Structures

Introduisons le problème suivant : un responsable d'année décide de recoder le système d'information de son école car ce dernier met bien trop de temps pour générer les bulletins de notes [1]. Afin de décider du sort de ses étudiants, il doit pouvoir les trier facilement par nom, par nombre de crédits validés, ou par moyenne générale.

Votre objectif est donc d'aider ce responsable d'année qui n'est pas bien compétent en programmation en C. Vous allez ainsi découvrir au travers de ce chapitre la notion de *structure* qui vous permettra (entre autres) de définir un étudiant en langage C.

1 Problématique

Un étudiant est défini par les champs suivants :

- un **nom**,
- un **prénom**,
- un tableau de trois **notes** de bloc,
- une **moyenne** générale,
- un nombre de **crédits** validés.

Les actions que vous devrez réaliser pour mener à bien votre travail sont les suivantes. Il vous faut dans un premier temps définir la structure de données `Etudiant`. Ce chapitre vous indiquera comment faire. Il vous faudra ensuite allouer un tableau d'étudiants et le remplir. Enfin, il faudra développer un algorithme de tri sur des clés au choix (nom, prénom, note ...). Ces dernières notions seront abordées lors d'un prochain TP.

2 Introduction aux structures

Définition 1 (Structure). Une *structure* est un regroupement de plusieurs variables sous un seul nouveau type de données.

Exemple 1. Supposons que l'on veuille définir un point de $\mathbb{R}^2$. Un tel élément est composé de deux réels que l'on peut noter x et y et regrouper sous un terme générique que l'on appellerait `Point`. Ainsi par exemple, un `Point` A peut être défini par un couple de réels $A = (x, y) = (1, 3)$.

L'exemple précédent montre que l'utilisation d'un `Point` requiert plusieurs étapes :

- définition du type `Point` : il se compose de deux réels x et y ;
- déclaration d'un nouveau `Point` A : action similaire à une déclaration de variable ;
- affectation de valeurs au `Point` A : modification des *champs* x et y de la variable A.

1. Ceci est une pure fiction. Toute ressemblance avec des personnes ou des institutions existantes serait purement fortuite.

2.1 En pratique

La définition d'une structure est réalisée de façon générique à l'aide des lignes de code suivantes.

```
struct TypeStruct          // définition du type de la structure
{
  type1 nom1;                    // type et nom du premier champ
  type2 nom2;                     // type et nom du second champ
  ...
};          // on n'oublie pas le point-virgule après l'accolade
```

Cette définition est à réaliser dans un fichier `.h`. En effet, ces lignes ne contiennent que des définitions pour un nouveau type de données. Elles ne réalisent aucune modification de la mémoire. Vous remplacerez bien évidemment le mot `TypeStruct` par le nom de votre nouveau type structuré, et les noms `type1`, `type2`, ... par les types de données à regrouper au sein de la structure.

Exemple 2. La structure `Point` détaillée précédemment doit contenir deux champs : un réel x représentant la première coordonnée du point, et un réel y représentant la seconde coordonnée du point. On la définit alors de la façon suivante dans un fichier `.h`.

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

Convention

Afin de repérer facilement un type structuré que vous venez de créer, nous vous recommandons fortement de débuter vos types par une majuscule.

TD **Exercice 1.**

Une fois le nouveau type structuré défini, il est possible de déclarer des nouvelles variables à l'aide de ce nouveau type. La déclaration se fait alors de manière (pas) très classique.

```
// Déclaration seule
struct TypeStruct var1;
// Déclaration et initialisation des champs
struct TypeStruct var2 = {valeur_nom1, valeur_nom2, ... };
```

Comme pour toute variable, il est obligatoire de spécifier en premier son type. Dans notre cas, le type est un type structuré que nous venons de définir. Il est alors obligatoire de répéter le mot clé `struct` avant le type structuré afin d'indiquer au compilateur que ce type est « fait maison ». La seconde ligne nous indique qu'il est possible, comme pour un tableau, d'initialiser les champs d'une variable de type structuré lors de sa déclaration. Il suffit pour cela d'écrire entre accolades les valeurs de chacun des champs, dans l'ordre

d'apparition dans la structure.

Exemple 3. Poursuivons l'utilisation de notre structure `Point`. La définition de deux nouveaux points A et B se fait à l'aide des lignes de code ci-dessous.

```
struct Point A;       // Déclaration d'une variable A de type Point
struct Point B = {3.14, 3.73};     // Déclaration et initialisation
```

TD **Exercice 2.**

La dernière étape consiste à accéder aux différents champs de la structure. Cet accès est réalisé à l'aide de l'opérateur « . ». Pour accéder à un champ d'une variable, on rédige une des lignes de code suivantes.

```
var1.nom1 = valeur; // acces au champ nom1 de la variable var1
var2.nom1 = valeur; // acces au champ nom1 de la variable var2
```

Exemple 4. Une fois notre variable `A` de type `Point` déclarée, on souhaite modifier ses coordonnées. On peut alors écrire les lignes suivantes.

```
A.x = 42.73; // Modification du champ x de la variable A
A.y = A.x;   // Modification du champ y de la variable A
```

Remarque 1. L'accès à un champ de la structure est autorisé dès la déclaration de la variable, **sous condition de compiler avec la norme c99** [2]. Ainsi, il est possible d'initialiser certains champs, en laissant les autres à 0. Il est également possible d'initialiser les champs dans n'importe quel ordre, comme le montre l'exemple suivant.

```
// Initialisation du champ x seulement
struct Point C = {.x = -1.0};
// Initialisation des champs dans un ordre différent.
struct Point C = {.y = 2.0, .x = 1.3};
```

Hors initialisation, il est en revanche impossible de modifier plus d'un champ à la fois. Une nouvelle fois, comme pour les tableaux, chaque champ doit être modifié à l'aide d'une instruction individuelle.

La comparaison entre les structures et les tableaux ne s'arrête pas là. En effet, les différents champs d'une structure sont mis à la suite en mémoire, tout comme les cases d'un tableau. Un dépassement sur le premier champ risque donc d'entrainer la modification de la valeur du champ suivant.

TD **Exercice 3.**

2.2 Le mot clé typedef

Vous avez naturellement remarqué la lourdeur de la déclaration d'une nouvelle variable de type structuré. En effet, il est pour le moment obligatoire d'inscrire le mot clé `struct`

2. On ajoute pour cela l'option `-std=c99` à la ligne de compilation

lors de la déclaration, et lors de la définition. Parce qu'un bon codeur est un codeur fainéant [3], le langage C possède le mot clé `typedef`.

Ce mot clé permet de remplacer un mot ou une série de mots (autrement dit une *expression*) par une nouvelle expression. Par exemple, il peut être intéressant de définir un type booléen pouvant prendre la valeur 0 ou 1. Ce type `bool` peut être plus clair dans la lecture d'un code qu'un type `char` ou `int`. Une telle redéfinition de type et son usage sont alors rédigés de la façon suivante.

```
typedef char bool;  // Redéfinition de type
bool test = 0;      // Déclaration d'une variable de type bool
```

Dans notre contexte de structures, on souhaite remplacer l'expression `struct TypeStruct` par une nouvelle expression contenant un seul mot. Dès lors, il est possible d'ajouter l'instruction suivante dans le fichier `.h` à la suite de la définition de la structure.

```
typedef struct TypeStruct TypeStruct;
```

L'instruction précédente reste cependant peu claire, en particulier à cause de la répétition du type structuré. Un moyen de contourner cela est d'ajouter les caractères `_s` à la fin du type structuré, et de les supprimer du typedef.

Exemple 5. Reprenons la définition de notre structure `Point` et modifions-là dans notre `.h` pour ajouter le `typedef`.

```
struct Point_s   // Définition du type Point_s
{
  double x;     // Contient deux champs
  double y;
};
typedef struct Point_s Point; // Redéfinition de type
```

Dès lors, la déclaration d'une variable de type `Point` peut être réalisée de la façon suivante.

```
Point A = {0.42, 0.37};  // Déclaration et initialisation
Point B;  // Déclaration sans initialisation
```

TD **Exercice 4.**

2.3 Pointeurs et structures

Nous venons de voir comment manipuler des variables de type structuré. Il est maintenant possible de rencontrer un jour, au hasard d'un code, un pointeur sur une variable de type structuré. Sa manipulation peut se faire de façon traditionnelle, mais l'accès à un champ peut alourdir l'écriture, comme le montre l'exemple suivant.

3. Mais tout de même compétent...

Exemple 6. Supposons que l'on dispose d'un pointeur sur une variable de type `Point`. Sa déclaration peut avoir été réalisée de la façon suivante.

```
Point A;         // Déclaration d'une variable de type Point
Point *p = &A;   // Déclaration d'un pointeur sur une var. de type
 ↪  Point
```

L'accès à un champ de la variable `A` par le biais du pointeur nécessite les opérations suivantes :

- déréférencement du pointeur (accès au contenu, donc à la variable `A`),
- accès à un champ de la variable structurée.

Les parenthèses permettent de donner une priorité à certaines opérations. Un déréférencement s'écrivant avec une étoile avant le pointeur, et un accès à un champ s'écrivant avec un point après la variable, on en déduit l'expression suivante :

```
(*pointeur).champ    // * = déréférencement,      . = accès au champ
```

Dès lors, supposons que l'on dispose d'un pointeur `p` sur une variable de type `Point`. Pour modifier le champ `x` de la variable à l'aide du pointeur `p`, on écrit :

```
(*p).x = 13.37;
```

Cette notation `(*x).y` demande donc 4 caractères en plus des noms des variables `x` et `y` : c'est trop ! Le C introduit la notation équivalente `x->y`, composé d'un tiret et d'un symbole supérieur. On n'utilise plus que deux caractères, soit deux fois moins de caractères !

Exemple 7. L'accès au champ `x` du pointeur `p` s'écrit alors :

```
(*p).x = 13.37;  // Deux écritures équivalentes pour l'accès
p->x = 13.37;    //             à un champ depuis un pointeur
```

Synthèse structures. Définition :

```
struct TypeStruct_s     // définition du nom de la structure
{
  type1 nom1;                // type et nom du premier champ
  type2 nom2;                 // type et nom du second champ
  ...
};      // on n'oublie pas le point-virgule après l'accolade
typedef struct TypeStruct_s TypeStruct; // Renommage de type
```

Déclaration d'une nouvelle variable : `TypeStruct var;`

Déclaration et initialisation : `TypeStruct var = {XXX, YYY, ...};`

Accès aux champs :

- à partir d'une variable, on utilise le point : `var.nom1 = ...;`
- à partir d'un pointeur, on utilise la flèche : `p->nom1 = ...;`

TD **Exercices 5 et 6.**

3 Autres types structurés

3.1 Les unions

Dans une structure, chaque champ possède une adresse différente des autres. En effet, les différents champs sont organisés comme les cases d'un tableau. Il y a autant d'écart entre les adresses de deux champs successifs qu'il y a d'octets formant le premier des deux champs. C'est cette organisation qui nous assure de ne pas modifier le champ X lorsqu'on modifie le champ Y.

Une union n'est qu'une structure particulière. Tout ce qui a été vu avec les structures s'applique sur les unions, en changeant le mot clé `struct` par le mot clé `union` :

- la définition de l'union débute par `union TypeUnion_s { ... champs ... };`
- il est possible de faire un `typedef` : `typedef union TypeUnion_s TypeUnion;`
- la déclaration d'une variable de type `union` n'a dès lors rien d'exceptionnel : `TypeUnion var;`
- l'accès aux champs est identique à celui d'une structure : `var.X = ...` ou, avec un pointeur : `p->X = ...`

Définition 2 (union). Une *union* est une structure dans laquelle tous les champs ont la même adresse.

La différence majeure entre une structure et une union est que cette dernière regroupe l'ensemble des champs à la même adresse. La taille de l'union est donc celle du plus grand élément la composant.

Si le principe est extrêmement simple, il peut paraitre inutile. Pourquoi mettre plusieurs variable sur la même adresse ? La modification d'une de ces variables entrainerait

en effet la modification des autres... Et c'est exactement le but recherché !

L'usage le plus répandu des unions est fait au travers des images. Un pixel est un triplet d'octets RVB, plus un octet de transparence pour les images qui le supportent. Ces quatre octets peuvent alors être stockés de deux façons : un tableau de 4 cases ou un entier de 4 octets. L'entier est intéressant pour des couleurs particulières comme le noir (valeur 0) ou le blanc (tous les octets à 255 soit `0xFFFFFFFF`). L'entier est également intéressant pour l'affichage : il ne nécessite qu'un `printf` alors qu'il en faut 4 pour le tableau. En complément, le tableau de 4 cases est intéressant pour toutes les couleurs quelconques. Ainsi, celui qui souhaite jongler entre ces deux écritures peut utiliser les unions, comme le montre le code suivant.

```
union Pixel_u  // Définition d'une union nommée Pixel_u
{
  unsigned char RGBA[4];      // Premier champ: tableau de 4 octets
  unsigned int code;          // Second champ: entier sur 4 octets
};
typedef union Pixel_u Pixel;  // Redéfinition de type
...
Pixel pix;                    // Déclaration d'un var. de type Pixel
pix.RGBA[0] = 1;              // Accès par le tableau
pix.RGBA[1] = 3;
pix.RGBA[2] = 3;
pix.RGBA[3] = 7;
printf("code pixel: 0x%08X\n", pix.code); // Affiche 0x01030307
```

TD **Exercices 7 et 8.**

3.2 Les énumérations

Définition 3 (énumération). Une *énumération* permet de donner un type à un ensemble de constantes.

Une énumération est donc relativement différente d'une structure. Il n'y a pas de champ à proprement parler. On ne peut pas modifier les éléments constituant une énumération. Le seul usage qui est fait d'une énumération est la déclaration d'une variable pouvant prendre un certain nombre de valeurs définies. La définition d'un nouveau type `enum` se rapproche quant à elle de ce qu'on a vu jusque là. Son utilisation est en revanche plus limitée.

```
enum TypeEnum_e         // Définition d'une nouvelle énumération
{
  VAL_1, VAL_2, ... , VAL_N       // Pas de ; à la fin
};
typedef enum TypeEnum_e TypeEnum; // Redéfinition de type
...
TypeEnum var = VAL_X;  // Déclaration d'une var. de type TypeEnum
var = VAL_Y;           // Modification de la variable
```

L'énumération ainsi créée permet de définir un ensemble de constantes nommées `VAL_1`, `VAL_2` etc. À chaque constante est attribuée une valeur entière de façon automatique. La constante `VAL_1` prend la valeur 0, `VAL_2` prend la valeur 1 etc. Il est également possible d'attribuer manuellement une valeur à une constante en écrivant, lors de la définition `..., VAL_X = Y, ...` La constante suivant `VAL_X` aura alors la valeur `Y+1`, la constante suivante `Y+2` etc.

Exemple 8. Parce que le plus simple est de montrer un exemple, voyons comment les énumérations peuvent caractériser facilement les Peugeot [4]. Intéressons nous à la série de modèles des compactes : 205, 206 etc. On peut alors définir l'énumération suivante.

```
enum Peugeot_e   // Définition du type énuméré Peugeot
{
  PGO_TRES_VIEILLE = 205,
  PGO_VIEILLE,   // Prend automatiquement la valeur 206
  PGO_RECENTE,   // Prend automatiquement la valeur 207
  PGO_NOUVELLE   // Prend automatiquement la valeur 208
};
typedef enum Peugeot_e Peugeot;
```

Si la voiture de vos rêves est une 206, vous pouvez alors le crier au monde entier à l'aide de l'instruction suivante.

```
Peugeot monReve = PGO_VIEILLE;
```

TD **Exercices 9 et 10.**

4 En mémoire

Cette section se consacre exclusivement à la représentation des types structurés en mémoire. Débutons avec le simple d'entre eux, la structure classique. La figure ci-dessous présente l'état de la mémoire à la fin de l'exécution du code.

4. Oui c'est une marque, non nous n'avons pas d'actions chez eux, nous ne gagnerons rien si vous en achetez une. Ils ont simplement des noms de modèles rigolos.

```
1  struct Test_s
2  {
3    char x;
4    char y;
5    short a;
6  };
7  typedef struct Test_s Test;
8
9  int main ()
10 {
11   Test A;
12   Test B = {3, 7, 1337};
13
14   A.x = 0;
15   A.y = B.y;
16   A.a = 42;
17
18   return 0;
19 }
```

Adresse	Valeur	Nom de variable	
0x0160	? 0	A.x	A
0x0161	? 7	A.y	
0x0162	? 42	A.a	
0x0163			
0x0164	3	B.x	B
0x0165	7	B.y	
0x0166	1337	B.a	
0x0167			
0x0168			
0x0169			
0x016A			
0x016B			

On rappelle que les instructions qui ont permis de réserver des emplacements mémoire sont les déclarations de variables des lignes 11 et 12. La définition de la structure des lignes 1 à 7 a simplement permis de décrire comment la mémoire allait être organisée lors de la définition de variables.

On peut alors observer que chaque champ de la variable `A` possède sa propre adresse :

- l'adresse de la variable `A` est `0x0160` ;
- l'adresse du champ `x` de la variable `A` est `0x0160` ;
- l'adresse du champ `y` de la variable `A` est `0x0161` ;
- l'adresse du champ `a` de la variable `A` est `0x0162` ;

Le même procédé s'applique bien évidemment pour la variable `B`. Lors d'une allocation dynamique, l'organisation de la mémoire ne change pas, si ce n'est que les éléments sont alloués sur le tas et non sur la pile.

Dans le cas des unions, l'état de la mémoire est plus difficile à représenter. En effet, chacun des champs possède la même adresse : celle de la variable. Étudions l'exemple suivant.

```
1  union Test2_u
2  {
3    unsigned char tab[2];
4    unsigned int a;
5  };
6  typedef union Test2_u
     ↪ Test2;
7
8  int main ()
9  {
10   Test2 A;
11
12   A.tab[0] = 0;
13   A.tab[1] = 42;
14   A.a = 0xE1FFE165;
15
16   printf ("%hhd %hhd",
       ↪ A.tab[0], A.tab[1]);
17
18   return 0;
19 }
```

Adresse	Valeur octet	Nom variable		
0x0160	? 0 0x65	A.tab[0]	A.a	A
0x0161	? 42 0xE1	A.tab[1]		
0x0162	? 0xFF			
0x0163	? 0xE1			
0x0164				
0x0165				
0x0166				
0x0167				
0x0168				
0x0169				
0x016A				
0x016B				

L'union est définie à partir de deux champs : un tableau de 2 caractères nommé `tab` et un entier nommé `a`. La variable `A` étant déclarée à l'adresse `0x0160`, les deux champs de l'union sont positionnés sur la même adresse. Voyons maintenant l'action des différentes instructions.

1. Ligne 12 : modification de la case 0 du champ `tab` ; l'octet à l'adresse `0x0160` prend la valeur 0.
2. Ligne 13 : modification de la case 1 du champ `tab` ; l'octet à l'adresse `0x0161` prend la valeur 42.
3. Ligne 14 : modification du champ `a` ; les adresses `0x0160` à `0x0163` représentent l'entier `0xE1FFE165`.
4. Ligne 16 : l'instruction affiche les valeurs 101 (0x65) et 225 (0xE1).

Notez que la représentation de l'entier `a` est effectuée en notation « petit-boutiste[5] », c'est à dire que l'entier est écrit octet par octet, en commençant par l'octet de poids faible. Toute valeur entière stockée sur plus de un octet est représentée de cette façon.

TD **Exercice 11.**

5. ou *little-endian* en anglais, c'est tout de suite plus stylé ...