Structures en C

Hicham Janati

1 Introduction

1.1 L'utilité des structures

Supposons que vous programmez un jeu où vous pouvez choisir votre héros parmi une liste de personnages. Chaque personnage est caractérisé par sa taille, sa vitesse (0-100) et sa corpulence (0-100). Pour définir plusieurs personnages, on peut très bien définir trois variables pour chaque personnage du type :

```
1 float taille1 = 1.75;
2 int vitesse1 = 80;
3 int corpulence1 = 44;
4 float taille2 = 1.9;
5 int vitesse2 = 25;
6 int corpulence2 = 94;
7 // .. etc
```

Pas mal, mais avec beaucoup de personnages et beaucoup de caractéristiques, notre programme sera très difficile à gérer et à comprendre. C nous permet de mieux structurer notre code avec des *structures !*

Une structure est un **type** de variables personnalisé. Vous pouvez donc définir votre première structure que l'on appellera *Personnage* comme suit :

```
1 struct Personnage{
2     float taille;
3     int vitesse;
4     int corpulence;
5 };
```

Et créer plusieurs personnages dans votre main :

```
1
2 // Définition du TYPE 'Personnage':
3 struct Personnage{
4     float taille;
5     int vitesse;
6     int corpulence;
7 };
8
9 int main(){
10     // Création de plusieurs VARIABLES de type 'Personnage':
11     struct Personnage heros1;
12     struct Personnage heros2;
13     struct Personnage heros3;
14     return 0;
15     }
```

Nous avons créés ci-dessus trois personnages nommés heros1, heros2 et heros3. Pour accéder à leurs caractéristiques (taille, vitesse, corpulence), il suffit de suivre leur nom d'un point puis du nom de la variable. Ainsi pour reprendre l'exemple du début, on peut compléter notre main par :

```
1  // saisie manuelle
2 heros1.taille =  1.75;
```

```
heros1.vitesse = 80;
heros1.corpulence = 44;
heros2.taille = 1.9;

// saisie par l'utilisateur:
printf("Quelle est la vitesse de votre héros?\n");
scanf("%d",&heros2.vitesse);
// .. etc

printf("La taille de heros1: %f\n", heros1.taille);
printf("La vitesse de heros2: %d\n", heros2.vitesse);
```

Ainsi, vous pouvez traiter les attributs de la structure comme des variables !

1. **Reprenez la définition de la structure Personnage et créez un personnage en saisissant au clavier les 3 caractéristiques puis affichez-les à l'écran.**

1.2 Une bonne habitude

Devoir écrire struct au début de chaque déclaration de variable est .. fatiguant non ? Ce serait bien de pouvoir déclarer une variable de type Personnage, comme on déclare un float par exemple : *Personnage NomDeLaVariable*. Eh ben c'est possible en utilisant un alias !

Vous pouvez créer un raccourci avec typedef : en gros, vous dites à C de remplacer l'instruction "struct Personnage" par un seul mot "Personnage" :

```

typedef struct Personnage Personnage;
struct Personnage{
    float taille;
    int vitesse;
    int corpulence;
};

int main(){
    // Nous n'avons plus besoin de struct à la déclaration:
    // "Personnage" est désormais équivalent à "struct Personnage"
    Personnage heros1;
    Personnage heros2;
    Personnage heros3;
    return 0;
    }
```

1.3 Initialisation des structures

On peut initialiser nos structures comme avec les tableaux. À la déclaration d'un personnage Monstre par exemple, vous pouvez lui attribuer les caractéristiques de la façon suivante.

```

int main(){
    Personnage Monstre  = {1.95, 94, 81};
    printf("Taille de monstre : %f\n",Monstre.taille);
    printf("Vitesse de monstre : %d\n",Monstre.vitesse);
    printf("Vitesse de monstre : %d\n",Monstre.corpulence);
    return 0;
    }
```

2. **L'initialisation se fait donc dans l'ordre des caractéristiques de la structure. Que contient Monstre si l'on ne précise aucune, une ou seulement deux valeurs ? Pour y répondre, affichez les attributs des personnages suivants :**

```
1     Personnage Monstre1;
2     Personnage Monstre2 = {0};
3     Personnage Monstre3 = {34};
4     Personnage Monstre3 = {3.3, 90};
```

L'initialisation des structures est donc très similaire à celle des tableaux. Mais celle-ci devient lourde lorsqu'il y a beaucoup d'attributs : il faut faire attention à l'ordre dans lequel ils sont définis. C'est pourquoi il est préférable d'initialiser les structures avec des fonctions ! On verra cela un peu plus bas.

1.4 Tableaux comme attributs

Cool, on a créé nos personnages avec différentes variables de vitesse, corpulence et taille. Et si l'on veut aussi ajouter leur 10 dernières performances ? Ou leur attribuer un nom ? Les munir d'un texte descriptif ? Vous l'avez deviné : on peut également définir un tableau ou une chaîne de caractères comme attribut d'une structure :

```
1  typedef struct Personnage Personnage;
2  struct Personnage{
3      float taille;
4      int vitesse;
5      int corpulence;
6      char description[100];
7
8  };
9
10 int main(){
11     Personnage Heros = {189.3,5,90, "Monstreux ! Mais trop lent !"};
12     return 0;
13     }
```

Rappel chaînes :

Vous pouvez laisser l'utilisateur saisir la description du personnage avec le format %s des *string*. Ainsi, on peut remplir les variables d'un personnage comme suit :

```
1
2  int main(){
3      Personnage Monstre;
4      printf("La taille ? \n");
5      scanf("%f",&Monstre.taille);
6      printf("La vitesse ? \n");
7      scanf("%d",&Monstre.vistesse);
8      printf("La corpulence? \n");
9      scanf("%d",&Monstre.corpulence);
10     printf("La description ? \n");
11     scanf("%s",Monstre.description);
12     return 0;
13     }
```

Rappel pointeurs :


Pourquoi n'a-t-on pas mis un & avant Monstre.description ?

Monstre.description est une chaîne de caractères, donc ... un tableau. On a vu que le nom d'un tableau est en fait un pointeur sur sa première case. Ainsi, Monstre.description est bien l'adresse de notre chaîne, et à scanf on donne toujours l'adresse de la variable à saisir.

Conclusion

— Une structure est un type de variable personnalisé qui peut comporter plusieurs variables et tableaux que l'on désigne par *attributs*.

— Les attributs d'une structure peuvent être de type différent.

— On définit une structure de la manière suivante, avant le main :

```
1 struct NomDeLaStructure{
2 type1 attribut1;
3 type2 attribut2;
4 ...
5 };
```

— On peut créer, dans le main, plusieurs variables de type *NomDeLaStructure* dites *instances* de la structure :

```
1 struct NomDeLaStructure instance1;
2 struct NomDeLaStructure instance2;
```

— Il est possible de s'en passer du "struct" dans le main à condition de définir un alias avant le main :

```
1 typedef struct NomDeLaStructure NomDeLaStructure;
```

— On accède aux attributs des instances de structures avec un point :

```
1 instance1.attribut1
```

2 Fonctions opérant sur des structures

Un bon programmeur doit non seulement assurer que son code fonctionne mais qu'il est aussi compréhensible par une personne n'ayant aucun a priori sur le sujet. Toujours dans l'esprit des bonnes habitudes, nous allons dorénavant définir nos structures dans les fichiers .h, c'est à dire avec les prototypes des fonctions (s'il y en a). À gauche notre fichier main.c, à droite notre fichier "fonctions.h". Pour l'instant, le fichier "fonctions.c" ne contient rien !

main.c

```
1 #include<stdio.h>
2 #include"fonctions.h"
3
4 int main(){
5     Personnage Heros;
6     return 0;
7     }
```

fonctions.h

```
1 typedef struct Personnage Personnage;
2 struct Personnage{
3     float taille;
4     int vitesse;
5     int corpulence;
6     char description[100];
7 };
```

Nous allons définir une fonction qui va prendre une structure en paramètre pour l'initialiser. Or, comme vous le savez déjà (cf Pointeurs), pour modifier une variable en mémoire par une fonction, il faut faire un passage par adresse : donc un pointeur sur une structure !

On définit un pointeur sur une structure de la même manière qu'avec les autres types de variables :

```
1 #include<stdio.h>
2 #include"fonctions.h"
3
4 int main(){
5     Personnage Heros;
6     Personnage* P_Heros;
7     P_Heros = &Heros;
```

```
8       return 0;
9       }
```

Ainsi, si P_Heros est un pointeur sur Heros, *P_Heros est équivalent à Heros. Pour accéder aux attributs de Heros à travers le pointeur, il suffit d'ajouter un point puis le nom de l'attribut non ?
En effet, mais Attention ! Il faut préciser que l'* ne concerne que le pointeur sur la structure. Il faut donc éviter de faire :

CECI EST FAUX

*P_Heros.vitesse

Mais plutôt :

Là ça va

(*P_Heros).vitesse

3. **Créez une fonction InitialiserHeros qui prend en paramètre un pointeur sur une structure de type Personnage et qui lui attribue les valeurs de votre choix.**

```
1 void InitialiserHeros(Personnage *pers){
2
3
4
5
6
7 }
```

Il existe un raccourci utile qui peut vous simplifier la vie : lorsqu'on travaille avec des pointeurs sur des structures, on a tendance à écrire (*Structure).attribut des centaines de fois ... et cette écriture peut être parfois dérangeante (Les programmeurs sont maniaques ...). Il existe une forme plus simple : Si P_Heros est toujours un pointeur sur une structure Personnage, on peut de manière équivalente au cadre ci-dessus écrire :

Là ça va

P_Heros-> vitesse

3 Listes chaînées

Pour manipuler plusieurs *instances* de votre structure, il est tout à fait possible de créer un tableau de structures ! Pour créer un tableau à 10 Personnages par exemple, on peut :

```
1 Personnage TableauDePersonnages[10];
```

Chaque case TableauDePersonnages[i] est donc une structure Personnage. Et vous pouvez donc accéder à ses attributs avec TableauDePersonnages[i] où i est bien entendu < 10.

Imaginez que vous avez un tableau à 1000 cases et que vous souhaitez insérer un personnage au milieu, il faudra translater beaucoup beaucoup de cases : donc effectuer beaucoup beaucoup de copies. Et si votre tableau était déjà tout rempli, vous ne pouvez plus changer sa taille pour ajouter une case ! Pour y remédier, on utilise .. les pointeurs ! Au lieu d'avoir nos variables (structures) juxtaposées dans la mémoire (avec un tableau), nous allons les éparpiller (les définir indépendamment) mais en ajoutant un lien entre elles : des pointeurs ! Ainsi, on ajoute à chaque structure un pointeur vers une autre structure, formant une *chaîne* !

Liste chaînée

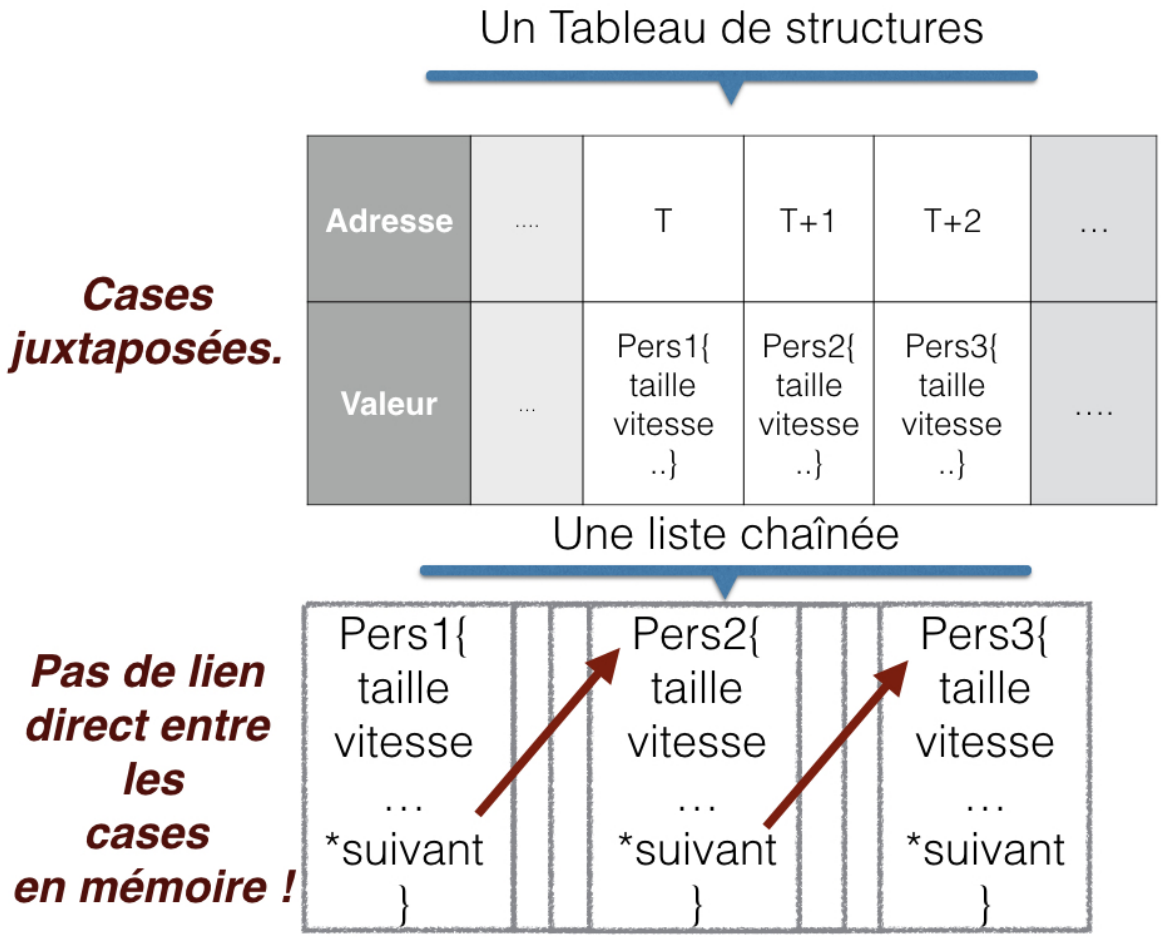
Une liste chaînée est un ensemble d'instances d'une même structure ayant comme attribut un pointeur sur la même structure.

Dit comme ça, ça reste abstrait. Un exemple !

Imaginons que nous souhaitons organiser une fille d'attente avec nos personnages. Chaque personnage est situé devant un autre etc ... Nous modifions notre structure comme suit :

```
1 struct Personnage{
2     float taille;
3     int vitesse;
4     int corpulence;
5     char description[100];
6     Personnage * suivant;
7 };
```

pers_suivant est un pointeur sur un autre personnage ! À travers ce pointeur, on pourra passer d'un élément à un autre.



Il reste néanmoins un élément important à définir : les extrémités de la chaîne. Le dernier élément de la chaîne pointe en général sur NULL. Ainsi, à la création d'une liste chaînée, on a en général une seule structure dont l'attribut *suivant* pointe sur NULL.