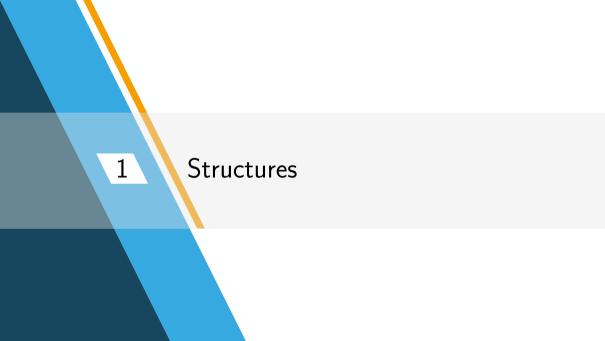
Introduction au langage C Structures - Fichiers





Lorsqu'on veut définir une variable contenant plusieurs éléments qui ne sont pas tous du même type, on ne peut pas utiliser le type <u>tableau</u>. 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 :

- on déclare un <u>modèle</u> de structure, en expliquant quels types déjà existants entrent dans sa composition, et on leur donne des noms,
- 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 :

```
Structure Modele

type_1 champ_1

type_2 champ_2

...

type_n champ_n

FinStructure
```

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 <u>langages orientés objets</u> : 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.

<sup>1.</sup> 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.

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

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

```
#include <math.h>
    // déclaration du modèle de type structuré
    struct Complexe{
3
      double reel; // partie réelle
      double imag; // partie imaginaire
5
    typedef struct Complexe complexe; // on définit ici le nouveau type
    int main(){
      // 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);
      printf("norme de (\%f + i \%f) = \%f \n",z.reel,z.imag,norme);
13
      return 0:
14
15
```

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

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

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

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

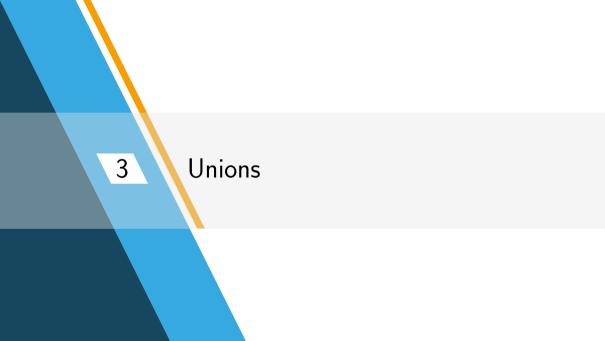
## Définition d'un ensemble de constante

- 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
  - . . . .

```
// Définition
enum NomEnum{
VAL1, VAL2, ..., VALN
};
// Déclaration
enum NomEnum nomVar;
// Affectation
nomVar = VAL2;
```

11 Exemple

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



12 Union

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

```
// Définition
union uNomUnion{
    type1 var1;
    type2 var2;
    //...
};
// Déclaration
union uNomUnion nomVar;
// Affectation
nomVar.var1 = x;
```

13

# Exemple

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

14

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



15 Fichier

- Un fichier est un 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

17 Fichier

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
а	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
```

```
#include <stdio.h>
    #include <stdlib.h>
    int main (int argc, char **argv){
      FILE *pInfile = NULL;
      char c:
5
      if (argc<2) return EXIT_FAILURE; // Test des arguments
      pInfile = fopen (argv[1], "r"); // Ouverture du fichier
      if (!pInfile){ // Test de l'ouverture
        printf ("Impossible d'ouvrir le fichier %s\n", argv[1]);
9
        return EXIT_FAILURE;
10
1.1
      do{ // Lecture et affichage des caractères
        fread ( &c, sizeof(char), 1, pInfile);
13
        printf ("%c", c);
14
        }while (!feof(pInfile)); // Fin de fichier
15
      fclose(pInfile); // Fermeture du fichier
16
      return EXIT_SUCCESS;}
17
```

#### Lecture

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

### Ecriture

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

## Navigation dans le fichier

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