[LC070]

biblio : [TCU page 206] [K&R page 115]

# Structures & Unions

## Structures

En langage C une variable structurée (ou simplement structure) correspond à ce qu'on appelle dans d'autres langages un enregistrement (ou record) car cela sert en général pour les différents éléments (enregistrements) d'une base de données.

Une structure apparaît en fait comme un bloc constitué de plusieurs variables. Ces variables, qui peuvent être de différents types, sont appelées les membres de la structure.

Il y a trois manières de déclarer/définir des structures : directement, par un "type structure étiqueté" (structure tag) et par un "type structure défini".

### Déclarations directes de variables structurées

Dans ce cas on déclare les variables structurées en précisant directement leur "structure", c'est-à-dire de quels membres elles sont constituées.

On notera dans ces déclarations :

- avant le bloc : le mot-clef **struct**,
- dans le bloc : les déclarations des membres, chacune étant terminée par un point-virgule,
- après le bloc : le nom de la (les) structure(s) déclarée(s) et éventuellement un modifieur tel que [ ] pour un tableau ou * pour un pointeur,
- le point-virgule final.

```
struct {char * nom ; unsigned ipays ; float lati,longi ;
        float d_hor ;} ville ;

struct {char * nom, * nlatin ; } constell[200], * ptconst ;

struct {char * nom ; int iconst ; float decli, ascdte ; couleur coul ;
        int magn ; } etoil1, star, * ptetoil ;
```

### Déclarations de structures par un "type structure étiqueté"

On définit d'abord un tel type en faisant suivre le mot-clef **struct** d'une étiquette de structure, puis du bloc indiquant les différents membres et enfin d'un point-virgule :

```
struct date { int jour, mois, année ; } ;
```

On peut ensuite déclarer/définir des variables du type correspondant en utilisant le mot **struct** suivi de l'étiquette :

```
struct date hier,demain ;
struct date naiss[3]={{17,5,61},{3,2,63},{55,8,91}} ;
```

Il est à noter que l'étiquette n'est pas utilisable sans le mot **struct**.

### Déclarations de structures par un "type structure défini" (avec "typedef")

Un tel type est défini avec un "typedef" : La syntaxe est la même que la déclaration directe précédée du mot "**typedef**". Dans ce cas le nom qui suit est celui du nouveau type défini :

```
typedef struct {double ad, de ; } coordequat ;
```

On peut ensuite utiliser ce type pour déclarer/définir des variables :

```
coordequat cquat[NBPLANET], cqetoil,cqlieu ;
```

### Utilisation : Accès aux membres d'une structure

Exemple : Les membres de la structure `hier` sont hier.jour, `hier.mois`, `hier.annee`.

On peut utiliser les membres d'une structure comme n'importe quelle variable de même type.

Exemples :

```
ville.nom = "CLERMONT-FERRAND" ; ville.ipays = 1 ;
```

```
printf("%s latitude: %g longitude: %g\n", ville.nom, ville.lati, ville.longi) ;
strcpy (nom,ville.nom) ;
K = KRAD * cqetoil.ad - 90 ;
if (strcmp ( constell[5].nom, "GRANDE OURSE")) ...
```

### Utilisation globale d'une structure

On peut se référer à une structure ( globalement ) par son adresse (que l'on affecte à un pointeur de type convenable). Exemple :

```
ptetoil = &etoil1 ; ptconst = constell + 71 ;
```

La norme AWSI autorise également d'affecter (recopier) une structure globalement. Exemple :

```
hier = aujourd ; aujourd = demain ;
```

### Notation spéciale pour les pointeurs vers structure

Exemple :

```
(* ptetoil).coul = VERT ;
```

pourra s'écrire plus simplement sous la forme suivante :

```
ptetoil -> coul = VERT;
```

### A quoi peuvent servir les structures ?

**1) enregistrements de bases de données** (pour les entrées/sorties)

Voir le chapitre sur les Entrées/sorties.

**2) tableaux structurés**

Exemple d'après K&R page 119 :

```
/* ------------ version 1 ------------ */
char * motclef[NB] = {"break","case","char",...,"while"} ;
int compt_mc[NB] ; char mot[12] ; int i ;
...
...
for (i=0 ; i< NB && strcmp(mot,(une bb motclef)[i])>0 ; i++)   (;)
if (strcmp(mot,motclef[i] == 0) compt_mc[i]++ ;
...
...
/* ------------ version 2 ------------ */
struct mc { char * motclef ; int compt ; } ; /* déf du type étiqueté */
struct mc *pt;          /* déf des variables */
struct mc tabl_mc[]={"break",0,"case",0,"char",0,...,"while",0,NULL} ;
char mot[12];
...
...
for (pt = tabl_mc ; pt->motclef && strcmp(mot,pt->motclef)>0 ; pt++) ;
if (strcmp(mot,motclef[i] == 0) pt->compt++ ;
...
...
```

```
/* ------------ version 3 ------------ */
struct mc { char motclef[SIZ_MC] ; int compt ; } ; /* déf type étiqueté */
struct mc *pt;          /* déf des variables */
struct mc tabl_mc[NB]={"break",0,"case",0,"char",0,...,"while",0 } ;
char mot[SIZ_MC];
...
...
pt = (struct mc *) bsearch(mot,tabl_mc,NB,sizeof(              ),strcmp);
if (pt) pt->compt++ ;
...
...
```

recherche dichotomique dans le tableau.

### 3) listes chaînées

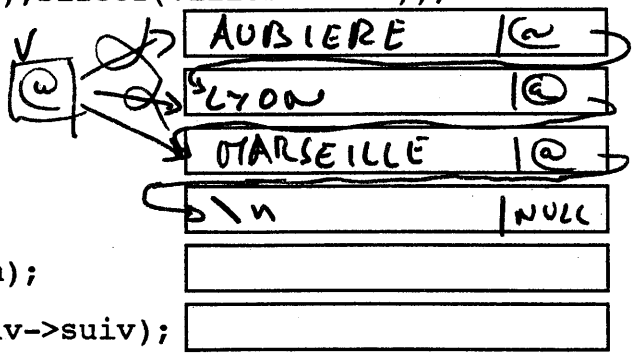
```
/* EXEMPLE des VILLES      STRUVILL.C  18/10/95 ( : ) JPBlanc      IUT Info  63 Aubière */
/* variante de STR_DS91.C sans allocations mémoire */   /* exercice sur les structures (extrait du DS du 24/6/91) */

#include <stdio.h>
#include <string.h>

#define NB_VILL   53  /* on suppose qu'il n'y a jamais plus de 53 villes  */
#define SIZN_VILL 30  /* et que chacune ne fait pas plus de 29 caractères */

struct tville {char nom[SIZN_VILL]; struct tville * suiv;};

struct tville villes[NB_VILL];

typedef struct tville  * ptvi;
ptvi v;

main()
{
  v=villes;
  puts("donnez le nom de la première ville:");
  while ( strcmp(gets(v->nom),""))   /* Boucle de saisie des noms de villes : ... */
        {                            /* .. une entrée vide indique la fin de saisie */
         v=((v->suiv)=v+1);
         puts("..de la ville suivante:");
        }
  v->suiv=NULL;
```

v est un ptr vers le 1er elem du tableau.

/* On suppose que sont saisies les trois villes suivantes dans l'ordre : AUBIERE, LYON, MARSEILLE
puis une entrée vide. Que donnent les affichages suivants ? */

```
  printf("%d,%d \n",sizeof(struct tville),sizeof(villes->suiv));
  printf("%s,%s \n",v->nom,villes->nom);
  printf("%s \n",villes->suiv->nom);
  printf("%s \n",(*(villes->suiv)).nom);
  printf("%s \n",villes->suiv->suiv->nom);
  printf("%d \n",villes->suiv->suiv->suiv->suiv);
  return 0;
}
```

34 — 4 — "" — aubiere — LYON — LYON — MARSEILLE — 0 (zero)



# Unions

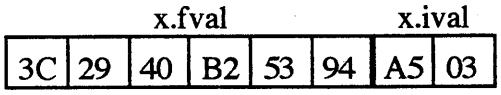
Les unions utilisent les mêmes règles syntaxiques que les structures, mais au lieu d'être constituées de membres juxtaposés, il s'agit de membres superposés.

Autrement dit, alors que les structures permettent de regrouper plusieurs variables de types différents (ou non), les unions permettent d'interpréter une (ou plusieurs) variable(s) sous différentes formes, c'est-à-dire comme des variables de différents types.

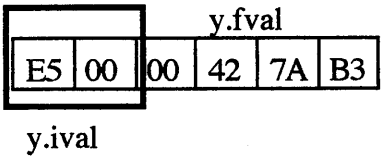
*Exemples* :

```
struct {float fval ; int ival ; } x ;
union  {float fval ; int ival ; } y ;
```

les membres de la *x* sont deux variables (composantes) différentes :

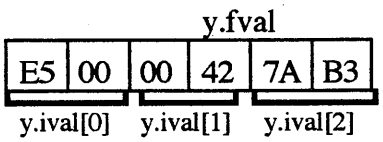



les membres de l'union *y* sont deux de la même variable :




Si on souhaite pouvoir accéder aux 3 "bioctets" de la variable union **y**, on déclarera plutôt :

```
union {float fval ; int ival[3] ; } y ;
```




On peut le faire aussi en utilisant un membre qui est lui-même une structure :

```
union {  float fval ;
         struct {int i1,i2,i3 ;} ival ;
      } y ;
```

*Exercice* : Quel est le nom complet du 2ème entier (celui qui vaut 66) dans cette dernière déclaration ?

*Remarque* : Cette utilisation des unions est elle portable ?

## *Utilisation des unions*

• Considérer une même comme ayant différents (voir exemples précédents)

• Faire des de pointeurs :

```
union {
        char * cp ;
        short * sp ;
        int * ip ;
      } ptr ;
```

Alors : ptr.cp pointe vers un [ ]

ptr.sp pointe vers un [ ]

ptr.ip pointe vers un [ ]

• champs dans des structures représentant des enregistrements (records) de bases de données.

*Exemple* :

```
struct date { int jour ;
              int mois ;
              union { int année ;
                      struct { char h ; char mn ; } hm ;
                    } u ;
              int flag ;
            } d ;
```

Dans cet exemple, l'année est : [ ] l'heure est [ ]

(Rappelons que sous Unix, la commande *ls* affiche les dates de modifications des fichiers sous l'une des formes :

```
jour mois h:mn
```

ou

```
jour mois année
```

suivant que cette date est plus récente ou plus vieille que 6 mois.)