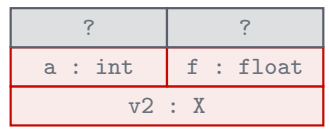
l. 6 :





l. 8 :





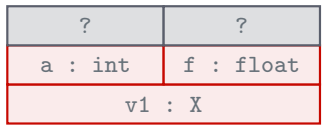
Affectation de variables d'un type structuré

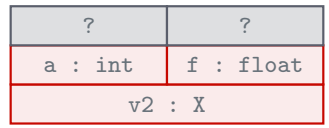
Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

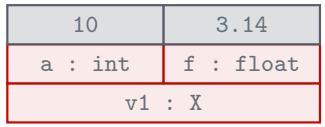
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

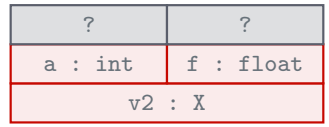
l. 6 :



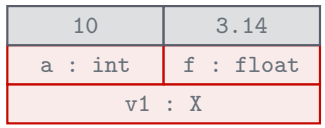


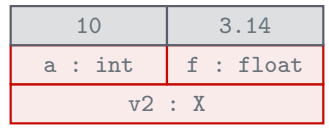
l. 8 :





l. 9 :





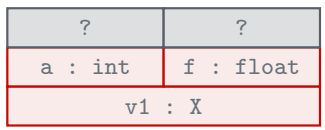
Affectation de variables d'un type structuré

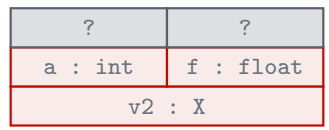
Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

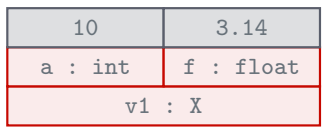
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

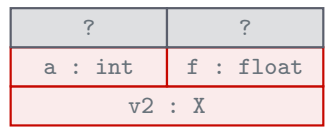
l. 6 :



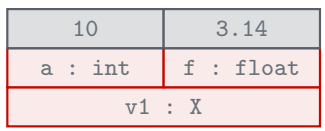


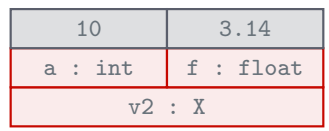
l. 8 :



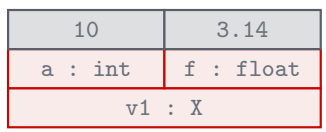


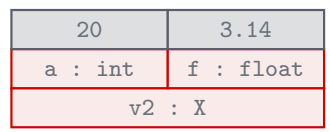
l. 9 :





l. 10 :





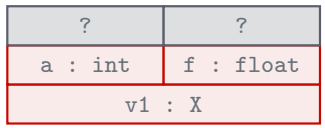
Affectation de variables d'un type structuré

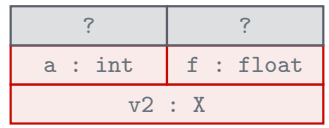
Considérons le code

```
1  typedef struct {
2      int a;
3      float f;
4  } X;
5  ...
```

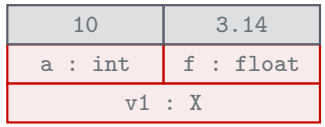
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

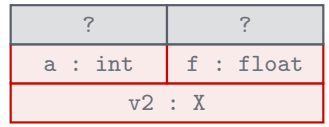
l. 6 :



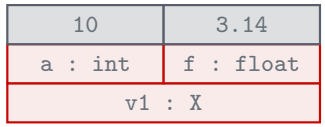


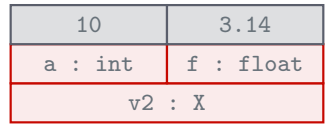
l. 8 :



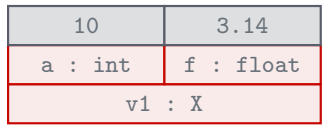


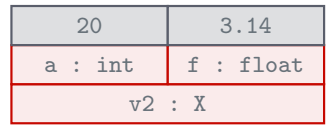
l. 9 :





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

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;
```

```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

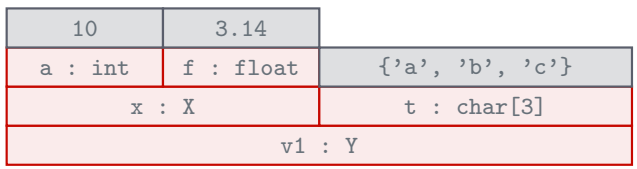
Affectation de variables d'un type structuré

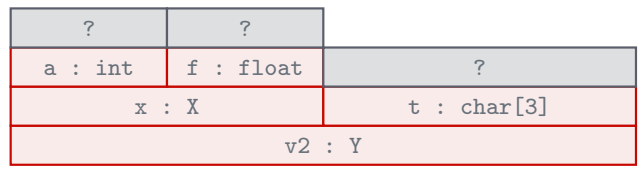
Considérons le code

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;
```

```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

l. 9 :





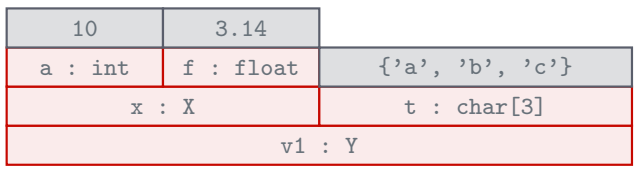
Affectation de variables d'un type structuré

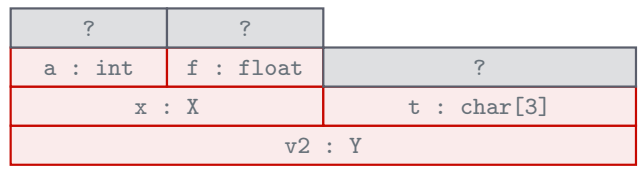
Considérons le code

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;
```

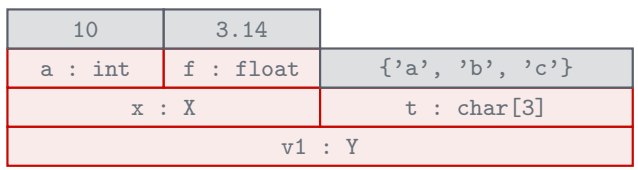
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

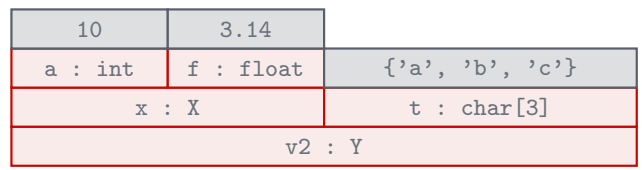
l. 9 :





l. 10 :





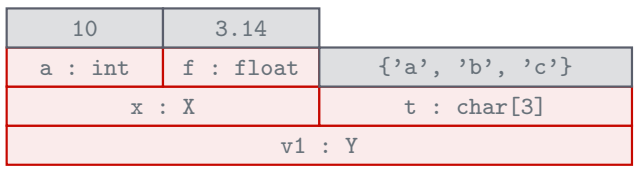
Affectation de variables d'un type structuré

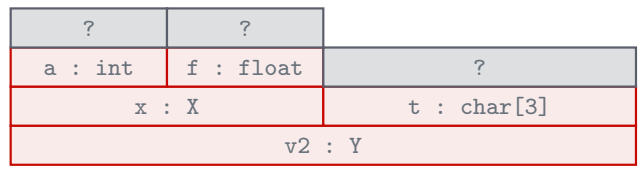
Considérons le code

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;
```

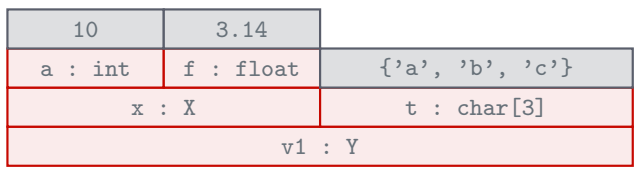
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

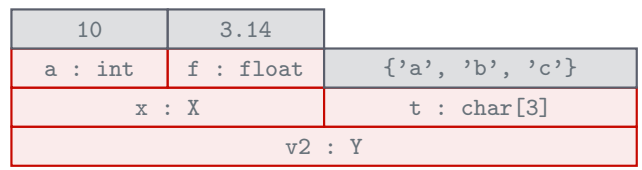
l. 9 :



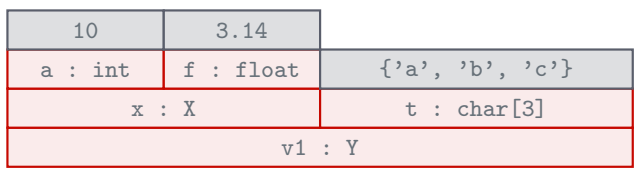


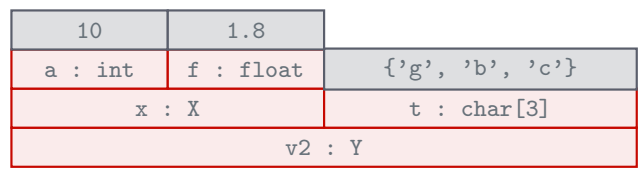
l. 10 :





l. 12 :





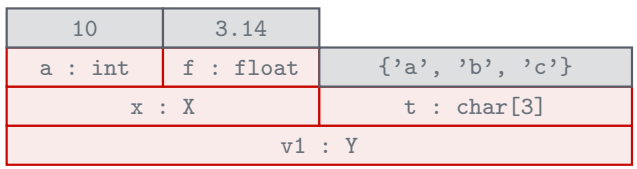
Affectation de variables d'un type structuré

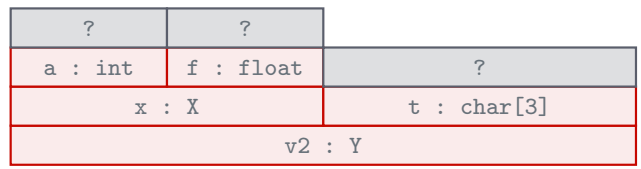
Considérons le code

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;
```

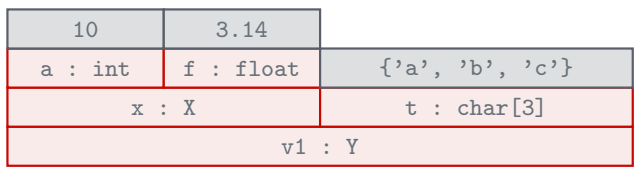
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

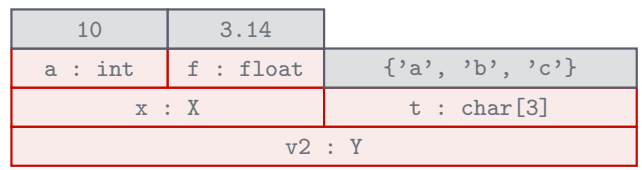
l. 9 :



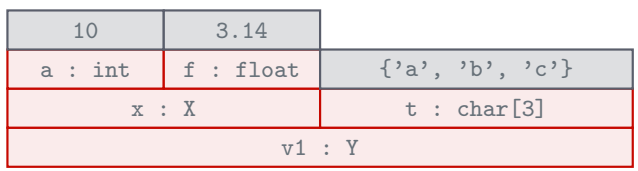


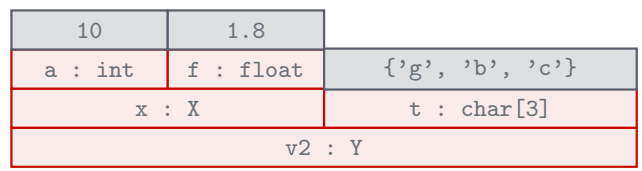
l. 10 :





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

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
```

```
6  T v1, v2;
7  v1.t = malloc(3);
8  v1.n = 3;
9  v1.t[0] = 'a';
10 v1.t[1] = 'b';
```

```
11 v1.t[2] = 'c';
12 v2 = v1;
13 v2.n = 2;
14 v2.t[0] = 'g';
```

Affectation de variables d'un type structuré

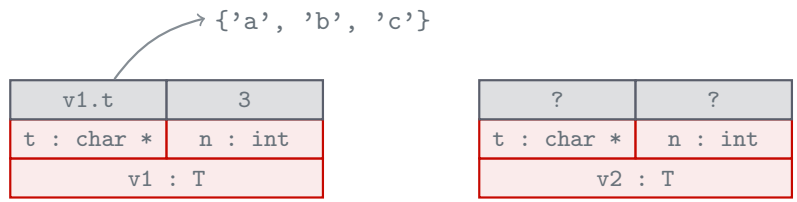
Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
```

```
6  T v1, v2;
7  v1.t = malloc(3);
8  v1.n = 3;
9  v1.t[0] = 'a';
10 v1.t[1] = 'b';
```

```
11 v1.t[2] = 'c';
12 v2 = v1;
13 v2.n = 2;
14 v2.t[0] = 'g';
```

l. 11 :



Affectation de variables d'un type structuré

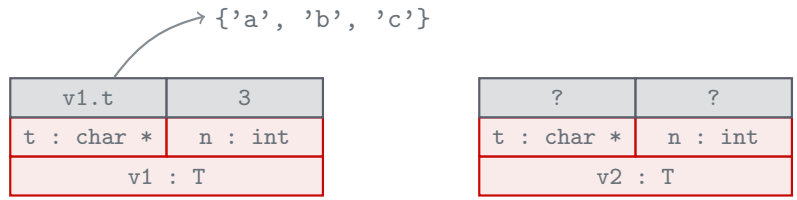
Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
```

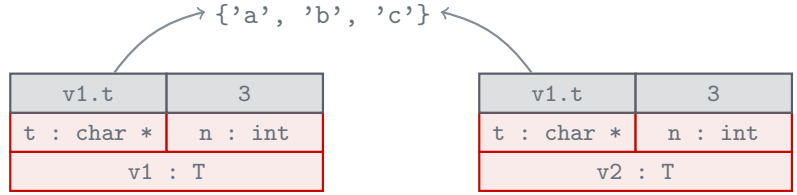
```
6   T v1, v2;
7   v1.t = malloc(3);
8   v1.n = 3;
9   v1.t[0] = 'a';
10  v1.t[1] = 'b';
```

```
11  v1.t[2] = 'c';
12  v2 = v1;
13  v2.n = 2;
14  v2.t[0] = 'g';
```

l. 11 :



l. 12 :



Affectation de variables d'un type structuré

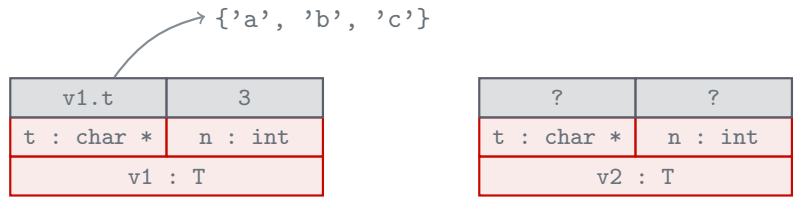
Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
```

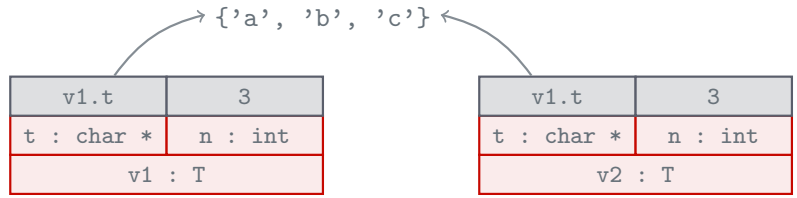
```
6  T v1, v2;
7  v1.t = malloc(3);
8  v1.n = 3;
9  v1.t[0] = 'a';
10 v1.t[1] = 'b';
```

```
11 v1.t[2] = 'c';
12 v2 = v1;
13 v2.n = 2;
14 v2.t[0] = 'g';
```

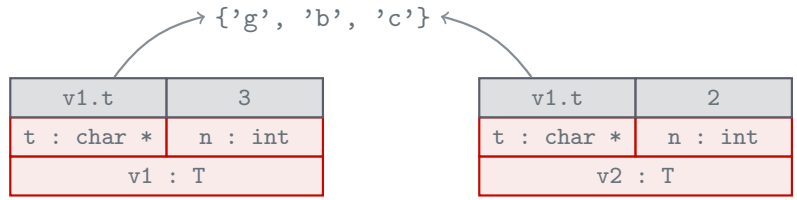
l. 11 :



l. 12 :



l. 14 :



Affectation de variables d'un type structuré

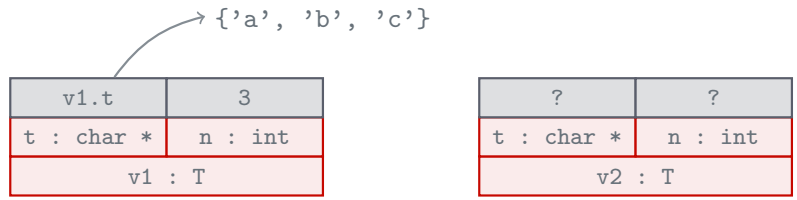
Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
```

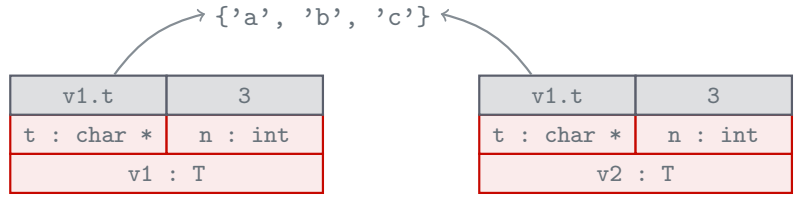
```
6   T v1, v2;
7   v1.t = malloc(3);
8   v1.n = 3;
9   v1.t[0] = 'a';
10  v1.t[1] = 'b';
```

```
11  v1.t[2] = 'c';
12  v2 = v1;
13  v2.n = 2;
14  v2.t[0] = 'g';
```

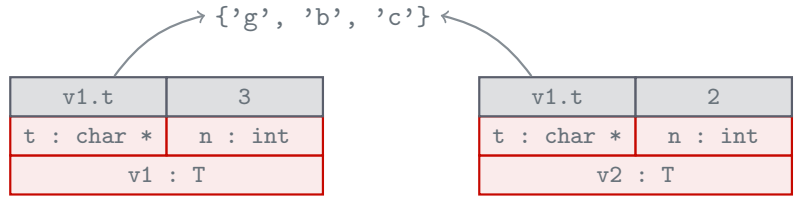
l. 11 :



l. 12 :



l. 14 :



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`.

Par exemple, la définition du type `T` précédent s'accompagne de la définition de la fonction

```
1  int copier_T(const T *v1, T *v2) {
2      int i;
3      assert(v1 != NULL);
4      assert(v2 != NULL);
5      v2->n = v1->n;
6      v2->t = (char *) malloc(sizeof(char) * v1->n);
7      if (v2->t == NULL) return 0;
8      for (i = 0 ; i < v1->n ; ++i)
9          v2->t[i] = v1->t[i];
10     return 1;
11 }
```

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

Comparaison de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      int a;
3      int b;
4  } A;
5  ...
```

```
6  A v1, v2;
7  ...
8  if (v1 == v2) {...}
9  ...
10 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`.

Par exemple, la définition du type `A` précédent s'accompagne de la définition des fonctions

```
1  int sont_ega_A(A *v1, A *v2) {
2      assert(v1 != NULL);
3      assert(v2 != NULL);
4      return (v1->a == v2->a)
5          && (v1->b == v2->b);
6  }
```

```
7  int sont_dif_A(A *v1, A *v2) {
8      assert(v1 != NULL);
9      assert(v2 != NULL);
10     return !sont_ega_A(v1, v2);
11 }
```

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`.

Par exemple, la déclaration du type `B` suivant s'accompagne de la définition de la fonction

```
1  typedef struct {
2      int *tab;
3      int n;
4  } B;
5
```

```
6  void detruire_B(B *v) {
7      assert(v != NULL);
8      free(v->tab);
9      *v = NULL;
10 }
```

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

Types structurés
- Déclaration et initialisation
- Affectation et comparaison
- Dans les fonctions
- Alignement en mémoire

Renvoi d'une variable d'un type structuré

Le code

```
1  typedef struct {
2      int x;
3      int y;
4  } Couple;
5
6
```

```
7  Couple twist(Couple c) {
8      Couple res;
9      res.x = c.y;
10     res.y = c.x;
11     return res;
12 }
```

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, il n'est pas efficace car, à chaque appel de fonction

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

la variable `res`, qui vit dans la pile, doit être recopiée.

Paramètre variable d'un type structuré

Le code

```
1  typedef struct {
2      int tab1[2048];
3      int tab2[2048];
4  } DeuxTab;
5
```

```
6  int prem_egaux(DeuxTab x) {
7      return x.tab1[0]
8          == x.tab2[0];
9  }
```

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, il 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`.

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, ...) { ... }
```

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, ...) { ... }
```

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 :

```
1  tyepdef struct {
2      int *tab;
3      int n;
4  } Tab;
5
```

```
6  void init(Tab t, int k) {
7      int i;
8      for (i = 0 ; i < t.n ; ++i)
9          t.tab[i] = k;
10 }
```

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 :

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 ;

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.

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) ;

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` ;

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) ;

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

P.ex., voici le nécessaire pour calculer la somme pondérée de deux points selon les conventions établies :

```
1  typdef struct {
2      float x;
3      float y;
4  } Point;
5
6  void somme_points(const Point *p1, const Point *p2,
7          float coeff1, float coeff2,
8          Point *res) {
9
10     assert(p1 != NULL);
11     assert(p2 != NULL);
12     assert(res != NULL);
13
14     res->x = coeff1 * p1->x + coeff2 * p2->x;
15     res->y = coeff1 * p1->y + coeff2 * p2->y;
16 }
```

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).

Plan

Types structurés
- Déclaration et initialisation
- Affectation et comparaison
- Dans les fonctions
- Alignement en mémoire

Alignement en mémoire

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

Alignement en mémoire

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

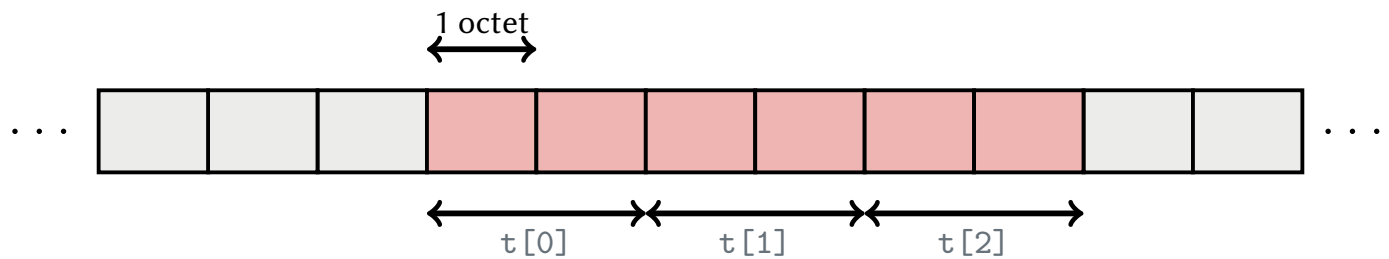
Par exemple, nous avons déjà vu que les **tableaux** de taille `n` d'éléments d'un type `T` sont organisés en un segment contigu de `sizeof(T) * n` octets.

Alignement en mémoire

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

Par 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

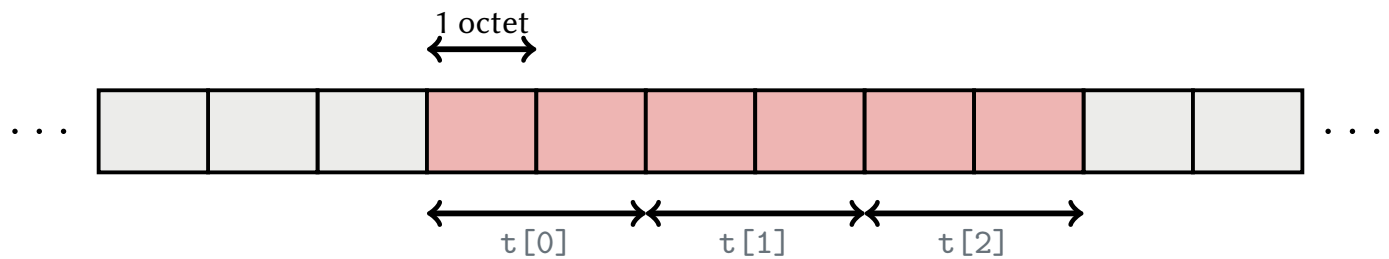


Alignement en mémoire

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

Par 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

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

```c
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,

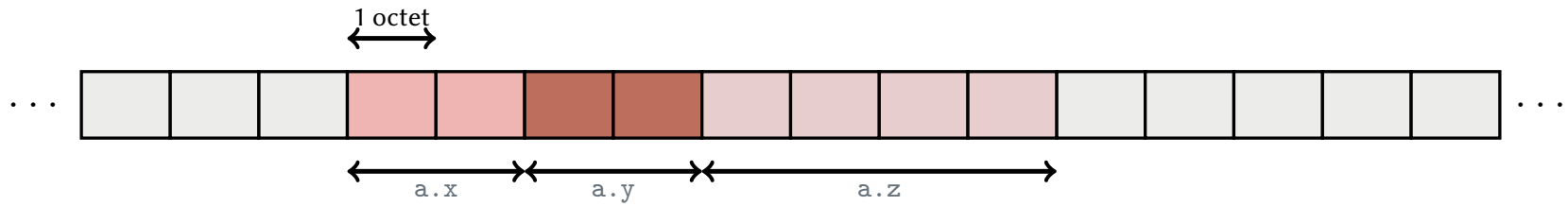
```c
printf("%lu %lu\n", sizeof(A), sizeof(B));
```

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.

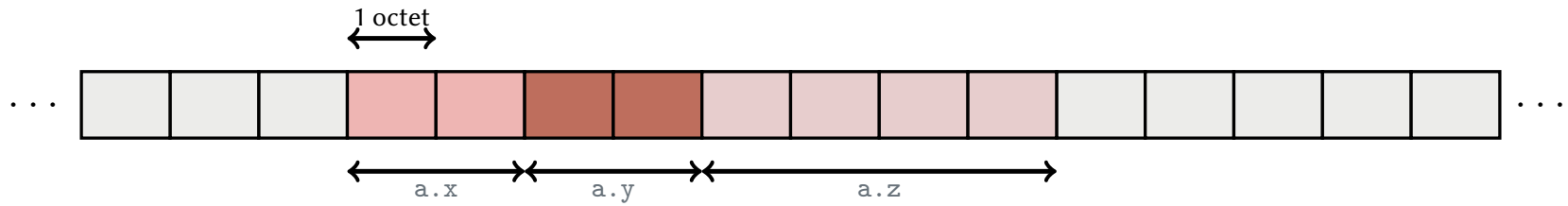
Alignement en mémoire des variables d'un type structuré

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

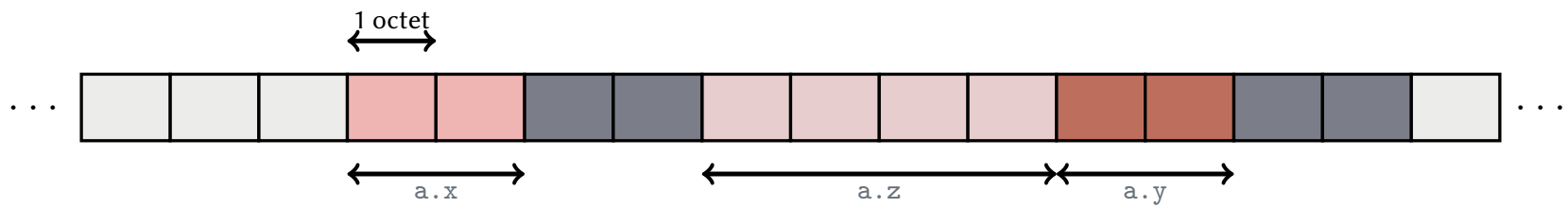


Alignement en mémoire des variables d'un type structuré

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

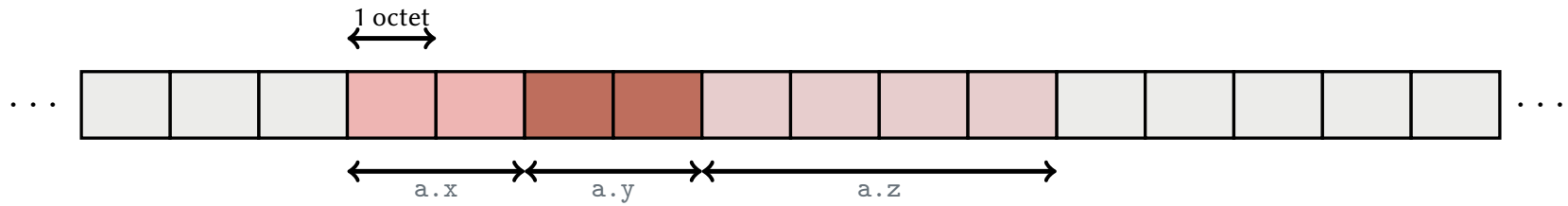


Soit `b` une variable de type `B`. Cette variable est organisée en mémoire en

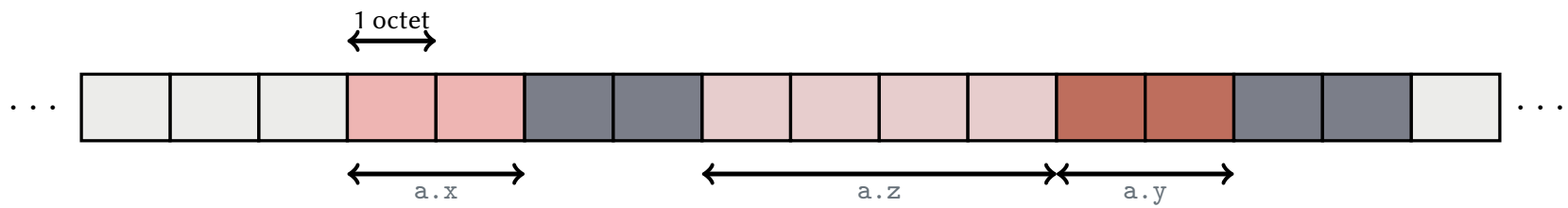


Alignement en mémoire des variables d'un type structuré

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.

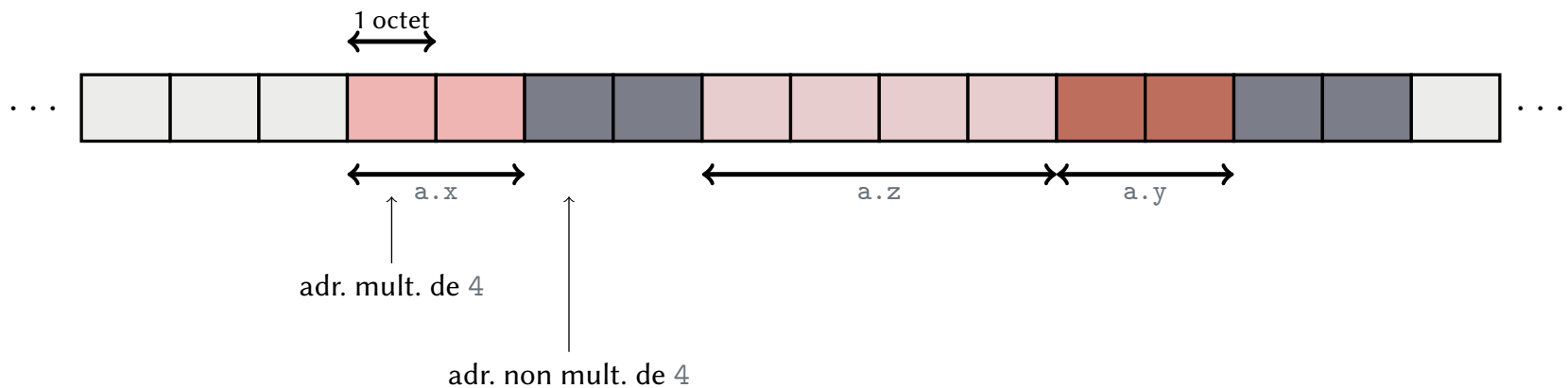
Octets de complétion

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.

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`.

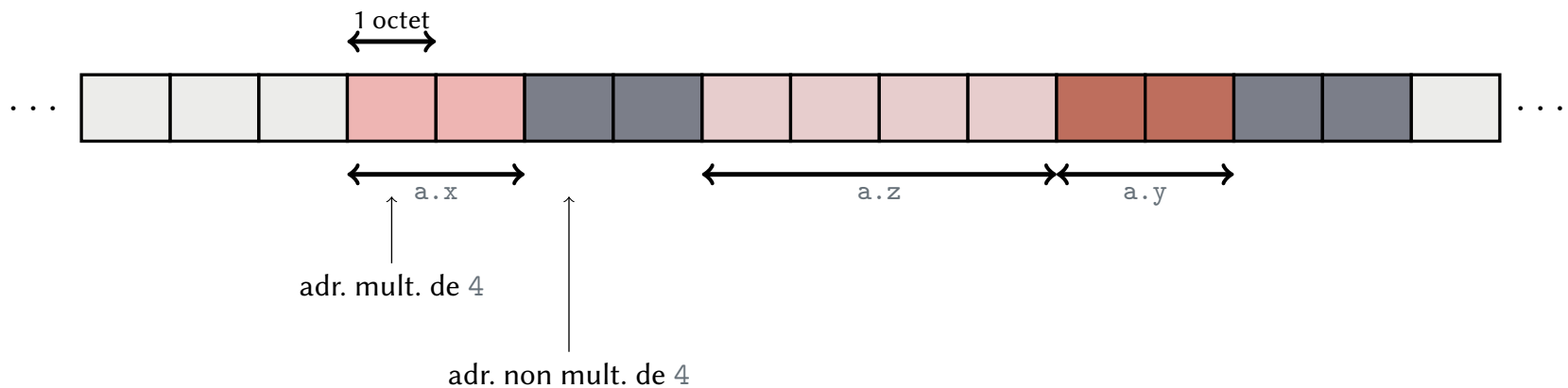
Dans notre exemple, en sachant que tout champ de type `short` (resp. `int`) doit commencer à une adresse multiple de 2 (resp. 4), on explique l'alignement en mémoire de `b` précédent :



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 :



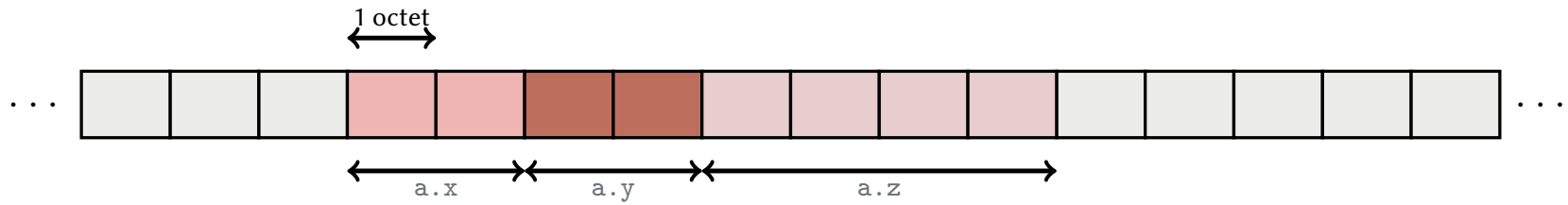
Les derniers octets de complétion sont introduits pour que les tableaux de variables de type `B` puissent être représentés en vérifiant cet alignement en mémoire.

Accès manuel aux champs

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

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

Cette variable est organisée en mémoire en



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

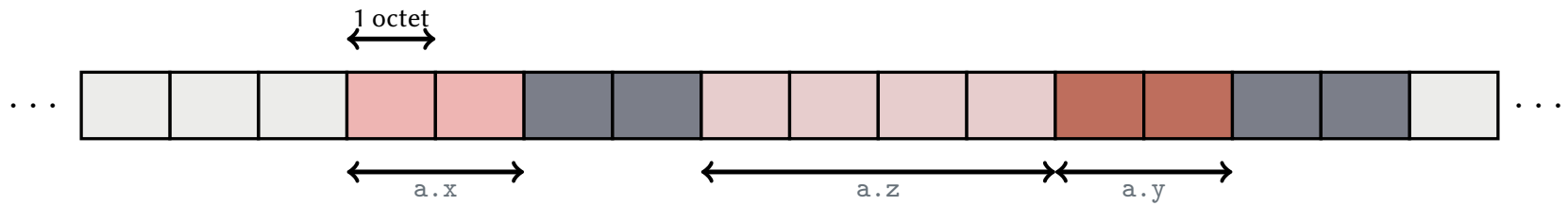
```
1  short x, y;
2  int z;
3  void *p;
4  p = &a;
5  x = *((short *) p); /* Equivalent a x = a.x; */
6  p += 2; /* p est de type void * : l'adresse p est incrementee de 2 */
7  y = *((short *) p); /* Equivalent a y = a.y; */
8  p += 2;
9  z = *((int *) p); /* Equivalent a z = a.z; */
```

Accès manuel aux champs

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

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

Cette variable est organisée en mémoire en



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

```
1  short x, y;
2  int z;
3  void *p;
4  p = &b;
5  x = *((short *) p); /* Equivalent a x = a.x; */
6  p += 4;
7  z = *((int *) p); /* Equivalent a z = a.z; */
8  p += 4;
9  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

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

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.

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**.

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**.

Dans la pratique, on ne cherchera pas à déclarer des types structurés qui ne nécessitent pas d'octet de complétion. Au contraire : il est de loin préférable de déclarer les champs dans un ordre logique favorisant la relecture et la maintenance.