

COURS 3

- Boucles (suite et fin)
- Expressions et instructions spéciales (suite et fin)
- Types composés : tableaux, structures et unions

Instruction faire ... tant que

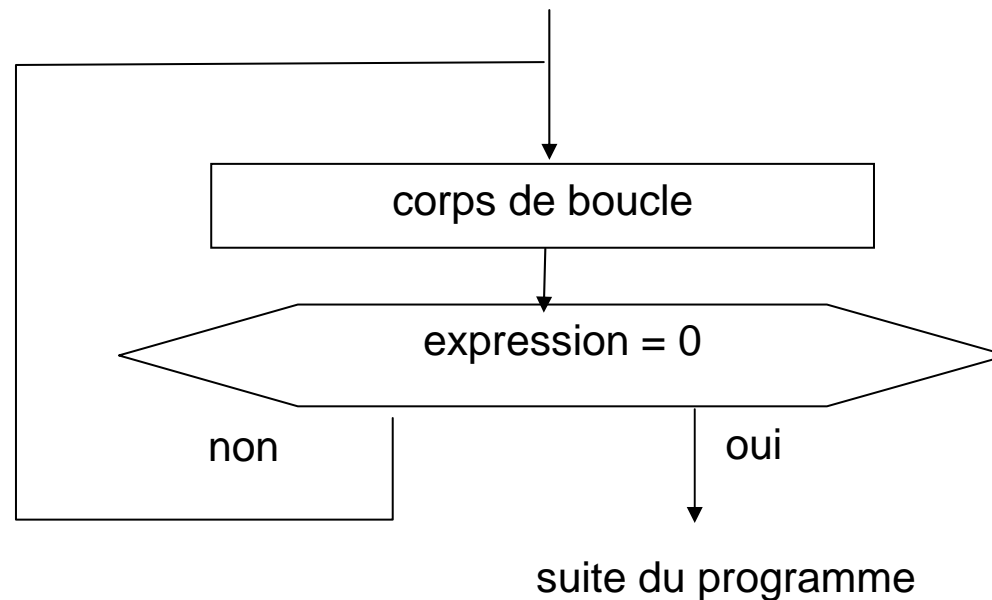
instruction_faire_tantque → **do** *corps_de_boucle* **while** (*expression*);

On exécute les instructions du corps de boucle

Puis on évalue l'expression qui suit le **while**.

Si cette expression est vraie (non nulle), on recommence

Sinon on passe à l'instruction qui suit l'instruction faire ... tant que.



La différence avec le **while** est que le corps de boucle est exécuté au moins une fois.

Remarquer aussi le « ; » en fin d'instruction.

Exemples

```
float N;
do
{
    printf("Introduisez un nombre entre 1 et 10 :");
    scanf("%f", &N);
}
while (N<1 || N>10);
```

```
int n, div;
printf("Entrez le nombre à diviser : ");
scanf("%i", &n);
do
{
    printf("Entrez le diviseur ( 0) : ");
    scanf("%i", &div);
}
while (!div);
printf("%i / %i = %f\n", n, div, (float)n/div);
```

Choix de la structure répétitive

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez **while** ou **for**.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez **do - while**.
- Si le nombre d'exécutions du bloc d'instructions dépend d'une ou de plusieurs variables qui sont modifiées à la fin de chaque répétition, alors utilisez **for**.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie (p.ex aussi longtemps qu'il y a des données dans le fichier d'entrée), alors utilisez **while**.

Le choix entre **for** et **while** n'est souvent qu'une question de préférence ou d'habitudes:

- **for** nous permet de réunir les instructions qui influencent le nombre de répétitions au début de la structure.
- **while** a l'avantage de correspondre plus exactement aux structures d'autres langages de programmation
- **for** a le désavantage de favoriser la programmation de structures surchargées et par la suite illisibles.
- **while** a le désavantage de mener parfois à de longues structures, dans lesquelles il faut chercher pour trouver les instructions qui influencent la condition de répétition.

Instructions de rupture de séquence

instruction_de_rupture → *instruction_break* / *instruction_continue* / *instruction_retour*

instruction_break → **break** ;

- Ne peut être placée qu'à l'intérieur d'un corps de boucle ou d'une instruction d'aiguillage.
- Interrompt l'exécution des instructions de la boucle ou de l'aiguillage
- Donne le contrôle à l'instruction qui suit immédiatement l'instruction répétitive ou l'aiguillage.

instruction_continue → **continue** ;

- Ne peut être placée qu'à l'intérieur d'un corps de boucle
- Interrompt l'itération en cours d'exécution
- Démarre l'itération suivante (et l'évaluation de l'expression de modification des variables de contrôle dans le cas d'une boucle pour) .

instruction_retour → **return** *<expression>* ;

- Ne peut être placée qu'à l'intérieur d'un corps de fonction
- Termine l'exécution de la fonction
- Renvoie la valeur de *expression* comme résultat, après conversion éventuelle du type de *expression* vers le type de la fonction.
- Le contrôle passe ensuite à l'instruction qui suit immédiatement l'appel de la fonction.

Si *expression* est absente, le résultat de la fonction est indéterminé. Cette fonction devrait être de type **void** .

Instruction d'aiguillage

instruction_d_aiguillage → **switch** (*expression*)
 {**case** k_1 *instruction*⁺
 case k_2 *instruction*⁺
 ...
 case k_n *instruction*⁺
 <default *instruction*⁺ }

Les k_i sont des constantes toutes différentes, d'un type compatible avec celui de *expression*.

On évalue *expression* et on le compare à chacun des k_i :

- si la valeur de l'expression est égale à l'un des k_i pour un certain i , on exécute les instructions qui suivent ce k_i .
- Attention, on ne s'arrête pas là : toutes les instructions qui suivent, y compris celles des alternatives qui suivent la $i^{\text{ème}}$ sont exécutées, à moins qu'on ne rencontre une instruction de rupture de séquence auquel cas on saute à l'instruction qui suit l'instruction d'aiguillage (**break**) ou on sort de la fonction qui englobe l'aiguillage (**return**),.
- si la valeur de l'expression est différente de tous les k_i on exécute les instructions qui suivent l'étiquette **default** et on passe à l'instruction qui suit l'instruction d'aiguillage.

Noter que l'alternative **default** est optionnelle. Si elle est absente et dans le cas où la valeur de l'expression est différente de tous les k_i , on passe directement à l'instruction qui suit l'instruction d'aiguillage.

Instruction bloc

instruction_bloc → { *déclaration_de_variable** *instruction** }

L'instruction bloc ne se termine pas sur un « ; ».

Les variables sont locales au bloc et donc visibles dans ce bloc seulement sauf si elles sont **static** ou **extern**,

Expression séquence

$exp_seq \rightarrow exp1 , exp2$

Evaluation de $exp1$ suivie de l'évaluation de $exp2$. N'est utile que si $exp1$ a un effet de bord.

séquence	type	valeur	R/Lvalue	effet de bord
$exp1$	quelconque	$v1$	Rvalue	-
$exp2$	quelconque	$v2$	Rvalue	-
$exp1 , exp2$	celui de $exp2$	$v2$	Rvalue	celui de $exp1$ suivi de celui de $exp2$

Expression conditionnelle

$exp_cond \rightarrow exp ? exp1 : exp2$

conditionnelle	type	valeur	R/Lvalue	effet de bord
exp	quelconque	v	Rvalue	-
$exp1$	types compatibles	$v1$	Rvalue	-
$exp2$		$v2$	Rvalue	-
exp_cond	celui de $exp1$ ou $exp2$	$v1$ si $v = 0$. $v2$ n'est pas calculée $v2$ si $v \neq 0$. $v1$ n'est pas calculée	Rvalue	aucun

Exemple : calculer le plus grand de 2 nombres :

```
MAX = (A > B) ? A : B;
```

Exemple : respecter l'accord du pluriel dans un message calculé :

```
printf("Vous avez %i carte%c \n", N, (N==1) ? ' ' : 's');
```

Attention : les *règles de conversion de types* s'appliquent aussi aux opérateurs conditionnels ? :

Pour N de type **int** et F de type **float**, l'expression

```
(N>0) ? N : F
```

va *toujours* fournir un résultat du type **float**, que N soit plus grand ou plus petit que zéro!

Expressions bit à bit

Ces expressions portent sur le codage interne des opérandes en base 2, bit à bit.

$exp_bin \rightarrow exp1\ op_bin\ exp2\ | \text{décalage} | \text{complément_à_un}$

$op_bin \rightarrow \wedge\ |\ |\ \&$

expr. bin	type	valeur	R/Lvalue	effet de bord
$exp1$	entier	$v1$	Rvalue	-
$exp2$	entier	$v2$	Rvalue	-
$exp1\ \&\ exp2$	le plus général après conversion	« et » logique bit à bit des représentations binaires de $v1$ et $v2$	Rvalue	aucun
$exp1\ \wedge\ exp2$		« ou » logique bit à bit des représentations binaires de $v1$ et $v2$	Rvalue	aucun
$exp1\ \ exp2$		« ou » exclusif bit à bit des représentations binaires de $v1$ et $v2$	Rvalue	aucun

$complément_à_un \rightarrow \sim exp$

complément	type	valeur	R/Lvalue	effet de bord
exp	entier	v	Rvalue	-
$\sim exp$	celui de exp	valeur obtenue en inversant chaque bit de la représentation binaire de v : 0 devient 1 et vice-versa.	Rvalue	aucun

Expressions bit à bit (suite)

décalage $\rightarrow exp1 \gg exp2$ / $exp1 \ll exp$

décalage	type	valeur	R/Lvalue	effet de bord
$exp1$	entier	$v1$	Rvalue	
$exp2$	entier	$v2$	Rvalue	
$exp1 \ll exp2$	celui de $exp1$	valeur obtenue en décalant de $v2$ positions vers la gauche les bits de la représentation binaire de $v1$. Les bits entrants sont des 0. Les bits sortants sont perdus.	Rvalue	aucun
$exp1 \gg exp2$	celui de $exp1$	valeur obtenue en propageant de $v2$ positions vers la droite le bit de gauche de la représentation binaire de $v1$. Les bits sortants sont perdus.	Rvalue	-

Remarque : le traitement des expressions binaires dépend de la plateforme. Leur portabilité n'est donc pas assurée.

7. Types composés

Défini par l'utilisateur

Un objet composé regroupe plusieurs objets plus simples

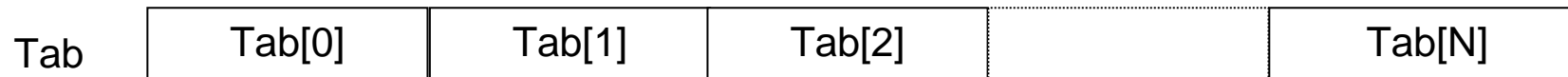
- tous de même type → tableau
- ou de types différents → structures et unions.

On sait désigner directement l'objet composé et accéder individuellement à ses composants.

Tableaux

Un objet de type tableau est le regroupement d'un nombre fini et connu d'autres objets

- Tous les composants (éléments) du tableau sont de même type.
- Ces éléments sont ordonnés et indexés.
- L'accès à ces éléments se fait par cet index.



déclaration_de_tableau → *qualifieur mode nom_de_type nom_de_tableau dimensions*
nom_de_tableau → *identificateur*
dimensions → [*constante_entière*]*

- Dimension : une constante entière N,
- Les éléments du tableau sont indexés de 0 à N – 1 : Tab[0] est le 1^{er} élément et Tab[N-1] le dernier.
- L'expression peut être omise lorsque la déclaration n'implique pas l'allocation effective de mémoire :
 - s'il s'agit de la dernière dimension d'un tableau déclaré comme argument formel d'une fonction
 - si le tableau est déclaré **extern**
- Si un tableau de type T est formé de N composantes et si une composante a besoin de M octets en mémoire, alors le tableau occupera de N*M octets : `N*sizeof(T)` .

Initialisation des tableaux

La valeur à affecter est la liste des valeurs des éléments entre accolades :

```
int Tab[4]= {1,2,3,4} ;
```

Les éléments manquants sont initialisés à 0 ou restent indéterminés si la variable tableau est locale.

L'initialisation permet d'omettre la taille du tableau :

```
int Tab[] = {1,2,3,4} ;
```

déclare un tableau de 4 entiers.

Si le tableau est multidimensionnel, on peut imbriquer les accolades.

Le tableau est rempli ligne par ligne

Les éléments manquants sont remplacés par des 0 ou restent indéterminés

```
int Tab[3][3] = {{1,2},{3,4}} donne :
```

1	2	0
3	4	0
0	0	0

Accès aux éléments d'un tableau

L'accès à un élément d'un tableau est réalisé au moyen de l'opérateur « [] » :

$exp_tableau \rightarrow nom_de_tableau[exp]$

expr. tableau	type	valeur	R/Lvalue	effet de bord
<i>exp</i>	entier	<i>k</i>	Lvalue	-
<i>exp_tableau</i>	T	$k^{ième}$ élément du tableau	Lvalue	-

Exemple : Tab[12]

Exemple : Affichage du contenu d'un tableau

La structure **for** se prête particulièrement bien au travail avec les tableaux.

```
main()
{
    int A[5];
    int I; /* Compteur */
    for (I=0; I<5; I++)
        printf("%d ", A[I]);
    return 0;
    printf("\n");
}
```

- Avant de pouvoir afficher les composantes d'un tableau, il faut évidemment leur affecter des valeurs.
- La deuxième condition dans la structure **for** n'est pas une condition d'arrêt, mais une condition de répétition! Ainsi la commande d'affichage sera répétée *aussi longtemps* que **I** est inférieur à 5. La boucle sera donc bien exécutée pour les indices 0,1,2,3 et 4 !
- L'instruction **printf** doit être informée du type exact des données à afficher. (Ici: **%d** ou **%i** pour des valeurs du type **int**)
- Pour être sûr que les valeurs sont bien séparées lors de l'affichage, il faut inclure au moins un espace dans la chaîne de format. Autres possibilités:

```
printf("%d\t", A[I]); /* tabulateur */
printf("%7d", A[I]); /* format d'affichage */
```

Exemple : initialisation d'un tableau par lecture

```
main()  
{  
    int A[5];  
    int I; /* Compteur */  
    for (I=0; I<5; I++)  
        scanf("%d", &A[I]);  
    return 0;  
}
```

- Comme **scanf** a besoin des adresses des différentes composantes du tableau, il faut faire précéder le terme A[I] par l'opérateur adresse '&'.
- La commande de lecture **scanf** doit être informée du type exact des données à lire. (Ici: %d ou %i pour lire des valeurs du type **int**)

Chaînes de caractères

- Une chaîne de caractères est un tableau de caractères.
- La fin de la chaîne est marquée par le caractère nul « \0 ».
- Les chaînes transmises aux fonctions standards doivent se terminer par « \0 ».
- Une variable chaîne est une constante dont la valeur est l'adresse de son 1^{er} caractère.

```
char *k = "bonjour" ; /* k contient l'adresse d'un caractère */
/* est correcte */
```

```
char q[ ] = "bonjour" ; /* est correcte : "bonjour" est considérée comme un raccourci de :
{ 'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0' } */
```

- Pour un texte de **n** caractères, il faut **n+1** octets : **six** octets pour la chaîne "Hello",
- Les chaînes de caractères constantes sont indiquées entre guillemets. La chaîne de caractères vide est alors: ""
- Dans les chaînes de caractères, nous pouvons utiliser toutes les séquences d'échappement
ici le symbole ' est représenté à l'intérieur d'une liste de caractères par la séquence d'échappement \ ' :
`{ 'L', '\'', 'a', 's', 't', 'u', 'c', 'e', '\0' }`
- Des constantes chaînes séparées par des signes d'espacement (espaces, tabulateurs ou interlignes) dans le texte du programme seront réunies en une seule chaîne constante lors de la compilation:
`"un " "deux"`
`" trois"` sera évalué à `"un deux trois"`
- `'x'` est une constante caractère qui a la valeur numérique 120 dans le code ASCII. Codé sur 1 octet
- `"x"` est un tableau de 2 caractères : la lettre `'x'` et le caractère nul `'\0'`. Codé sur 2 octets

Ordres : Le code ASCII induit l'ordre suivant: ... ,0,1,2, ... ,9, ... ,A,B,C, ... ,Z, ... ,a,b,c, ... ,z, ...

Les symboles spéciaux (' ,+ , - ,/ , { , } , ...) et les lettres accentuées (é ,è ,à ,û , ...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).

- '0' est inférieur à 'Z' noté '0' < 'Z'
- La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
- La chaîne A = "a₁a₂a ... a_p" (p caractères) précède la chaîne B = "b₁b₂ ... b_m" (m caractères) si l'une des deux conditions suivantes est remplie:

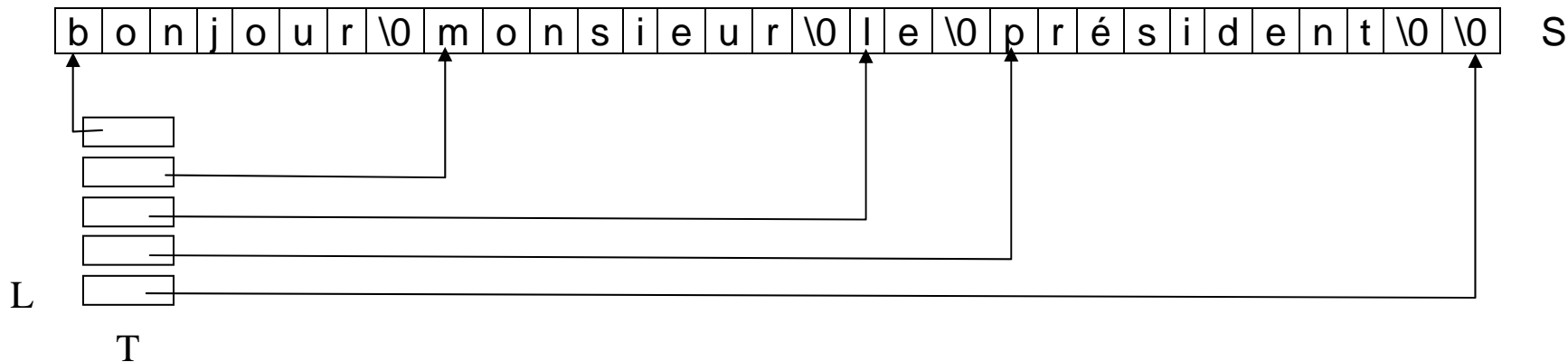
- 'a₁' < 'b₁'
- 'a₁' = 'b₁' et
- "a₂a₃ ... a_p" < "b₂b₃ ... b_m"

"ABC" précède "BCD" car 'A' < 'B'
"ABC" précède "B" car 'A' < 'B'
"Abc" précède "abc" car 'A' < 'a'
"ab" précède "abcd" car "" précède "cd"
" ab" précède "ab" car ' ' < 'a'

- On peut contrôler le type du caractère (chiffre, majuscule, minuscule).
`if (C>='0' && C<='9') printf("Chiffre\n", C);`
`if (C>='A' && C<='Z') printf("Majuscule\n", C);`
`if (C>='a' && C<='z') printf("Minuscule\n", C);`
- On peut convertir des lettres majuscules dans des minuscules: `if (C>='A' && C<='Z') C = C-'A'+'a';`
ou vice-versa: `if (C>='a' && C<='z') C = C-'a'+'A';`

Tableaux multidimensionnels (suite)

Une autre façon de faire est d'utiliser un espace linéaire, où chaque ligne n'occupe que le nombre de caractères strictement nécessaire :



On utilise en plus un tableau de pointeurs qui référence le début de chaque ligne. `L` désigne toujours l'indice de la 1^{ère} ligne libre. Les déclarations correspondantes sont :

```
char S[maxcar] ;  
char *T[N]  
int L=0 ;
```

Le $k^{\text{ème}}$ caractère de la $i^{\text{ème}}$ ligne est toujours référencé par `T[i][k]`.

8. Types composés (suite) : Structures et Unions

Une structure est un nouveau type d'objet, défini par le programmeur comme la juxtaposition d'un nombre fini et connu d'autres objets

- Les composants (champs) de la structure ne sont pas nécessairement de même type
- Ces champs sont nommés.
- L'accès à un champ se fait par son nom.

Déclaration des structures

déclaration_de_structure → **struct** *nom_de_struct* { *champ*⁺ } ;

nom_de_struct → *identificateur*

champ → *déclaration_de_variable*

- Les champs peuvent être de n'importe quel type, y compris des structures.
- Une structure peut donc contenir d'autres structures, sauf elle-même.
- Une structure peut contenir un pointeur vers une structure de même type : données récursives

Une fois la structure déclarée, la création de variables de ce type se fait par :

déclaration_de_variables_struct → **struct** *nom_de_struct* *liste_de_variables* ;

liste_de_variables → *nom_de_variable* ⟨ , *nom_de_variable* ⟩^{*}

→ la répétition de **struct** dans la déclaration de variable de type structure et le « ; » après le dernier champ.

Déclaration des structures (suite)

Exemple 1 : définition de la structure personne, composée de plusieurs champs :

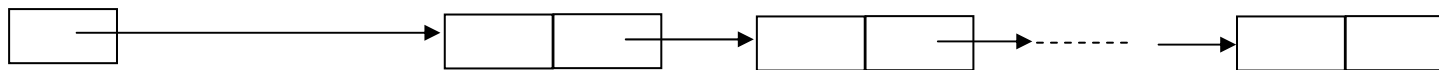
- le nom qui est une chaîne de 80 caractères
- le jour, le mois et l'année de naissance, chacun étant un entier
- le pays de résidence, une chaîne de 50 caractères.

personne	nom	jour	mois	an	résidence
----------	-----	------	------	----	-----------

```
struct personne {char nom [80];
                short jour, mois, an ;
                char residence[50];} ;
```

et les variables : struct personne jean, paul ;

Exemple 2 : coder les maillons d'une liste. Chaque maillon porte une information représentée par un entier et a au plus un seul maillon suivant dans la liste. La liste elle-même est l'adresse du 1^{er} maillon.



```
struct maillon {int info ; struct maillon *suivant ;} ;
```

et la liste : struct maillon *L ;

→ Notez la répétition de **struct** dans la déclaration de variable de type structure et le « ; » après le dernier champ.

Déclaration des structures (suite)

En fait, la déclaration de structure revient à la déclaration d'un nouveau type :

déclaration_de__structure → **typedef** *qualifieur* **struct** *nom_de_struct* { *champ*⁺ } *nom_de_type_struct* ;
nom_de_type_struct → *identificateur*

La création de variables de ce type se fait par :

déclaration_de_variables_struct → *nom_de_type_struct* *liste_de_variables* ;

avec **l'exemple 1** ci-dessus :

```
typedef struct date{short jour, mois,an ;} DATE;
typedef struct personne  {char nom[80];
    DATE date_de_naissance ;
    char residence[50];} PERSONNE;
```

```
et: PERSONNE jean, paul ;
```

Avec **l'exemple 2** :

```
typedef struct maillon
    {int info; struct maillon *suivant;}MAILLON;
typedef MAILLON *liste ;
```

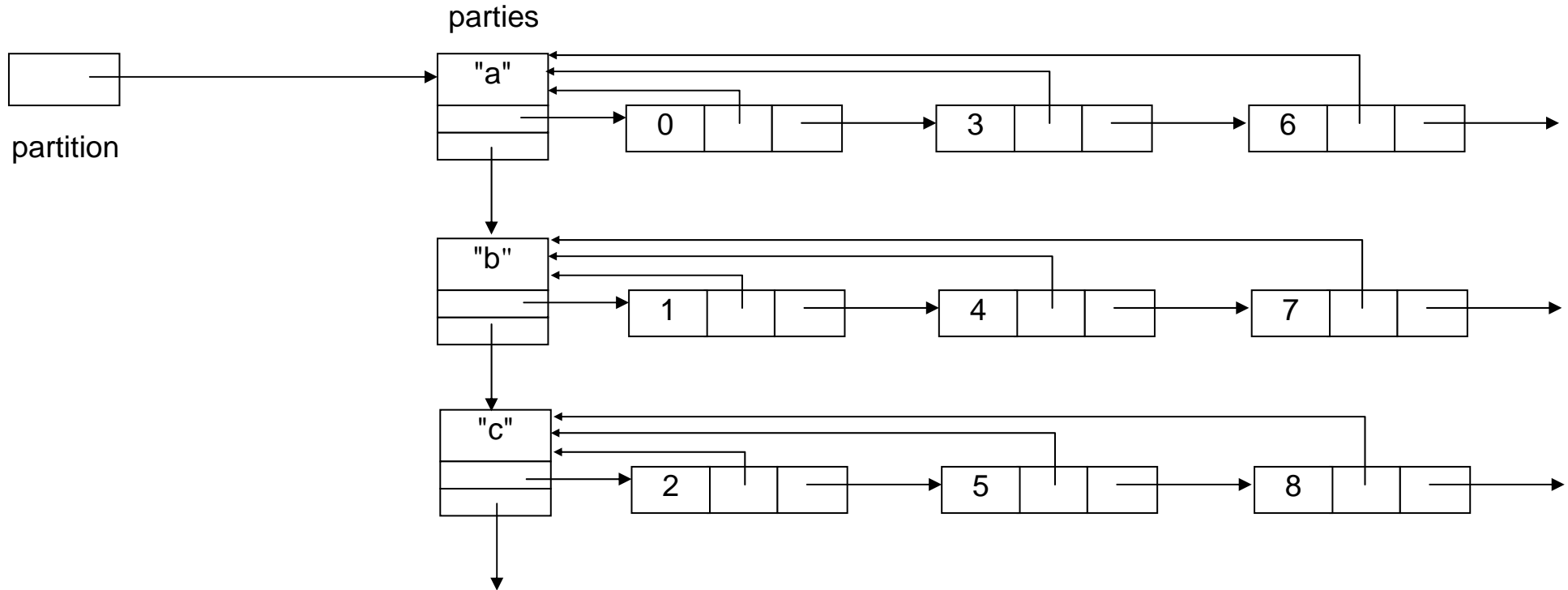
Et la déclaration d'une liste L est :

```
liste L ;
```

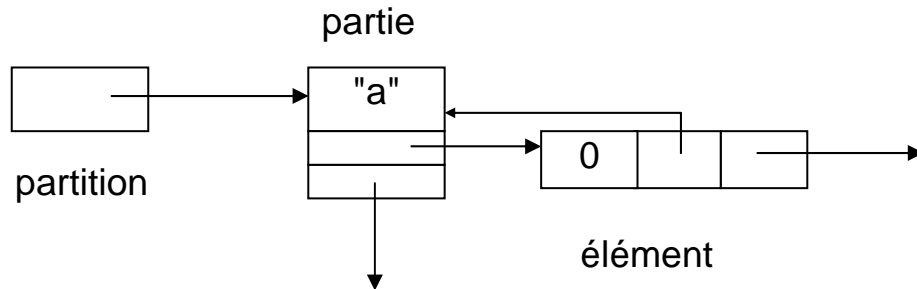
- la distinction entre le nom de la structure et le nom du nouveau type structure défini. C'est presque la même chose, et pour le signaler on garde le même identificateur mais en majuscules.
- La déclaration de variables utilise le nom du type seulement. Le mot-clé **struct** a disparu.

Déclaration des structures (suite)

Voici comment représenter des données qui se font mutuellement référence comme ici des partitions d'un ensemble d'éléments qui eux-mêmes font référence à la partie à laquelle ils appartiennent :



Déclaration des structures (fin)



- partition est un pointeur sur partie
- partie est un triplet : nom de la partie, pointeur sur la liste de ses éléments, pointeur sur la partie suivante
- élément est un triplet : valeur de l'élément, pointeur sur la partie, pointeur sur l'élément suivant.

Les structures partie et élément se référencent l'une l'autre. Heureusement, C permet de déclarer un pointeur vers une structure non encore définie, à condition de préciser que l'objet pointé est bien une structure :

```
typedef struct partie
{char nom[20] ;
 struct element *liste_elements ;
 struct partie *partie_suiv ;}
PARTIE;

typedef struct element
{int valeur ;
 struct partie *appartient_a ;
 struct element *element_suiv ;}
ELEMENT;

typedef PARTIE *PARTITION ;
```

ou:

```
typedef struct partie PARTIE;
typedef struct element ELEMENT;

struct partie {char nom[20];
 ELEMENT *liste_elements;
 PARTIE *partie_suiv;};

struct element {int valeur ;
 PARTIE *appartient_a ;
 ELEMENT *element_suiv ;}

typedef PARTIE *PARTITION ;
```

Structures : accès aux champs

L'accès à un champ d'une structure est réalisé au moyen de l'opérateur « . » :

$$exp_champ \rightarrow exp . nom_de_champ$$

expr. champ	type	valeur	R/Lvalue	effet de bord
<i>exp</i>	struct	<i>v</i>	Lvalue	-
<i>nom_de_champ</i>	T	<i>k</i>		-
<i>exp_champ</i>	T	champ <i>k</i> de la structure <i>v</i>	Lvalue	-

Exemple : `jean.nom` ou : `jean.date_de_naissance.jour`

Opérations sur les structures : les structures sont des objets « globaux »:

- On peut affecter une structure à une autre
- Le résultat d'une fonction peut être une structure
- Une structure est passée par valeur lors de l'appel d'une fonction.

Initialisation des structures

Une variable de type structure peut être initialisée lors de sa déclaration :

```
PERSONNE jean = { " Dupont ", {18, 12, 1974}, "France" } ;
```

S'il y a moins de valeurs que de champs, les champs restant sont initialisés à 0.

Unions

Il arrive que dans une structure, seul un champ est significatif (ou a un sens). Ceci est notamment le cas lorsque le contenu d'une même zone de mémoire doit être interprété différemment selon le contexte. En voici un exemple :

- Pour une voiture on s'intéresse à la puissance (nombre de chevaux) et la marque.
- Pour un vélo on s'intéresse au nombre de vitesses et à la présence d'un double plateau.

Pour représenter un véhicule, on est tenté d'utiliser une structure à 5 champs :

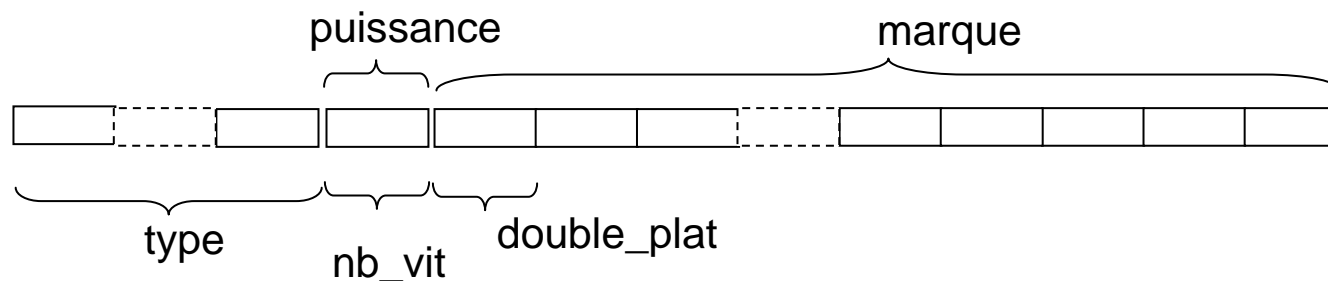
```
struct vehicule {char type[80] ;  
    short puissance ; char marque[20];  
    short nb_vit, double_plat ;} V;
```

Comme un véhicule n'est pas en même temps une voiture et un vélo, l'espace alloué à 2 de ces champs est perdu.

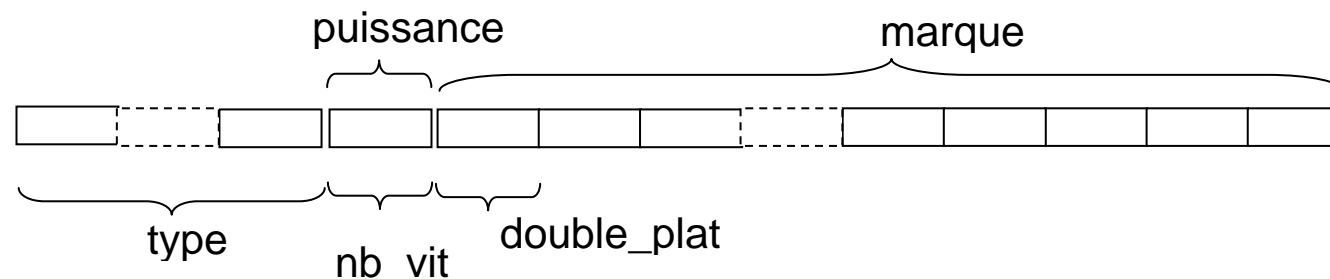
On évite cela avec le concept d'union :

- dans une structure les champs se suivent sans se chevaucher,
- les champs d'une union commencent tous au même endroit et donc se superposent.

```
struct vehicule {char type[80] ;  
    union { struct voit {short puissance ; char[10] marque ;} ;  
            struct velo {short nb_vit, double_plat ;}} data ;} v ;
```



Unions (suite)



Les 2 struct qui constituent l'union occupent la même zone de mémoire. Si l'on fait :

```
v.puissance = 12 ;  
v.marque = "Renault" ;  
x = v.nb_vit ;  
y = v.double_plat ;
```

on trouvera dans `x` la valeur 12 et dans `y` la valeur Ascii du caractère 'R'.

Attention aux surprises ! Avec la déclaration :

```
union {float R ; int I ;} x ;
```

et les affectations :

```
x.R = 12.5 ;  
k = x.I ;
```

on obtiendra dans `k` l'entier dont la représentation binaire coïncide avec celle du réel 12.5 ; il n'y a aucune raison pour que ce soit 12, et en effet c'est le nombre 1 095 237 632 !