

# 5

## Introduction au langage C Structures - Fichiers



1

# Structures

Lorsqu'on veut définir une variable contenant plusieurs éléments qui ne sont pas tous du même type, on ne peut pas utiliser le type tableau. On définit alors un **type structuré** qu'on appellera **structure**.

Par définition, une **structure** est une suite finie d'objets de types différents.

Pour la déclaration, on procède en deux étapes :

- 1 on déclare un modèle de structure, en expliquant quels types déjà existants entrent dans sa composition, et on leur donne des noms,
- 2 puis on déclare une (ou des) variable(s) qui sont des objets de ce type

Les différents éléments qui interviennent dans une structure sont appelés des **champs**.  
Chaque champ reçoit un nom.

Pour déclarer une variable **bidule**, du type structuré **Modele**, on adopte donc le schéma suivant :

1 Structure **Modele**

type\_1 champ\_1

type\_2 champ\_2

...

type\_n champ\_n

FinStructure

2 Structure **Modele** **bidule**

Les différents éléments sont repérés **par le nom du champ** (et non plus par un indice). On y accède en faisant suivre le nom de la variable par un point puis par le nom du champ :  
`bidule.champ_i`

On peut aller plus loin en définissant un nouveau type ayant le nom du modèle de structure. Exemple : on pose **Point** comme étant un nouveau nom de type qu'on utilisera par la suite comme tous les autres types déjà connus.

Extension : ceci rejoint le principe de base des langages orientés objets : les **classes** ne sont autres que des **types abstraits** qui étendent la notion de type structuré. En POO<sup>1</sup>, chaque classe est caractérisée par des **attributs**, qui correspondent aux champs de nos types structurés, et des **méthodes** qui définissent les codes des manipulations autorisées sur tous les objets de cette classe.

---

1. POO = Programmation Orientée Objet

N.B. Même si les champs sont, finalement, d'un même type, on peut avoir intérêt à déclarer une variable dans un type structuré plutôt que dans un tableau.

```
1  #include <math.h>
2  // déclaration du modèle de type structuré
3  struct Complexe{
4      double reel; // partie réelle
5      double imag; // partie imaginaire
6  };
7  int main(){
8      // déclaration d'une variable de type Complexe
9      struct Complexe z;
10     // manipulations sur cette variable
11     float norme = sqrt(z.reel * z.reel + z.imag * z.imag);
12     printf("norme de (%f + i %f) = %f \n",z.reel,z.imag,norme);
13     return 0;
14 }
```

Le mot-clé **typedef** permet de définir un **nouveau nom de type**. Reprenons l'exemple ci-dessus :

```
1  #include <math.h>
2  // déclaration du modèle de type structuré
3  struct Complexe{
4      double reel; // partie réelle
5      double imag; // partie imaginaire
6  };
7  typedef struct Complexe complexe; // on définit ici le nouveau type
8  int main(){
9      // déclaration d'une variable de type Complexe
10     complexe z;
11     // manipulations sur cette variable
12     float norme = sqrt(z.reel * z.reel + z.imag * z.imag);
13     printf("norme de (%f + i %f) = %f \n",z.reel,z.imag,norme);
14     return 0;
15 }
```



L'utilisation de typedef permet de simplifier la déclaration de la structure

```
1  #include <math.h>
2  // déclaration du modèle de type structuré
3  typedef struct Complexe{
4      double reel; // partie réelle
5      double imag; // partie imaginaire
6  }Complexe;
7  int main(){
8      // déclaration d'une variable de type Complexe
9      Complexe z;
10     // manipulations sur cette variable
11     float norme = sqrt(z.reel * z.reel + z.imag * z.imag);
12     printf("norme de (%f + i %f) = %f \n",z.reel,z.imag,norme);
13     return 0;
14 }
```

On peut initialiser les champs d'une structure lors de sa déclaration

```
1  #include <math.h>
2  // déclaration du modèle de type structuré
3  typedef struct Complexe{
4      double reel; // partie réelle
5      double imag; // partie imaginaire
6  }Complexe;
7  int main(){
8      // déclaration d'une variable de type Complexe
9      Complexe z={3.5,5.5};
10     // manipulations sur cette variable
11     float norme = sqrt(z.reel * z.reel + z.imag * z.imag);
12     printf("norme de (%f + i %f) = %f \n",z.reel,z.imag,norme);
13     return 0;
14 }
```

Si **ptr** est un pointeur sur une **structure**, on peut accéder à un champ de la structure pointée par l'expression :

`(*ptr).champ`

Les **parenthèses sont indispensables** car l'opérateur d'indirection `*` a une priorité plus faible que l'opérateur de champ de structure.

Cette notation peut être simplifiée grâce à l'opérateur **pointeur de champ** de structure, noté `->` et donc l'expression précédente est strictement équivalente à :

`ptr->champ`



2

## Enumérations

### Définition d'un ensemble de constante

- Regroupement sous un seul "type"
- Affectation manuelle de la valeur
- affectation automatique lors de la définition de l'énumération :
  - Première constante : 0
  - Seconde constante : 1
  - ...

```
1 // Définition
2 enum NomEnum{
3     VAL1, VAL2, ..., VALN
4 };
5 // Déclaration
6 enum NomEnum nomVar;
7 // Affectation
8 nomVar = VAL2;
```

```
1  typedef enum {
2      ORANGE = 1,
3      BLEU,
4      ROUGE,
5      VERT
6  } Couleur;
7  int main(int argc, char ** argv)
8  {
9      Couleur coul1;
10     coul1 = ROUGE;
11     printf("%d\n",coul1);
12     return 0;
13 }
```



3

Unions

Une zone mémoire pour un ensemble d'éléments

- Taille de la zone = taille du plus grand élément
- La modification d'un élément implique la modification des autres

```
1 // Définition
2 union uNomUnion{
3     type1 var1;
4     type2 var2;
5     //...
6 };
7 // Déclaration
8 union uNomUnion nomVar;
9 // Affectation
10 nomVar.var1 = x;
```



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef struct eVector3{
4      enum TYPE {REEL, COULEUR};
5      union{
6          struct{ float x, y, z; };
7          struct{ int r, g, b; };
8      };
9  } Vector3D;
10 void affiche_vect(Vector3D v){
11     if (v.TYPE = REEL) printf("%f %f %f \n", v.x,v.y,v.z);
12     else if (v.TYPE = COULEUR) printf("%d %d %d\n", v.r,v.g,v.b);
13 }
14
```

```
1  int main(){
2      Vector3D couleur;
3      couleur.TYPE = COULEUR;
4      couleur.r = 20;couleur.g = 30;couleur.b = 40;
5      affiche_vect(couleur);
6
7      Vector3D vec;
8      vec.TYPE = REEL;
9      vec.x = 20.5;vec.y = 30.5;vec.z = 40.5;
10     affiche_vect(vec);
11
12
13     return 0;
14 }
```



4

Fichiers

- Un fichier est une série d'octets enregistrée sur le disque dur d'une machine.
- Entrées/sorties connues : clavier/console (`stdin stdout stderr`).
- Possède un chemin pour l'identifier :
- Ex : `/home/user/Documents/img.png`
  - `C:\\Users\\user\\Documents\\img.png`
  - `../../others/img.png`

- Possibilité de lire, écrire, se déplacer dans des fichiers
- Utilisation :
  - Ouverture
  - Lecture / écriture / déplacement
  - Fermeture
- Utilisation d'un pointeur sur fichier : FILE \*
  - Equivalent à un curseur
  - Se déplacer en même temps que l'écriture ou la lecture

## ■ Ouverture

```
FILE * fopen (const char * filename, const char * mode);
```

## ■ modes :

Mode	Description	Fichier existant ?	Création auto ?	Position du pointeur
r	Lecture	Obligatoire	NON	Début
w	Ecriture	Ecrase	OUI	Début
a	Ajout	–	OUI	Fin
r+	Lecture + écriture	Obligatoire	NON	Début
w+	Ecriture + lecture	Ecrase	OUI	Début
a+	Ajout + lecture	–	OUI	Dépendant actions
b	Mode binaire	/	/	/

## ■ Lecture

```
FILE * fread (void *ptr, size_t size, size_t count, FILE * stream);
```

- void \* ptr : buffer de réception des données
- size\_t size : taille en octet de chaque élément
- size\_t count : nombre d'élément
- FILE \* stream : pointeur sur le fichier

## ■ Ecriture

```
FILE * fwrite (const void *ptr, size_t size, size_t count, FILE * stream);
```

- const void \*ptr : buffer à écrire
- size\_t size : taille en octet de chaque élément
- size\_t count : nombre d'élément
- FILE \* stream : pointeur sur le fichier

## ■ Fermeture

```
FILE * fclose (FILE * stream);
```

- FILE \* stream : pointeur sur le fichier

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main (int argc, char **argv){
4      FILE *pInfile = NULL;
5      char c;
6      if (argc<2) return EXIT_FAILURE; // Test des arguments
7      pInfile = fopen (argv[1], "r"); // Ouverture du fichier
8      if (!pInfile){ // Test de l'ouverture
9          printf ("Impossible d'ouvrir le fichier %s\n", argv[1]);
10         return EXIT_FAILURE;
11     }
12     do{ // Lecture et affichage des caractères
13         fread ( &c, sizeof(char), 1, pInfile);
14         printf ("%c", c);
15     }while (!feof(pInfile)); // Fin de fichier
16     fclose(pInfile); // Fermeture du fichier
17     return EXIT_SUCCESS;}
```



## ■ Lecture

```
1 FILE * fread ( void *ptr, size_t size, size_t count, FILE * stream );
2 int fscanf ( FILE * stream, const char * format, ... );
3 int fgetc ( FILE * stream );
4 char * fgets ( char * str, int num, FILE * stream );
```

## ■ Ecriture

```
1 FILE * fwrite ( const void *ptr, size_t size, size_t count, FILE *stream );
2 int fprintf ( FILE * stream, const char * format, ... );
3 int fputc ( int character, FILE * stream );
4 int fputs ( const char * str, FILE * stream );
```

## ■ Navigation dans le fichier

```
1 int fseek ( FILE * stream, long int offset, int origin );
2 long int ftell ( FILE * stream );
3 void rewind ( FILE * stream );
```