Plan

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
=	affect.	2	⟵	une var. et une val.

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
`=`	affect.	2	$\longleftarrow$	une var. et une val.
`+=`, `-=`, `*=`, `/=`, `%=`	affect. compo. arith.	2	$\longleftarrow$	une var. num. et une val. num

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
`=`	affect.	2	⟵	une var. et une val.
`+=`, `-=`, `*=`, `/=`, `%=`	affect. compo. arith.	2	⟵	une var. num. et une val. num
`&=`, `|=`, `^=`, `<<=`, `>>=`	affect. compo. bit à bit	2	⟵	une var. ent. et une val. ent.

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
`=`	affect.	2	⟵	une var. et une val.
`+=`, `-=`, `*=`, `/=`, `%=`	affect. compo. arith.	2	⟵	une var. num. et une val. num
`&=`, `|=`, `^=`, `< <=`, `> >=`	affect. compo. bit à bit	2	⟵	une var. ent. et une val. ent.

Toute expression de la forme `a X= b` est équivalente à `a = a X b`.

Opérateurs d'affectation

Toutes les expressions d'affectation produisent une valeur qui est la valeur qui vient d'être affectée.

Opérateurs d'affectation

Toutes les expressions d'affectation produisent une valeur qui est la valeur qui vient d'être affectée.

Par exemple, dans

```
int a, b;
a = 2;
b = 5;
a *= b += 3;
```

à cause de l'associativité des opérateurs d'affectation, la l. 4 s'interprète comme `a *= (b += 3);`.

Ainsi, comme `b += 3` produit la valeur `8`, `a` vaut finalement `16`.

Plan

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	une var. ou un type

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	une var. ou un type
`(T)`	coercition `T` est un type	1	–	une val.

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	une var. ou un type
`(T)`	coercition `T` est un type	1	–	une val.
`? :`	condition	3	–	une val. num. et deux val.

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	une var. ou un type
`(T)`	coercition `T` est un type	1	–	une val.
`? :`	condition	3	–	une val. num. et deux val.
`,`	séquence	2	$\longrightarrow$	deux val

L'opérateur de séquence

Dans l'expression `V1, V2`, où `V1` et `V2` sont des valeurs, on commence par évaluer `V1` puis ensuite `V2`. Cette expression produit la valeur `V2`.

L'opérateur de séquence

Dans l'expression `V1, V2`, où `V1` et `V2` sont des valeurs, on commence par évaluer `V1` puis ensuite `V2`. Cette expression produit la valeur `V2`.

L'opérateur `,` est le plus souvent utilisé dans les **champs des boucles** `for`.

L'opérateur de séquence

Dans l'expression `V1, V2`, où `V1` et `V2` sont des valeurs, on commence par évaluer `V1` puis ensuite `V2`. Cette expression produit la valeur `V2`.

L'opérateur `,` est le plus souvent utilisé dans les **champs des boucles** `for`.

P.ex.,

```
int i, j, l;
...
for (i = 0, j = l - 1 ; i < j ; ++i, --j) {
...
}
```

permet d'obtenir une boucle `for` avec **deux compteurs** : `i` croît et `j` décroît dans l'intervalle allant de `0` à `l - 1`.

Plan

Plan

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Cependant, au même titre qu'une variable, toute fonction possède une **adresse** en mémoire. Il devient alors possible de réaliser des opérations sur les fonctions au moyen de leur adresse.

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Cependant, au même titre qu'une variable, toute fonction possède une **adresse** en mémoire. Il devient alors possible de réaliser des opérations sur les fonctions au moyen de leur adresse.

On parle alors de pointeur de fonction.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

```
&fct
```

permet d'accéder à l'adresse de `fct`.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'adresse de `fct`.

```
#include <stdio.h>
int somme(int a, int b) {
    return a + b;
}
int produit(int a, int b) {
    return a * b;
}
int main() {
    printf("%p\n", &somme);
    printf("%p\n", &produit);
    return 0;
}
```

Ce programme affiche `0x40052d` et `0x400541`, respectivement les adresses des fonctions `somme` et `produit`.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'adresse de `fct`.

```
#include <stdio.h>
int somme(int a, int b) {
    return a + b;
}
int produit(int a, int b) {
    return a * b;
}
int main() {
    printf("%p\n", &somme);
    printf("%p\n", &produit);
    return 0;
}
```

Ce programme affiche `0x40052d` et `0x400541`, respectivement les adresses des fonctions `somme` et `produit`.

Note : aux lignes 9 et 10, il est possible de ne pas mentionner les `&`. Le compilateur comprend implicitement qu'il s'agit de pointeurs de fonction.

Le type pointeur de fonction

La syntaxe

```
T (*FCT)(T1, ..., TN);
```

où

- `T`, `T1`, …, `TN` sont des types ;
- `FCT` est un identificateur ;

permet de déclarer un pointeur de fonction.

Le type pointeur de fonction

La syntaxe

```
T (*FCT)(T1, ..., TN);
```

où

- `T`, `T1`, …, `TN` sont des types ;
- `FCT` est un identificateur ;

permet de déclarer un pointeur de fonction.

Celui-ci a `FCT` pour identificateur et peut être l'adresse d'une fonction de type de retour `T` et de signature `(T1, ..., TN)`.

.

Le type pointeur de fonction

La syntaxe

```
T (*FCT)(T1, ..., TN);
```

où

- `T`, `T1`, ..., `TN` sont des types ;
- `FCT` est un identificateur ;

permet de déclarer un pointeur de fonction.

Celui-ci a `FCT` pour identificateur et peut être l'adresse d'une fonction de type de retour `T` et de signature `(T1, ..., TN)`.

```
float moyenne(int a, int b) {
    return (0.0 + a + b) / 2;
}
...
/* Decl. d'un ptr de fonction */
float (*moy)(int, int);

/* Utilisation */
moy = &moyenne;
printf("%f\n", moy(2, 3));
```

Pour la même raison que dans l'exemple précédent, il est possible à la ligne 9 de ne pas mentionner le `&`.

Cependant, pour la clarté du code, nous prenons la **convention de mentionner tous les** `&`.

Champs pointeurs de fonction

Un champ d'un type structuré peut être un pointeur sur une fonction.

Champs pointeurs de fonction

Un champ d'un type structuré peut être un pointeur sur une fonction.

Ceci déclare un type structuré sensé modéliser des suites d'entiers :

```
typedef struct {
    int t;
    int (*t_suiv)(int);
} Suite;
```

`t` contient le terme courant de la suite et `t_suiv` est la fonction qui, étant donné un terme en entrée, calcule le terme suivant.

Champs pointeurs de fonction

Un champ d'un type structuré peut être un pointeur sur une fonction.

Ceci déclare un type structuré sensé modéliser des suites d'entiers :

```
typedef struct {
    int t;
    int (*t_suiv)(int);
} Suite;
```

`t` contient le terme courant de la suite et `t_suiv` est la fonction qui, étant donné un terme en entrée, calcule le terme suivant.

```
int suivant(Suite *s) {
    s->t = s->t_suiv(s->t);
    return s->t;
}

int mul_2(int x) {
    return 2 * x;
}
```

```
...
int i;
Suite s;
s.t = 1;
s.t_suiv = &mul_2;
for (i = 0 ; i < 6 ; ++i) {
    printf("%d ", suivant(&s));
}
```

Ceci affiche 1 2 4 8 16 32.

Tableaux de pointeurs de fonction

Il est possible de manipuler des tableaux de pointeurs de fonction.

Tableaux de pointeurs de fonction

Il est possible de manipuler des tableaux de pointeurs de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

Tableaux de pointeurs de fonction

Il est possible de manipuler des tableaux de pointeurs de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

2. on déclare ensuite le tableau de manière usuelle. Syntaxe :

```
FCT tab[M];
```

Tableaux de pointeurs de fonction

Il est possible de manipuler des tableaux de pointeurs de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

2. on déclare ensuite le tableau de manière usuelle. Syntaxe :

```
FCT tab[M];
```

La syntaxe plus directe

```
T (*tab[M])(T1, ..., TN);
```

existe mais rend le code plus difficile à lire. Elle déclare un tableau `tab` de pointeurs de fonction sans la déclaration de type préalable de la 1re méthode.

Tableaux de pointeurs de fonction

```
/* Alias pour ptr. de fct. */
typedef int (*opb)(int, int);

int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}
```

```
...
opb tab[2];

tab[0] = &add;
tab[1] = &mul;
printf("%d %d\n",
    tab[0](10, 20),
    tab[1](10, 20));
```

Ceci affiche 30 200.

Tableaux de pointeurs de fonction

```
/* Alias pour ptr. de fct. */
typedef int (*opb)(int, int);

int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}
```

```
...
opb tab[2];

tab[0] = &add;
tab[1] = &mul;
printf("%d %d\n",
    tab[0](10, 20),
    tab[1](10, 20));
```

Ceci affiche 30 200.

Les tableaux de pointeurs de fonction peuvent être **dynamiques**. La ligne 10 peut être remplacée par

```
opb *tab;
tab = (opb *) malloc(sizeof(opb) * 2);
```

Plan

Pointeur de fonction en paramètre

Une fonction peut être paramétrée par un pointeur de fonction. Un paramètre pointeur de fonction est spécifié avec la même syntaxe que celle qui sert à le déclarer.

Pointeur de fonction en paramètre

Une fonction peut être paramétrée par un pointeur de fonction. Un paramètre pointeur de fonction est spécifié avec la même syntaxe que celle qui sert à le déclarer.

P.ex.,

```
int appliquer(int n, int k, int (*f)(int)) {
    int i;
    for (i = 0 ; i < k ; ++i)
        n = f(n);
    return n;
}
```

est une fonction est paramétrée par un pointeur de fonction acceptant un entier et renvoyant un entier.

Pointeur de fonction en paramètre

```
int add_1(int n) {
    return n + 1;
}

int mul_2(int n) {
    return 2 * n;
}
```

```
...
printf("%d\n",
    appliquer(3, 4, &add_1));

printf("%d\n",
    appliquer(3, 4, &mul_2));
```

Pointeur de fonction en paramètre

```
int add_1(int n) {
    return n + 1;
}

int mul_2(int n) {
    return 2 * n;
}
```

```
...
printf("%d\n",
    appliquer(3, 4, &add_1));

printf("%d\n",
    appliquer(3, 4, &mul_2));
```

Le 1er appel à `appliquer` calcule

```
(((3 + 1) + 1) + 1) + 1
```

et affiche donc 7.

Pointeur de fonction en paramètre

```
int add_1(int n) {
    return n + 1;
}

int mul_2(int n) {
    return 2 * n;
}
```

```
...
printf("%d\n",
    appliquer(3, 4, &add_1));

printf("%d\n",
    appliquer(3, 4, &mul_2));
```

Le 1er appel à `appliquer` calcule

```
(((3 + 1) + 1) + 1) + 1
```

et affiche donc 7.

Le 2e appel à `appliquer` calcule

```
(((3 * 2) * 2) * 2) * 2
```

et affiche donc 48.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le type de retour est un pointeur de fonction.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le type de retour est un pointeur de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias `R` pour le pointeur de fonction que l'on souhaite renvoyer ;

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le type de retour est un pointeur de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias `R` pour le pointeur de fonction que l'on souhaite renvoyer ;
2. on définit la fonction souhaitée, dont le type de retour est `R`.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le type de retour est un pointeur de fonction.

Pour cela, on procède en deux étapes :

1. on déclare un type alias `R` pour le pointeur de fonction que l'on souhaite renvoyer ;
2. on définit la fonction souhaitée, dont le type de retour est `R`.

La syntaxe plus directe

```
R (*FCT(T1 ARG1, ..., TN ARGN))(R1, ..., RM) {
    ...
}
```

permet de définir directement une fonction `FCT` de signature `(T1, ..., TN)` renvoyant l'adresse d'une fonction de type de retour `R` et de signature `(R1, ..., RM)`. Cependant, le code devient illisible.

Renvoi d'un pointeur de fonction

Exemple : opération aléatoire sur des entiers.

```
/* Definition des operations */
int add(int a, int b) {
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}
int mod(int a, int b) {
    return a % b;
}
```

```
/* Type de retour */
typedef int (*opb)(int, int);

opb op_alea() {
    opb tab[3];
    tab[0] = &add;
    tab[1] = &sub;
    tab[2] = &mod;
    return tab[rand() % 3];
}
```

Renvoi d'un pointeur de fonction

Exemple : opération aléatoire sur des entiers.

```
/* Definition des operations */
int add(int a, int b) {
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}
int mod(int a, int b) {
    return a % b;
}
```

```
/* Type de retour */
typedef int (*opb)(int, int);

opb op_alea() {
    opb tab[3];
    tab[0] = &add;
    tab[1] = &sub;
    tab[2] = &mod;
    return tab[rand() % 3];
}
```

On peut utiliser `op_alea` de la manière suivante :

```
int n;
n = op_alea()(3, 4);
```

Ceci affecte, de manière aléatoire, `7`, `-1` ou `3` à `n`.

Plan

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite générique si elle peut représenter des données dont le type n'est pas fixé.

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite générique si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- une liste dont les éléments sont d'un type non fixé ;

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite générique si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- une liste dont les éléments sont d'un type non fixé ;
- un arbre binaire dont les éléments sont d'un type non fixé ;

Principe de généricité

Une **fonction** est dite générique si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- une fonction qui teste si deux valeurs sont égales ;
- une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite générique si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- une liste dont les éléments sont d'un type non fixé ;
- un arbre binaire dont les éléments sont d'un type non fixé ;
- un tableau dont les éléments sont d'un type non fixé.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé type pointeur générique.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé type pointeur générique.

Pour convertir un pointeur générique `ptr_g` vers un pointeur d'un type connu `T`, on utilise l'**opérateur de coercition**

```
(T *) ptr_g.
```

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé type pointeur générique.

Pour convertir un pointeur générique `ptr_g` vers un pointeur d'un type connu `T`, on utilise l'**opérateur de coercition**

```
(T *) ptr_g.
```

Avant de pouvoir interpréter (c.-à-d. déréférencer) la valeur située à une adresse spécifiée par un pointeur générique, **le convertir est primordial**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {




}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;




}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;




}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)



}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i])
            return 0;

}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i])
            return 0;
    return 1;
}
```

La fonction ega est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i])
            return 0;
    return 1;
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

On l'utilise de la manière suivante :

```
ega(sizeof(T), &t1, &t2)
```

pour comparer deux variables `t1` et `t2` de type `T`.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {




}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;




}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {


    }
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {
        aff_elt(tab[i]);

    }
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {
        aff_elt(tab[i]);
        printf(" ");
    }
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {



}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;


}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);

}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}

/* Version raccourcie. */
void aff_int(void *x) {
    printf("%d", *((int *) x));
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}

/* Version raccourcie. */
void aff_int(void *x) {
    printf("%d", *((int *) x));
}
...
aff_tab((void **) tab, 13, &aff_int);
```

Fonction générique d'affichage de tableau

Pour afficher un tableau `tab` de taille `23` de pointeurs sur des variables de **type structuré** `Date` :

```
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;
...
void aff_date(void *d) {
    Date dd;
    dd = *((Date *) d);
    printf("%d-%d-%d", dd.jour, dd.mois, dd.annee);
}
...
aff_tab((void **) tab, 23, &aff_date);
```

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {
    struct _Cellule *suiv;
    void *e;
} Cellule;

typedef Cellule *Liste;
```

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {
    struct _Cellule *suiv;
    void *e;
} Cellule;

typedef Cellule *Liste;
```

Le type `Liste` permet ainsi de représenter des listes génériques.

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {
    struct _Cellule *suiv;
    void *e;
} Cellule;

typedef Cellule *Liste;
```

Le type `Liste` permet ainsi de représenter des listes génériques.

C'est une structure de donnée générique car le **type des éléments** que les futures listes pourront contenir **n'est pas connu lors de l'écriture de la fonction**.

Listes génériques

La fonction

```
void aff_lst(Liste lst, void (*aff_elt)(void *)) {
    Cellule *x;

    assert(lst != NULL);
    assert(aff_elt != NULL);

    for (x = lst ; x != NULL ; x = x->suiv) {
        aff_elt(x->e);
        printf(" ");
    }
}
```

est une fonction générique pour l'affichage des éléments d'une liste générique.

Listes génériques

On l'utilise de la manière suivante (dans le cas ici d'une liste d'entiers).

```
void aff_int(void *e) {
    printf("%d", *((int *) e));
}
...
Liste lst;
int a, b, c;
a = 3; b = 14; c = 414;

lst = (Cellule *) malloc(sizeof(Cellule));
lst->e = &a;
lst->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->e = &b;
lst->suiv->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->suiv->e = &c;
lst->suiv->suiv->suiv = NULL;
aff_lst(lst, &aff_int);
```

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`
4. `void *max(Liste lst, int (*est_inf)(void *, void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques. Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`
4. `void *max(Liste lst, int (*est_inf)(void *, void *));`

Les cas 3 et 4 supposent que les éléments représentés par les listes sont comparables au moyen d'une fonction `est_inf` à fournir.

Plan

Plan

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**.

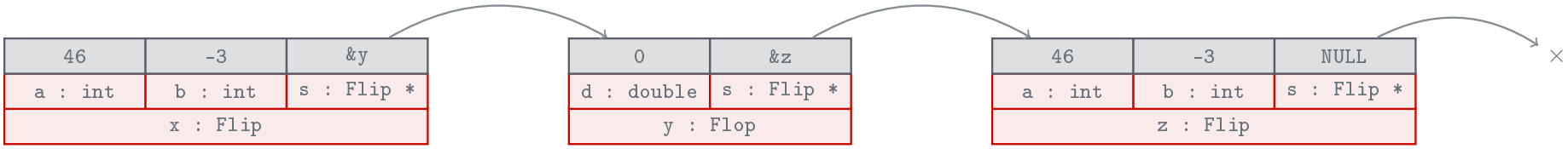
Le système ne peut pas connaître pas la taille de ce champ.

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

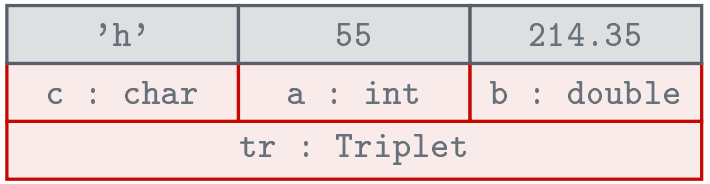
Il est possible d'initialiser les champs d'une variable d'un type structuré au moment de sa déclaration.

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

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 :

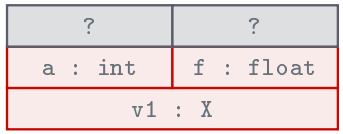
Affectation de variables d'un type structuré

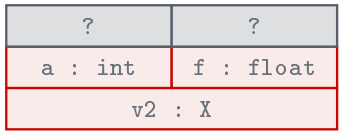
Considérons le code

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

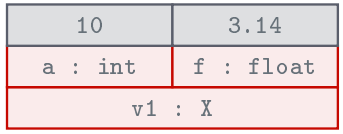
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

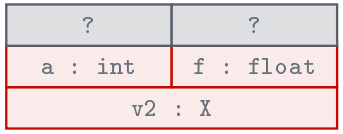
l. 6 :





l. 8 :





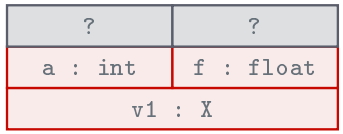
Affectation de variables d'un type structuré

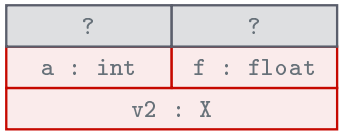
Considérons le code

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

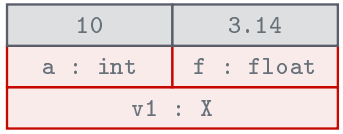
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

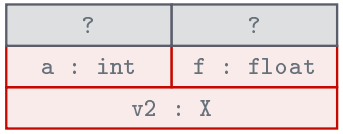
l. 6 :



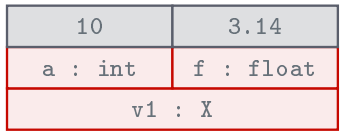


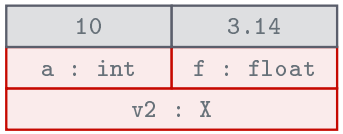
l. 8 :





l. 9 :





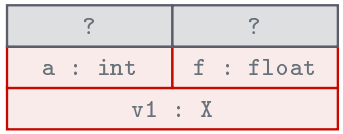
Affectation de variables d'un type structuré

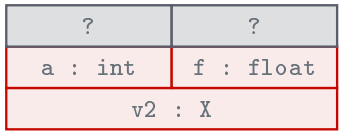
Considérons le code

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

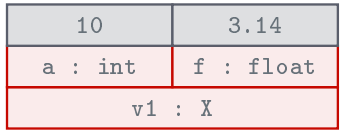
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

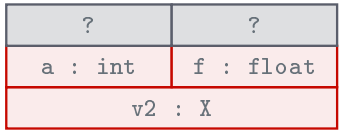
l. 6 :



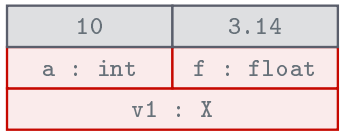


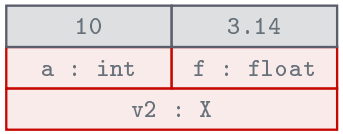
l. 8 :



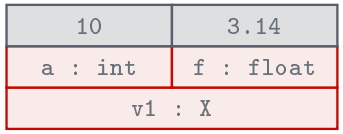


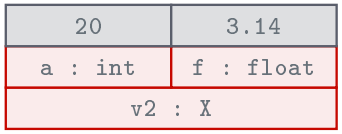
l. 9 :





l. 10 :





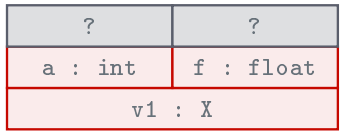
Affectation de variables d'un type structuré

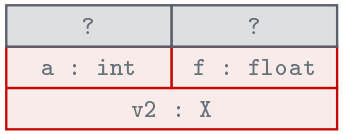
Considérons le code

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

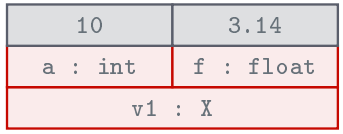
```
6  X v1, v2;
7  v1.a = 10;
8  v1.f = 3.14;
9  v2 = v1;
10 v2.a = 20;
```

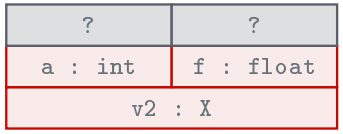
l. 6 :



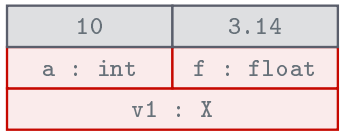


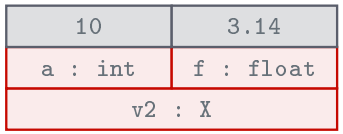
l. 8 :



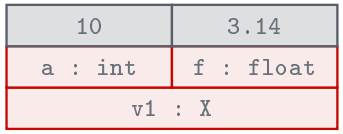


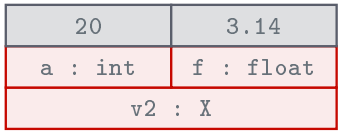
l. 9 :





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';
```

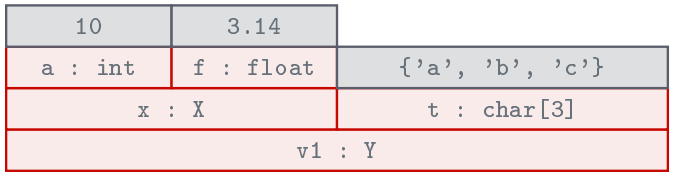
Affectation de variables d'un type structuré

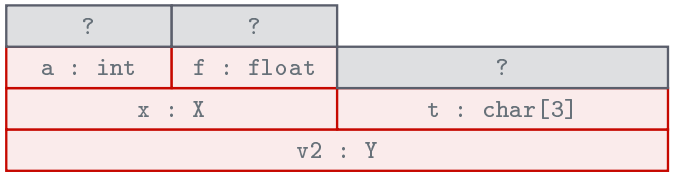
Considérons le code

```
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 :





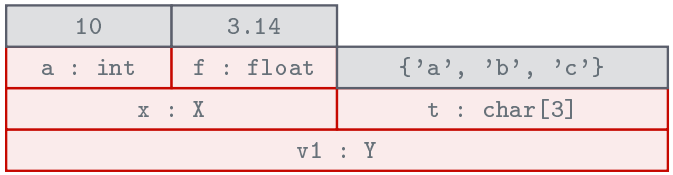
Affectation de variables d'un type structuré

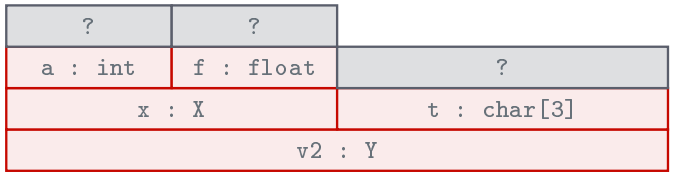
Considérons le code

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

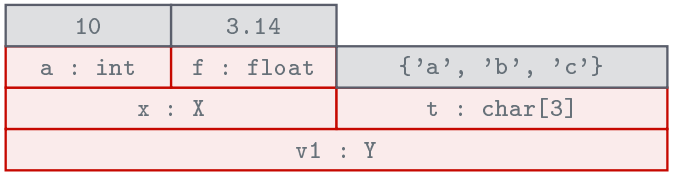
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

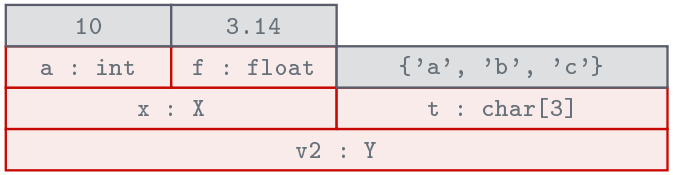
l. 9 :





l. 10 :





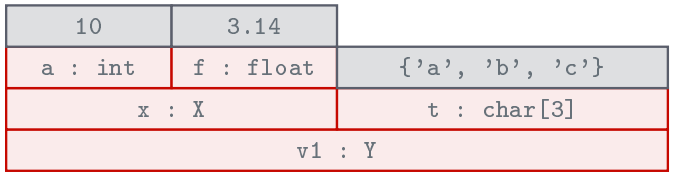
Affectation de variables d'un type structuré

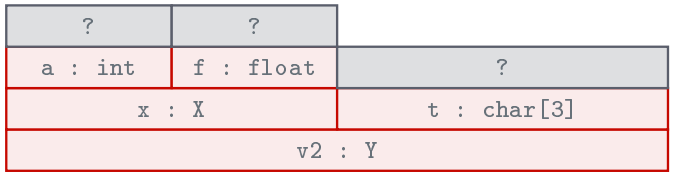
Considérons le code

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

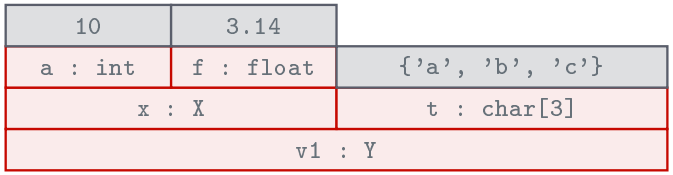
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

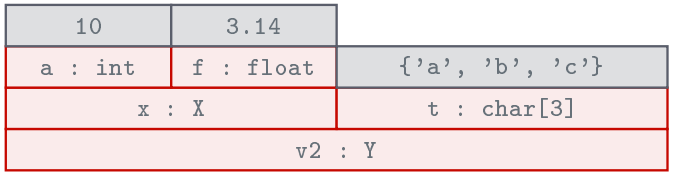
l. 9 :



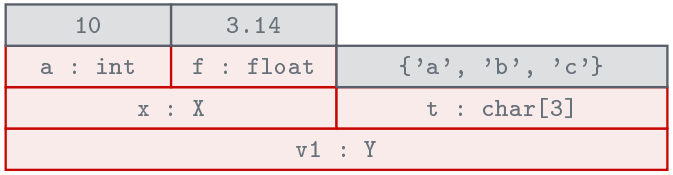


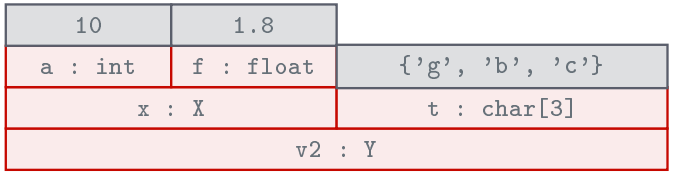
l. 10 :





l. 12 :





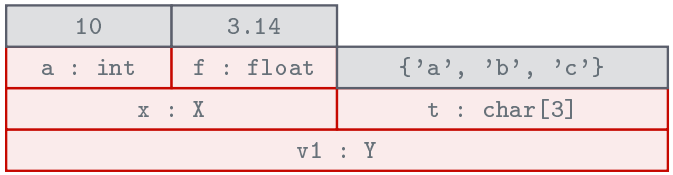
Affectation de variables d'un type structuré

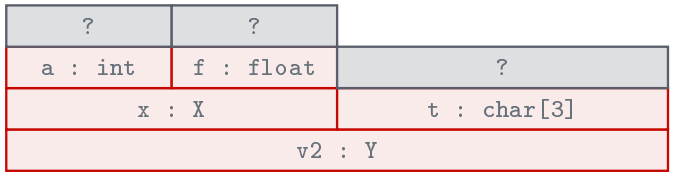
Considérons le code

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

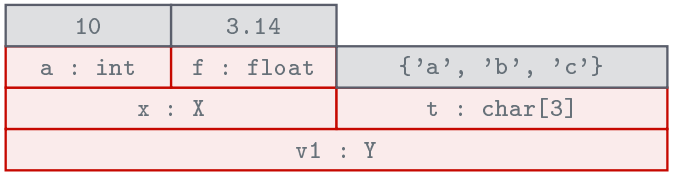
```
7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

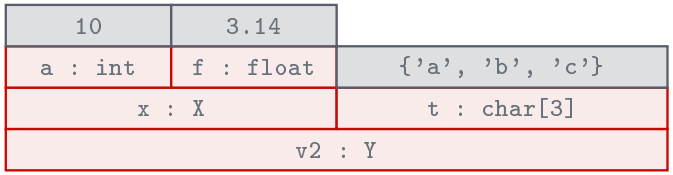
l. 9 :



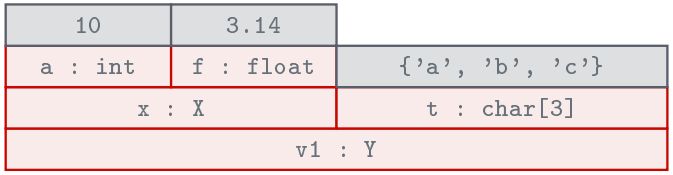


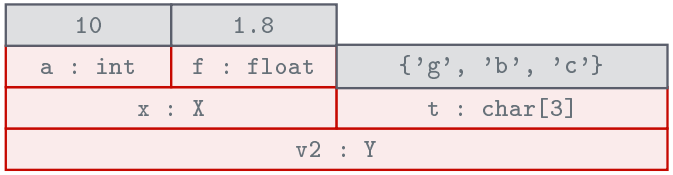
l. 10 :





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

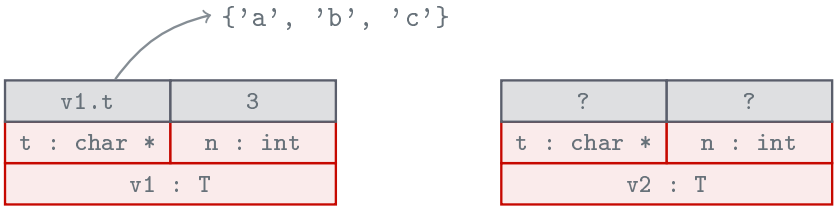
Considérons le code

```
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

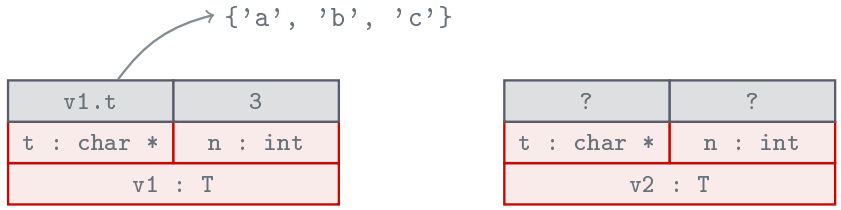
Considérons le code

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

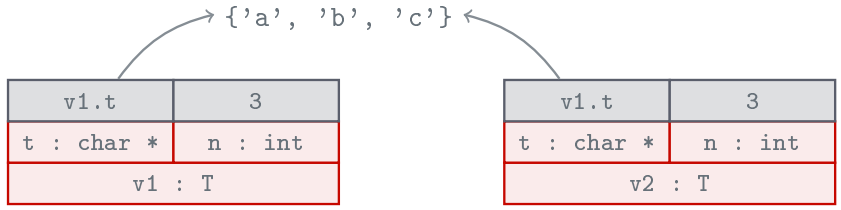
```
6  T v1, v2;
7  v1.t = malloc(3);
8  v1.n = 3;
9  v1.t[0] = 'a';
10 v1.t[1] = 'b';
```

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

l. 11 :



l. 12 :

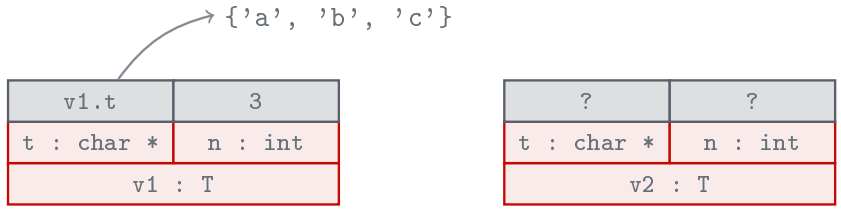


Affectation de variables d'un type structuré

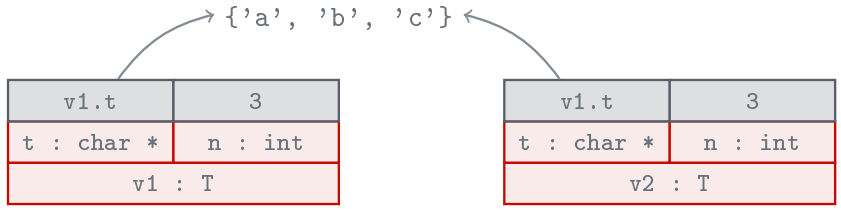
Considérons le code

```
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';
```

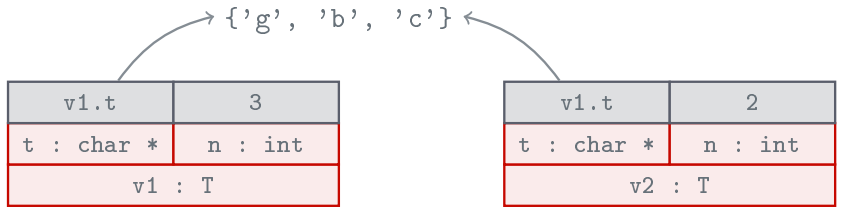
l. 11 :



l. 12 :



l. 14 :



Affectation de variables d'un type structuré

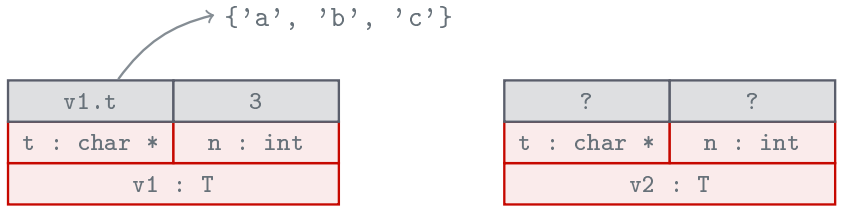
Considérons le code

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

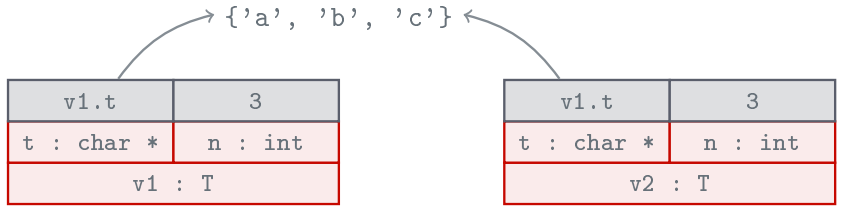
```
6  T v1, v2;
7  v1.t = malloc(3);
8  v1.n = 3;
9  v1.t[0] = 'a';
10 v1.t[1] = 'b';
```

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

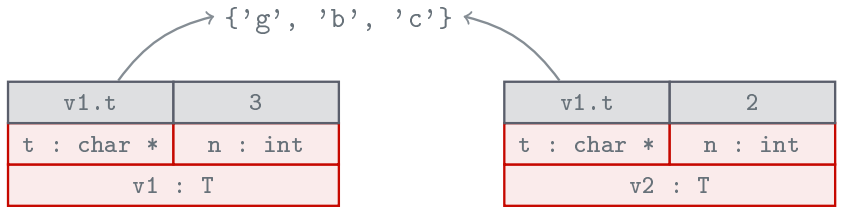
l. 11 :



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

Renvoi d'une variable d'un type structuré

Le code

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

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

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

```c
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).