Chapitre 2

Les types composés

A partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types composés*, qui permettent de représenter des ensembles de données organisées.

2.1 Les tableaux

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.

La déclaration d'un tableau à une dimension se fait de la façon suivante :

`type nom-du-tableau[nombre-éléments];`

où *nombre-éléments* est une expression constante entière positive. Par exemple, la déclaration `int tab[10];` indique que `tab` est un tableau de 10 éléments de type `int`. Cette déclaration alloue donc en mémoire pour l'objet `tab` un espace de 10×4 octets consécutifs.

Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre-éléments* par une directive au préprocesseur, par exemple
`#define nombre-éléments 10`

On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments d'un tableau sont toujours numérotés de 0 à *nombre-éléments* `-1`. Le programme suivant imprime les éléments du tableau `tab` :

```
#define N 10
main()
{
  int tab[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Un tableau correspond en fait à un pointeur vers le premier élément du tableau. Ce pointeur est constant. Cela implique en particulier qu'aucune opération globale n'est autorisée sur un tableau. Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation.

Par exemple, on ne peut pas écrire "`tab1 = tab2;`". Il faut effectuer l'affectation pour chacun des éléments du tableau :

```
#define N 10
main()
{
  int tab1[N], tab2[N];
  int i;
  ...
  for (i = 0; i < N; i++)
    tab1[i] = tab2[i];
}
```

On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

type nom-du-tableau`[N] = {`*constante-1*`,`*constante-2*`,...,`*constante-N*`};`

Par exemple, on peut écrire

```
#define N 4
int tab[N] = {1, 2, 3, 4};
main()
{
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à zéro si le tableau est une variable globale (extérieure à toute fonction) ou une variable locale de classe de mémorisation `static` (*cf.* page 64).

De la même manière un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec un caractère nul `'\0'`. Il faut donc que le tableau ait au moins un élément de plus que le nombre de caractères de la chaîne littérale.

```
#define N 8
char tab[N] = "exemple";
main()
{
  int i;
  for (i = 0; i < N; i++)
    printf("tab[%d] = %c\n",i,tab[i]);
}
```

Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation. Ainsi le programme suivant imprime le nombre de caractères du tableau `tab`, ici 8.

```
char tab[] = "exemple";
```

```
main()
{
  int i;
  printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char));
}
```

De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

`type nom-du-tableau[nombre-lignes][nombre-colonnes]`

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression "`tableau[i][j]`". Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes :

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};

main()
{
  int i, j;
  for (i = 0 ; i < M; i++)
    {
      for (j = 0; j < N; j++)
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```

2.2 Les structures

Une *structure* est une suite finie d'objets de types différents. Contrairement aux tableaux, les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.

On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné. La déclaration d'un modèle de structure dont l'identificateur est *modele* suit la syntaxe suivante :

```
struct  modele
{
    type-1 membre-1;
    type-2 membre-2;
     ...
    type-n membre-n;
};
```

Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

struct *modele objet*;

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct  modele
{
   type-1 membre-1;
   type-2 membre-2;
    ...
   type-n membre-n;
}  objet;
```

On accède aux différents membres d'une structure grâce à l'opérateur *membre de structure*, noté ".". Le `i`-ème membre de *objet* est désigné par l'expression

objet.membre-i

On peut effectuer sur le `i`-ème membre de la structure toutes les opérations valides sur des données de type *type-i*. Par exemple, le programme suivant définit la structure `complexe`, composée de deux champs de type `double` ; il calcule la norme d'un nombre complexe.

```
#include <math.h>
struct complexe
{
  double reelle;
  double imaginaire;
};

main()
{
  struct complexe z;
  double norme;
   ...
  norme = sqrt(z.reelle * z.reelle + z.imaginaire * z.imaginaire);
  printf("norme de (%f + i %f) = %f \n",z.reelle,z.imaginaire,norme);
}
```

Les règles d'initialisation d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :
`struct complexe z = {2. , 2.};`
En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
...
main()
{
  struct complexe z1, z2;
```

```
  ...
  z2 = z1;
}
```

2.3 Les champs de bits

Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (`int` ou `unsigned int`). Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration. Par exemple, la structure suivante

```
struct registre
{
  unsigned int actif : 1;
  unsigned int valeur : 31;
};
```

possède deux membres, `actif` qui est codé sur un seul bit, et `valeur` qui est codé sur 31 bits. Tout objet de type `struct registre` est donc codé sur 32 bits. Toutefois, l'ordre dans lequel les champs sont placés à l'intérieur de ce mot de 32 bits dépend de l'implémentation.

Le champ `actif` de la structure ne peut prendre que les valeurs 0 et 1. Aussi, si `r` est un objet de type `struct registre`, l'opération `r.actif += 2;` ne modifie pas la valeur du champ.

La taille d'un champ de bits doit être inférieure au nombre de bits d'un entier. Notons enfin qu'un champ de bits n'a pas d'adresse ; on ne peut donc pas lui appliquer l'opérateur `&`.

2.4 Les unions

Une *union* désigne un ensemble de variables de types différents susceptibles d'occuper *alternativement* une même zone mémoire. Une union permet donc de définir un objet comme pouvant être d'un type au choix parmi un ensemble fini de types. Si les membres d'une union sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.

Les déclarations et les opérations sur les objets de type union sont les mêmes que celles sur les objets de type `struct`. Dans l'exemple suivant, la variable `hier` de type `union jour` peut être soit un entier, soit un caractère.

```
union jour
{
  char lettre;
  int numero;
};

main()
{
  union jour hier, demain;
  hier.lettre = 'J';
```

```
  printf("hier = %c\n",hier.lettre);
  hier.numero = 4;
  demain.numero = (hier.numero + 2) % 7;
  printf("demain = %d\n",demain.numero);
}
```

Les unions peuvent être utiles lorsqu'on a besoin de voir un objet sous des types différents (mais en général de même taille). Par exemple, le programme suivant permet de manipuler en même temps les deux champs de type `unsigned int` d'une structure en les identifiant à un objet de type `unsigned long` (en supposant que la taille d'un entier long est deux fois celle d'un `int`).

```
struct coordonnees
{
  unsigned int x;
  unsigned int y;
};
union point
{
  struct coordonnees coord;
  unsigned long mot;
};

main()
{
  union point p1, p2, p3;
  p1.coord.x = 0xf;
  p1.coord.y = 0x1;
  p2.coord.x = 0x8;
  p2.coord.y = 0x8;
  p3.mot = p1.mot ^ p2.mot;
  printf("p3.coord.x = %x \t p3.coord.y = %x\n", p3.coord.x, p3.coord.y);
}
```

2.5 Les énumérations

Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumération est défini par le mot-clef `enum` et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet :

`enum` *`modele`* {*`constante-1, constante-2,...,constante-n`*};

En réalité, les objets de type `enum` sont représentés comme des `int`. Les valeurs possibles *`constante-1, constante-2,...,constante-n`* sont codées par des entiers de 0 à *`n-1`*. Par exemple, le type `enum booleen` défini dans le programme suivant associe l'entier 0 à la valeur `faux` et l'entier 1 à la valeur `vrai`.

```
main()
```

```
{
  enum booleen {faux, vrai};
  enum booleen b;
  b = vrai;
  printf("b = %d\n",b);
}
```

On peut modifier le codage par défaut des valeurs de la liste lors de la déclaration du type énuméré, par exemple :
`enum booleen {faux = 12, vrai = 23};`

2.6 Définition de types composés avec `typedef`

Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de `typedef` :

`typedef` *`type synonyme`*`;`

Par exemple,

```
struct complexe
{
  double reelle;
  double imaginaire;
};
typedef struct complexe complexe;

main()
{
  complexe z;
   ...
}
```