

Objectifs

Modificateurs de
types

Type énuméré

Tableaux

typedef

Structures

Autres types

Programmation « orientée système »

LANGAGE C – TYPES AVANCÉS

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours d'aujourd'hui

Présenter des **types de données plus avancées** (que les types élémentaires) :

- ▶ Types élémentaires avancés
- ▶ Tableaux (de taille fixe)
- ▶ Alias de types
- ▶ Structures

Rencontre du 4^e type

À ce stade du cours, la représentation des **données** se réduit aux types élémentaires **int**, **double** et **char**.

Ils permettent de représenter, dans des **variables**, des concepts simples du monde modélisé dans le programme :
dimensions, sommes, tailles, expressions logiques, ...

Cependant, de nombreuses données **plus sophistiquées** ne se réduisent pas à un objet informatique élémentaire.

- ✎ un langage de programmation évolué doit donc fournir le moyen de **composer les types élémentaires** pour construire des types plus complexes, les **types composés**.

(Rappel : en C, pas d'objet !)

Exemples de données structurées

Âge
20
35
26
38
22

Nom	Taille	Âge	Sexe
Dupond	1.75	41	M
Dupont	1.75	42	M
Durand	1.85	26	F
Dugenou	1.70	38	oui
Pahut	1.63	22	F

- ☞ tableaux, structures de données hétérogènes (par exemple, « un enregistrement » dans la liste ci-dessus), chaînes de caractères (par exemple, le « nom »), etc...

Types élémentaires « avancés »

différent de Java !

Avant d'examiner les types composés, signalons qu'il existe aussi d'autres types élémentaires, dérivés des types élémentaires présentés.

Trois **modificateurs** peuvent être utilisés :

- ▶ pour les `int` et les `double`, on peut demander d'avoir une *plus grande précision* de représentation à l'aide du modificateur `long`.

Exemple : `long int nb_etoiles;`

- ▶ pour les `int`, on peut aussi demander d'avoir une *moins grande précision* de représentation à l'aide du modificateur `short`.

Exemple : `short int nb_cantons;`

- ▶ pour les `int` (et les `char`), on peut demander de travailler avec des données *positives*, à l'aide du modificateur `unsigned`.

Exemple : `unsigned int nb_cacahouetes;`

On peut bien sûr combiner :

```
unsigned long int nb_etoiles;
```

```
unsigned short int nb_cantons;
```



Types élémentaires « avancés »



Depuis C99, il existe également les types :

- ▶ `long long int`
- ▶ `double complex` et `double imaginary`
(définis dans `complex.h`)
- ▶ `bool` (défini dans `stdbool.h`)
- ▶ `int8_t`, `uint8_t`, ..., `int64_t`, `uint64_t`
(définis dans `stdint.h`)

Types élémentaires « avancés »

En C, la taille des types n'est pas spécifiée dans la norme.

Seules indications :

- ▶ le plus petit type est `char`
- ▶ les inégalités suivantes sont toujours vérifiées sur les tailles mémoires :
 $\text{char} \leq \text{short int} \leq \text{int} \leq \text{long int}$
 $\text{double} \leq \text{long double}$

Cependant, les tailles généralement utilisées sur une architecture 32 bits sont

8 bits pour les `char`
16 bits pour les `short int`
32 bits pour les `long int`

Mais attention ! Cela peut changer d'une architecture à l'autre !

☞ Ne jamais considérer ces valeurs comme absolues (portabilité !) !

Types élémentaires « avancés »

Ces choix typiques conduisent aux bornes suivantes :

type	min.	max.
<code>signed char</code>	-128	127
<code>unsigned char</code>	0	255
<code>short int</code>	-32768	32767
<code>unsigned short int</code>	0	65535
<code>long int</code>	-2147483648	2147483647
<code>unsigned long int</code>	0	4294967295

type	min. (valeur absolue)	max.	précision
<code>double</code>	2.22507e-308	1.79769e+308	2.22045e-16
<code>long double</code>	3.3621e-4932	1.18973e+4932	1.0842e-19

Note : « *précision* » correspond au plus petit nombre x tel que $1 + x \neq 1$.

Types élémentaires « avancés »

qui sont définies dans `limits.h` :

type	min.	max.
<code>signed char</code>	<code>SCHAR_MIN</code>	<code>SCHAR_MAX</code>
<code>unsigned char</code>	<code>0</code>	<code>UCHAR_MAX</code>
<code>short int</code>	<code>SHRT_MIN</code>	<code>SHRT_MAX</code>
<code>unsigned short int</code>	<code>0</code>	<code>USHRT_MAX</code>
<code>long int</code>	<code>LONG_MIN</code>	<code>LONG_MAX</code>
<code>unsigned long int</code>	<code>0</code>	<code>ULONG_MAX</code>

et dans `float.h` :

type	min. (valeur absolue)	max.	précision
<code>double</code>	<code>DBL_MIN</code>	<code>DBL_MAX</code>	<code>DBL_EPSILON</code>
<code>long double</code>	<code>LDBL_MIN</code>	<code>LDBL_MAX</code>	<code>LDBL_EPSILON</code>

Note : « *précision* » correspond au plus petit nombre x tel que $1 + x \neq 1$.

Connaissez-vous bien votre arithmétique ?

comme en Java

(ceci est **aussi** valable en Java !)

Que pensez-vous du code suivant (pas de problème sur `MAX`, qui est bien ≥ 1) ?

```
int index = demander_nombre();  
if (index < 0)    { index = -index; }  
if (index >= MAX) { index = MAX-1; }  
utilisation(tableau[index]);
```

Connaissez-vous bien votre arithmétique ?

comme en Java

(ceci est **aussi** valable en Java !)

Et de ce code ?

```
int i;  
...  
if ( abs(i) < 0 ) { ... }
```

ou de ce code ?

```
int i;  
...  
if ( i == -i ) { ... }
```

Quels sont les `int x` tel que `x == -x` ?

Connaissez-vous bien votre arithmétique ?

comme en Java

(ceci est **aussi** valable en Java !)

Quels sont les `int x` tel que `x == -x` ?

0 **ET** -2147483648 (pour un `int` sur 32 bits), ou `INT_MIN` si vous préférez.
(pour un entier quelconque :

```
~((unsigned)(~0) >> 1)  
)
```

Type énuméré

presque comme en Java

En C, il est également possible de donner des noms aux valeurs de types **énumérés**, comme par exemple la liste des **couleurs**, la liste des **cantons**, etc...

Cela permet d'utiliser ensuite ces valeurs *sans* avoir à *se préoccuper* de leur *codage* effectif.

Ceci se fait à l'aide du mot clé `enum`. Pour déclarer un type énuméré, la syntaxe est la suivante :

```
enum Type { valeur1, valeur2, ... };
```

Par exemple :

```
enum CantonRomand { Vaud, Valais, Geneve, Neuchatel,  
                    Fribourg, Jura };
```

Type énuméré (2)

On peut alors ensuite utiliser simplement ces valeurs comme pour un type entier :

```
enum CantonRomand moncanton = Vaud;  
...  
moncanton = Valais;  
...  
switch (moncanton) {  
    case Valais: ...; break;  
    case Vaud:   ...; break;  
}
```

Type énuméré (2)

On peut même les utiliser comme entiers, sachant que la convention utilisée est que la première valeur énumérée (`Vaud` dans l'exemple précédent) correspond à `0`.

On pourrait alors par exemple faire :

```
int const NB_CANTONS_ROMANDS = Jura+1;
```

ou encore

```
for (i = Vaud; i <= Jura; ++i) ...
```

Remarque : on peut aussi les utiliser pour indexer des tableaux (les tableaux sont présentés plus loin) :

```
population[moncanton] = 616;
```

Note : en C, la conversion `int` vers `enum` est aussi possible.
En C++, elle est interdite (implicitement en tout cas).

Les types composés : les tableaux

presque comme en Java

Une première façon de *composer les types élémentaires* est de faire des **tableaux**.

Les tableaux en C sont des types composés **homogènes**, c'est-à-dire constitués d'éléments qui sont tous du **même type**.

On pourra donc définir des tableaux d'**int**, de **double**, de **char**,
... mais aussi de types composés, par exemple des **tableaux de tableaux**

Dans un premier temps, nous allons nous intéresser aux
tableaux **de taille fixe**

Les tableaux de *taille variable* **n'existent pas en C**.

Nous verrons comment y remédier avec les **pointeurs** et l'**allocation dynamique de mémoire**.

Les différentes sortes de tableaux

Il existe en général quatre sortes de tableaux :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	1.	2.
	non	3.	4.

Exemples :

1. gestion des notes des étudiants d'un cours (inscriptions/désinscriptions)
2. (rare) informations sur les communes/cantons/départements dans une région (connu au départ mais pourrait changer)
3. tableau des scores des participants à un jeu (une fois le nombre de joueurs connus, il ne change plus)
4. vecteur dans l'espace : tableau de 3 nombres réels

Les différentes sortes de tableaux

Il existe en général quatre sortes de tableaux :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	1.	2.
	non	3.	4.

Remarques :

- ▶ avec le premier type de tableau (1.), on peut faire tous les autres
- ▶ pratiquement aucun langage de programmation n'offre les 4 variantes

Les tableaux en C

En C, on a :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	—	—
	non	(C99) VLA	<i>type</i> [N]

Pour rappel en Java, on utilise :

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	ArrayList	
	non	<i>type</i> []	

Déclaration d'un tableau

comme en Java

La syntaxe de déclaration d'un tableau de taille fixe est :

type identificateur[taille];

où *type* est le type des éléments constitutifs du tableau,

identificateur est le nom du tableau

et *taille* est le nombre d'éléments que contient le tableau. Ce nombre doit être **connu à l'avance**

Exemples :

```
int age[5];
```

correspond à la déclaration d'un **tableau de 5 entiers**.

```
size_t const NB_CANTONS = 26;  
double superficie[NB_CANTONS];
```

correspond à la déclaration d'un tableau de 26 **double**.

Déclaration d'un tableau (2)

Conseil : N'écrivez jamais explicitement la taille d'un tableau (valeur littérale)

```
int age[5];
```

mais préférez mettre sa taille **dans une constante** [norme C99] :

```
size_t const NB_CANTONS = 26;  
double superficie[NB_CANTONS];
```

ou (en C quelconque) **dans une macro**
(on verra les macros plus en détails plus tard)

```
#define NB_CANTONS 26  
double superficie[NB_CANTONS];
```

Cela vous permet par la suite de référencer à plusieurs endroit la taille (par exemple dans des boucles) sans recopier la valeur explicite. Ainsi en cas de changement du programme vous n'aurez qu'une seule valeur à modifier : l'initialisation de la constante.

(Note : C99 supporte les « *variable length array* » (VLA). L'exemple `superficie` ci-dessus est en fait une VLA, *même* avec le `const` de `NB_CANTONS` !)



variable length array (VLA)



Depuis C99, il existe des tableaux dont la taille n'est pas connue à la compilation (mais une fois fixée, elle ne change plus) :

```
size_t taille_lue = 0;
...
scanf("%zu", &taille_lue);
double mon_tableau[taille_lue];
...
```

Initialisation d'un tableau

comme en Java

Comme pour les variables de type élémentaire, un tableau de taille fixe peut être initialisé directement lors de sa déclaration.

La syntaxe est similaire à celle des autres types :

```
type identificateur[taille] = { val1, ... , valtaille };
```

(**sauf que**, en C 89, { *val*₁, ... , *val*_{taille} } *n'est pas* une valeur littérale de type tableau (en C99 oui, en C++98 non, mais en C++11 oui !))

Exemple d'initialisation :

```
int age[5] = { 20, 35, 26, 38, 22 };
```



Accès aux éléments d'un tableau

presque comme en Java

Le $i+1^{\text{ème}}$ élément d'un tableau d'identificateur `tablo` est référencé par `tablo[i]`

Attention ! Les indices correspondant aux éléments d'un tableau de taille `taille` varient entre 0 et `taille-1`

et il n'y a **pas de contrôle de débordement !!**

`tablo[10000]`

☞ corruption mémoire / « buffer overflow » (cf cours dans 2 semaines)

Le 1^{er} élément d'un tableau `tablo` précédemment déclaré est donc `tablo[0]` et son 10^{ème} élément est `tablo[9]`

Exemple :

```
#define TAILLE 5
int age[TAILLE];
size_t i;

for(i = 0; i < TAILLE; ++i) {
    printf("Age de l'employé %d?", i+1);
    scanf("%d", &age[i])
}
```


Passage d'un tableau à une fonction (1)

différent de Java !

La déclaration d'un tableau en argument d'une fonction s'écrit comme pour n'importe quel autre type :

```
int f(double tableau[TAILLE]);
```

On peut par contre omettre de spécifier la taille
(qui en fait ne sert absolument à rien ici) :

```
int f(double tableau[]);
```

Dans tous les cas, la fonction `f` prend comme argument un **pointeur**, et en particulier ne s'intéresse pas à la taille effectivement utilisée !! :

```
int f(double* tableau);
```

Passage d'un tableau à une fonction (2)

Conseil : Veillez à ce que la taille du tableau soit connue de la fonction, par exemple en faisant :

```
int f(double tableau[], size_t const taille);
```

ou en utilisant une macro (`#define`).



Attention ! Un tableau **ne** peut **pas** être un type de retour pour une fonction.

👉 voir les pointeurs



Passage d'un tableau à une fonction (3)

Attention ! Il faut avoir conscience que le passage d'un tableau à une fonction se fait toujours **par référence**, bien que ce ne soit pas explicitement marqué par le signe *****.

Exemple :

```
#define TAILLE 3
void g(int tab[], size_t const taille) {
    size_t i;
    for (i = 0; i < taille; ++i) tab[i] = i+1;
}
int main(void) {
    size_t i;
    int t[TAILLE] = { 0, 0, 0 };
    g(t, TAILLE);
    for (i = 0; i < TAILLE; ++i)
        printf("%d ", t[i]);
    return 0;
}
```

👉 affiche 1 2 3
et non pas 0 0 0

...donc l'appel à la fonction **g** **modifie les éléments** du tableau **tab**.

Si vous voulez évitez les effets de bord, ajoutez **const**

Tableaux à plusieurs dimensions

comme en Java

Comment déclarer un tableau à plusieurs dimensions ?

👉 On ajoute simplement un niveau de `[]` de plus :

C'est en fait un tableau de tableaux...

Exemples :

```
double rotation[2][2];  
int statistiques[nb_cantons][nb_statistiques];  
double tenseur[3][2][4];  
  
statistique[Vaud][population] = 616000;
```

En faisant une analogie avec les mathématiques, un tableau à une dimension représente donc un **vecteur**, un tableau à deux dimensions une **matrice** et un tableau de plus de deux dimensions un **tenseur**.

Tableaux à plusieurs dimensions

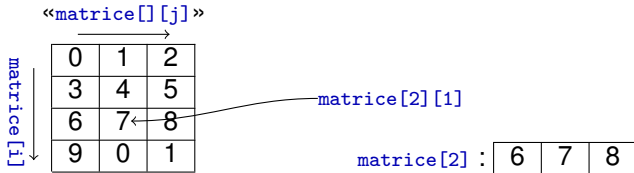
comme en Java

Les tableaux multidimensionnels peuvent également être initialisés lors de leur déclaration.

Il faut bien sûr spécifier autant de valeurs que les dimensions et ceci pour chacune des dimensions.

Exemple :

```
int matrice[4][3] = {  
    { 0, 1, 2 },  
    { 3, 4, 5 },  
    { 6, 7, 8 },  
    { 9, 0, 1 }  
};
```





Passage de tableaux à plusieurs dimensions



Attention ! Lors de la déclaration d'un tableau multidimensionnel comme argument, seule la première taille a le droit de ne pas être spécifiée. Toutes les autres *doivent* l'être :

```
void f(int tab[][3][5]);
```

Mais elles ne sont pas plus utilisables que la première dans le corps de la fonction

👉 Conseil : passer **toutes** les tailles comme arguments supplémentaires à la fonction !

```
#define N 3
#define M 3

void f(int tab[][M], size_t nb_lignes, size_t nb_colonnes);
...
int main(void)
{
    int tablo[N][M]; ...
    f(tablo, N, M);
    return 0;
}
```

Attention avec des tableaux en C !

Les tableaux en C :

- ▶ sont toujours passés « par référence »

En fait

```
Type1 f(Type2 t[TAILLE]);
```

est *remplacé* par

```
Type1 f(Type2* t);
```

- ▶ n'ont pas connaissance de leur propre taille, en *aucune* façon
- ▶ ne peuvent pas être manipulés globalement (pas de « = »)
- ▶ ne peuvent pas être retournés par une fonction
- ▶ (C89) ont une syntaxe d'initialisation particulière :

```
{ val1, val2, etc. }
```

 n'est **pas** une valeur

Depuis *C99* (et en C++11, mais pas en C++98), on peut par contre écrire par exemple la valeur littérale (« *compound literals* »)

```
(double[2]){ 1.2, 3.4 }
```

(mais l'intérêt est limité vu qu'on n'a pas l'affectation (de « = »))

- ☞ Beaucoup de ces inconvénients peuvent être contournés en incluant le tableau dans une *structure*...



Les tableaux



déclaration : `type identificateur[taille];`

déclaration/initiaisation :

`type identificateur[taille] = {val1, ... , valtaille};`

Accès aux éléments : `tab[i]`

`i` entre **0** et **taille-1**

Le passage `type1 f(type2 tab[]);` d'un tableau `tab` à une fonction `f` se fait
automatiquement **par référence**
pour éviter les effet de bords :

`type1 f(type2 const tab[]);`

tableau multidimensionnel :

`type identificateur[taille1][taille2];`
`tab[i][j];`

Les tableaux ne peuvent pas être des types de retour pour les fonctions. :-(

Alias de types

différent de Java !

Pour des types composés complexes, dont l'utilisation directe est difficile, on peut utiliser la commande `typedef` pour

définir un nouveau nom de type

Syntaxe : comme une déclaration de variable mais précédée de `typedef`, typiquement : `typedef type alias;`
où *alias* est le nouveau nom de type et *type* un type élémentaire ou composé.

Exemples :

```
typedef unsigned long int Compteur;  
typedef double Matrice[3][3];
```

Un tableau bidimensionnel d'entiers pourra alors être déclaré plus simplement (et plus lisiblement) par :

```
Matrice rotation;
```

Mieux :

```
typedef double Vecteur[3];  
typedef Vecteur Matrice[3];
```

Alias de types

De telles définitions de nouveaux noms de types sont particulièrement utiles, pour :

- ▶ *bien identifier les types* des objets que l'on manipule

Exemple : `typedef int distance;`

- ▶ meilleure identification des « concepts »
si tout est `int`, on ne distingue plus les choux des carottes (e.g. les `distances` des `ages`, des `couleurs`, etc.)
- ▶ changements ultérieurs de types plus faciles
(e.g. toutes les distances deviennent des `double`)

- ▶ les *arguments de fonctions*

- ▶ Écriture plus claire, plus compacte et plus systématique

Exemple :

```
typedef double Vecteur[N];  
double produit_scalaire(Vecteur, Vecteur);
```

- ▶ les déclarations de *tableaux*

- ▶ améliore également la lecture, l'écriture et la manipulation

Exemples :

```
#define N 3  
typedef double Vecteur[N];  
typedef Vecteur Matrice[N];
```

Données Structurées

différent de Java !

Comme vu précédemment, un programme peut avoir à représenter des **données structurées**, par exemple :

Âge
20
35
26
38
22

Nom	Taille	Âge	Sexe
Dupond	1.75	41	M
Dupont	1.75	42	M
Durand	1.85	26	F
Dugenou	1.70	38	oui
Pahut	1.63	22	F

Les tableaux permettent de représenter des structures de données **homogènes**, c'est-à-dire des listes constituées d'éléments qui sont tous du **même type**.

Exemple : `unsigned short int ages[5];`

QUID des données non homogènes ?

☞ On les homogénéise dans un type composé : les **structures**

Les structures

différent de Java !

La seconde façon de composer les types élémentaires est donc de créer des **structures** regroupant des types **hétérogènes**.

La syntaxe pour déclarer un type « structure » est la suivante :

```
struct Nom_du_type {  
    type1 identificateur1 ;  
    type2 identificateur2 ;  
    ...  
};
```

où

Nom_du_type est le nom que vous souhaitez donner à votre type structuré,
et les

type; identificateur; sont les **déclarations des types** et **identificateurs** des **champs**
de la structure.

Déclaration d'une structure

Exemples :

```
struct Personne {  
    char nom[TAILLE_MAX_NOM];  
    double taille;  
    int age;  
    char sexe;  
};
```

déclare un nouveau type, `Personne`, comme une `structure` composée de quatre **champs** : un de type `char[]`, un autre de type `double`, un troisième de type `int` et un dernier de type `char`.

Autre exemple :

```
struct Complexe {  
    double x;  
    double y;  
};
```

Déclaration d'une structure (2)

Note : Les types des champs d'une structure peuvent aussi être des **types composés**, par exemple des tableaux ou des structures.

Exemple :

```
struct Simple {  
    int souschamp1;  
    double souschamp2;  
};  
  
struct Compliquee {  
    double champ1[3];  
    int champ2;  
    struct Simple champ3;  
};
```

Déclaration d'une structure (3)

Une fois le type de la structure déclarée, on peut utiliser son nom, précédé du mot **struct**, comme tout autre type pour déclarer des variables :

```
struct Nom_du_type nom_de_la_variable;
```

Exemples :

```
struct Personne {  
    char nom[TAILLE_MAX_NOM];  
    double taille;  
    int age;  
    char sexe;  
};
```

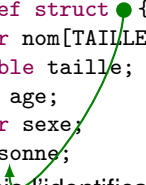
```
struct Personne untel;
```

```
struct Complexe {  
    double x;  
    double y;  
};  
  
struct Complexe z;
```

Utilisation de typedef

La façon la plus pratique de définir un type « structure » est sûrement d'utiliser conjointement `typedef` :

```
typedef struct {  
    char nom[TAILLE_MAX_NOM] ;  
    double taille;  
    int age;  
    char sexe;  
} Personne;
```



associe l'identificateur `Personne` au type `struct` défini.

Une structure définie de cette façon pourra alors être utilisée **sans** avoir besoin d'**ajouter** « `struct` » avant :

`Personne untel;`

`Complexe z;`

Note : on peut aussi faire cela (utiliser `typedef`) pour les types énumérés ; de la même façon (déplacement du nom de type).

Initialisation d'une structure

Les structures peuvent être initialisées avec la syntaxe suivante :

Type identificateur = { val1, val2, ...};

où chaque *val_i* est une valeur du type du champs correspondant.

Exemple :

```
typedef struct {  
    char nom[TAILLE_MAX_NOM];  
    double taille;  
    int age;  
    char sexe;  
} Personne;
```

```
Personne untel = { "Dupontel", 1.75, 20, 'M' };
```

Attention ! En C 89, { *val1, val2, ...* }; n'est **pas** une valeur littérale de type **struct**

(depuis C99, oui, avec casting ; en C++98 non, mais en C++11 oui !)



Accès aux champs d'une structure

On peut accéder aux champs d'une structure en utilisant la syntaxe suivante :

structure.champ

Exemples :

```
untel.taille = 1.75;  
  
++(untel.age); /* déjà un an de plus ! */  
  
printf("%c\n", untel.sexe);
```

Exemple complet

```
typedef struct {  
    double taille;  
    int age;  
    char sexe;  
} Personne;  
  
void affiche_personne(Personne p) {  
    printf("taille: %f\n", p.taille);  
    printf("age   : %d\n", p.age);  
    printf("sexe  : %c\n", p.sexe);  
}
```

```
Personne naissance(void) {  
    Personne p;  
  
    puts("Saisie d'une nouvelle personne");  
  
    printf("  Entrez sa taille :");  
    scanf("%lf", &(p.taille));  
  
    printf("  Entrez son age: ");  
    scanf("%d", &(p.age));  
  
    printf("    Homme [M] ou Femme [F] : ");  
    scanf("%c", &(p.sexe));  
  
    return p;  
}
```

Affectation de structures

Une variable de type composé `struct` peut être **directement affectée** par une variable du même type

Exemple :

```
Personne p1 = { "Durand", 1.75, 20, 'M' };  
Personne p2;  
p2 = p1;
```

La valeur de chaque champ de `p1` est affectée au champ correspondant de `p2` (copie profonde)

- ☞ L'instruction `p2=p1` est équivalente à la séquence d'instructions
`p2.nom=p1.nom; p2.taille=p1.taille; p2.age=p1.age; p2.sexe=p1.sexe;`

Retour à l'exemple du début

Les structures sont particulièrement utiles pour les tableaux hétérogènes :

👉 **tableaux de structures**

Exemple :

```
typedef struct {  
    char nom[TAILLE_MAX_NOM];  
    double taille;  
    int age;  
    char sexe;  
} Personne;  
  
Personne personnes[5] = {  
    { "Dupond", 1.75, 41, 'M' },  
    { "Dupont", 1.75, 42, 'M' },  
    { "Durand", 1.85, 26, 'F' },  
    { "Dugenou", 1.70, 38, 'M' },  
    { "Pahut", 1.63, 22, 'F' }  
};
```



Les structures



Déclaration du type correspondant :

```
struct Nom_du_type {  
    type1 champ1 ;  
    type2 champ2 ;  
    ...  
};
```

Déclaration d'une variable :

```
struct Nom_du_type identificateur;
```

Déclaration/Initialisation d'une variable :

```
struct Nom_du_type identificateur = { val1, val2, ...};
```

Accès à un champs donné de la structure :

```
identificateur.champ
```

Affectation globale de structures :

```
identificateur1 = identificateur2
```



Autres modificateurs



Nous avons vu les modificateurs `const`, `long`, `short`, `unsigned` (, `signed`)

Il en existe encore

- `extern` objet déclaré ailleurs. Voir cours n° 10 & 11.
- `static` limitation de portée. Dans une fonction, partage la valeur à tous les appels.
- `register` optimisation : *propose* le stockage dans un registre.
- `restrict` pour les pointeurs uniquement : aucun autre pointeur pointe au même endroit.
- `volatile` supprime toute optimisation liée à cette variable.

(`auto` est le contraire de `static` et est le mode par défaut, donc jamais utilisé.)



union



Si l'on considère les `struct` comme un « ET » entre champs, `union` correspond au « OU » : il permet de regrouper plusieurs façons de voir la même zone mémoire.

Exemple :

```
typedef union {  
    int i;  
    double d;  
} Int_or_Double;  
  
Int_or_Double x;  
  
x.i = 3;  
...  
x.d = 9.87;  
...
```

Le **GROS INCONVÉNIENT** de ce genre de choses est qu'il faut garantir la consistance de l'utilisation : le champs/type utilisé en lecture doit être compatible avec le dernier champs/type affecté !!



union, remarques



En toute rigueur cela ne sert donc à rien car exactement le même rôle peut être joué par un pointeur générique `void*` (voir cours suivants).

Les seuls avantages de `union` pourraient être

1. que l'allocation à la plus grande taille est effectuée automatiquement
2. la lisibilité, pour ceux qui n'aiment pas lire les castings de `void*`.



Bit fields



Les « bit fields » permettent de spécifier bit à bit l'utilisation d'une zone mémoire.

Exemple :

```
typedef struct {  
    unsigned int sign : 1 ;  
    unsigned int exp : 15 ;  
    unsigned int mant : 32 ;  
} FloatNumber;
```

Il pourrait alors dans ce contexte être utile de connaître les opérateurs de manipulation binaire :

$\sim x$ inversion de tous les bits

$x | y$ « OU » bit à bit, $x \wedge y$ pour le « OU exclusif »

$x \& y$ « ET » bit à bit

$x \ll n$ décalage à gauche de n bits

$x \gg n$ décalage à droite de n bits

Attention ! Ne pas confondre $\&\&$ et $\&$: $1 \&\& 2$ s'évalue comme « vrai », $1 \& 2$ vaut 0, qui s'évalue comme « faux » !