Chapitre 14

Regrouper des Données : Structures, Unions, et Champs de Bits

Ce chapitre va nous permettre d'étudier les structures, les unions, et les champs de bits, qui sont plusieurs méthodes que le C met à notre disposition pour regrouper des données.

I:\ Structures

Lorsque l'on programme, il arrive que l'on ressente le besoin de regrouper des données au sein d'une seule variable.
Au cours de notre Tutorial, nous avons eu l'occasion de parler des tableaux, qui permettent de regrouper en une variable des données qui sont toutes du même type.
A présent, nous allons étudier les structures, qui sont le moyen que fournit le langage C pour regrouper plusieurs données de types qui peuvent être différents au sein d'une seule variable, d'un seul "enregistrement".

A: Déclaration et utilisation

Pour déclarer une structure, on utilise le mot-clef `struct`, de la manière qui suit :

```
struct
{
    type1 variable1;
    type2 variable2;
    type3 variable3;
} nom_structure;
```

Ceci va déclarer une variable nommmée `nom_structure`, qui correspondra à une structure contenant trois variables, de types que vous demandez.

Ensuite, pour utiliser cette variable, et les éléments la composant, on utilisera une syntaxe de ce type :

```
nom_structure.variable
```

Ceci pouvant être utilisé aussi bien en partie gauche, qu'en partie droite, d'une expression, tout comme n'importe quel autre type de variable que nous avons jusqu'à présent étudié.

B: Exemple

Par exemple, pour déclarer une variable nommée `ma_structure`, et contenant deux entiers `a` et `b`, nous utiliserons la syntaxe suivante :

```
struct
{
      short  a;
      short  b;
}  ma_structure;
```

Ensuite, il nous est possible d'utiliser cette structure en accèdant à ses composantes de la manière dont nous l'avons indiqué un peu plus haut.
Pour affecter des valeurs aux deux variables composant la structure :

```
ma_structure.a  =  10;
ma_structure.b  =  20;
```

Et, pour relire ces valeurs, nous utilisons la même écriture :

```
printf("%d ; %d", ma_structure.a, ma_structure.b);
```

Ce qui nous permettra d'afficher ces deux valeurs.

C: Soyons brefs !

Devoir écrire tout ce qui compose la structure à chaque fois que nous souhaitons en déclarer une est extrêmement long... surtout si nous avons un grand nombre de variables à déclarer.
Bien évidemment, nous pourrions utiliser ceci :

```
struct
{
      short  a;
      short  b;
} var_1,  var_2,  var_3,  var_4;
```

Ce qui nous permet de déclarer, ici, quatre variables correspondant à notre structure.
Mais si nous souhaitons déclarer des variables de ce type dans, par exemple, deux fonctions, il nous faudra écrire la description complète de la structure deux fois, comme ceci :

```
void fonction_1(void)
{
    struct
    {
        short a;
        short b;
    } ma_variable_1;

    ma_variable_1.a = 10;
    ma_variable_1.b = 20;

    // ...

} // Fin fonction_1

void fonction_2(void)
{
    struct
    {
        short a;
        short b;
    } ma_variable_2;

    ma_variable_2.a = 135;
    ma_variable_2.b = 246;

    // ...

} // Fin fonction_2
```

Si nous souhaitons apporter une modification à cette structure, il nous faudra la modifier en deux endroits, ce qui peut rapidement devenir une source d'erreurs.

Voila pourquoi il est possible de donner un nom à la strucure, comme ceci :

```
struct ma_structure
{
      short a;
      short b;
};
```

Et ensuite, chaque fois que nous voudrons déclarer une variable de type correspondant à cette structure, nous utiliserons ce type de syntaxe :

```
struct ma_structure ma_variable_3;
```

Ce qui est plus court, et, bien évidemment, plus sécurisé, puisqu'il n'y a plus, en cas de besoin, qu'à modifier la structure en un seul endroit.

Cependant, le C, avec ses habitudes de concision, permet d'aller encore plus loin.
En effet, il nous est possible d'indiquer au compilateur qu'il doit considérer notre structure comme un type, au même sens que `short` ou `float`, par exemple.
Pour cela, nous utiliserons l'instruction `typedef`, de la manière suivante :

```
typedef struct
{
      short a;
      short b;
} MON_NOM_TYPE;
```

Ceci va indiquer au compilateur qu'il doit considérer `MON_NOM_TYPE` comme un nouveau type de données, correspondant à notre structure.
Notons que, en général, nous écrivons en majuscules les noms de types que nous définissons à l'aide d'un `typedef`, afin de plus rapidement pouvoir les identifier, dans le but de s'y retrouver au sein de notre code source.
Ensuite, pour déclarer une variable du type que nous avons choisi pour notre structure, nous agirons exactement de la même manière que pour tout autre type de variable :

```
MON_NOM_TYPE ma_variable_5;
```

Ceci va nous permettre de déclarer une variable, nommée `ma_variable_5`, de type correspondant à notre structure.
Par la suite, nous pourrons l'utiliser exactement de la même manière que ce que nous avons fait jusqu'à présent :

```
ma_variable_5.a = 89;
ma_variable_5.b = 90;
```

Le choix d'utiliser un `typedef` ou d'en rester à la solution que je proposais juste au dessus (conserver le mot-clef `struct`) dépend généralement des programmeurs...
Personnellement, je préfére utiliser un `typedef` ; mais le choix final vous revient.

D: Affectation de structures

Le langage C permet d'effectuer des affectations de structures, en utilisant l'opérateur d'affectation =, comme pour n'importe quel autre type de données.
Notez que celui-ci affecte l'ensemble des champs de la structure ; si vous ne souhaitez copier que quelques uns de ses champs, il vous faudra les copier un par un.
De plus, il n'est possible d'affecter que des structures de même type !

Voici un exemple d'affectations de structures :

```
#include <tigcclib.h>

typedef struct
{
    short a;
    float y;
    long d;
} MA_STRUCTURE;

void affiche(MA_STRUCTURE a);

void _main(void)
{
    clrscr();

    MA_STRUCTURE a, b;

    a.a = 10;
    a.y = 3.14;
    a.d = 567;

    affiche(a);

    b = a;

    affiche(b);

    b.y = 35.6;

    affiche(b);

    ngetchx();
}

void affiche(MA_STRUCTURE a)
{
    printf("%d - %f - %ld\n", a.a, a.y, a.d);
}
```

Exemple Complet

En quelques mots, voici ce que nous faisons au cours de cet exemple :
Nous commençons par déclarer un type correspondant à une structure : `MA_STRUCTURE`.
Ensuite, nous déclarons le prototype d'une fonction que nous avons choisi de nommer `affiche`. Cette fonction, dont la définition est placée en fin de notre code source, n'a d'autre but que l'affichage du contenu d'une variable de type de structure avec lequel nous avons choisi de travailler.

Après cela, notre fonction principale efface l'écran, déclare deux variables du type correspondant à notre structure, initialise les éléments de l'une d'entre elle, affecte celle-ci à la seconde, puis modifie l'un des champs de la seconde.
Le tout en affichant, à chaque étape, le contenu de la structure sur laquelle nous venons de travailler.

F: Initialisation d'une structure

Le langage C nous permet d'initialiser une structure entière en une seule affectation, de la manière qui suit :

```
MA_STRUCTURE a = {10, 3.14, 567};
```

Notez toutefois que ce type d'affectation ne peut se faire qu'à la déclaration, et nulle part ailleur. De plus, en utilisant cette syntaxe, vous devez initialiser tous les champs de la structure.

Si vous ne souhaitez initialiser que certains des champs de la structure, vous pouvez, sous TIGCC (en tant que dérivé de GCC), utiliser la syntaxe illustrée ci-dessous :

```
MA_STRUCTURE a = {.a=10, .d=567};
```

L'effet sera exactement le même que si vous initialisiez vous-même à 0 les champs non initialisés, y compris en occupation mémoire - en fait, le compilateur fera l'initialisation des champs non précisés lui-même, sans vous l'indiquer.
En somme, cette syntaxe n'est, par rapport à la précédente, qu'un raccourci d'écriture, prévue pour vous éviter d'avoir à préciser des valeurs pour les champs qui ne vous intéressent pas.

E: Structures et pointeurs

Pour en terminer avec les structures, nous allons traiter de l'utilisation de structures via des pointeurs.
Pour cette partie, nous allons considérer que nous avons déclaré une structure, et un pointeur sur une structure, de la manière suivante ; toujours, naturellement, en travaillant sur le type de structure que nous utilisons depuis tout à l'heure :

```
MA_STRUCTURE a = {10, 3.14, 89};
MA_STRUCTURE *p;
```

Exactement comme avec les types avec lesquels nous avons déjà travaillé au cours de ce tutorial, la seconde ligne permet de déclarer un pointeur, nommé `p`, à même de pointer sur une structure du type qui nous intéresse.
A présent, faisons pointer notre pointeur sur la variable de type structure que nous avons déclaré :

```
p = &a;
```

Encore une fois, cela se fait de la même manière que pour les autres types de variables : en utilisant l'opérateur unaire &.

Pour ce qui est d'accèder à un champ de la structure, il nous faut commencer par accèder à celle-ci, en utilisant, comme pour n'importe quel pointeur, l'opérateur `*` ; ensuite, comme nous l'avons vu précédemment, nous utilisons l'opérateur `.` pour accèder au champ qui nous intéresse.
Par exemple :

```
(*p).a = 16;
```

Cela dit, ce type d'écriture est particulièrement lourd, notamment du fait que les parenthèses sont nécessaires, en raison de priorité des opérateurs...
Voila pourquoi le C, fidèle à ses habitudes de concision, propose un moyen alternatif : l'opérateur `->` que nous pouvons utiliser de la manière qui suit :

```
p->a = 16;
```

Ceci correspondant exactement à la même écriture que la précédente.

II:\ Unions

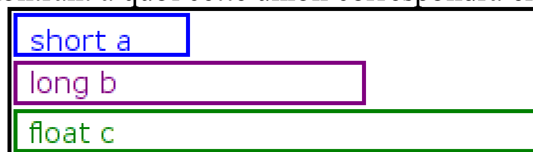
Comme nous l'avons vu un peu plus haut, une structure permet d'enregistrer plusieurs données, éventuellement de types différents, en les regroupant au sein d'une seule variable.
Une `union` permet de désigner un seul espace mémoire, et de permettre d'y accèder de plusieurs moyens différents. En somme, une union est un regroupement de plusieurs types de données, qui permettent tous d'accèder au même emplacement mémoire.

A: Déclaration et Utilisation

Voici comment on déclare une `union` ; exactement de la même manière que pour une structure :

```
union
{
    short a;
    long b;
    float c;
} mon_union;
```

Voici un rapide schéma montrant à quoi cette union correspondra en mémoire :



Le type `float` étant, des trois types que nous avons groupé au sein de notre union, celui qui occupe le plus de place en mémoire, une variable du type de cette union occupera en mémoire la taille d'un `float`, dans ce cas.
Ensuite, si nous adressons une variable du type de cette union par le champ de type `short`, nous n'utiliserons qu'une partie de l'espace mémoire occupé par cette variable, puisqu'un `short` est moins gros en mémoire qu'un `float`.

Pour ce qui est de l'utilisation, il en va exactement de la même manière que pour les structures. Par exemple :

```
mon_union.a = 10;
mon_union.b = 70900;
mon_union.c = 3.14;
```

(En travaillant avec l'union que nous avons déclaré juste au dessus)

B: Soyons plus brefs !

De la même façon que pour les structures, il existe des syntaxes plus concises pour la déclaration d'unions :

```
union union_2
{
    short a;
    float b;
};

union union_2 a;

a.a = 10;
a.b = 3.14;
```

Et, en utilisant un typedef :

```
typedef union
{
    short a;
    float b;
} MON_UNION;

MON_UNION b;
b.a = 10;
b.b = 3.14;
```

Comme nous le voyons, dans un cas comme dans l'autre, la syntaxe est exactement la même que ce que nous avons présenté précédemment pour les structures.

C: Démonstration

A présent, pour que vous compreniez bien le fonctionnement d'une union, et pour vous prouver que les différents champs la composant sont bien des alias du même espace mémoire, j'aimerais attirer votre attention sur la portion de code suivante :

```
typedef union
{
    short a;
    long b;
} MY_UNION;

MY_UNION test;

test.b = 0xA0B0C0D0;

clrscr();
printf("%X - %lX", test.a, test.b);
ngetchx();
```

Au cours de cet exemple, nous affectons la valeur 0xA0B0C0D0 au champ de type `long` de notre variable de type `union` capable de contenir un `long` et un `short`.
Ensuite, nous affichons le contenu de cette variable, en y accédant par le champ de type `long`, ce qui nous affiche bien la valeur à laquelle nous nous attendions, à savoir, 0xA0B0C0D0, ainsi que le contenu de cette même variable, en y accédant, cette fois-ci, par le champ de type `short` ; et là, la valeur affichée est 0xA0B0, ce qui correspond à la première moitié de l'espace mémoire correspondant à notre variable.
En somme, nous venons de prouver que les deux champs composant notre variable correspondent bien au même espace mémoire, de la taille du plus grand de ceux-ci.

D: Utilité ?

Maintenant que nous savons comment déclarer, et utiliser, des unions, je pense qu'un petit exemple concret de cas où l'emploi d'une union peut être bénéfique s'avére nécessaire, pour que vous compreniez bien à quoi cela sert... Cet exemple nous permettra aussi d'utiliser une structure en "situation réelle".

Nous allons nous placer dans la situation d'un programmeur qui commence à réfléchir à la manière dont il pourait coder en mémoire les niveaux de son futur jeu de type Shoot'em Up (Pour quelques exemples correspondant à ce à quoi je pense, en sachant que j'ai volontairement simplifié mon exemple ici, pensez à des titres tels Solar Striker, ou Krypton - Cf TimeToTeam)

Nos niveaux peuvent contenir, sur chaque ligne, soit rien, soit un ennemi, soit un power-up, un niveau étant composé d'autant de lignes que nous le souhaitons, à raison d'un power-up ou ennemi par ligne du niveau. Ce niveau pourrait donc être enregistré sous forme d'un tableau d'unions représentant soit un ennemi, soit un power-up ; mais ceci ne nous regarde pas ici : c'est au niveau de l'implémentation du jeu lui-même que ceci devrait être décidé.

Dans notre niveau, un ennemi peut être décrit comme une structure ainsi définie :

```
typedef struct
{
     short nombre_canons;
     short vitesse;
     short energie;
}ENNEMI;
```

Comme nous pouvons le constater, nos ennemis disposent d'un nombre variable de canons, d'une vitesse donnée, et d'un certain nombre de points d'énergie, tout ceci étant défini par le créateur du niveau.

Pour ce qui est des power-ups, leur seul spécificité est le type de bonus dont il s'agit ; voici comment un power-up peut donc être défini au sein de notre niveau :

```
typedef struct
{
     short type_bonus;
}POWER_UP;
```

Ici aussi, donc, un typedef d'une structure.

Finalement, pour notre niveau, il pourrait nous suffir d'une union contenant soit un ENNEMI, soit un POWER_UP.
Cela dit, il nous faut un moyen de déterminer si l'espace mémoire correspondant à une ligne de notre niveau doit être interprété comme un descripteur d'ennemi, ou comme un descripteur de power-up. Pour cela, nous ajoutons un champ "type", pour lequel nous acceptons les valeurs 0 pour rien, 1 pour un ennemi, et 2 pour un power-up.
Voici la structure que nous utiliserons :

```
typedef struct
{
     short type;   // 1 => ennemi ; 2 => power-up ; 0 => rien
     short x, y;
     union
     {
          ENNEMI ennemi;
          POWER_UP power_up;
     };
}ENTITE;
```

Notons l'utilisation de deux champs nommés `x` et `y`, qui définiront la position de création de notre ennemi ou power-up ; pourquoi les placer à la fois dans l'union ENNEMI et dans l'union POWER_UP ? alors qu'il est si simple de ne les éfinir qu'une seule fois ?
Notons aussi que nous avons groupé nos deux types de descripteur au sein d'une union non nommée. Ceci est possible dans le cas ou notre union non nommée (ou une structure non nommée) est placée au sein d'une autre union ou structure, à condition qu'il n'y ait pas de définition ambigüe de noms de champs.

III:\ Champs de bits

Ce qu'on appelle, en C, un Champ de Bits, est une structure un peu particulière : au lieu que les champs la composant soient différentes variables de différents types, comme pour une structure "normale", les champs qui la composent sont en fait des portions d'un entier : comme son nom l'indique, un champ de bit est une structure dont les différents champs font le nombre de bits que vous voulez.
Pour déclarer un champ de bit, voici le type d'écriture que vous utiliserez :

```
typedef struct
{
      unsigned short
            champ_1: 6,
            champ_2: 6,
            champ_3: 1,
            champ_4: 1,
            champ_5: 2;
}BIT_FIELD_1;
```

Votre structure ne contient qu'un seul type entier, dont elle fera la taille (`short` ou `long`, par exemple), et vous découpez ce type en plusieurs champs, chacun faisant le nombre de bits que vous souhaitez.
Dans le cas de l'exemple que nous venons de présenter, notre champ de bits est composé de deux champs de 6 bits de long, de deux champs de 1 bit, et d'un champ de 2 bits, ce qui fait bien les 16 bits qui composent un `unsigned short`.

Pour accèder aux champs composant un champ de bits, vous agissez exactement de la même manière que pour accèder à ceux qui composeraient une structure : après tout, un champ de bits est une structure !
Par exemple :

```
BIT_FIELD_1 mon_champ;

mon_champ.champ_1 = 16;
mon_champ.champ_4 = 0;
```

Cette portion de code déclarant une variable du type du champ de bits que nous avons défini plus haut, et affectant des valeurs à deux des champs de celle-ci.

En deux mots, la principale utilité des champs de bits est de vous permettre de tirer profit de tous les bits d'un espace mémoire, ce qui est utile dans le cas où vous avez besoin de mémoriser beaucoup d'informations, en ne disposant que peu d'espace mémoire.

Nous venons d'atteindre la fin de ce chapître de notre tutorial, destiné à nous apprendre à regrouper des données par l'utilisation de structures, d'unions, et de champs de bits. Le prochain chapître va nous permettre de découvrir ce qu'est l'allocation dynamique de mémoire, et comment en tirer parti.